

VII

Applications

- 48 IP Router Tables** *Sartaj Sahni, Kun Suk Kim, and Haibin Lu* **48-1**
Introduction • Longest Matching-Prefix • Highest-Priority Matching • Most-Specific-Range Matching
- 49 Multi-Dimensional Packet Classification** *Pankaj Gupta* **49-1**
Introduction • Performance Metrics for Classification Algorithms • Classification Algorithms • Summary
- 50 Data Structures in Web Information Retrieval** *Monika Henzinger* **50-1**
Introduction • Inverted Indices • Fingerprints • Finding Near-Duplicate Documents • Conclusions
- 51 The Web as a Dynamic Graph** *S. N. Maheshwari* **51-1**
Introduction • Experimental Observations • Theoretical Growth Models • Properties of Web Graphs and Web Algorithmics • Conclusions
- 52 Layout Data Structures** *Dinesh P. Mehta* **52-1**
Introduction • VLSI Technology • Layout Data Structures: an Overview • Corner Stitching • Corner Stitching Extensions • Quad Trees and Variants • Concluding Remarks
- 53 Floorplan Representation in VLSI** *Zhou Feng, Bo Yao, and Chung-Kuan Cheng* **53-1**
Introduction • Graph Based Representations • Placement Based Representations • Relationships of the Representations • Rectilinear Shape Handling • Conclusions • Acknowledgment
- 54 Computer Graphics** *Dale McMullin and Alyn Rockwood* **54-1**
Introduction • Basic Applications • Data Structures • Applications of Previously Discussed Structures
- 55 Geographic Information Systems** *Bernhard Seeger and Peter Widmayer* **55-1**
Geographic Information Systems: What They Are All About • Space Filling Curves: Order in Many Dimensions • Spatial Join • Models, Toolboxes and Systems for Geographic Information
- 56 Collision Detection** *Ming C. Lin and Dinesh Manocha* **56-1**
Introduction • Convex Polytopes • General Polygonal Models • Penetration Depth Computation • Large Environments
- 57 Image Data Structures** *S. S. Iyengar, V. K. Vaishnavi, and S. Gunasekaran* **57-1**
Introduction • What is Image Data? • Quadrees • Virtual Quadrees • Quadrees and R-trees • Octrees • Translation Invariant Data Structure (TID) • Content-Based Image Retrieval System • Summary • Acknowledgments

58	Computational Biology	<i>Stefan Kurtz and Stefano Lonardi</i>	58-1
	Introduction • Discovering Unusual Words • Comparing Whole Genomes		
59	Elimination Structures in Scientific Computing	<i>Alex Pothen and Sivan Toledo</i>	59-1
	The Elimination Tree • Applications of Etrees • The Clique Tree • Clique Covers and Quotient Graphs • Column Elimination Trees and Elimination DAGS		
60	Data Structures for Databases	<i>Joachim Hammer and Markus Schneider</i>	60-1
	Overview of the Functionality of a Database Management System • Data Structures for Query Processing • Data Structures for Buffer Management • Data Structures for Disk Space Management • Conclusion		
61	Data Mining	<i>Vipin Kumar, Pang-Ning Tan, and Michael Steinbach</i>	61-1
	Introduction • Classification • Association Analysis • Clustering • Conclusion		
62	Computational Geometry: Fundamental Structures	<i>Mark de Berg and Bettina Speckmann</i>	62-1
	Introduction • Arrangements • Convex Hulls • Voronoi Diagrams • Triangulations		
63	Computational Geometry: Proximity and Location	<i>Sunil Arya and David M. Mount</i>	63-1
	Introduction • Point Location • Proximity Structures • Nearest Neighbor Searching • Sources and Related Material		
64	Computational Geometry: Generalized Intersection Searching	<i>Prosenjit Gupta, Ravi Janardan, and Michiel Smid</i>	64-1
	Geometric Intersection Searching Problems • Summary of Known Results • Techniques • Conclusion and Future Directions • Acknowledgment		

48

IP Router Tables

Sartaj Sahni <i>University of Florida</i>	48.1	Introduction.....	48-1
Kun Suk Kim <i>University of Florida</i>	48.2	Longest Matching-Prefix Linear List • End-Point Array • Sets of Equal-Length Prefixes • Tries • Binary Search Trees • Priority Search Trees	48-4
Haibin Lu <i>University of Florida</i>	48.3	Highest-Priority Matching The Data Structure BOB • Search for the Highest-Priority Matching Range	48-18
	48.4	Most-Specific-Range Matching Nonintersecting Ranges • Conflict-Free Ranges	48-21

48.1 Introduction

An Internet router classifies incoming packets into flows* utilizing information contained in packet headers and a table of (classification) rules. This table is called the *rule table* (equivalently, *router table*). The packet-header information that is used to perform the classification is some subset of the source and destination addresses, the source and destination ports, the protocol, protocol flags, type of service, and so on. The specific header information used for packet classification is governed by the rules in the rule table. Each rule-table rule is a pair of the form (F, A) , where F is a filter and A is an action. The action component of a rule specifies what is to be done when a packet that satisfies the rule filter is received. Sample actions are drop the packet, forward the packet along a certain output link, and reserve a specified amount of bandwidth. A rule filter F is a tuple that is comprised of one or more fields. In the simplest case of destination-based packet forwarding, F has a single field, which is a destination (address) prefix and A is the next hop for packets whose destination address has the specified prefix. For example, the rule $(01*, a)$ states that the next hop for packets whose destination address (in binary) begins with 01 is a . IP (Internet Protocol) multicasting uses rules in which F is comprised of the two fields source prefix and destination prefix; QoS routers may use five-field rule filters (source-address prefix, destination-address prefix, source-port range, destination-port range, and protocol); and firewall filters may have one or more fields.

In the d -dimensional packet classification problem, each rule has a d -field filter. In this chapter, we are concerned solely with 1-dimensional packet classification. It should be noted, that data structures for multidimensional packet classification are usually built on top of

*A **flow** is a set of packets that are to be treated similarly for routing purposes.

Prefix Name	Prefix	Range Start	Range Finish
P1	*	0	31
P2	0101*	10	11
P3	100*	16	19
P4	1001*	18	19
P5	10111	23	23

FIGURE 48.1: Prefixes and their ranges.

data structures for 1-dimensional packet classification. Therefore, the study of data structures for 1-dimensional packet classification is fundamental to the design and development of data structures for d -dimensional, $d > 1$, packet classification.

For the 1-dimensional packet classification problem, we assume that the single field in the filter is the destination field and that the action is the next hop for the packet. With these assumptions, 1-dimensional packet classification is equivalent to the destination-based packet forwarding problem. Henceforth, we shall use the terms rule table and router table to mean tables in which the filters have a single field, which is the destination address. This single field of a filter may be specified in one of two ways:

1. *As a range.* For example, the range $[35, 2096]$ matches all destination addresses d such that $35 \leq d \leq 2096$.
2. *As an address/mask pair.* Let x_i denote the i th bit of x . The address/mask pair a/m matches all destination addresses d for which $d_i = a_i$ for all i for which $m_i = 1$. That is, a 1 in the mask specifies a bit position in which d and a must agree, while a 0 in the mask specifies a don't care bit position. For example, the address/mask pair 101100/011101 matches the destination addresses 101100, 101110, 001100, and 001110.

When all the 1-bits of a mask are to the left of all 0-bits, the address/mask pair specifies an address prefix. For example, 101100/110000 matches all destination addresses that have the prefix 10 (i.e., all destination addresses that begin with 10). In this case, the address/mask pair is simply represented as the prefix 10^* , where the $*$ denotes a sequence of don't care bits. If W is the length, in bits, of a destination address, then the $*$ in 10^* represents all sequences of $W - 2$ bits. In IPv4 the address and mask are both 32 bits, while in IPv6 both of these are 128 bits.

Notice that every prefix may be represented as a range. For example, when $W = 6$, the prefix 10^* is equivalent to the range $[32, 47]$. A range that may be specified as a prefix for some W is called a *prefix range*. The specification 101100/011101 may be abbreviated to $?011?0$, where $?$ denotes a don't-care bit. This specification is not equivalent to any single range. Also, the range specification $[3,6]$ isn't equivalent to any single address/mask specification.

Figure 48.1 shows a set of five prefixes together with the start and finish of the range for each. This figure assumes that $W = 5$. The prefix $P1 = *$, which matches all legal destination addresses, is called the *default* prefix.

Suppose that our router table is comprised of five rules R1–R5 and that the filters for these five rules are P1–P5, respectively. Let N1–N5, respectively, be the next hops for these five rules. The destination address 18 is matched by rules R1, R3, and R5 (equivalently, by prefixes P1, P3, and P5). So, N1, N3, and N5 are candidates for the next hop for incoming packets that are destined for address 18. Which of the matching rules (and associated action) should be selected? When more than one rule matches an incoming packet, a *tie*

occurs. To select one of the many rules that may match an incoming packet, we use a *tie breaker*.

Let RS be the set of rules in a rule table and let FS be the set of filters associated with these rules. $rules(d, RS)$ (or simply $rules(d)$ when RS is implicit) is the subset of rules of RS that match/cover the destination address d . $filters(d, FS)$ and $filters(d)$ are defined similarly. A tie occurs whenever $|rules(d)| > 1$ (equivalently, $|filters(d)| > 1$).

Three popular tie breakers are:

1. *First matching rule in table.* The rule table is assumed to be a linear list ([15]) of rules with the rules indexed 1 through n for an n -rule table. The action corresponding to the first rule in the table that matches the incoming packet is used. In other words, for packets with destination address d , the rule of $rules(d)$ that has least index is selected.

For our example router table corresponding to the five prefixes of Figure 48.1, rule R1 is selected for every incoming packet, because P1 matches every destination address. When using the first-matching-rule criteria, we must index the rules carefully. In our example, P1 should correspond to the last rule so that every other rule has a chance to be selected for at least one destination address.

2. *Highest-priority rule.* Each rule in the rule table is assigned a priority. From among the rules that match an incoming packet, the rule that has highest priority wins is selected. To avoid the possibility of a further tie, rules are assigned different priorities (it is actually sufficient to ensure that for every destination address d , $rules(d)$ does not have two or more highest-priority rules).

Notice that the first-matching-rule criteria is a special case of the highest-priority criteria (simply assign each rule a priority equal to the negative of its index in the linear list).

3. *Most-specific-rule matching.* The filter $F1$ is **more specific** than the filter $F2$ iff $F2$ matches all packets matched by $F1$ plus at least one additional packet. So, for example, the range $[2, 4]$ is more specific than $[1, 6]$, and $[5, 9]$ is more specific than $[5, 12]$. Since $[2, 4]$ and $[8, 14]$ are disjoint (i.e., they have no address in common), neither is more specific than the other. Also, since $[4, 14]$ and $[6, 20]$ intersect[†], neither is more specific than the other. The prefix 110^* is more specific than the prefix 11^* .

In most-specific-rule matching, ties are broken by selecting the matching rule that has the most specific filter. When the filters are destination prefixes, the most-specific-rule that matches a given destination d is the longest[‡] prefix in $filters(d)$. Hence, for prefix filters, the most-specific-rule tie breaker is equivalent to the longest-matching-prefix criteria used in router tables. For our example rule set, when the destination address is 18, the longest matching-prefix is P4.

When the filters are ranges, the most-specific-rule tie breaker requires us to select the most specific range in $filters(d)$. Notice also that most-specific-range matching is a special case of the the highest-priority rule. For example, when the filters are prefixes, set the prefix priority equal to the prefix length. For the case of ranges, the range priority equals the negative of the range size.

[†]Two ranges $[u, v]$ and $[x, y]$ intersect iff $u < x \leq v < y \vee x < u \leq y < v$.

[‡]The length of a prefix is the number of bits in that prefix (note that the $*$ is not used in determining prefix length). The length of P1 is 0 and that of P2 is 4.

In a *static* rule table, the rule set does not vary in time. For these tables, we are concerned primarily with the following metrics:

1. *Time required to process an incoming packet.* This is the time required to search the rule table for the rule to use.
2. *Preprocessing time.* This is the time to create the rule-table data structure.
3. *Storage requirement.* That is, how much memory is required by the rule-table data structure?

In practice, rule tables are seldom truly static. At best, rules may be added to or deleted from the rule table infrequently. Typically, in a “static” rule table, inserts/deletes are batched and the rule-table data structure reconstructed as needed.

In a *dynamic* rule table, rules are added/deleted with some frequency. For such tables, inserts/deletes are not batched. Rather, they are performed in real time. For such tables, we are concerned additionally with the time required to insert/delete a rule. For a dynamic rule table, the initial rule-table data structure is constructed by starting with an empty data structures and then inserting the initial set of rules into the data structure one by one. So, typically, in the case of dynamic tables, the preprocessing metric, mentioned above, is very closely related to the insert time.

In this paper, we focus on data structures for static and dynamic router tables (1-dimensional packet classification) in which the filters are either prefixes or ranges.

48.2 Longest Matching-Prefix

48.2.1 Linear List

In this data structure, the rules of the rule table are stored as a linear list ([15]) L . Let $LMP(d)$ be the longest matching-prefix for address d . $LMP(d)$ is determined by examining the prefixes in L from left to right; for each prefix, we determine whether or not that prefix matches d ; and from the set of matching prefixes, the one with longest length is selected. To insert a rule q , we first search the list L from left to right to ensure that L doesn't already have a rule with the same filter as does q . Having verified this, the new rule q is added to the end of the list. Deletion is similar. The time for each of the operations determine $LMP(d)$, insert a rule, delete a rule is $O(n)$, where n is the number of rules in L . The memory required is also $O(n)$.

Note that this data structure may be used regardless of the form of the filter (i.e., ranges, Boolean expressions, etc.) and regardless of the tie breaker in use. The time and memory complexities are unchanged.

Although the linear list is not a suitable data structure for a purely software implementation of a router table with a large number of prefixes, it leads to a very practical solution using TCAMs (ternary content-addressable memories) [28, 34]. Each memory cell of a TCAM may be set to one of three states 0, 1, and don't care. The prefixes of a router table are stored in a TCAM in descending order of prefix length. Assume that each word of the TCAM has 32 cells. The prefix 10^* is stored in a TCAM word as $10???...?$, where ? denotes a don't care and there are 30 ?s in the given sequence. To do a longest-prefix match, the destination address is matched, in parallel, against every TCAM entry and the first (i.e., longest) matching entry reported by the TCAM arbitration logic. So, using a TCAM and a sorted-by-length linear list, the longest matching-prefix can be determined in $O(1)$ time. A prefix may be inserted or deleted in $O(W)$ time, where W is the length of

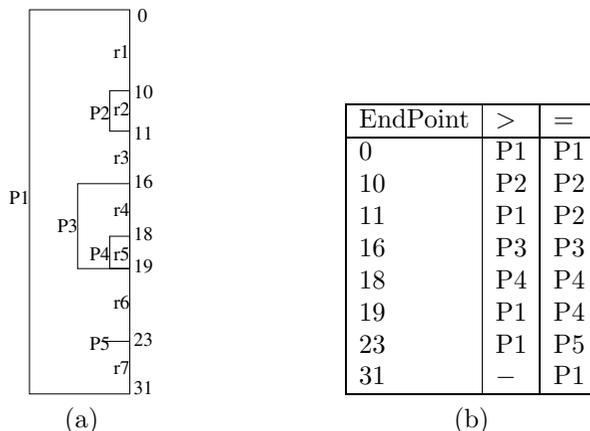


FIGURE 48.2: (a) Pictorial representation of prefixes and ranges (b) Array for binary search.

the longest prefix [28][§]. For example, an insert of a prefix of length 3 (say) can be done by relocating a the first prefix of length 1 to the end of the list; filling the vacated slot with the first prefix of length 2; and finally filling this newly vacated spot with the prefix of length 3 that is to be inserted.

Despite the simplicity and efficiency of using TCAMs, TCAMs present problems in real applications [3]. For example, TCAMs consume a lot of power and board area.

48.2.2 End-Point Array

Lampson, Srinivasan, and Varghese [17] have proposed a data structure in which the end points of the ranges defined by the prefixes are stored in ascending order in an array. $LMP(d)$ is found by performing a binary search on this ordered array of end points.

Prefixes and their ranges may be drawn as nested rectangles as in Figure 48.2(a), which gives the pictorial representation of the five prefixes of Figure 48.1.

In the data structure of Lampson et al. [17], the distinct range end-points are stored in ascending order as in Figure 48.2(b). The distinct end-points (range start and finish points) for the prefixes of Figure 48.1 are [0, 10, 11, 16, 18, 19, 23, 31]. Let $r_i, 1 \leq i \leq q \leq 2n$ be the distinct range end-points for a set of n prefixes. Let $r_{q+1} = \infty$. With each distinct range end-point, $r_i, 1 \leq i \leq q$, the array stores $LMP(d)$ for d such that (a) $r_i < d < r_{i+1}$ (this is the column labeled “>” in Figure 48.2(b)) and (b) $r_i = d$ (column labeled “=”). Now, $LMP(d), r_1 \leq d \leq r_q$ can be determined in $O(\log n)$ time by performing a binary search to find the unique i such that $r_i \leq d < r_{i+1}$. If $r_i = d$, $LMP(d)$ is given by the “=” entry; otherwise, it is given by the “>” entry. For example, since $d = 20$ satisfies $19 \leq d < 23$ and since $d \neq 19$, the “>” entry of the end point 19 is used to determine that $LMP(20)$ is P1.

As noted by Lampson et al. [17], the range end-point table can be built in $O(n)$ time (this assumes that the end points are available in ascending order). Unfortunately, as stated in [17], updating the range end-point table following the insertion or deletion of a prefix also takes $O(n)$ time because $O(n)$ “>” and/or “=” entries may change. Although Lampson

[§]More precisely, W may be defined to be the number of different prefix lengths in the table.

et al. [17] provide ways to reduce the complexity of the search for the LMP by a constant factor, these methods do not result in schemes that permit prefix insertion and deletion in $O(\log n)$ time.

It should be noted that the end-point array may be used even when ties are broken by selecting the first matching rule or the highest-priority matching rule. Further, the method applies to the case when the filters are arbitrary ranges rather than simply prefixes. The complexity of the preprocessing step (i.e., creation of the array of ordered end-points) and the search for the rule to use is unchanged. Further, the memory requirements are the same, $O(n)$ for an n -rule table, regardless of the tie breaker and whether the filters are prefixes or general ranges.

48.2.3 Sets of Equal-Length Prefixes

Waldvogel et al. [32] have proposed a data structure to determine $LMP(d)$ by performing a binary search on prefix length. In this data structure, the prefixes in the router table T are partitioned into the sets S_0, S_1, \dots such that S_i contains all prefixes of T whose length is i . For simplicity, we assume that T contains the default prefix $*$. So, $S_0 = \{*\}$. Next, each S_i is augmented with markers that represent prefixes in S_j such that $j > i$ and i is on the binary search path to S_j . For example, suppose that the length of the longest prefix of T is 32 and that the length of $LMP(d)$ is 22. To find $LMP(d)$ by a binary search on length, we will first search S_{16} for an entry that matches the first 16 bits of d . This search[¶] will need to be successful for us to proceed to a larger length. The next search will be in S_{24} . This search will need to fail. Then, we will search S_{20} followed by S_{22} . So, the path followed by a binary search on length to get to S_{22} is S_{16}, S_{24}, S_{20} , and S_{22} . For this to be followed, the searches in S_{16}, S_{20} , and S_{22} must succeed while that in S_{24} must fail. Since the length of $LMP(d)$ is 22, T has no matching prefix whose length is more than 22. So, the search in S_{24} is guaranteed to fail. Similarly, the search in S_{22} is guaranteed to succeed. However, the searches in S_{16} and S_{20} will succeed iff T has matching prefixes of length 16 and 20. To ensure success, every length 22 prefix P places a *marker* in S_{16} and S_{20} , the marker in S_{16} is the first 16 bits of P and that in S_{20} is the first 20 bits in P . Note that a marker M is placed in S_i only if S_i doesn't contain a prefix equal to M . Notice also that for each i , the binary search path to S_i has $O(\log l_{max}) = O(\log W)$, where l_{max} is the length of the longest prefix in T , S_j s on it. So, each prefix creates $O(\log W)$ markers. With each marker M in S_i , we record the longest prefix of T that matches M (the length of this longest matching-prefix is necessarily smaller than i).

To determine $LMP(d)$, we begin by setting $leftEnd = 0$ and $rightEnd = l_{max}$. The repetitive step of the binary search requires us to search for an entry in S_m , where $m = \lfloor (leftEnd + rightEnd)/2 \rfloor$, that equals the first m bits of d . If S_m does not have such an entry, set $rightEnd = m - 1$. Otherwise, if the matching entry is the prefix P , P becomes the longest matching-prefix found so far. If the matching entry is the marker M , the prefix recorded with M is the longest matching-prefix found so far. In either case, set $leftEnd = m + 1$. The binary search terminates when $leftEnd > rightEnd$.

One may easily establish the correctness of the described binary search. Since, each prefix creates $O(\log W)$ markers, the memory requirement of the scheme is $O(n \log W)$. When each set S_i is represented as a hash table, the data structure is called SELPH (sets of equal length prefixes using hash tables). The expected time to find $LMP(d)$ is $O(\log W)$ when

[¶]When searching S_i , only the first i bits of d are used, because all prefixes in S_i have exactly i bits.

the router table is represented as an SELPH. When inserting a prefix, $O(\log W)$ markers must also be inserted. With each marker, we must record a longest-matching prefix. The expected time to find these longest matching-prefixes is $O(\log^2 W)$. In addition, we may need to update the longest-matching prefix information stored with the $O(n \log W)$ markers at lengths greater than the length of the newly inserted prefix. This takes $O(n \log^2 W)$ time. So, the expected insert time is $O(n \log^2 W)$. When deleting a prefix P , we must search all hash tables for markers M that have P recorded with them and then update the recorded prefix for each of these markers. For hash tables with a bounded loading density, the expected time for a delete (including marker-prefix updates) is $O(n \log^2 W)$. Waldvogel et al. [32] have shown that by inserting the prefixes in ascending order of length, an n -prefix SELPH may be constructed in $O(n \log^2 W)$ time.

When each set is represented as a balanced search tree (see [Chapter 10](#)), the data structure is called SELPT. In an SELPT, the time to find $LMP(d)$ is $O(\log n \log W)$; the insert time is $O(n \log n \log^2 W)$; the delete time is $O(n \log n \log^2 W)$; and the time to construct the data structure for n prefixes is $O(W + n \log n \log^2 W)$.

In the full version of [32], Waldvogel et al. show that by using a technique called marker partitioning, the SELPH data structure may be modified to have a search time of $O(\alpha + \log W)$ and an insert/delete time of $O(\alpha \sqrt{n} W \log W)$, for any $\alpha > 1$.

Because of the excessive insert and delete times, the sets of equal-length prefixes data structure is suitable only for static router tables. Note that in an actual implementation of SELPH or SELPT, we need only keep the non-empty S_i s and do a binary search over the collection of non-empty S_i s. Srinivasan and Varghese [30] have proposed the use of controlled prefix-expansion to reduce the number of non-empty sets S_i . The details of their algorithm to reduce the number of lengths are given in [29]. The complexity of their algorithm is $O(nW^2)$, where n is the number of prefixes, and W is the length of the longest prefix. The algorithm of [29] does not minimize the storage required by the prefixes and markers for the resulting set of prefixes. Kim and Sahni [16] have developed an algorithm that minimizes storage requirement but takes $O(nW^3 + kW^4)$ time, where k is the desired number of non-empty S_i s. Additionally, Kim and Sahni [16] propose improvements to the heuristic of [29].

We note that Waldvogel's scheme is very similar to the k -ary search-on-length scheme developed by Berg et al. [4] and the binary search-on-length schemes developed by Willard [33]. Berg et al. [4] use a variant of stratified trees [10] for one-dimensional point location in a set of n disjoint ranges. Willard [33] modified stratified trees and proposed the y-fast trie data structure to search a set of disjoint ranges. By decomposing filter ranges that are not disjoint into disjoint ranges, the schemes of [4, 33] may be used for longest-prefix matching in static router tables. The asymptotic complexity for a search using the schemes of [4, 33] is the same as that of Waldvogel's scheme. The decomposition of overlapping ranges into disjoint ranges is feasible for static router tables but not for dynamic router tables because a large range may be decomposed into $O(n)$ disjoint small ranges.

48.2.4 Tries

1-Bit Tries

A *1-bit trie* is a tree-like structure in which each node has a left child, left data, right child, and right data field. Nodes at level^{||} $l - 1$ of the trie store prefixes whose length is l . If the rightmost bit in a prefix whose length is l is 0, the prefix is stored in the left data field of a node that is at level $l - 1$; otherwise, the prefix is stored in the right data field of a node that is at level $l - 1$. At level i of a trie, branching is done by examining bit i (bits are numbered from left to right beginning with the number 0) of a prefix or destination address. When bit i is 0, we move into the left subtree; when the bit is 1, we move into the right subtree. Figure 48.3(a) gives the prefixes in the 8-prefix example of [30], and Figure 48.3(b) shows the corresponding 1-bit trie. The prefixes in Figure 48.3(a) are numbered and ordered as in [30].

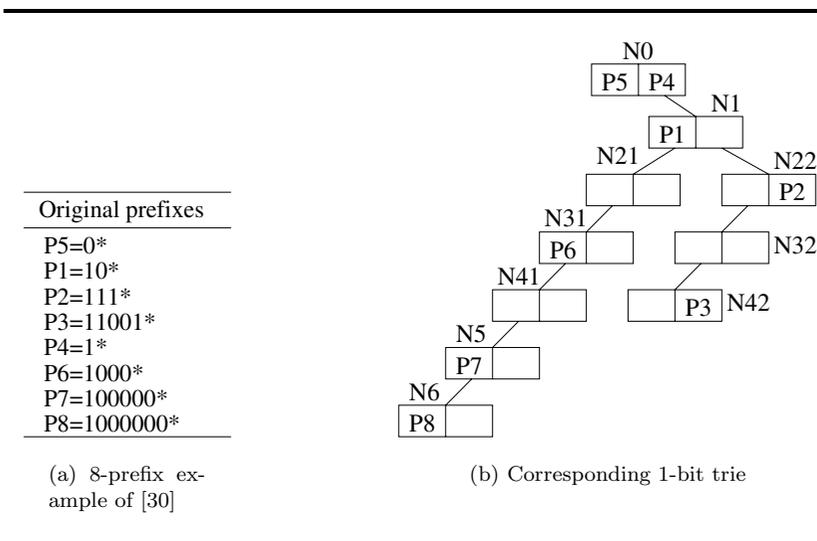


FIGURE 48.3: Prefixes and corresponding 1-bit trie.

The 1-bit tries described here are an extension of the 1-bit tries described in [15]. The primary difference being that the 1-bit tries of [15] are for the case when no key is a prefix of another. Since in router-table applications, this condition isn't satisfied, the 1-bit trie representation of [15] is extended so that keys of length l are stored in nodes at level $l - 1$ of the trie. Note that at most two keys may be stored in a node; one of these has bit l equal to 0 and other has this bit equal to 1. In this extension, every node of the 1-bit trie has 2 child pointers and 2 data fields. The height of a 1-bit trie is $O(W)$.

For any destination address d , all prefixes that match d lie on the search path determined by the bits of d . By following this search path, we may determine the longest matching-

^{||}Level numbers are assigned beginning with 0 for the root level.

prefix, the first prefix in the table that matches d , as well as the highest-priority matching-prefix in $O(W)$ time. Further, prefixes may be inserted/deleted in $O(W)$ time. The memory required by the 1-bit trie is $O(nW)$.

IPv4 backbone routers may have more than 100 thousand prefixes. Even though the prefixes in a backbone router may have any length between 0 and W , there is a concentration of prefixes at lengths 16 and 24, because in the early days of the Internet, Internet address assignment was done by classes. All addresses in a class B network have the same first 16 bits, while addresses in the same class C network agree on the first 24 bits. Addresses in class A networks agree on their first 8 bits. However, there can be at most 256 class A networks (equivalently, there can be at most 256 8-bit prefixes in a router table). For our backbone routers that occur in practice [24], the number of nodes in a 1-bit trie is between $2n$ and $3n$. Hence, in practice, the memory required by the 1-bit-trie representation is $O(n)$.

Fixed-Stride Tries

Since the trie of Figure 48.3(b) has a height of 6, a search into this trie may make up to 7 memory accesses, one access for each node on the path from the root to a node at level 6 of the trie. The total memory required for the 1-bit trie of Figure 48.3(b) is 20 units (each node requires 2 units, one for each pair of (child, data) fields).

When 1-bit tries are used to represent IPv4 router tables, the trie height may be as much as 31. A lookup in such a trie takes up to 32 memory accesses.

Degermark et al. [8] and Srinivasan and Varghese [30] have proposed the use of fixed-stride tries to enable fast identification of the longest matching prefix in a router table. The *stride* of a node is defined to be the number of bits used at that node to determine which branch to take. A node whose stride is s has 2^s child fields (corresponding to the 2^s possible values for the s bits that are used) and 2^s data fields. Such a node requires 2^s memory units. In a *fixed-stride trie* (FST), all nodes at the same level have the same stride; nodes at different levels may have different strides.

Suppose we wish to represent the prefixes of Figure 48.3(a) using an FST that has three levels. Assume that the strides are 2, 3, and 2. The root of the trie stores prefixes whose length is 2; the level one nodes store prefixes whose length is 5 (2 + 3); and level three nodes store prefixes whose length is 7 (2 + 3 + 2). This poses a problem for the prefixes of our example, because the length of some of these prefixes is different from the storeable lengths. For instance, the length of P5 is 1. To get around this problem, a prefix with a nonpermissible length is expanded to the next permissible length. For example, P5 = 0* is expanded to P5a = 00* and P5b = 01*. If one of the newly created prefixes is a duplicate, natural dominance rules are used to eliminate all but one occurrence of the prefix. For instance, P4 = 1* is expanded to P4a = 10* and P4b = 11*. However, P1 = 10* is to be chosen over P4a = 10*, because P1 is a longer match than P4. So, P4a is eliminated. Because of the elimination of duplicate prefixes from the expanded prefix set, all prefixes are distinct. Figure 48.4(a) shows the prefixes that result when we expand the prefixes of Figure 48.3 to lengths 2, 5, and 7. Figure 48.4(b) shows the corresponding FST whose height is 2 and whose strides are 2, 3, and 2.

Since the trie of Figure 48.4(b) can be searched with at most 3 memory references, it represents a time performance improvement over the 1-bit trie of Figure 48.3(b), which requires up to 7 memory references to perform a search. However, the space requirements of the FST of Figure 48.4(b) are more than that of the corresponding 1-bit trie. For the root of the FST, we need 8 fields or 4 units; the two level 1 nodes require 8 units each; and the level 3 node requires 4 units. The total is 24 memory units.

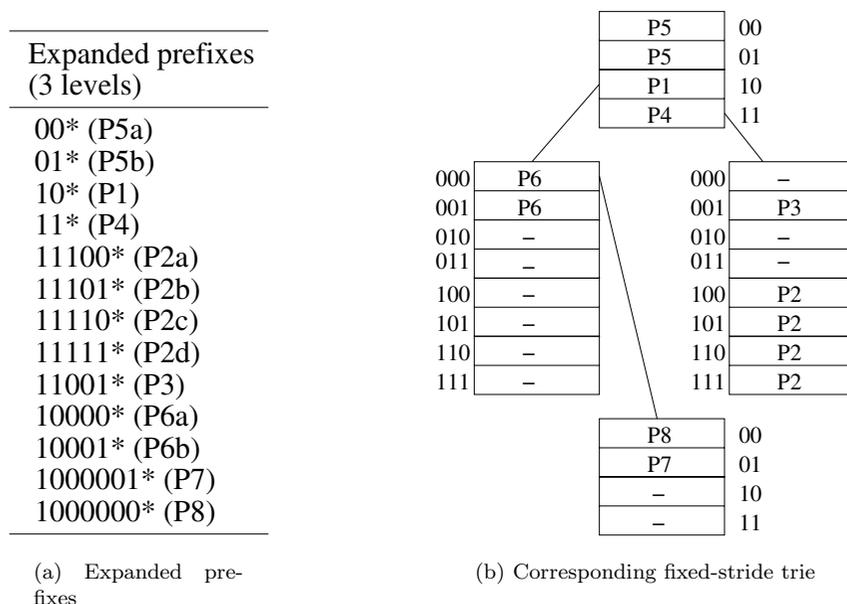


FIGURE 48.4: Prefix expansion and fixed-stride trie.

We may represent the prefixes of Figure 48.3(a) using a one-level trie whose root has a stride of 7. Using such a trie, searches could be performed making a single memory access. However, the one-level trie would require $2^7 = 128$ memory units.

For IPv4 prefix sets, Degermark et al. [8] propose the use of a three-level trie in which the strides are 16, 8, and 8. They propose encoding the nodes in this trie using bit vectors to reduce memory requirements. The resulting data structure requires at most 12 memory accesses. However, inserts and deletes are quite expensive. For example, the insertion of the prefix 1^* changes up to 2^{15} entries in the trie's root node. All of these changes may propagate into the compacted storage scheme of [8].

Lampson et al. [17] have proposed the use of hybrid data structures comprised of a stride-16 root and an auxiliary data structure for each of the subtrees of the stride-16 root. This auxiliary data structure could be the end-point array of Section 48.2.2 (since each subtree is expected to contain only a small number of prefixes, the number of end points in each end-point array is also expected to be quite small). An alternative auxiliary data structure suggested by Lampson et al. [17] is a 6-way search tree for IPv4 router tables. In the case of these 6-way trees, the keys are the remaining up to 16 bits of the prefix (recall that the stride-16 root consumes the first 16 bits of a prefix). For IPv6 prefixes, a multicolumn scheme is suggested [17]. None of these proposed structures is suitable for a dynamic table.

In the *fixed-stride trie optimization* (FSTO) problem, we are given a set P of prefixes and an integer k . We are to select the strides for a k -level FST in such a manner that the k -level FST for the given prefixes uses the smallest amount of memory.

For some P , a k -level FST may actually require more space than a $(k - 1)$ -level FST. For example, when $P = \{00^*, 01^*, 10^*, 11^*\}$, the unique 1-level FST for P requires 4 memory units while the unique 2-level FST (which is actually the 1-bit trie for P) requires 6 memory units. Since the search time for a $(k - 1)$ -level FST is less than that for a k -level tree, we

would actually prefer $(k - 1)$ -level FSTs that take less (or even equal) memory over k -level FSTs. Therefore, in practice, we are really interested in determining the best FST that uses at most k levels (rather than exactly k levels). The *modified* MSTO problem (MFSTO) is to determine the best FST that uses at most k levels for the given prefix set P .

Let O be the 1-bit trie for the given set of prefixes, and let F be any k -level FST for this prefix set. Let s_0, \dots, s_{k-1} be the strides for F . We shall say that level 0 of F covers levels $0, \dots, s_0 - 1$ of O , and that level j , $0 < j < k$ of F covers levels a, \dots, b of O , where $a = \sum_0^{j-1} s_q$ and $b = \sum_0^j s_q - 1$. So, level 0 of the FST of Figure 48.4(b) covers levels 0 and 1 of the 1-bit trie of Figure 48.3(b). Level 1 of this FST covers levels 2, 3, and 4 of the 1-bit trie of Figure 48.3(b); and level 2 of this FST covers levels 5 and 6 of the 1-bit trie. We shall refer to levels $e_u = \sum_0^u s_q$, $0 \leq u < k$ as the *expansion levels* of O . The expansion levels defined by the FST of Figure 48.4(b) are 0, 2, and 5.

Let $nodes(i)$ be the number of nodes at level i of the 1-bit trie O . For the 1-bit trie of Figure 48.3(a), $nodes(0 : 6) = [1, 1, 2, 2, 2, 1, 1]$. The memory required by F is $\sum_0^{k-1} nodes(e_q) * 2^{s_q}$. For example, the memory required by the FST of Figure 48.4(b) is $nodes(0) * 2^2 + nodes(2) * 2^3 + nodes(5) * 2^2 = 24$.

Let $C(j, r)$ be the cost of the best FST that uses *at most* r expansion levels. A simple dynamic programming recurrence for C is [24]:

$$C(j, r) = \min_{m \in \{-1..j-1\}} \{C(m, r-1) + nodes(m+1) * 2^{j-m}\}, j \geq 0, r > 1 \quad (48.1)$$

$$C(-1, r) = 0 \text{ and } C(j, 1) = 2^{j+1}, j \geq 0 \quad (48.2)$$

Let $M(j, r)$, $r > 1$, be the smallest m that minimizes

$$C(m, r-1) + nodes(m+1) * 2^{j-m},$$

in Equation 48.1.

THEOREM 48.1 [Sahni and Kim [24]] $\forall(j \geq 0, k > 2)[M(j, k) \geq \max\{M(j-1, k), M(j, k-1)\}]$.

Theorem 48.1 results in an algorithm to compute $C(W-1, k)$ in $O(kW^2)$. Using the computed M values, the strides for the OFST that uses at most k expansion levels may be determined in an additional $O(k)$ time. Although the resulting algorithm has the same asymptotic complexity as does the optimization algorithm of Srinivasan and Varghese [30], experiments conducted by Sahni and Kim [24] using real IPv4 prefix-data-sets indicate that the algorithm based on Theorem 48.1 runs 2 to 4 times as fast.

Basu and Narliker [2] consider implementing FST router tables on a pipelined architecture. Each level of the FST is assigned to a unique pipeline stage. The optimization problem to be solved in this application requires an FST that has a number of levels no more than the number of pipeline stages, the memory required per level should not exceed the available per stage memory, and the total memory required is minimum subject to the stated constraints.

Variable-Stride Tries

In a *variable-stride trie* (VST) [30], nodes at the same level may have different strides. Figure 48.5 shows a two-level VST for the 1-bit trie of Figure 48.3. The stride for the root

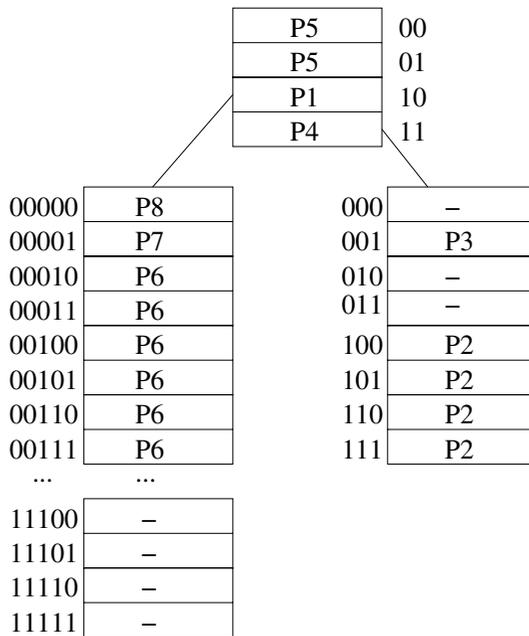


FIGURE 48.5: Two-level VST for prefixes of Figure 48.3(a).

is 2; that for the left child of the root is 5; and that for the root’s right child is 3. The memory requirement of this VST is 4 (root) + 32 (left child of root) + 8 (right child of root) = 44.

Since FSTs are a special case of VSTs, the memory required by the best VST for a given prefix set P and number of expansion levels k is less than or equal to that required by the best FST for P and k . Despite this, FSTs may be preferred in certain router applications “because of their simplicity and slightly faster search time” [30].

Let r -VST be a VST that has at most r levels. Let $Opt(N, r)$ be the cost (i.e., memory requirement) of the best r -VST for a 1-bit trie whose root is N . Nilsson and Karlsson [23] propose a greedy heuristic to construct optimal VSTs. The resulting VSTs are known as LC-tries (level-compressed tries) and were first proposed in a more general context by Andersson and Nilsson [1]. An LC-tries obtained from a 1-bit trie by replacing full subtrees of the 1-bit trie by single multibit nodes. This replacement is done by examining the 1-bit trie top to bottom (i.e., from root to leaves). Srinivasan and Varghese [30], have obtained the following dynamic programming recurrence for $Opt(N, r)$.

$$Opt(N, r) = \min_{s \in \{1 \dots 1+height(N)\}} \{2^s + \sum_{M \in D_s(N)} Opt(M, r - 1)\}, \quad r > 1 \tag{48.3}$$

where $D_s(N)$ is the set of all descendants of N that are at level s of N . For example, $D_1(N)$ is the set of children of N and $D_2(N)$ is the set of grandchildren of N . $height(N)$ is the maximum level at which the trie rooted at N has a node. For example, in Figure 48.3(b), the height of the trie rooted at N_1 is 5. When $r = 1$,

$$Opt(N, 1) = 2^{1+height(N)} \tag{48.4}$$

Srinivasan and Varghese [30], describe a way to determine $Opt(R, k)$ using Equations 48.3 and 48.4. The complexity of their algorithm is $O(p * W * k)$, where p is the number of nodes in the 1-bit trie for the prefixes ($p = O(n)$ for realistic router tables). Sahni and Kim [25] provide an alternative way to compute $Opt(R, k)$ in $O(pWk)$ time. The algorithm of [25], however, performs fewer operations and has fewer cache misses. When the cost of operations dominates the run time, the algorithm of [25] is expected to be about 6 times as fast as that of [30] (for available router databases). When cache miss time dominates the run time, the algorithm of [25] could be 12 times as fast when $k = 2$ and 42 times as fast when $k = 7$.

We describe the formulation used in [25]. Let

$$Opt(N, s, r) = \sum_{M \in D_s(N)} Opt(M, r), \quad s > 0, \quad r > 1,$$

and let $Opt(N, 0, r) = Opt(N, r)$. From Equations 48.3 and 48.4, it follows that:

$$Opt(N, 0, r) = \min_{s \in \{1 \dots 1 + height(N)\}} \{2^s + Opt(N, s, r - 1)\}, \quad r > 1 \quad (48.5)$$

and

$$Opt(N, 0, 1) = 2^{1 + height(N)}. \quad (48.6)$$

For $s > 0$ and $r > 1$, we get

$$\begin{aligned} Opt(N, s, r) &= \sum_{M \in D_s(N)} Opt(M, r) \\ &= Opt(LeftChild(N), s - 1, r) \\ &+ Opt(RightChild(N), s - 1, r). \end{aligned} \quad (48.7)$$

For Equation 48.7, we need the following initial condition:

$$Opt(null, *, *) = 0 \quad (48.8)$$

With the assumption that the number of nodes in the 1-bit trie is $O(n)$, we see that the number of $Opt(*, *, *)$ values is $O(nWk)$. Each $Opt(*, *, *)$ value may be computed in $O(1)$ time using Equations 48.5 through 48.8 provided the Opt values are computed in postorder. Therefore, we may compute $Opt(R, k) = Opt(R, 0, k)$ in $O(pWk)$ time. The algorithm of [25] requires $O(W^2k)$ memory for the $Opt(*, *, *)$ values. To see this, notice that there can be at most $W + 1$ nodes N whose $Opt(N, *, *)$ values must be retained at any given time, and for each of these at most $W + 1$ nodes, $O(Wk)$ $Opt(N, *, *)$ values must be retained. To determine the optimal strides, each node of the 1-bit trie must store the stride s that minimizes the right side of Equation 48.5 for each value of r . For this purpose, each 1-bit trie node needs $O(k)$ space. Since the 1-bit trie has $O(n)$ nodes in practice, the memory requirements of the 1-bit trie are $O(nk)$. The total memory required is, therefore, $O(nk + W^2k)$.

In practice, we may prefer an implementation that uses considerably more memory. If we associate a cost array with each of the p nodes of the 1-bit trie, the memory requirement increases to $O(pWk)$. The advantage of this increased memory implementation is that the optimal strides can be recomputed in $O(W^2k)$ time (rather than $O(pWk)$) following each insert or delete of a prefix. This is so because, the $Opt(N, *, *)$ values need be recomputed only for nodes along the insert/delete path of the 1-bit trie. There are $O(W)$ such nodes.

Faster algorithms to determine optimal 2- and 3-VSTs also are developed in [25].

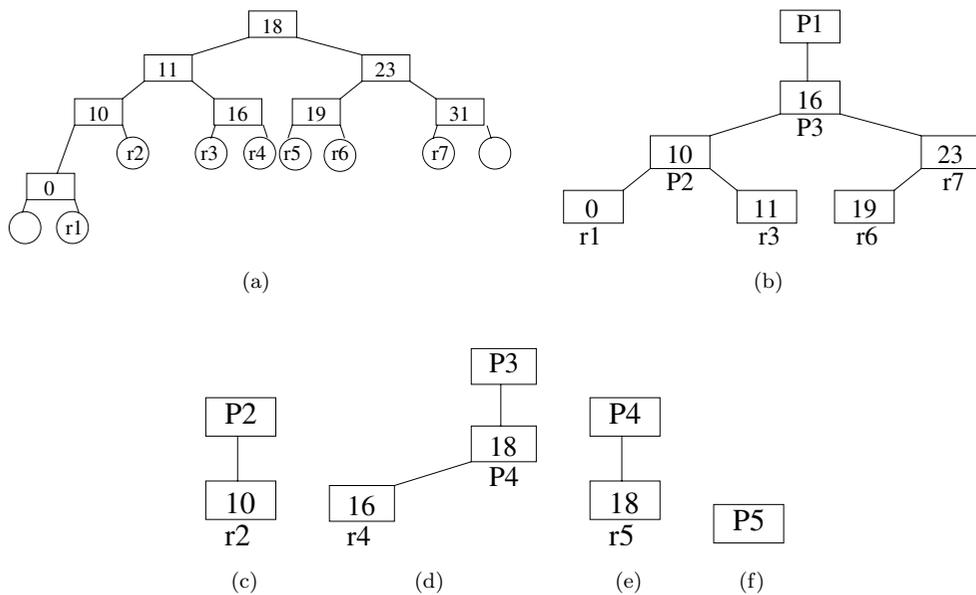


FIGURE 48.6: CBST for Figure 48.2(a). (a) base interval tree (b) prefix tree for P1 (c) prefix tree for P2 (d) prefix tree for P3 (e) prefix tree for P4 (f) prefix tree for P5.

48.2.5 Binary Search Trees

Sahni and Kim [26] propose the use of a collection of red-black trees (see Chapter 10) to determine $LMP(d)$. The CRBT comprises a front-end data structure that is called the *binary interval tree* (BIT) and a back-end data structure called a *collection of prefix trees* (CPT). For any destination address d , define the *matching basic interval* to be a basic interval with the property that $r_i \leq d \leq r_{i+1}$ (note that some d s have two matching basic intervals).

The BIT is a binary search tree that is used to search for a matching basic interval for d . The BIT comprises internal and external nodes and there is one internal node for each r_i . Since the BIT has q internal nodes, it has $q + 1$ external nodes. The first and last of these, in inorder, have no significance. The remaining $q - 1$ external nodes, in inorder, represent the $q - 1$ basic intervals of the given prefix set. Figure 48.6(a) gives a possible (we say possible because, any red-black binary search tree organization for the internal nodes will suffice) BIT for our five-prefix example of Figure 48.2(a). Internal nodes are shown as rectangles while circles denote external nodes. Every external node has three pointers: *startPointer*, *finishPointer*, and *basicIntervalPointer*. For an external node that represents the basic interval $[r_i, r_{i+1}]$, *startPointer* (*finishPointer*) points to the header node of the prefix tree (in the back-end structure) for the prefix (if any) whose range start and finish points are r_i (r_{i+1}). Note that only prefixes whose length is W can have this property. *basicIntervalPointer* points to a prefix node in a prefix tree of the back-end structure. In Figure 48.6(a), the labels in the external (circular) nodes identify the represented basic interval. The external node with r1 in it, for example, has a *basicIntervalPointer* to the rectangular node labeled r1 in the prefix tree of Figure 48.6(b).

For each prefix and basic interval, x , define $next(x)$ to be the smallest range prefix (i.e., the longest prefix) whose range includes the range of x . For the example of Figure 48.2(a), the $next()$ values for the basic intervals r1 through r7 are, respectively, P1, P2, P1, P3, P4, P1, and P1. Notice that the next value for the range $[r_i, r_{i+1}]$ is the same as the “>” value for r_i in Figure 48.2(b), $1 \leq i < q$. The $next()$ values for the nontrivial prefixes P1 through P4 of Figure 48.2(a) are, respectively, “-”, P1, P1, and P3.

The back-end structure, which is a collection of prefix trees (CPT), has one prefix tree for each of the prefixes in the router table. Each prefix tree is a red-black tree. The prefix tree for prefix P comprises a header node plus one node, called a *prefix node*, for every nontrivial prefix or basic interval x such that $next(x) = P$. The header node identifies the prefix P for which this is the prefix tree. The prefix trees for each of the five prefixes of Figure 48.2(a) are shown in Figures 48.6(b)-(f). Notice that prefix trees do not have external nodes and that the prefix nodes of a prefix tree store the start point of the range or prefix represented by that prefix node. In the figures, the start points of the basic intervals and prefixes are shown inside the prefix nodes while the basic interval or prefix name is shown outside the node.

The search for $LMP(d)$ begins with a search of the BIT for the matching basic interval for d . Suppose that external node Q of the BIT represents this matching basic interval. When the destination address equals the left (right) end-point of the matching basic interval and *startPointer* (*finishPointer*) is not null, $LMP(d)$ is pointed to by *startPointer* (*finishPointer*). Otherwise, the back-end CPT is searched for $LMP(d)$. The search of the back-end structure begins at the node $Q.basicIntervalPointer$. By following parent pointers from $Q.basicIntervalPointer$, we reach the header node of the prefix tree that corresponds to $LMP(d)$.

When a CRBT is used, $LMP(d)$ may be found in $O(\log n)$ time. Inserts and deletes also take $O(\log n)$ time when a CRBT is used. In [27], Sahni and Kim propose an alternative BIT structure (ABIT) that has internal nodes only. Although the ABIT structure increases the memory requirements of the router table, the time to search, insert, and delete is reduced by a constant factor [27]. Suri et al. [31] have proposed a B-tree data structure for dynamic router tables. Using their structure, we may find the longest matching prefix in $O(\log n)$ time. However, inserts/deletes take $O(W \log n)$ time. The number of cache misses is $O(\log n)$ for each operation. When W bits fit in $O(1)$ words (as is the case for IPv4 and IPv6 prefixes) logical operations on W -bit vectors can be done in $O(1)$ time each. In this case, the scheme of [31] takes $O(\log W * \log n)$ time for an insert and $O(W + \log n) = O(W)$ time for a delete. An alternative B-tree router-table design has been proposed by Lu and Sahni [21]. Although the asymptotic complexity of each of the router-table operations is the same using either B-tree router-table design, the design of Lu and Sahni [21] has fewer cache misses for inserts and deletes; the number of cache misses when searching for $lmp(d)$ is the same using either design. Consequently, inserts and deletes are faster when the design of Lu and Sahni [21] is used.

Several researchers ([6, 11, 14, 27], for example), have investigated router table data structures that account for bias in access patterns. Gupta, Prabhakar, and Boyd [14], for example, propose the use of ranges. They assume that access frequencies for the ranges are known, and they construct a bounded-height binary search tree of ranges. This binary search tree accounts for the known range access frequencies to obtain near-optimal IP lookup. Although the scheme of [14] performs IP lookup in near-optimal time, changes in the access frequencies, or the insertion or removal of a prefix require us to reconstruct the data structure, a task that takes $O(n \log n)$ time.

Ergun et al. [11] use ranges to develop a biased skip list structure that performs longest prefix-matching in $O(\log n)$ expected time. Their scheme is designed to give good expected

performance for bursty** access patterns". The biased skip list scheme of Ergun et al. [11] permits inserts and deletes in $O(\log n)$ time only in the severely restricted and impractical situation when all prefixes in the router table are of the same length. For the more general, and practical, case when the router table comprises prefixes of different length, their scheme takes $O(n)$ expected time for each insert and delete. Sahni and Kim [27] extend the biased skip lists of Ergun et al. [11] to obtain a biased skip lists structure in which longest prefix-matching as well as inserts and deletes take $O(\log n)$ expected time. They also propose a splay tree scheme (see Chapter 12) for bursty access patterns. In this scheme, longest prefix-matching, insert and delete have an $O(\log n)$ amortized complexity.

48.2.6 Priority Search Trees

A priority-search tree (PST) [22] (see Chapter 18) is a data structure that is used to represent a set of tuples of the form $(key1, key2, data)$, where $key1 \geq 0$, $key2 \geq 0$, and no two tuples have the same $key1$ value. The data structure is simultaneously a min-tree on $key2$ (i.e., the $key2$ value in each node of the tree is \leq the $key2$ value in each descendant node) and a search tree on $key1$. There are two common PST representations [22]:

1. In a **radix priority-search tree** (RPST), the underlying tree is a binary radix tree on $key1$.
2. In a **red-black priority-search tree** (RBPST), the underlying tree is a red-black tree.

McCreight [22] has suggested a PST representation of a collection of ranges with distinct finish points. This representation uses the following mapping of a range r into a PST tuple:

$$(key1, key2, data) = (finish(r), start(r), data) \quad (48.9)$$

where $data$ is any information (e.g., next hop) associated with the range. Each range r is, therefore mapped to a point $map1(r) = (x, y) = (key1, key2) = (finish(r), start(r))$ in 2-dimensional space.

Let $ranges(d)$ be the set of ranges that match d . McCreight [22] has observed that when the mapping of Equation 48.9 is used to obtain a point set $P = map1(R)$ from a range set R , then $ranges(d)$ is given by the points that lie in the rectangle (including points on the boundary) defined by $x_{left} = d$, $x_{right} = \infty$, $y_{top} = d$, and $y_{bottom} = 0$. These points are obtained using the method $enumerateRectangle(x_{left}, x_{right}, y_{top}) = enumerateRectangle(d, \infty, d)$ of a PST (y_{bottom} is implicit and is always 0).

When an RPST is used to represent the point set P , the complexity of

$$enumerateRectangle(x_{left}, x_{right}, y_{top})$$

is $O(\log maxX + s)$, where $maxX$ is the largest x value in P and s is the number of points in the query rectangle. When the point set is represented as an RBPST, this complexity becomes $O(\log n + s)$, where $n = |P|$. A point (x, y) (and hence a range $[y, x]$) may be inserted into or deleted from an RPST (RBPST) in $O(\log maxX)$ ($O(\log n)$) time [22].

**In a *bursty* access pattern the number of different destination addresses in any subsequence of q packets is $\ll q$. That is, if the destination of the current packet is d , there is a high probability that d is also the destination for one or more of the next few packets. The fact that Internet packets tend to be bursty has been noted in [7, 18], for example.

Data Structure	Search	Update
Linear List	$O(n)$	$O(n)$
End-point Array	$O(\log n)$	$O(n)$
Sets of Equal-Length Prefixes	$O(\alpha + \log W)$ expected	$O(\alpha \sqrt[3]{nW \log W})$ expected
1-bit tries	$O(W)$	$O(W)$
s-bit Tries	$O(W/s)$	-
CRBT	$O(\log n)$	$O(\log n)$
ACRBT	$O(\log n)$	$O(\log n)$
BSLPT	$O(\log n)$ expected	$O(\log n)$ expected
CST	$O(\log n)$ amortized	$O(\log n)$ amortized
PST	$O(\log n)$	$O(\log n)$

TABLE 48.1 Time complexity of data structures for longest matching-prefix.

Let R be a set of prefix ranges. For simplicity, assume that R includes the range that corresponds to the prefix $*$. With this assumption, $LMP(d)$ is defined for every d . One may verify that $LMP(d)$ is the prefix whose range is

$$[\maxStart(ranges(d)), \minFinish(ranges(d))].$$

Lu and Sahni [19] show that R must contain such range. To find this range easily, we first transform $P = \text{map1}(R)$ into a point set $\text{transform1}(P)$ so that no two points of $\text{transform1}(P)$ have the same x -value. Then, we represent $\text{transform1}(P)$ as a PST. For every $(x, y) \in P$, define $\text{transform1}(x, y) = (x', y') = (2^W x - y + 2^W - 1, y)$. Then, $\text{transform1}(P) = \{\text{transform1}(x, y) \mid (x, y) \in P\}$.

We see that $0 \leq x' < 2^{2W}$ for every $(x', y') \in \text{transform1}(P)$ and that no two points in $\text{transform1}(P)$ have the same x' -value. Let $PST1(P)$ be the PST for $\text{transform1}(P)$. The operation

$$\text{enumerateRectangle}(2^W d - d + 2^W - 1, \infty, d)$$

performed on $PST1$ yields $ranges(d)$. To find $LMP(d)$, we employ the

$$\text{minXinRectangle}(x_{left}, x_{right}, y_{top})$$

operation, which determines the point in the defined rectangle that has the least x -value. It is easy to see that

$$\text{minXinRectangle}(2^W d - d + 2^W - 1, \infty, d)$$

performed on $PST1$ yields $LMP(d)$.

To insert the prefix whose range is $[u, v]$, we insert $\text{transform1}(\text{map1}([u, v]))$ into $PST1$. In case this prefix is already in $PST1$, we simply update the next-hop information for this prefix. To delete the prefix whose range is $[u, v]$, we delete $\text{transform1}(\text{map1}([u, v]))$ from $PST1$. When deleting a prefix, we must take care not to delete the prefix $*$. Requests to delete this prefix should simply result in setting the next-hop associated with this prefix to \emptyset .

Since, minXinRectangle , insert, and delete each take $O(W)$ ($O(\log n)$) time when $PST1$ is an RPST (RBPST), $PST1$ provides a router-table representation in which longest-prefix matching, prefix insertion, and prefix deletion can be done in $O(W)$ time each when an RPST is used and in $O(\log n)$ time each when an RBPST is used.

Tables 48.1 and 48.2 summarize the performance characteristics of various data structures for the longest matching-prefix problem.

Data Structure	Memory Usage
Linear List	$O(n)$
End-point Array	$O(n)$
Sets of Equal-Length Prefixes	$O(n \log W)$
1-bit tries	$O(nW)$
s-bit Tries	$O(2^s nW/s)$
CRBT	$O(n)$
ACRBT	$O(n)$
BSLPT	$O(n)$
CST	$O(n)$
PST	$O(n)$

TABLE 48.2 Memory complexity of data structures for longest matching-prefix.

48.3 Highest-Priority Matching

The trie data structure may be used to represent a dynamic prefix-router-table in which the highest-priority tie-breaker is in use. Using such a structure, each of the dynamic router-table operations may be performed in $O(W)$ time. Lu and Sahni [20] have developed the binary tree on binary tree (BOB) data structure for highest-priority dynamic router-tables. Using BOB, a lookup takes $O(\log^2 n)$ time and cache misses; a new rule may be inserted and an old one deleted in $O(\log n)$ time and cache misses. Although BOB handles filters that are non-intersecting ranges, specialized versions of BOB have been proposed for prefix filters. Using the data structure PBOB (prefix BOB), a lookup, rule insertion and deletion each take $O(W)$ time and cache misses. The data structure LMPBOB (longest matching-prefix BOB) is proposed in [20] for dynamic prefix-router-tables that use the longest matching-prefix rule. Using LMPBOB, the longest matching-prefix may be found in $O(W)$ time and $O(\log n)$ cache misses; rule insertion and deletion each take $O(\log n)$ time and cache misses. On practical rule tables, BOB and PBOB perform each of the three dynamic-table operations in $O(\log n)$ time and with $O(\log n)$ cache misses. Other data structures for maximum-priority matching are developed in [12, 13].

48.3.1 The Data Structure BOB

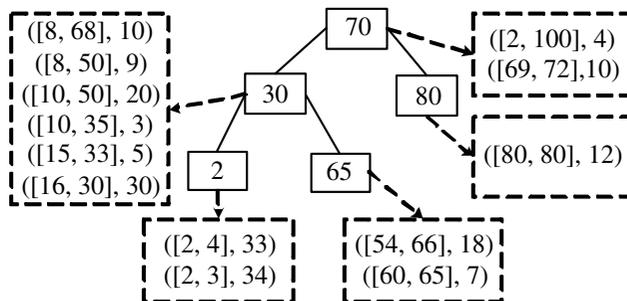
The data structure binary tree on binary tree (BOB) comprises a single balanced binary search tree at the top level. This top-level balanced binary search tree is called the point search tree (PTST). For an n -rule NHRT, the PTST has at most $2n$ nodes (we call this the PTST **size constraint**). With each node z of the PTST, we associate a point, $point(z)$. The PTST is a standard red-black binary search tree (actually, any binary search tree structure that supports efficient search, insert, and delete may be used) on the $point(z)$ values of its node set [15]. That is, for every node z of the PTST, nodes in the left subtree of z have smaller point values than $point(z)$, and nodes in the right subtree of z have larger point values than $point(z)$.

Let R be the set of nonintersecting ranges. Each range of R is stored in exactly one of the nodes of the PTST. More specifically, the root of the PTST stores all ranges $r \in R$ such that $start(r) \leq point(root) \leq finish(r)$; all ranges $r \in R$ such that $finish(r) < point(root)$ are stored in the left subtree of the root; all ranges $r \in R$ such that $point(root) < start(r)$ (i.e., the remaining ranges of R) are stored in the right subtree of the root. The ranges allocated to the left and right subtrees of the root are allocated to nodes in these subtrees using the just stated **range allocation rule** recursively.

For the range allocation rule to successfully allocate all $r \in R$ to exactly one node of the PTST, the PTST must have at least one node z for which $start(r) \leq point(z) \leq finish(r)$. Figure 48.7 gives an example set of nonintersecting ranges and a possible PTST for this set

<i>range</i>	<i>priority</i>
[2, 100]	4
[2, 4]	33
[2, 3]	34
[8, 68]	10
[8, 50]	9
[10, 50]	20
[10, 35]	3
[15, 33]	5
[16, 30]	30
[54, 66]	18
[60, 65]	7
[69, 72]	10
[80, 80]	12

(a)



(b)

FIGURE 48.7: (a) A nonintersecting range set and (b) A possible PTST.

of ranges (we say possible, because we haven't specified how to select the $point(z)$ values and even with specified $point(z)$ values, the corresponding red-black tree isn't unique). The number inside each node is $point(z)$, and outside each node, we give $ranges(z)$.

Let $ranges(z)$ be the subset of ranges of R allocated to node z of the PTST^{††}. Since the PTST may have as many as $2n$ nodes and since each range of R is in exactly one of the sets $ranges(z)$, some of the $ranges(z)$ sets may be empty.

The ranges in $ranges(z)$ may be ordered using the $<$ relation for non-intersecting ranges^{‡‡}. Using this $<$ relation, we put the ranges of $ranges(z)$ into a red-black tree (any balanced binary search tree structure that supports efficient search, insert, delete, join, and split may be used) called the range search-tree or $RST(z)$. Each node x of $RST(z)$ stores exactly one range of $ranges(z)$. We refer to this range as $range(x)$. Every node y in the left (right) subtree of node x of $RST(z)$ has $range(y) < range(x)$ ($range(y) > range(x)$). In addition, each node x stores the quantity $mp(x)$, which is the maximum of the priorities of the ranges associated with the nodes in the subtree rooted at x . $mp(x)$ may be defined recursively as below.

$$mp(x) = \begin{cases} p(x) & \text{if } x \text{ is leaf} \\ \max \{mp(leftChild(x)), mp(rightChild(x)), p(x)\} & \text{otherwise} \end{cases}$$

where $p(x) = priority(range(x))$. Figure 48.8 gives a possible RST structure for $ranges(30)$ of Figure 48.7(b).

LEMMA 48.1 [Lu and Sahni [20]] Let z be a node in a PTST and let x be a node in $RST(z)$. Let $st(x) = start(range(x))$ and $fn(x) = finish(range(x))$.

^{††}We have overloaded the function $ranges$. When u is a node, $ranges(u)$ refers to the ranges stored in node u of a PTST; when u is a destination address, $ranges(u)$ refers to the ranges that match u .

^{‡‡}Let r and s be two ranges. $r < s \Leftrightarrow start(r) < start(s) \vee (start(r) = start(s) \wedge finish(r) > finish(s))$.

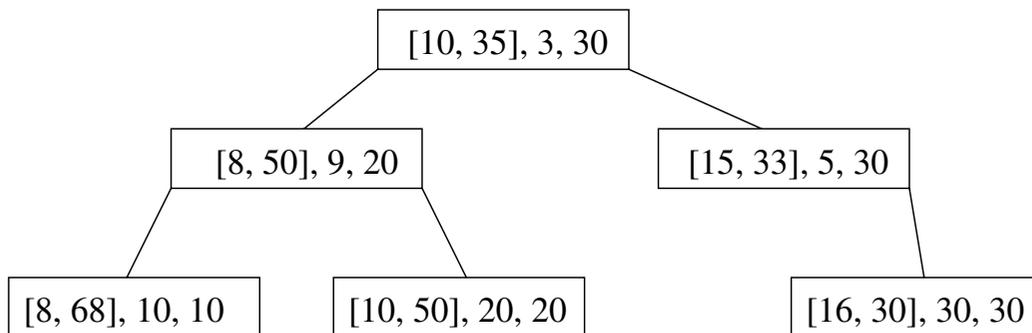


FIGURE 48.8: An example RST for $\text{ranges}(30)$ of Figure 48.7(b). Each node shows $(\text{range}(x), p(x), mp(x))$.

1. For every node y in the right subtree of x , $st(y) \geq st(x)$ and $fn(y) \leq fn(x)$.
2. For every node y in the left subtree of x , $st(y) \leq st(x)$ and $fn(y) \geq fn(x)$.

Both BOB and BOT (the binary tree on trie structure of Gupta and McKeown [13]) use a range allocation rule identical to that used in an interval tree [5] (See Chapter 18). While BOT may be used with any set of ranges, BOB applies only to a set of non-intersecting ranges. However, BOB reduces the search complexity of BOT from $O(W \log n)$ to $O(\log^2 n)$ and reduces the update complexity from $O(W)$ to $O(\log n)$.

48.3.2 Search for the Highest-Priority Matching Range

The highest-priority range that matches the destination address d may be found by following a path from the root of the PTST toward a leaf of the PTST. Figure 48.9 gives the algorithm. For simplicity, this algorithm finds $hp = \text{priority}(hpr(d))$ rather than $hpr(d)$. The algorithm is easily modified to return $hpr(d)$ instead.

We begin by initializing $hp = -1$ and z is set to the root of the PTST. This initialization assumes that all priorities are ≥ 0 . The variable z is used to follow a path from the root toward a leaf. When $d > \text{point}(z)$, d may be matched only by ranges in $RST(z)$ and those in the right subtree of z . The method $RST(z) \rightarrow \text{hpRight}(d, hp)$ (Figure 48.10) updates hp to reflect any matching ranges in $RST(z)$. This method makes use of the fact that $d > \text{point}(z)$. Consider a node x of $RST(z)$. If $d > fn(x)$, then d is to the right (i.e., $d > \text{finish}(\text{range}(x))$) of $\text{range}(x)$ and also to the right of all ranges in the right subtree of x . Hence, we may proceed to examine the ranges in the left subtree of x . When $d \leq fn(x)$, $\text{range}(x)$ as well as all ranges in the left subtree of x match d . Additional matching ranges may be present in the right subtree of x . $hpLeft(d, hp)$ is the analogous method for the case when $d < \text{point}(z)$.

Complexity The complexity of the invocation $RST(z) \rightarrow \text{hpRight}(d, hp)$ is readily seen to be $O(\text{height}(RST(z))) = O(\log n)$. Consequently, the complexity of $hp(d)$ is $O(\log^2 n)$. To determine $hpr(d)$ we need only add code to the methods $hp(d)$, $hpRight(d, hp)$, and $hpLeft(d, hp)$ so as to keep track of the range whose priority is the current value of hp . So, $hpr(d)$ may be found in $O(\log^2 n)$ time also.

The details of the insert and delete operation as well of the BOB variants PBOB and LMPBOB may be found in [20].

```

Algorithm hp(d) {
    // return the priority of hpr(d)
    // easily extended to return hpr(d)
    hp = -1; // assuming 0 is the smallest priority value
    z = root; // root of PTST
    while (z != null) {
        if (d > point(z)) {
            RST(z)->hpRight(d, hp);
            z = rightChild(z);
        }
        else if (d < point(z)) {
            RST(z)->hpLeft(d, hp);
            z = leftChild(z);
        }
        else // d == point(z)
            return max{hp, mp(RST(z)->root)};
    }
    return hp;
}

```

FIGURE 48.9: Algorithm to find $priority(hpr(d))$.

```

Algorithm hpRight(d, hp) {
    // update hp to account for any ranges in RST(z) that match d
    // d > point(z)
    x = root; // root of RST(z)
    while (x != null)
        if (d > fn(x))
            x = leftChild(x);
        else {
            hp = max{hp, p(x), mp(leftChild(x))};
            x = rightChild(x);
        }
}

```

FIGURE 48.10: Algorithm $hpRight(d, hp)$.

48.4 Most-Specific-Range Matching

Let $msr(d)$ be the most-specific range that matches the destination address d . For static tables, we may simply represent the n ranges by the up to $2n - 1$ basic intervals they induce. For each basic interval, we determine the most-specific range that matches all points in the interval. These up to $2n - 1$ basic intervals may then be represented as up to $4n - 2$ prefixes [12] with the property that $msr(d)$ is uniquely determined by $LMP(d)$. Now, we may use any of the earlier discussed data structures for static tables in which the filters are prefixes. In this section, therefore, we discuss only those data structures that are suitable for dynamic tables.

48.4.1 Nonintersecting Ranges

Let R be a set of nonintersecting ranges. For simplicity, assume that R includes the range z that matches all destination addresses ($z = [0, 2^{32} - 1]$ in the case of IPv4). With this assumption, $msr(d)$ is defined for every d . Similar to the case of prefixes, for nonintersecting ranges, $msr(d)$ is the range $[maxStart(ranges(d)), minFinish(ranges(d))]$ (Lu and Sahni [19] show that R must contain such a range). We may use

$$PST1(transform1(map1(R)))$$

to find $msr(d)$ using the same method described in Section 48.2.6 to find $LMP(d)$.

Insertion of a range r is to be permitted only if r does not intersect any of the ranges of R . Once we have verified this, we can insert r into $PST1$ as described in Section 48.2.6. Range intersection may be verified by noting that there are two cases for range intersection. When inserting $r = [u, v]$, we need to determine if $\exists s = [x, y] \in R[u < x \leq v < y \vee x < u \leq y < v]$. We see that $\exists s \in R[x < u \leq y < v]$ iff $map1(R)$ has at least one point in the rectangle defined by $x_{left} = u$, $x_{right} = v - 1$, and $y_{top} = u - 1$ (recall that $y_{bottom} = 0$ by default). Hence, $\exists s \in R[x < u \leq y < v]$ iff $minXinRectangle(2^W u - (u - 1) + 2^W - 1, 2^W (v - 1) + 2^W - 1, u - 1)$ exists in $PST1$.

To verify $\exists s \in R[u < x \leq v < y]$, map the ranges of R into 2-dimensional points using the mapping, $map2(r) = (start(r), 2^W - 1 - finish(r))$. Call the resulting set of mapped points $map2(R)$. We see that $\exists s \in R[u < x \leq v < y]$ iff $map2(R)$ has at least one point in the rectangle defined by $x_{left} = u + 1$, $x_{right} = v$, and $y_{top} = (2^W - 1) - v - 1$. To verify this, we maintain a second PST, $PST2$ of points in $transform2(map2(R))$, where $transform2(x, y) = (2^W x + y, y)$. Hence, $\exists s \in R[u < x \leq v < y]$ iff $minXinRectangle(2^W (u + 1), 2^W v + (2^W - 1) - v - 1, (2^W - 1) - v - 1)$ exists.

To delete a range r , we must delete r from both $PST1$ and $PST2$. Deletion of a range from a PST is similar to deletion of a prefix as discussed in Section 48.2.6.

The complexity of the operations to find $msr(d)$, insert a range, and delete a range are the same as that for these operations for the case when R is a set of ranges that correspond to prefixes.

48.4.2 Conflict-Free Ranges

The range set R has a **conflict** iff there exists a destination address d for which $ranges(d) \neq \emptyset \wedge msr(d) = \emptyset$. R is **conflict free** iff it has no conflict. Notice that sets of prefix ranges and sets of nonintersecting ranges are conflict free. The two-PST data structure of Section 48.4.1 may be extended to the general case when R is an arbitrary conflict-free range set. Once again, we assume that R includes the range z that matches all destination addresses. $PST1$ and $PST2$ are defined for the range set R as in Sections 48.2.6 and 48.4.1.

Lu and Sahni [19] have shown that when R is conflict free, $msr(d)$ is the range

$$[maxStart(ranges(d)), minFinish(ranges(d))]$$

Hence, $msr(d)$ may be obtained by performing the operation

$$minXinRectangle(2^W d - d + 2^W - 1, \infty, d)$$

on $PST1$. Insertion and deletion are complicated by the need to verify that the addition or deletion of a range to/from a conflict-free range set leaves behind a conflict-free range set. To perform this check efficiently, Lu and Sahni [19] augment $PST1$ and $PST2$ with a representation for the chains in the normalized range-set, $norm(R)$, that corresponds to

R. This requires the use of several red-black trees. The reader is referred to [19] for a description of this augmentation.

The overall complexity of the augmented data structure of [19] is $O(\log n)$ for each operation when *RBPSTs* are used for *PST1* and *PST2*. When *RPSTs* are used, the search complexity is $O(W)$ and the insert and delete complexity is $O(W + \log n) = O(W)$.

Acknowledgment

This work was supported, in part, by the National Science Foundation under grant CCR-9912395.

References

- [1] A. Andersson and S. Nillson, Improved behavior of tries by adaptive branching, *Information Processing Letters*, 46, 295-300, 1993.
- [2] A. Basu and G. Narlikar, Fast incremental updates for pipelined forwarding engines, *IEEE INFOCOM*, 2003.
- [3] F. Baboescu, S. Singh, and G. Varghese, Packet classification for core routers: Is there an alternative to CAMs?, *IEEE INFOCOM*, 2003.
- [4] M. Berg, M. Kreveld, and J. Snoeyink, Two- and three-dimensional point location in rectangular subdivisions, *Journal of Algorithms*, 18(2), 256-277, 1995.
- [5] M. deBerg, M. vanKreveld, and M. Overmars, Computational geometry: Algorithms and applications, Second Edition, Springer Verlag, 1997.
- [6] G. Cheung and S. McCanne, Optimal routing table design for IP address lookups under memory constraints, *IEEE INFOCOM*, 1999.
- [7] K. Claffy, H. Braun, and G. Polyzos, A parameterizable methodology for internet traffic flow profiling, *IEEE Journal of Selected Areas in Communications*, 1995.
- [8] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, Small forwarding tables for fast routing lookups, *ACM SIGCOMM*, 3-14, 1997.
- [9] W. Doeringer, G. Karjoth, and M. Nassehi, Routing on longest-matching prefixes, *IEEE/ACM Transactions on Networking*, 4, 1, 86-97, 1996.
- [10] P.V. Emde Boas, R. Kass, and E. Zijlstra, Design and implementation of an efficient priority queue, *Mathematical Systems Theory*, 10, 99-127, 1977.
- [11] F. Ergun, S. Mittra, S. Sahinalp, J. Sharp, and R. Sinha, A dynamic lookup scheme for bursty access patterns, *IEEE INFOCOM*, 2001.
- [12] A. Feldman and S. Muthukrishnan, Tradeoffs for packet classification, *IEEE INFOCOM*, 2000.
- [13] P. Gupta and N. McKeown, Dynamic algorithms with worst-case performance for packet classification, *IFIP Networking*, 2000.
- [14] P. Gupta, B. Prabhakar, and S. Boyd, Near-optimal routing lookups with bounded worst case performance, *IEEE INFOCOM*, 2000.
- [15] E. Horowitz, S. Sahni, and D. Mehta, Fundamentals of data structures in C++, W.H. Freeman, NY, 1995, 653 pages.
- [16] K. Kim and S. Sahni, IP lookup by binary search on length. *Journal of Interconnection Networks*, to appear.
- [17] B. Lampson, V. Srinivasan, and G. Varghese, IP lookup using multi-way and multi-column search, *IEEE INFOCOM*, 1998.
- [18] S. Lin and N. McKeown, A simulation study of IP switching, *IEEE INFOCOM*, 2000.
- [19] H. Lu and S. Sahni, $O(\log n)$ dynamic router-tables for prefixes and ranges. Submitted.

- [20] H. Lu and S. Sahni, Dynamic IP router-tables using highest-priority matching. Submitted.
- [21] H. Lu and S. Sahni, A B-tree dynamic router-table design. Submitted.
- [22] E. McCreight, Priority search trees, *SIAM Jr. on Computing*, 14, 1, 257-276, 1985.
- [23] S. Nilsson and G. Karlsson, Fast address look-up for Internet routers, *IEEE Broadband Communications*, 1998.
- [24] S. Sahni and K. Kim, Efficient construction of fixed-stride multibit tries for IP lookup, *Proceedings 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, 2001.
- [25] S. Sahni and K. Kim, Efficient construction of variable-stride multibit tries for IP lookup, *Proceedings IEEE Symposium on Applications and the Internet (SAINT)*, 220-227, 2002.
- [26] S. Sahni and K. Kim, $O(\log n)$ dynamic packet routing, *IEEE Symposium on Computers and Communications*, 443-448, 2002.
- [27] S. Sahni and K. Kim, Efficient dynamic lookup for bursty access patterns, submitted.
- [28] D. Shah and P. Gupta, Fast updating algorithms for TCAMs, *IEEE Micro*, 21, 1, 36-47, 2002.
- [29] V. Srinivasan, Fast and efficient Internet lookups, *CS Ph.D Dissertation*, Washington University, Aug., 1999.
- [30] V. Srinivasan and G. Varghese, Faster IP lookups using controlled prefix expansion, *ACM Transactions on Computer Systems*, Feb:1-40, 1999.
- [31] S. Suri, G. Varghese, and P. Warkhede, Multiway range trees: Scalable IP lookup with fast updates, *GLOBECOM*, 2001.
- [32] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, Scalable high speed IP routing lookups, *ACM SIGCOMM*, 25-36, 1997.
- [33] Dan E. Willard, Log-logarithmic worst-case range queries are possible in space $\theta(N)$, *Information Processing Letters*, 17, 81-84, 1983.
- [34] F. Zane, G. Narlikar, and A. Basu, CoolCAMs: Power-efficient TCAMs for forwarding engines, *IEEE INFOCOM*, 2003.

49

Multi-Dimensional Packet Classification

49.1	Introduction.....	49-1
	Problem Statement	
49.2	Performance Metrics for Classification Algorithms	49-4
49.3	Classification Algorithms.....	49-4
	Background • Taxonomy of Classification Algorithms	
	• Basic Data Structures • Geometric Algorithms •	
	Heuristics • Hardware-Based Algorithms	
49.4	Summary	49-19

Pankaj Gupta
Cypress Semiconductor

49.1 Introduction

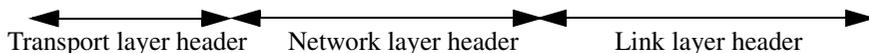
Chapter 48 discussed algorithms for 1-d packet classification. In this chapter we consider multi-dimensional classification in detail. First we discuss the motivation and then the algorithms for multi-dimensional classification. As we will see, packet classification on multiple fields is in general a difficult problem. Hence, researchers have proposed a variety of algorithms which, broadly speaking, can be categorized as “basic search algorithms,” geometric algorithms, heuristic algorithms, or hardware-specific search algorithms. In this chapter, we will describe algorithms that are representative of each category, and discuss which type of algorithm might be suitable for different applications.

Until recently, Internet routers provided only “best-effort” service, servicing packets in a first-come-first-served manner. Routers are now called upon to provide different qualities of service to different applications which means routers need new mechanisms such as admission control, resource reservation, per-flow queueing, and fair scheduling. All of these mechanisms require the router to distinguish packets belonging to different flows.

Flows are specified by *rules* applied to incoming packets. We call a collection of rules a *classifier*. Each rule specifies a flow that a packet may belong to based on some criteria applied to the packet header, as shown in Figure 49.1. To illustrate the variety of classifiers, consider some examples of how packet classification can be used by an ISP to provide different services. Figure 49.2 shows ISP_1 connected to three different sites: enterprise networks E_1 and E_2 and a Network Access Point* (NAP), which is in turn connected to

*A network access point is a network site which acts as an exchange point for Internet traffic. ISPs connect to the NAP to exchange traffic with other ISPs.

PAYLOAD	L4-SP 16b	L4-DP 16b	L4-PROT 8b	L3-SA 32b	L3-DA 32b	L3-PROT 8b	L2-SA 48b	L2-DA 48b
---------	--------------	--------------	---------------	--------------	--------------	---------------	--------------	--------------



L2 = Layer 2 (e.g., Ethernet)

L3 = Layer 3(e.g., IP)

L4 = Layer 4(e.g., TCP)

DA =Destination Address

SA = Source Address

PROT = Protocol

SP = Source Port

DP =Destination Port

FIGURE 49.1: This figure shows some of the header fields (and their widths) that might be used for classifying the packet. Although not shown in this figure, higher layer (e.g., application-level) headers may also be used.

Service	Example
Packet Filtering	Deny all traffic from ISP_3 (on interface X) destined to E_2 .
Policy Routing	Send all voice-over-IP traffic arriving from E_1 (on interface Y) and destined to E_2 via a separate ATM network.
Accounting & Billing	Treat all video traffic to E_1 (via interface Y) as highest priority and perform accounting for the traffic sent this way.
Traffic Rate Limiting	Ensure that ISP_2 does not inject more than 10Mbps of email traffic and 50Mbps of total traffic on interface X .
Traffic Shaping	Ensure that no more than 50Mbps of web traffic is injected into ISP_2 on interface X .

TABLE 49.1 Examples of services enabled by packet classification

Flow	Relevant Packet Fields:
Email and from ISP_2	Source Link-layer Address, Source Transport port number
From ISP_2	Source Link-layer Address
From ISP_3 and going to E_2	Source Link-layer Address, Destination Network-Layer Address
All other packets	-

TABLE 49.2 Flows that an incoming packet must be classified into by the router at interface X in Figure 49.2.

ISP_2 and ISP_3 . ISP_1 provides a number of different services to its customers, as shown in Table 49.1.

Table 49.2 shows the flows that an incoming packet must be classified into by the router at interface X . Note that the flows specified may or may not be mutually exclusive. For example, the first and second flows in Table 49.2 overlap. This is common in practice, and when no explicit priorities are specified, we follow the convention that rules closer to the top of the list take priority (referred to as the “*First matching rule in table*” tie-breaker rule in Chapter 48).

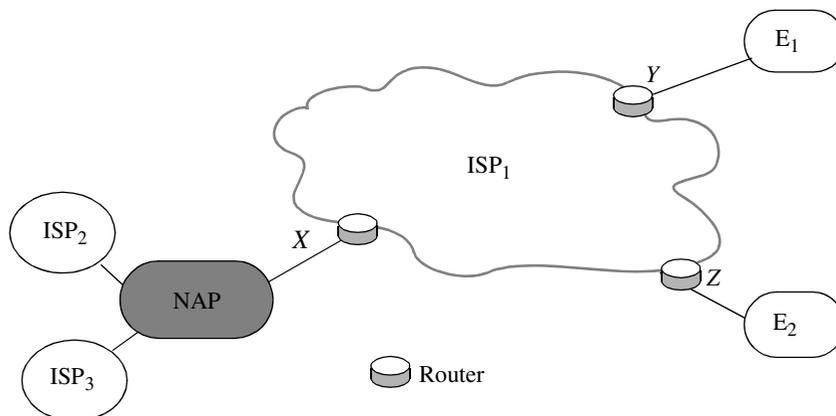


FIGURE 49.2: Example network of an ISP (ISP_1) connected to two enterprise networks (E_1 and E_2) and to two other ISP networks across a network access point (NAP).

Rule	Network-layer Destination (address/mask)	Network-layer Source (address/mask)	Transport-layer Destination	Transport-layer Protocol	Action
R1	152.163.190.69/255.255.255.255	152.163.80.11/255.255.255.255	*	*	Deny
R2	152.168.3.0/255.255.255.0	152.163.200.157/255.255.255.255	eq www	udp	Deny
R5	152.163.198.4/255.255.255.255	152.163.160.0/255.255.252.0	gt 1023	tcp	Permit
R6	0.0.0.0/0.0.0.0	0.0.0.0/0.0.0.0	*	*	Permit

TABLE 49.3 An example 4-dimensional real-life classifier.

49.1.1 Problem Statement

Each rule of a classifier has d components. $R[i]$ is the i^{th} component of rule R , and is a regular expression on the i^{th} field of the packet header. A packet P is said to match rule R , if $\forall i$, the i^{th} field of the header of P satisfies the regular expression $R[i]$. In practice, a rule component is not a general regular expression but is often limited by syntax to a simple address/mask or operator/number(s) specification. In an address/mask specification, a 0 (respectively 1) at bit position x in the mask denotes that the corresponding bit in the address is a don't care (respectively significant) bit. Examples of operator/number(s) specifications are *eq 1232* and *range 34-9339*. Note that a prefix can be specified as an address/mask pair where the mask is contiguous — i.e., all bits with value 1 appear to the left of bits with value 0 in the mask. It can also be specified as a range of width equal to 2^t where $t = 32 - \text{prefixlength}$. Most commonly occurring specifications can be represented by ranges. An example real-life classifier in four dimensions is shown in Table 49.3. By convention, the first rule R1 is of highest priority and rule R7 is of lowest priority. Some example classification results are shown in Table 49.4

Longest prefix matching for routing lookups is a special-case of one-dimensional packet classification. All packets destined to the set of addresses described by a common prefix may be considered to be part of the same flow. The address of the next hop where the

Packet Header	Network-layer Destination	Network-layer Source	Transport-layer Destination	Transport-layer Protocol	Best matching rule, Action
P1	152.163.190.69	152.163.80.11	www	tcp	R1, Deny
P2	152.168.3.21	152.163.200.157	www	udp	R2, Deny
P3	152.163.198.4	152.163.160.10	1024	tcp	R5, Permit

TABLE 49.4 Example classification results from the real-life classifier of Table 49.3.

packet should be forwarded to is the associated action. The length of the prefix defines the priority of the rule.

49.2 Performance Metrics for Classification Algorithms

1. *Search speed* — Faster links require faster classification. For example, links running at 10Gbps can bring 31.25 million packets per second (assuming minimum sized 40 byte TCP/IP packets).
2. *Low storage requirements* — Small storage requirements enable the use of fast memory technologies like SRAM (Static Random Access Memory). SRAM can be used as an on-chip cache by a software algorithm and as on-chip SRAM for a hardware algorithm.
3. *Ability to handle large real-life classifiers.*
4. *Fast updates* — As the classifier changes, the data structure needs to be updated. We can categorize data structures into those which can add or delete entries incrementally, and those which need to be reconstructed from scratch each time the classifier changes. When the data structure is reconstructed from scratch, we call it “pre-processing”. The update rate differs among different applications: a very low update rate may be sufficient in firewalls where entries are added manually or infrequently, whereas a router with per-flow queues may require very frequent updates.
5. *Scalability in the number of header fields used for classification.*
6. *Flexibility in specification* — A classification algorithm should support general rules, including prefixes, operators (range, less than, greater than, equal to, etc.) and wildcards. In some applications, non-contiguous masks may be required.

49.3 Classification Algorithms

49.3.1 Background

For the next few sections, we will use the example classifier in Table 49.5 repeatedly. The classifier has six rules in two fields labeled $F1$ and $F2$; each specification is a prefix of maximum length 3 bits. We will refer to the classifier as $C = \{R_j\}$ and each rule R_j as a 2-tuple: $\langle R_{j1}, R_{j2} \rangle$.

Rule	F1	F2
R_1	00*	00*
R_2	0*	01*
R_3	1*	0*
R_4	00*	0*
R_5	0*	1*
R_6	*	1*

TABLE 49.5 Example 2-dimensional classifier.

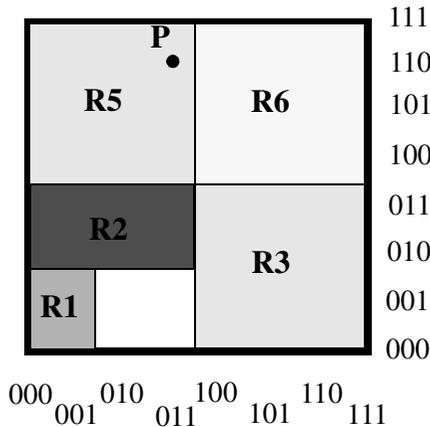


FIGURE 49.3: Geometric representation of the classifier in Table 49.5. A packet represents a point, for instance $P(011,110)$ in two-dimensional space. Note that R_4 is hidden by R_1 and R_2 .

Bounds from Computational Geometry

There is a simple geometric interpretation of packet classification. While a prefix represents a contiguous interval on the number line, a two-dimensional rule represents a rectangle in two-dimensional euclidean space, and a rule in d dimensions represents a d -dimensional hyper-rectangle. A classifier is therefore a collection of prioritized hyper-rectangles, and a packet header represents a point in d dimensions. For example, Figure 49.3 shows the classifier in Table 49.5 geometrically in which high priority rules overlay lower priority rules. Classifying a packet is equivalent to finding the highest priority rectangle that contains the point representing the packet. For example, point $P(011,110)$ in Figure 3 would be classified by rule R_5 .

There are several standard geometry problems such as *ray shooting*, *point location* and *rectangle enclosure* that resemble packet classification. Point location involves finding the enclosing region of a point, given a set of non-overlapping regions. The best bounds for point location in N rectangular regions and $d > 3$ dimensions are $O(\log N)$ time with $O(N^d)$ space;[†] or $O((\log N)^{d-1})$ time with $O(N)$ space [8, 9]. In packet classification, hyper-rectangles can overlap, making classification at least as hard as point location. Hence, a solution is either impracticably large (with 100 rules and 4 fields, N^d space is about 100MBytes) or too slow ($(\log N)^{d-1}$ is about 350 memory accesses).

We can conclude that: (1) Multi-field classification is considerably more complex than

[†]The time bound for $d \leq 3$ is $O(\log \log N)$ [8] but has large constant factors.

Range	Constituent Prefixes
[4, 7]	01**
[3, 8]	0011,01**,1000
[1, 14]	0001,001*,01**,10**,110*,1110

TABLE 49.6 Example 4-bit ranges and their constituent prefixes.

Category	Algorithms
<i>Basic data structures</i>	Linear search, caching, hierarchical tries, set-pruning tries
Geometry-based	Grid-of-tries, AQT, FIS
Heuristic	RFC, hierarchical cuttings, tuple-space search
Hardware	Ternary CAM, bitmap-intersection (and variants)

TABLE 49.7 Categories (not non-overlapping) of classification algorithms.

one-dimensional longest prefix matching, and (2) Complexity may require that practical solutions use heuristics.

Range lookups

Packet classification is made yet more complex by the need to match on ranges as well as prefixes. The cases of looking up static arbitrary ranges, and dynamic *conflict-free* ranges in one dimension have been discussed in Chapter 48. One simple way to handle dynamic arbitrary (overlapping) ranges is to convert each W -bit range to a set of $2W - 2$ prefixes (see Table 49.6) and then use any of the longest prefix matching algorithms detailed in Chapter 48, thus resulting in $O(NW)$ prefixes for a set consisting of N ranges.

49.3.2 Taxonomy of Classification Algorithms

The classification algorithms we will describe here can be categorized into the four classes shown in Table 49.7.

We now proceed to describe representative algorithms from each class.

49.3.3 Basic Data Structures

Linear search

The simplest data structure is a linked-list of rules stored in order of decreasing priority. A packet is compared with each rule sequentially until a rule is found that matches all relevant fields. While simple and storage-efficient, this algorithm clearly has poor scaling properties; the time to classify a packet grows linearly with the number of rules.

Hierarchical tries

A d -dimensional hierarchical radix trie is a simple extension of the one dimensional radix trie data structure, and is constructed recursively as follows. If d is greater than 1, we first construct a 1-dimensional trie, called the $F1$ -trie, on the set of prefixes $\{R_{j1}\}$, belonging to dimension $F1$ of all rules in the classifier, $C = \{R_j\} = \{< R_{j1}, R_{j2}, \dots, R_{jd} >\}$. For each prefix, p , in the $F1$ -trie, we recursively construct a $(d - 1)$ -dimensional hierarchical trie, T_p , on those rules which specify exactly p in dimension $F1$, i.e., on the set of rules $\{R_j : R_{j1} = p\}$. Prefix p is linked to the trie T_p using a next-trie pointer. The storage complexity of the data structure for an N -rule classifier is $O(NdW)$. The data structure for the classifier in Table 49.5 is shown in Figure 49.4. Hierarchical tries are sometimes called “multi-level tries,” “backtracking-search tries,” or “trie-of-tries”.

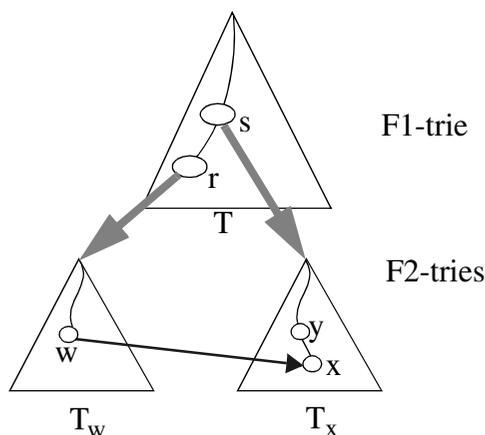


FIGURE 49.6: The conditions under which a switch pointer exists from node w to node x .

Set-pruning tries

A set-pruning trie data structure [13] is similar, but with reduced query time obtained by replicating rules to eliminate recursive traversals. The data structure for the classifier in Table 49.5 is shown in Figure 49.5. The query algorithm for an incoming packet (v_1, v_2, \dots, v_d) need only traverse the $F1$ -trie to find the longest matching prefix of v_1 , follow its next-trie pointer (if present), traverse the $F2$ -trie to find the longest matching prefix of v_1 , and so on for all dimensions. The rules are replicated to ensure that every matching rule will be encountered in the path. The query time is reduced to $O(dW)$ at the expense of increased storage of $O(N^d dW)$ since a rule may need to be replicated $O(N^d)$ times. Update complexity is $O(N^d)$, and hence, this data structure works only for relatively static classifiers.

49.3.4 Geometric Algorithms

Grid-of-tries

The grid-of-tries data structure, proposed by Srinivasan et al [11] for 2-dimensional classification, reduces storage space by allocating a rule to only one trie node as in a hierarchical trie, and yet achieves $O(W)$ query time by pre-computing and storing a *switch pointer* in some trie nodes. A switch pointer is labeled with '0' or '1' and guides the search process. The conditions which must simultaneously be satisfied for a switch pointer labeled b ($b = '0'$ or $'1'$) to exist from a node w in the trie T_w to a node x of another trie T_x are (see Figure 49.6):

1. T_x and T_w are distinct tries built on the prefix components of dimension $F2$. T_x and T_w are pointed to by two distinct nodes, say r and s respectively of the same trie, T , built on prefix components of dimension $F1$.
2. The bit-string that denotes the path from the root node to node w in trie T_w concatenated with the bit b is identical to the bit-string that denotes the path from the root node to node x in the trie T_x .
3. Node w does not have a child pointer labeled b , and
4. Node s in trie T is the closest ancestor of node r that satisfies the above conditions.

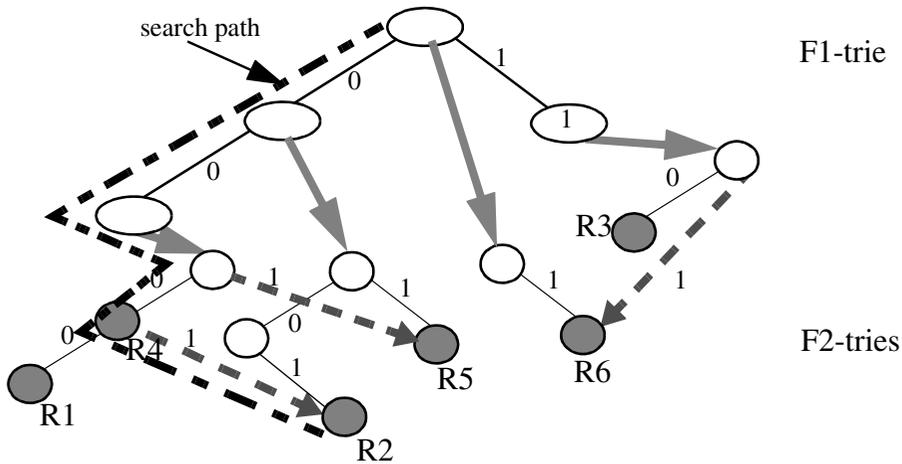


FIGURE 49.7: The grid-of-tries data structure. The switch pointers are shown dashed. The path traversed by the query algorithm on an incoming packet (000, 010) is shown.

If the query algorithm traverses paths $U1(s, root(T_x), y, x)$ and $U2(r, root(T_w), w, x)$ in a hierarchical trie, it need only traverse the path $(s, r, root(T_w), w, x)$ on a grid-of-tries. This is because paths $U1$ and $U2$ are identical (by condition 2 above) till $U1$ terminates at node w because it has no child branch (by condition 3 above). The switch pointer eliminates the need for backtracking in a hierarchical trie without the storage overhead of a set-pruning trie. Each bit of the packet header is examined at most once, so the time complexity reduces to $O(W)$, while storage complexity $O(NW)$ is the same as a 2-dimensional hierarchical trie. However, the presence of switch pointers makes incremental updates difficult, so the authors recommend rebuilding the data structure (in time $O(NW)$) for each update. An example of the grid-of-tries data structure is shown in Figure 49.7.

Reference [11] reports 2MBytes of storage for a 20,000 two-dimensional classifier with destination and source IP prefixes in a maximum of 9 memory accesses.

Grid-of-tries works well for two dimensional classification, and can be used for the last two dimensions of a multi-dimensional hierarchical trie, decreasing the classification time complexity by a factor of W to $O(NW^{d-1})$. As with hierarchical and set-pruning tries, grid-of-tries handles range specifications by splitting into prefixes.

Cross-producting

Cross-producting [11] is suitable for an arbitrary number of dimensions. Packets are classified by composing the results of separate 1-dimensional range lookups for each dimension as explained below.

Constructing the data structure involves computing a set of ranges (basic intervals), G_k , of size $s_k = |G_k|$, projected by rule specifications in each dimension $k, 1 \leq k \leq d$. Let $r_k^j, 1 \leq j \leq s_k$, denote the j^{th} range in G_k . r_k^j may be encoded simply as j in the k^{th} dimension. A cross-product table C_T of size $\prod_{k=1}^{k=d} s_k$ is constructed, and the best matching rule for each entry $(r_1^{i_1}, r_2^{i_2}, \dots, r_d^{i_d}), 1 \leq i_k \leq s_k, 1 \leq k \leq d$ is pre-computed and stored. Classifying a packet (v_1, v_2, \dots, v_d) involves a range lookup in each dimension k to identify the range $r_k^{i_k}$ containing point v_k . The tuple $\langle r_1^{i_1}, r_2^{i_2}, \dots, r_d^{i_d} \rangle$ (or, if using the above encoding for r_k^j , the tuple $\langle i_1, i_2, \dots, i_d \rangle$) is then found in the cross-product table C_T

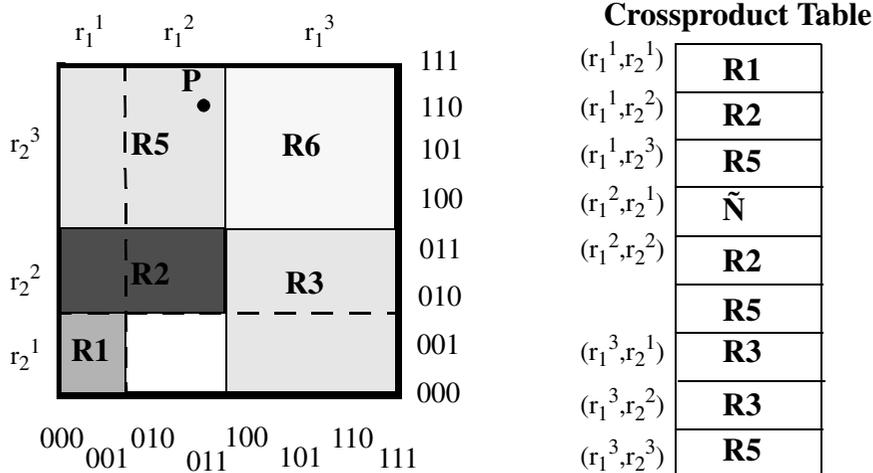


FIGURE 49.8: The table produced by the crossproducting algorithm and its geometric representation.

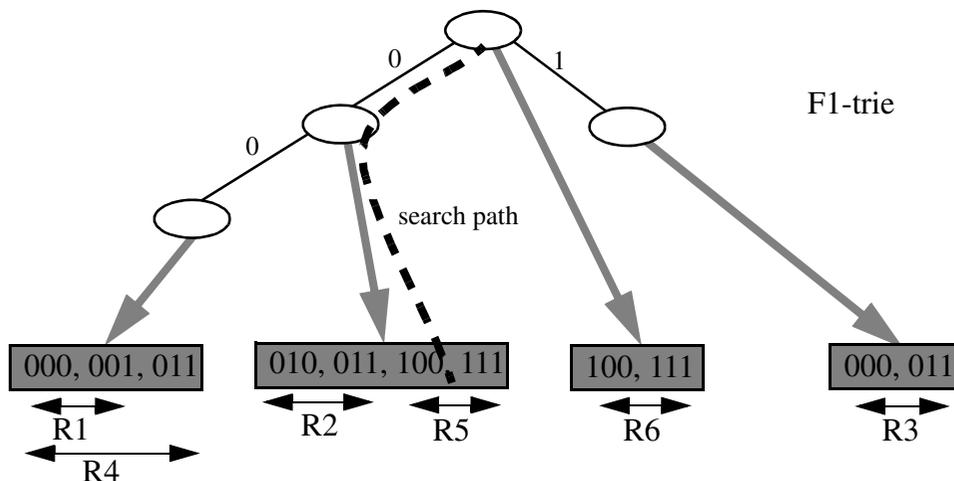


FIGURE 49.9: The data structure of [7] for the example classifier of Table 49.5. The search path for example packet P(011, 110) resulting in R5 is also shown.

which contains the pre-computed best matching rule. Figure 49.8 shows an example.

Given that N prefixes lead to at most $2N - 2$ ranges, $s_k \leq 2N$ and C_T is of size $O(N^d)$. The lookup time is $O(dt_{RL})$ where t_{RL} is the time complexity of finding a range in one dimension. Because of its high worst case storage complexity, cross-producting is suitable only for very small classifiers. Reference [11] proposes using an on-demand cross-producting scheme together with caching for classifiers bigger than 50 rules in five dimensions. Updates require reconstruction of the cross-product table, and so cross-producting is suitable for relatively static classifiers.

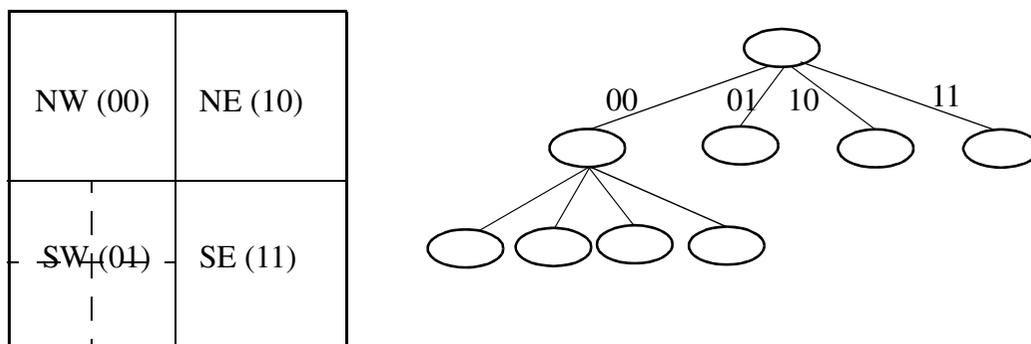


FIGURE 49.10: A quadtree constructed by decomposition of two-dimensional space. Each decomposition results in four quadrants.

A 2-dimensional classification scheme [7]

Lakshman and Stiliadis [7] propose a 2-dimensional classification algorithm where one dimension, say $F1$, is restricted to have prefix specifications while the second dimension, $F2$, is allowed to have arbitrary range specifications. The data structure first builds a trie on the prefixes of dimension $F1$, and then associates a set G_w of non-overlapping ranges to each trie node, w , that represents prefix p . These ranges are created by (possibly overlapping) projections on dimension $F2$ of those rules, S_w , that specify exactly p in dimension $F1$. A range lookup data structure (e.g., an array or a binary search tree) is then constructed on G_w and associated with trie node w . An example is shown in Figure 49.9.

Searching for point $P(v_1, v_2)$ involves a range lookup in data structure G_w for each trie node, w , encountered. The search in G_w returns the range containing v_2 , and hence the best matching rule. The highest priority rule is selected from the rules $\{R_w\}$ for all trie nodes encountered during the traversal.

The storage complexity is $O(NW)$ because each rule is stored only once in the data structure. Queries take $O(W \log N)$ time because an $O(\log N)$ range lookup is performed for every node encountered in the $F1$ -trie. This can be reduced to $O(W + \log N)$ using fractional cascading [1], but that makes incremental updates impractical.

Area-based quadtree

The Area-based Quadtree (AQT) was proposed by Buddhikot et al. [2] for two-dimensional classification. AQT allows incremental updates whose complexity can be traded off with query time by a tunable parameter. Each node of a quadtree [1] represents a two dimensional space that is decomposed into four equal sized quadrants, each of which is represented by a child node. The initial two dimensional space is recursively decomposed into four equal-sized quadrants till each quadrant has at most one rule in it (Figure 49.10 shows an example of the decomposition). Rules are allocated to each node as follows. A rule is said to cross a quadrant if it completely spans at least one dimension of the quadrant. For instance, rule R6 spans the quadrant represented by the root node in Figure 49.10, while R5 does not. If we divide the 2-dimensional space into four quadrants, rule R5 crosses the north-west quadrant while rule R3 crosses the south-west quadrant. We call the set of rules crossing the quadrant represented by a node in dimension k , the k -crossing filter set (k -CFS) of that node.

Two instances of the same data structure are associated with each quadtree node —

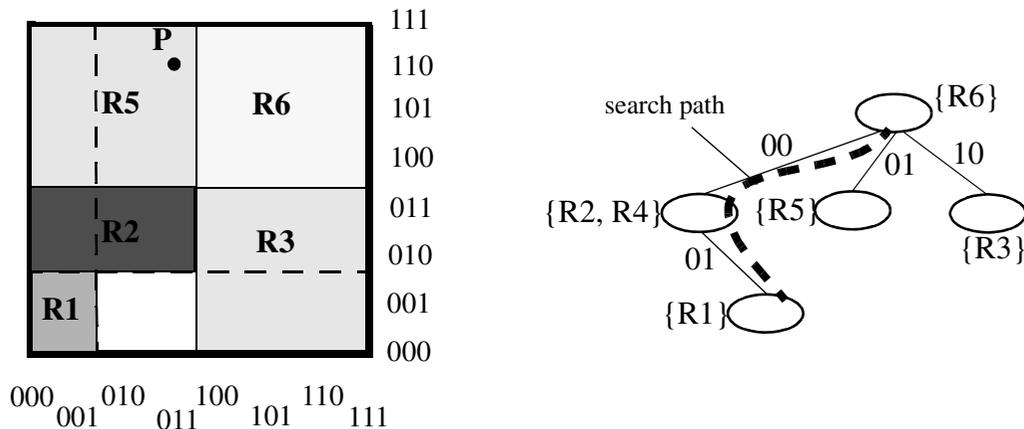


FIGURE 49.11: An AQT data structure. The path traversed by the query algorithm on an incoming packet (000, 010) yields R1 as the best matching rule.

each stores the rules in k -CFS ($k=1,2$). Since rules in crossing filter sets span at least one dimension, only the range specified in the other dimension need be stored. Queries proceed two bits at a time by transposing one bit from each dimension, with two 1-dimensional lookups being performed (one for each dimension on k -CFS) at each node. Figure 49.11 shows an example.

Reference [2] proposes an efficient update algorithm that, for N two-dimensional rules, has $O(NW)$ space complexity, $O(\alpha W)$ search time and $O(\alpha \sqrt[N]{N})$ update time, where α is a tunable integer parameter.

Fat Inverted Segment tree (FIS-tree)

Feldman and Muthukrishnan [3] propose the Fat Inverted Segment tree (FIS-tree) for two dimensional classification as a modification of a segment tree. A segment tree [1] stores a set S of possibly overlapping line segments to answer queries such as finding the highest priority line segment containing a given point. A segment tree is a balanced binary search tree containing the end points of the line segments in S . Each node, w , represents a range G_w , the leaves represent the original line segments in S , and parent nodes represent the union of the ranges represented by their children. A line segment is allocated to a node w if it contains G_w but not $G_{parent(w)}$. The highest priority line segment allocated to a node is pre-computed and stored at the node. A query traverses the segment tree from the root, calculating the highest priority of all the pre-computed segments encountered. Figure 49.12 shows an example segment tree.

An FIS-tree is a segment tree with two modifications: (1) The segment tree is compressed (made “fat” by increasing the degree to more than two) in order to decrease its depth and occupies a given number of levels l , and (2) Up-pointers from child to parent nodes are used. The data structure for 2-dimensions consists of an FIS-tree on dimension $F1$ and a range lookup data associated with each node. An instance of the range lookup data structure associated with node w of the FIS-tree stores the ranges formed by the $F2$ -projections of those classifier rules whose $F1$ -projections were allocated to w .

A query for point $P(v_1, v_2)$ first solves the range lookup problem on dimension $F1$. This returns a leaf node of the FIS-tree representing the range containing the point v_1 . The query algorithm then follows the up-pointers from this leaf node towards the root node, carrying

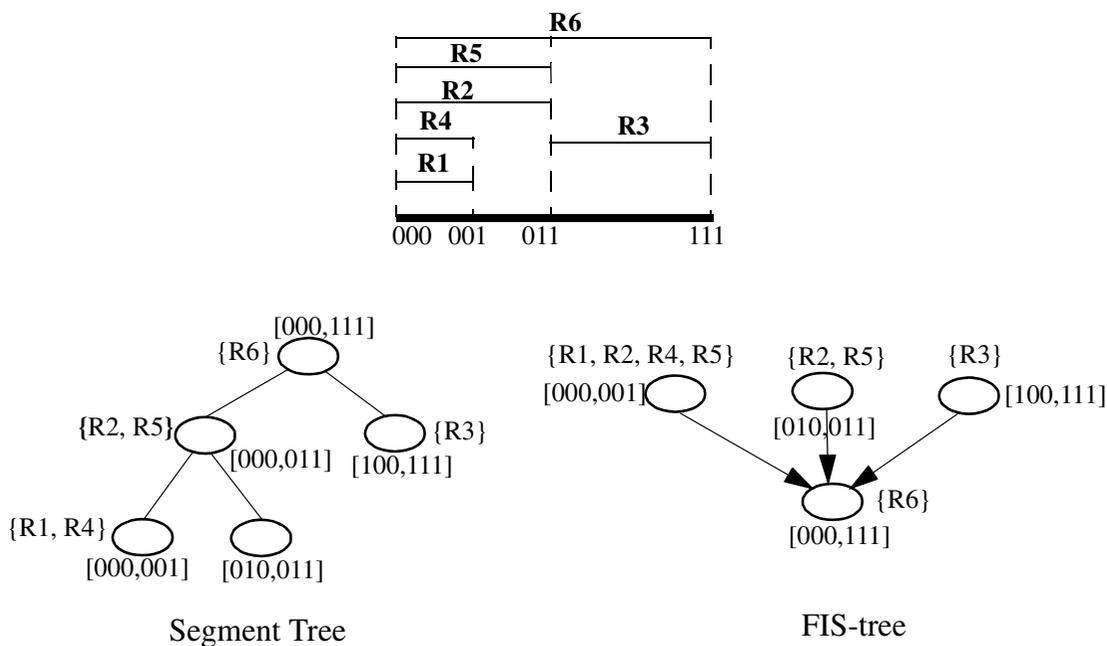


FIGURE 49.12: The segment tree and the 2-level FIS-tree for the classifier of Table 49.5.

out 1-dimensional range lookups at each node. The highest priority rule containing the given point is calculated at the end of the traversal.

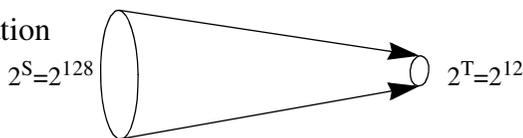
Queries on an l -level FIS-tree have complexity $O((l + 1)t_{RL})$ with storage complexity $O(ln^{1+1/l})$, where t_{RL} is the time for a 1-dimensional range lookup. Storage space can be traded off with search time by varying l . Modifications to the FIS-tree are necessary to support incremental updates — even then, it is easier to support inserts than deletes [3]. The static FIS-tree can be extended to multiple dimensions by building hierarchical FIS-trees, but the bounds are similar to other methods studied earlier [3].

Measurements on real-life 2-dimensional classifiers are reported in [3] using the static FIS-tree data structure. Queries took 15 or fewer memory operations with a two level tree, 4-60K rules and 5MBytes of storage. Large classifiers with one million 2-dimensional rules required 3 levels, 18 memory accesses per query and 100MBytes of storage.

Dynamic multi-level tree algorithms

Two algorithms, called *Heap-on-Trie (HoT)* and *Binarysearchtree-on-Trie (BoT)* are introduced in [6] that build a heap and binary search tree respectively on the last dimension, and multi-level tries on all the remaining $d - 1$ dimensions. If $W = O(\log N)$: *HoT* has query complexity $O(\log^d N)$, storage complexity $O(N \log^d N)$, and update complexity $O(\log^{d+1} N)$; and *BoT* has query complexity $O(\log^{d+1} N)$, storage complexity $O(N \log^d N)$, and update complexity $O(\log^d N)$. If $W \neq O(\log N)$, each of the above complexity formulae need to be modified to replace a factor $O(\log^{d-1} N)$ with $O(W^{d-1})$.

Simple One-step Classification



Recursive Classification

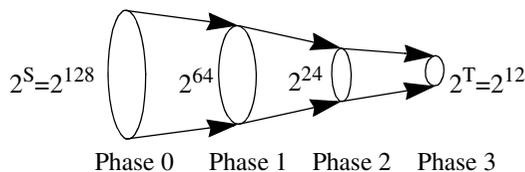


FIGURE 49.13: Showing the basic idea of Recursive Flow Classification. The reduction is carried out in multiple phases, with a reduction in phase I being carried out recursively on the image of the phase $I - 1$. The example shows the mapping of 2^S bits to 2^T bits in three phases.

49.3.5 Heuristics

As we saw in Section 49.3.1, the packet classification problem is expensive to solve in the worst-case — theoretical bounds state that solutions to multi-field classification either require storage that is geometric, or a number of memory accesses that is polylogarithmic, in the number of classification rules. We can expect that classifiers in real networks have considerable structure and redundancy that might be exploited by a heuristic. That is the motivation behind the algorithms described in this section.

Recursive Flow Classification (RFC)

RFC [4] is a heuristic for packet classification on multiple fields. Classifying a packet involves mapping S bits in the packet header to a T bit action identifier, where $T = \log N, T \ll S$. A simple, but impractical method could pre-compute the action for each of the 2^S different packet headers, yielding the action in one step. RFC attempts to perform the same mapping over several phases, as shown in Figure 49.13; at each stage the algorithm maps one set of values to a smaller set. In each phase a set of memories return a value shorter (i.e., expressed in fewer bits) than the index of the memory access. The algorithm, illustrated in Figure 49.14, operates as follows:

1. In the first phase, d fields of the packet header are split up into multiple chunks that are used to index into multiple memories in parallel. The contents of each memory are chosen so that the result of the lookup is narrower than the index.
2. In subsequent phases, memories are indexed using the results from earlier phases.
3. In the final phase, the memory yields the action.

The algorithm requires construction of the contents of each memory detailed in [4]. This paper reports that with real-life four-dimensional classifiers of up to 1,700 rules, RFC appears practical for 10Gbps line rates in hardware and 2.5Gbps rates in software. However, the storage space and pre-processing time grow rapidly for classifiers larger than 6,000 rules. An optimization described in [4] reduces the storage requirement of a 15,000 four-field classifier to below 4MBytes.

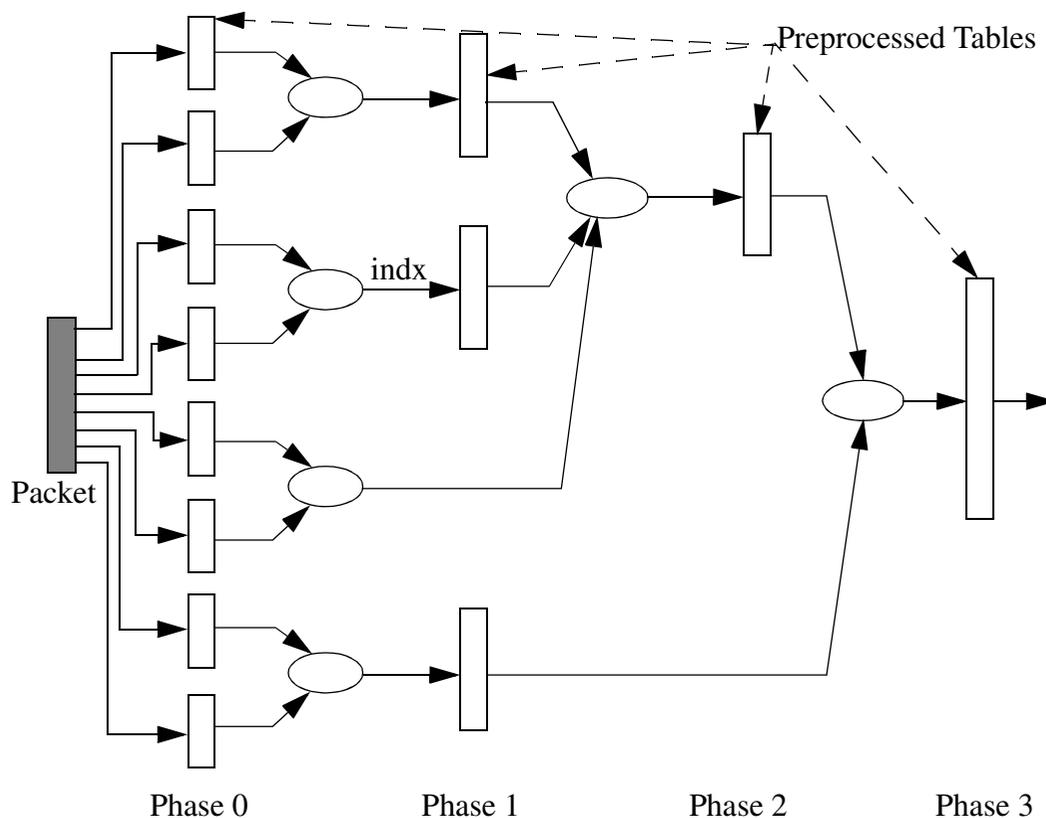


FIGURE 49.14: Packet flow in RFC.

Hierarchical Intelligent Cuttings (HiCuts)

HiCuts [5] partitions the multi-dimensional search space guided by heuristics that exploit the structure of the classifier. Each query leads to a leaf node in the HiCuts tree, which stores a small number of rules that can be searched sequentially to find the best match. The characteristics of the decision tree (its depth, degree of each node, and the local search decision to be made at each node) are chosen while pre-processing the classifier based on its characteristics (see [5] for the heuristics used).

Each node, v , of the tree represents a portion of the geometric search space. The root node represents the complete d -dimensional space, which is partitioned into smaller geometric sub-spaces, represented by its child nodes, by cutting across one of the d dimensions. Each sub-space is recursively partitioned until no sub-space has more than B rules, where B is a tunable parameter of the pre-processing algorithm. An example is shown in Figure 49.15 for two dimensions with $B = 2$.

Parameters of the HiCuts algorithm can be tuned to trade-off query time against storage requirements. On 40 real-life four-dimensional classifiers with up to 1,700 rules, HiCuts requires less than 1 MByte of storage with a worst case query time of 20 memory accesses, and supports fast updates.

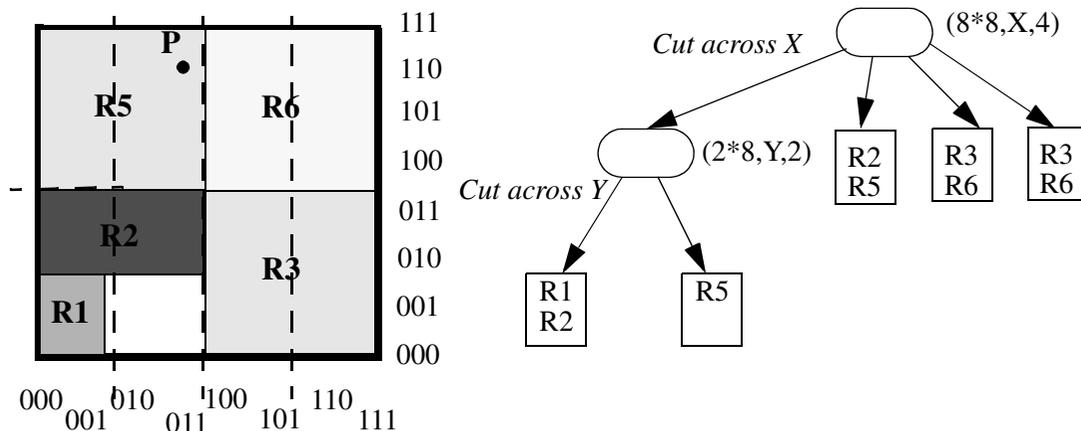


FIGURE 49.15: A possible HiCuts tree for the example classifier in Table 49.5. Each ellipse in the tree denotes an internal node v with a tuple \langle size of 2-dimensional space represented, dimension to cut across, number of children \rangle . Each square is a leaf node which contains the actual classifier rules.

Rule	Specification	Tuple
R1	(00*,00*)	(2,2)
R2	(0**,01*)	(1,2)
R3	(1**,0**)	(1,1)
R4	(00*,0**)	(2,1)
R5	(0**,1**)	(1,1)
R6	(***,1**)	(0,1)

Tuple	Hash Table Entries
(0,1)	{R6}
(1,1)	{R3,R5}
(1,2)	{R2}
(2,1)	{R4}
(2,2)	{R1}

FIGURE 49.16: The tuples and associated hash tables in the tuple space search scheme for the example classifier of Table 49.5.

Tuple Space Search

The basic tuple space search algorithm [12] decomposes a classification query into a number of exact match queries. The algorithm first maps each d -dimensional rule into a d -tuple whose i^{th} component stores the length of the prefix specified in the i^{th} dimension of the rule (the scheme supports only prefix specifications). Hence, the set of rules mapped to the same tuple are of a fixed and known length, and can be stored in a hash table. Queries perform exact match operations on each of the hash tables corresponding to all possible tuples in the classifier. An example is shown in Figure 49.16.

Query time is M hashed memory accesses, where M is the number of tuples in the classifier. Storage complexity is $O(N)$ since each rule is stored in exactly one hash table. Incremental updates are supported and require just one hashed memory access to the hashed table associated with the tuple of the modified rule. In summary, the tuple space search algorithm performs well for multiple dimensions in the average case if the number of tuples is small. However, the use of hashing makes the time complexity of searches and updates non-deterministic. The number of tuples could be very large, up to $O(W^d)$, in the worst

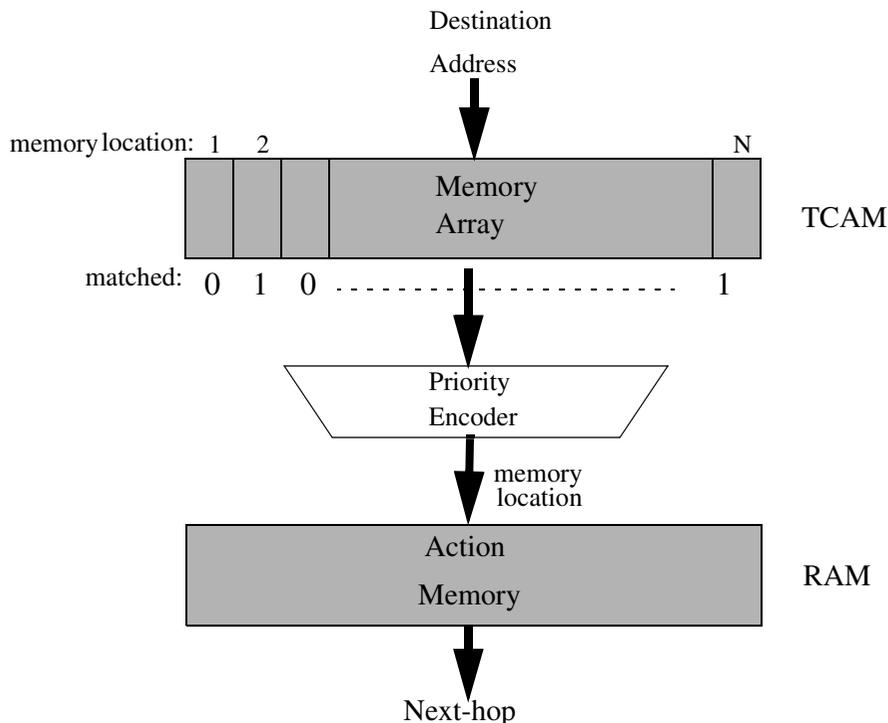


FIGURE 49.17: The classification operation using a ternary CAM. The packet header in this example is assumed to comprise only the destination address.

case. Furthermore, since the scheme supports only prefixes, the storage complexity increases by a factor of $O(W^d)$ for generic rules as each range could be split into W prefixes in the manner explained in Section 49.3.1. Of course, the algorithm becomes attractive for real-life classifiers that have a small number of tuples.

49.3.6 Hardware-Based Algorithms

Ternary CAMs

A TCAM stores each W -bit field as a $(val, mask)$ pair; where val and $mask$ are each W -bit numbers. A $mask$ of '0' wildcards the corresponding bit position. For example, if $W = 5$, a prefix 10^* will be stored as the pair $(10000, 11000)$. An element matches a given input key by checking if those bits of val for which the $mask$ bit is '1', match those in the key.

A TCAM is used as shown in Figure 49.17. The TCAM memory array stores rules in decreasing order of priorities, and compares an input key against every element in the array in parallel. The N -bit bit-vector, $matched$, indicates which rules match and so the N -bit priority encoder indicates the address of the highest priority match. The address is used to index into a RAM to find the action associated with this prefix. TCAMs are being increasingly deployed because of their simplicity of use and speed (as they are able to do classification in hardware at the rate of the hardware clock).

Several companies today ship 9Mb TCAMs capable of single and multi-field classification in as little as 10ns. Both faster and denser TCAMs can be expected in the near future. There are, however, some disadvantages to TCAMs:

1. A TCAM is less dense than a RAM, storing fewer bits in the same chip area. One bit in an SRAM typically requires 4-6 transistors, while one bit in a TCAM requires 11-15 transistors [10]. A 9Mb TCAM running at 100 MHz costs about \$200 today, while the same amount of SRAM costs less than \$10. Furthermore, range specifications need to be split into multiple masks, reducing the number of entries by up to $(2W - 2)^d$ in the worst case. If only two 16-bit dimensions specify ranges, this is a multiplicative factor of 900. Newer TCAMs, based on DRAM technology, have been proposed and promise higher densities.
2. TCAMs dissipate more power than RAM solutions because an address is compared against every TCAM element in parallel. At the time of writing, a 9Mb TCAM chip running at 100 MHz dissipates about 10-15 watts (the exact number varies with manufacturer). In contrast, a similar amount of SRAM running at the same speed dissipates 1W.
3. A TCAM is more unreliable while being in operational use in a router in the field than a RAM, because a *soft-error* (error caused by alpha particles and package impurities that can flip a bit of memory from 0 to 1, or vice-versa) could go undetected for a long amount of time. In a SRAM, only one location is accessed at any time, thus enabling easy on-the-fly error detection and correction. In a TCAM, wrong results could be given out during the time that the error is undetected – which is particularly problematic in such applications as filtering or security.

However, the above disadvantages of a TCAM need to be weighed against its enormous simplicity of use in a hardware platform. Besides, it is the only known “algorithm” that is capable of doing classification at high speeds without explosion in either storage space or time.

Due to their high cost and power dissipation, TCAMs will probably remain unsuitable in the near future for (1) Large classifiers (512K-1M rules) used for microflow recognition at the edge of the network, (2) Large classifiers (256-512K rules) used at edge routers that manage thousands of subscribers (with a few rules per subscriber). Of course, software-based routers need to look somewhere else.

Bitmap-intersection

The bitmap-intersection classification scheme, proposed in [7], is based on the observation that the set of rules, S , that match a packet is the intersection of d sets, S_i , where S_i is the set of rules that match the packet in the i^{th} dimension alone. While cross-producting pre-computes S and stores the best matching rule in S , this scheme computes S and the best matching rule during each classification operation.

In order to compute intersection of sets in hardware, each set is encoded as an N -bit bitmap where each bit corresponds to a rule. The set of matching rules is the set of rules whose corresponding bits are ‘1’ in the bitmap. A query is similar to cross-producting: First, a range lookup is performed in each of the d dimensions. Each lookup returns a bitmap representing the matching rules (pre-computed for each range) in that dimension. The d sets are intersected (a simple bit-wise AND operation) to give the set of matching rules, from which the best matching rule is found. See Figure 49.18 for an example.

Since each bitmap is N bits wide, and there are $O(N)$ ranges in each of d dimensions, the storage space consumed is $O(dN^2)$. Query time is $O(dt_{RL} + dN/w)$ where t_{RL} is the time to do one range lookup and w is the memory width. Time complexity can be reduced by a factor of d by looking up each dimension independently in parallel. Incremental updates

Dimension 1

r_1^1	{R1,R2,R4,R5,R6}	110111
r_1^2	{R2,R5,R6}	010011
r_1^3	{R3,R6}	001001

Dimension 2

r_2^1	{R1,R3,R4}	101100
r_2^2	{R2,R3}	011000
r_2^3	{R5,R6}	000111

Query on P(011,010):

010011	Dimension 1 bitmap
000111	Dimension 2 bitmap
000011	
R5	Best matching rule

FIGURE 49.18: Bitmap tables used in the “bitmap-intersection” classification scheme. See Figure 49.8 for a description of the ranges. Also shown is classification query on an example packet P(011, 110).

Algorithm	Worst-case time complexity	Worst-case storage complexity
Linear Search	N	N
Ternary CAM1	1	N
Hierarchical Tries	W^d	NdW
Set-pruning Tries	dW	N^d
Grid-of-Tries	W^{d-1}	NdW
Cross-producting	dW	N^d
FIS-tree	$(l+1)W$	$l * N^{1+1/l}$
RFC	d	N^d
Bitmap-intersection	$dW + N/memwidth$	dN^2
HiCuts	d	N^d
Tuple Space Search	N	N

TABLE 49.8 Complexity of various classification algorithms. N is the number of rules in d , W -bit wide, dimensions.

are not supported.

Reference [7] reports that the scheme can support up to 512 rules with a 33 MHz field-programmable gate array and five 1Mbit SRAMs, classifying 1Mpps. The scheme works well for a small number of rules in multiple dimensions, but suffers from a quadratic increase in storage space and linear increase in classification time with the size of the classifier. A variation is described in [7] that decreases storage at the expense of increased query time. This work has been extended significantly by [14] by aggregating bitmaps wherever possible and thus decreasing the time spent in reading the bitmaps. Though the bitmap-intersection scheme is primarily meant for hardware, it is easy to see how it can be used in software, where the aggregated bitmap technique of [14] could be especially useful.

49.4 Summary

Please see Table 49.8 for a summary of the complexities of classification algorithms described in this chapter.

References

- [1] M. de Berg, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 2nd rev. ed. 2000.

- [2] M.M. Buddhikot, S. Suri, and M. Waldvogel. "Space decomposition techniques for fast layer-4 switching," *Proceedings of Conference on Protocols for High Speed Networks*, pages 25-41, August 1999.
- [3] A. Feldman and S. Muthukrishnan. "Tradeoffs for packet classification," *Proceedings of Infocom*, vol. 3, pages 1193-202, March 2000.
- [4] P. Gupta and N. McKeown. "Packet Classification on Multiple Fields," *Proc. Sigcomm, Computer Communication Review*, vol. 29, no. 4, pp 147-60, September 1999, Harvard University.
- [5] P. Gupta and N. McKeown. "Packet Classification using Hierarchical Intelligent Cuttings," *Proc. Hot Interconnects VII*, August 99, Stanford. This paper is also available in *IEEE Micro*, pp 34-41, vol. 20, no. 1, January/February 2000.
- [6] P. Gupta and N. McKeown. "Dynamic Algorithms with Worst-case Performance for Packet Classification", *Proc. IFIP Networking*, pp 528-39, May 2000, Paris, France.
- [7] T.V. Lakshman and D. Stiliadis. "High-Speed Policy-based Packet Forwarding Using Efficient Multi-dimensional Range Matching", *Proceedings of ACM Sigcomm*, pages 191-202, September 1998.
- [8] M.H. Overmars and A.F. van der Stappen. "Range searching and point location among fat objects," *Journal of Algorithms*, vol. 21, no. 3, pages 629-656, November 1996.
- [9] F. Preparata and M. I. Shamos. *Computational geometry: an introduction*, Springer-Verlag, 1985.
- [10] F. Shafai, K.J. Schultz, G.F. R. Gibson, A.G. Bluschke and D.E. Somppi. "Fully parallel 30-Mhz, 2.5 Mb CAM," *IEEE Journal of Solid-State Circuits*, vol. 33, no. 11, November 1998.
- [11] V. Srinivasan, S. Suri, G. Varghese, and M. Waldvogel. "Fast and Scalable Layer four Switching," *Proceedings of ACM Sigcomm*, pages 203-14, September 1998.
- [12] V. Srinivasan, S. Suri, and G. Varghese. "Packet Classification using Tuple Space Search", *Proceedings of ACM Sigcomm*, pages 135-46, September 1999.
- [13] P. Tsuchiya. "A search algorithm for table entries with non-contiguous wildcarding," unpublished report, Bellcore.
- [14] F. Baboescu and G. Varghese. "Scalable packet classification," *Proc. Sigcomm*, 2001.

50

Data Structures in Web Information Retrieval

Monika Henzinger
Google Inc.

50.1	Introduction.....	50-1
50.2	Inverted Indices.....	50-1
	Index Compression • Index Granularity	
50.3	Fingerprints	50-3
50.4	Finding Near-Duplicate Documents	50-4
50.5	Conclusions	50-6

50.1 Introduction

Current search engines process thousands of queries per second over a collection of billions of web pages with a sub-second average response time. There are two reasons for this astonishing performance: Massive parallelism and a simple yet efficient data structure, called *inverted index*.

In this chapter we will describe inverted indices. The parallelism deployed by search engines is quite straightforward: Given a collection of documents and a user query the goal of information retrieval is to find all documents that are relevant to a user query and return them in decreasing order of relevance. Since on the Web there are often thousands of matches for a given user query, Web search engines usually return just the top 10 results and retrieve more results only upon request. This can be easily parallelized over m machines: Distribute the documents equally over $m - 1$ machines, find the best up to 10 documents for each machine and return them to the machine without documents, which then merges the lists to determine the overall top 10.

Since the users are only presented with the top 10 results they are usually annoyed if these results contain duplicates or near-duplicates. Thus, it is crucial for a search engine to detect near-duplicate web pages. In Section 50.4 we will describe a technique for doing so based on fingerprints, which we will introduce in Section 50.3.

50.2 Inverted Indices

Given a user query consisting of terms t_1, \dots, t_n , a search engine has to find all documents relevant to the query. Usually Web search engines make the following simplifying assumption: A document is relevant only if it contains *all* query terms. To find all these documents the search engine uses the *inverted (file) index data structure*. It consists of two parts:

- A data structure containing every word t that appears in at least one document together with a pointer p_t for each word and potentially a count f_t of the number of documents containing the word; and
- an *inverted list* for every term t that consists of all the documents containing t and is pointed to by p_t .

One popular implementation is to store the sorted list of words in a search tree whose leaves are connected by a linked list. This imposes an order on the words. The inverted lists are implemented as one large (virtual) array such that t 's inverted list consists of all documents between the position in the array pointed to by p_t and by $p_{t'}$, where t' is the term following t in the sorted list. In the web scenario this array does not fit onto a single machine and thus needs to be distributed over multiple machines, either in RAM or on disk.

Another implementation would be to store the list of words in any search tree or hash function together with the pointers p_t . A special terminator document id (like 0) is appended to the inverted lists and they can be stored either in one large array or in multiple arrays.

To determine all documents containing the query terms t_1, \dots, t_n (called a *Boolean AND*) the intersection of their inverted lists can be computed by traversing their lists in parallel. If one of the query terms is negated (*Boolean NOT*), i.e. the user does not want this term to be contained in the query, the negation of the corresponding list is used. To speed up the computation in the case that a very frequent term is combined with a very infrequent term the documents in the inverted list are stored in sorted order and a one-level or two-level search tree is stored for each inverted list. The one-level search tree consists of every $1/f$ -th entry in the inverted list together with a pointer to the location of this entry in the inverted list, where f is a constant like 100. At query time entries in the inverted list of the frequent term can be skipped quickly to get to the next document that contains the infrequent term. The space requirement is only $1/f$ of the space requirement of the space requirement of the inverted lists.

50.2.1 Index Compression

Of course, data compression is crucial when indexing billions of documents. For that it is usually assumed that the documents are numbered consecutively from 1 to N and that the documents are stored in the inverted list by increasing document id. A simple but powerful technique is called *delta-encoding*: Instead of storing the actual document ids in the inverted list, the document id of the first document is stored together with the difference between the i -th and the $i + 1$ -st document containing the term. When traversing the inverted list the first document id is summed up with delta-values seen so far to get the document id of the current document. Note that delta-encoding does not interfere with the one-level or two-level search tree stored on top of the inverted list. The advantage of delta-encoding is that it turns large integer (document ids) into mostly small integers (the delta values), depending of course on the value of f_t . A variable length encoding scheme, like Golomb codes, of the delta values then leads to considerable space saving. We sketch Golomb codes in the next paragraph. See [13] for more details on variable compression schemes.

There is a trade-off between the space used for the index and the time penalty encountered at query time for decompressing the index. If the index is read from disk, the time to read the appropriate part of the index dominates the compression time and thus more compression can be used. If the index is in RAM, then less compression should be used, except if sophisticated compression is the only way to make the index fit in RAM.

Let n be the number of unique words in all the documents and let f be the number of words in all the documents, including repetitions. Given an integer x it takes at least $\log x$

bits to represent x in binary. However, the algorithm reading the index needs to know how many bits to read. There are these two possibilities: (1) represent x in binary by $\log N$ bits in binary with the top $\log N - \log x$ bits set to 0; or (2) represent x in unary by x 1's followed by a 0. Usually the binary representation is chosen, but the unary representation is more space efficient when $x < \log N$. Golomb codes combine both approaches as follows.

Choose $b \approx 0.69 \frac{N \cdot n}{f}$. The *Golomb code* for an integer $x \geq 1$ consists of two parts:

- A unary encoding of q followed by 0, where $q = \lfloor (x - 1)/b \rfloor$; and
- a binary encoding of the remainder $r = x - qb - 1$. If b is a power of 2, this requires $\log b$ bits.

Note that $\frac{N \cdot n}{f}$ is the average distance between entries in the inverted index, i.e., the average value after the delta-encoding. The value of b was chosen to be roughly 69% of the average. Every entry of value b or less requires 1 bit for the unary part and $\log b$ bits for the binary part. This is a considerable space saving over the binary representation since $\log b + 1 < \log N$. As long as $(x - 1)/b < \log N - \log b - 1$, there is a space saving over implementing using the binary encoding. So only entries that are roughly $\log N$ times larger than the average require more space than the average.

If the frequency f_t of a term t is known, then the inverted list of t can be compressed even better using $b_t \approx 0.69 \frac{N}{f_t}$ instead of b . This implies that for frequent terms, b_t is smaller than b and thus fewer bits are used for small integers, which occur frequently in the delta-encoding of frequent terms.

50.2.2 Index Granularity

Another crucial issue is the level of granularity used in the inverted index. So far, we only discussed storing document ids. However to handle some query operators, like quotes, that require the query term to be next to each other in the document, the exact position of the document is needed. There are two possible ways to handle this:

(1) One way is to consider all documents to be concatenated into one large document and then store the positions in this large document instead of the document ids. Additionally one special inverted list is constructed that stores the position of the first word of each document. At query time an implicit Boolean AND operation is performed with this special inverted list to determine to which document a given position belongs. Note that the above compression techniques continue to work. This approach is very space efficient but incurs a run-time overhead at query time for traversing the special inverted list and performing the Boolean AND operation.

(2) Another solution is to store (document id,position)-pairs in the inverted list and to delta-encode the document ids and also the position within the same document. This approach uses more space, but does not incur the run-time overhead.

Another data structure that a search engine could use are suffix trees (see [Chapter 29](#)). They require, more space and are more complicated, but allow more powerful operations, like searching for syllables or letters.

50.3 Fingerprints

Fingerprints are short strings that represent larger strings and have the following properties:

- If the fingerprints of two strings are different then the strings are guaranteed to be different.

- If the fingerprints of two strings are identical then there is only a small probability that the strings are different.

If two different strings have the same fingerprint, it is called a *collision*. Thus, fingerprints are a type of hash functions where the hash table is populated only very sparsely in exchange for a very low collision probability.

Fingerprints are very useful. For example, search engines can use them to quickly check whether a URL that is contained on a web page is already stored in the index. We will describe how they are useful for finding near-duplicate documents in the next section. One fingerprinting scheme is due to Rabin [12] and is equivalent to cyclic redundancy checks. We follow here the presentation by Broder [3]. Let s be a binary string and let $s' = 1s$, i.e. s' equals s prefixed with a 1. The string $s' = (s_1, s_2, \dots, s_m)$ induces a polynomial $S(t)$ over Z_2 as follows:

$$S(t) = s_1 t^{m-1} + s_2 t^{m-2} + \dots + s_m.$$

Let $P(t)$ be an irreducible polynomial of degree k over Z_2 . The fingerprint of s is defined to be the polynomial

$$f(s) = S(t) \bmod P(t)$$

over Z_2 , which can be simply represented as a string of its coefficients. As shown in [3] the probability that two distinct strings of length m from a set of n strings have the same fingerprint is less than $nm^2/2^k$. Thus, given n and m the probability of collision can be reduced to the required value simply by increasing k , i.e. the length of a fingerprint.

Rabin's fingerprints have the property that given two overlapping strings $s = (s_1, s_2, \dots, s_m)$ and $t = (t_1, t_2, \dots, t_m)$ such that $t_{i+1} = s_i$ for all $1 \leq i < m$ then the fingerprint of t can be computed from the fingerprint of s very efficiently: If

$$f(s) = r_1 t^{k-1} + r_2 t^{k-2} + \dots + r_k,$$

then

$$f(t) = r_2 t^{k-1} + r_3 t^{k-2} + \dots + r_k t + t_m + (r_1 t^k) \bmod P(t).$$

Note that $Q(t) = t^k \bmod P(t)$ is equivalent to $P(t)$ with the leading coefficient removed, which means it can be computed easily. Thus

$$f(t) = r_2 t^{k-1} + r_3 t^{k-2} + \dots + r_k t + t_m + r_1 Q(t).$$

Hence $f(t)$ can be computed from $f(s)$ as follows: Assume $f(s)$ is stored in a shift register and $Q(t)$ is stored in another register. Shift left with t_m as input and if $r_1 = 1$ perform a bit-wise EX-OR operation with $Q(t)$. See [3] for a software implementation for a typical 32-bit word computer.

Other ways of computing fingerprints are cryptographic checksums like Sha-1 or MD5.

50.4 Finding Near-Duplicate Documents

Users of search engine strongly dislike getting near-duplicate results on the same results page. Of course it depends on the user what s/he considers to be near-duplicate. However, most users would agree that pages with the same content except for different ads or a different layout or a different header or footer are near-duplicates. By "near-duplicate" we mean these kinds of "syntactic" duplicates – semantic duplication is even harder detect.

Unfortunately, there are a variety of circumstances that create near-duplicate pages. The main reasons are (1) local copies of public-domain pages (for example the PERL-manual

or the preamble of the US constitution) or various databases and (2) multiple visits of the same page by the crawler (the part of the search engine that retrieves web pages to store in the index) without detection of the replication. The latter can happen because there were small changes to the URL and the content of the page. Usually, crawlers fingerprint the URL as well as the content of the page and thus can easily detect exact duplicates.

There are various approaches to detect near-duplicate pages. One approach that works very well in the Web setting was given by Broder [4] and was validated on the Web [6]. The basic idea is to associate a set of fingerprints with each document such that the similarity of two pages is proportional to the size of the intersection of their respective sets.

We describe first how to compute the fingerprints and show then how they can be used to efficiently detect near-duplicates. The set of fingerprints for a document is computed using the *shingling* technique introduced by Brin et al. [2]. Let a *token* be either a letter, a word, a line, or a sentence in a document. Each document consists of a sequence of token. We call a contiguous sequence of q tokens a q -*shingle* of the document. We want to compute fingerprints for all q -shingles in a document. Using Rabin's fingerprints this can be done efficiently if we start from the first shingle and then use a sliding window approach to compute the fingerprint of the shingle currently in the window.

Let S_D be the set of fingerprints generated for document D . The idea is simply to define the similarity or *resemblance* $r(A, B)$ of document A and B as

$$r(A, B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|}.$$

Experiments have indicated that a resemblance value close to 1 captures well the information notion of "syntactic" near-duplication that we discussed above.

Storing the whole set S_D would take a lot of space. It suffices, however, to keep a fixed number of fingerprints from S_D for each document D . This subset is called a *sketch*. As we will show below the sketches can be computed in time linear in the size of D and will be stored for each document. The resemblance of two documents can be approximated by using the sketches instead of the full sets of fingerprints. Thus, the time is only linear in the size of the sketches.

Sketches are determined as follows: Recall that each fingerprints requires k bits. Thus, $S_D \subseteq \{1, \dots, n\}$, where $n = 2^k$. Let π be a permutation chosen uniformly at random from the set of all permutations of $[n]$. Let $X = \min(\min\{\pi(S_A)\}, \min\{\pi(S_B)\})$.

The crucial observation is that if $\min\{\pi(S_A)\} = \min\{\pi(S_B)\}$, then $X = \min\{\pi(S_A)\}$ must belong to $\pi(S_A) \cap \pi(S_B)$. If $\min\{\pi(S_A)\} \neq \min\{\pi(S_B)\}$, then X belongs to either $\pi(S_A) - \pi(S_B)$ or to $\pi(S_B) - \pi(S_A)$, and, thus, X does not belong to $\pi(S_A) \cap \pi(S_B)$. It follows that $\min\{\pi(S_A)\} = \min\{\pi(S_B)\}$ if and only if X belongs to $\pi(S_A) \cap \pi(S_B)$. Since π was chosen uniformly at random the probability of the latter is

$$\frac{|S_A \cap S_B|}{|S_A \cup S_B|} = r(A, B).$$

It follows that

$$Pr(\min\{\pi(S_A)\} = \min\{\pi(S_B)\}) = r(A, B).$$

We choose p independent random permutations. For each document the sketch consists of the permuted fingerprint values $\min\{\pi_1(S_D)\}, \min\{\pi_2(S_D)\}, \dots, \min\{\pi_p(S_D)\}$. The resemblance of two documents is then estimated by the intersection of their sketches, whose expected value is proportional to the resemblance.

In practice, π cannot be chosen uniformly at random which led to the study of min-wise independent permutations [5].

To detect the near-duplicates in a set of documents we build for each shingle in a sketch the list of documents to which this shingle belongs. Using this list we generate for each shingle and each pair of document containing this shingle an (document id, document id)-pair. Finally we simply sort these pairs to determine the resemblance for each pair. The running time is proportional to the number of (document id, document id)-pairs. If each shingle only belongs to a constant number of documents, it is linear in the number of unique shingles.

In a web search engine it often suffices to remove the near-duplicates at query time. To do this the sketch is stored with each document. The search engine determines the top 50 or so results and then performs a pair-wise near-duplicate detection with some resemblance threshold, removing the near duplicate results, to determine the top 10 results.

The described near-duplicate detection algorithm assumes that the similarity between documents depends on the size of the overlap in their fingerprint sets. This corresponds to the Jaccard coefficient of similarity used in information retrieval. A more standard approach in information retrieval is to assign a weight vector of terms to each document and to define the similarity of two documents as the cosine of their (potentially normalized) term vectors. Charikar [7] gives an efficient near-duplicate detection algorithm based on this measure. He also presents a near-duplicate detection algorithms for another metric, the Earth Mover Distance.

Other similarity detection mechanisms are given in [2, 9–11]. Many near-duplicate web pages are created because a whole web host is duplicated. Thus, a large percentage of the near-duplicate web pages can be detected if near-duplicate web hosts can be found. Techniques for this problem are presented in [1, 8].

50.5 Conclusions

In this chapter we presented the dominant data structure for web search engines, the inverted index. We also described fingerprints, which are useful at multiple places in a search engine, and document sketches for near-duplicate detection. Other useful data structures for web information retrieval are adjacency lists: All web search engines claim to perform some type of analysis of the hyperlink structure. For this, they need to store the list of incoming links for each document, a use of the classic adjacency list representation.

Web search engines also sometimes analyze the log of the user queries issued at the search engine. With hundreds of millions of queries per day these logs are huge and can only be processed with one-pass data stream algorithms.

References

- [1] Bharat, K, Broder, A.Z, Dean, J., and Henzinger, M. R., 2000. A comparison of techniques to find mirrored hosts on the WWW. In *Journal of the American Society for Information Science*, 51(12):114-1122.
- [2] Brin, S., Davis, J., and Garcia-Molina, H., 1995. Copy detection mechanisms for digital documents. In *Proceedings 1995 ACM SIGMOD International conference on Management of Data*, 398-409.
- [3] Broder, A. Z., 1993. Some applications of Rabin's fingerprinting method. In Capocelli, R., De Santis, A., and Vaccaro, U., editors, *Sequences II: Methods in Communications, Security, and Computer Science*, Springer-Verlag, 143-152.
- [4] Broder, A. Z., 1997. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences 1997*, IEEE Computer Society, 21-29.

- [5] Broder, A. Z., Charikar, M., Frieze, A., and Mitzenmacher, M., 1998. Min-wise independent permutations. In *Proceedings 30th Annual ACM Symposium on Theory of Computing*, ACM Press, 327–336.
- [6] Broder, A. Z., Glassman, S. C., Manasse, M. S., Zweig, G., 1997. Syntactic clustering of the Web. In *Proceedings Sixth International World Wide Web Conference*, Elsevier Science, 391–404.
- [7] Charikar, M. S., 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings 34th Annual ACM Symposium on Theory of Computing*, ACM Press, 380–388.
- [8] Cho, J., Shivakumar, N., and Garcia-Molina, H., 2000. Finding replicated web collections. In *Proceedings 2000 ACM International Conference on Management of Data*, ACM Press, 355–366.
- [9] Heintze, N., 1996. Scalable document fingerprinting. In *Proceedings second USENIX Workshop on Electronic Commerce*, 191–200.
- [10] Manber, U., 1994. Finding similar files in a large file system. In *Proceedings Winter 1994 USENIX Conference*, 1–10.
- [11] Shivakumar, N. and Garcia-Molina, H., 1995. SCAM: A copy detection mechanism for digital documents. In *Proceedings Second Annual Conference on the Theory and Practice of Digital Libraries*.
- [12] Rabin, M. O., 1981. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University.
- [13] Witten, I. H., Moffat, A., and Bell, T. C., 1999. Managing Gigabytes. Compressing and Indexing Documents and Images. Morgan Kaufmann, San Francisco.

The Web as a Dynamic Graph

	51.1	Introduction.....	51-1
	51.2	Experimental Observations	51-2
	51.3	Theoretical Growth Models.....	51-3
	51.4	Properties of Web Graphs and Web Algorithmics	51-7
		Generating Function Framework • Average Path Length • Emergence of Giant Components • Search on Web Graphs • Crawling and Trawling	
S. N. Maheshwari <i>Indian Institute of Technology, Delhi</i>	51.5	Conclusions	51-13

51.1 Introduction

The World Wide Web (the Web) was started as an experimental network in 1991. Its growth since then can only be termed explosive. It has several billion pages today and is growing exponentially with time. This growth is totally distributed. There is no central authority to control the growth. The hyperlinks endow the Web with some structure in the sense that viewing the individual web pages as nodes and the hyperlinks as directed edges between them, the Web can be looked upon as a directed graph. What stands out is that this directed graph is not only dynamic—it is rapidly growing and changing—it has been much too large for some time to even have a complete snapshot. Experimental understanding of its structure is based on large but partial web crawls. What properties are being investigated is itself driven by the requirements of the increasingly sophisticated nature of the applications being developed as well as analogies and insights from fields like bibliometrics involving study of citations in academic literature [17].

Let us briefly consider topic search, discussed in detail in [Chapter 50](#), which involves searching for pages on the Web which correspond closely to a given search topic. The seminal work in this area is Kleinberg’s HITS algorithm [19] that assumes that for any topic on the Web there are pages which could be considered to be “authoritative” on that topic, and pages which are “hubs” in the sense that they contain links to relevant pages on that topic. Given a collection of pages and links between them, selected by some sampling method as pertaining to the given topic, HITS algorithm ranks the pages by weights which are representative of the quality of the pages as hubs or authorities. These weights are nothing but principal eigen values, and are, in some sense, a “measure” of the “denseness” of the interconnections between the pages. This model of dense interconnection between hubs and authorities of a given topic gave rise to the notion of “cyber communities” in the Web associated with different topics. Underlying this model of cyber communities was the hypothesis that a subgraph representing a Web community would contain “bipartite cores”.

Bipartite cores, as the name suggests, are complete bipartite graphs corresponding to the hubs and authorities around which the communities are supposed to have developed.

Experimental investigations of the structure of the web graph, taking place on the graphs extracted out of the partial crawls, has confirmed much of the above and more. The structural understanding resulting from the experimental investigations has fueled both, theoretical model building which attempts to explain experimentally observed phenomena, and development of new algorithmic techniques that solve traditional problems of search and information retrieval on the web graph in novel ways. Moreover, the reality of the Web as a structure which is too large and continuously changing, makes the standard off-line and on-line models for algorithm design totally inapplicable. Over the last seven-eight years researchers have attempted to grapple with these unique issues of complexity in the study of the Web. Interestingly, contributions to this study of the Web have come not only from computer scientists, but also from physicists who have brought to Web model building techniques from statistical mechanics that have been successful in predicting macro level behavior of a variety of natural phenomenon from millions of its constituent parts. In this chapter an attempt is made to put together what to the author are the main strands of this rapidly evolving model building. The rest of the chapter is organized as follows: Section 2 surveys the experimental observations and reflects what are the major trends in the findings. Section 3 contains the basic theoretical framework developed to explain the experimental findings. Section 4 contains examples of web algorithmics. Section 5 is crystal gazing and reflects what to the author are the grand challenges.

51.2 Experimental Observations

Recent literature contains reports of a number of experiments conducted to investigate topological properties satisfied by the web graph [6, 9, 22]. These experiments were conducted over a period of time, and using Web samples of varying sizes. Albert, Jeong and Barabasi [6] used nd.edu subset of the Web. Kumar et al.[22] used a cleaned up version of a 1997 web crawl carried out by Alexa Inc. Broder et al. [9] based their measurements on an Altavista crawl having about 200 million pages and 1.5 billion hyperlinks. The most fundamental observation that emerges from these experiments conducted at different times, focusing on different subparts of the Web, is that the degree distribution of nodes in the web graph follows a power law. The degree distribution is said to satisfy power law if the fraction of nodes of degree x is proportional to $x^{-\alpha}$ for $\alpha > 0$. Power law distribution is observed for both the indegrees and the outdegrees of the web graph. Broder et al. report that for indegrees the power coefficient —indexexperimental observations!indegree distribution $\alpha \approx 2.1$, and for outdegrees $\alpha \approx 2.72$. There is a very close match in literature in the value of α for indegree distribution. For outdegree distribution the value of α reported varies from 2.4 to 2.72 [6].

Broder et al. [9] also analysed the crawl for connectedness. Viewing the web graph as an undirected graph, it was observed that 91% of the nodes were connected to each other and formed a giant connected component. Interestingly, it was found that the distribution of the number of connected components by their sizes also satisfied power law ($\alpha \approx 2.5$). Power law distribution in the sizes of the components was observed even when the graph was viewed as directed. However, the size of the largest strongly connected component (giant SCC) was only 28% of the total web crawl. The giant SCC was reachable from about 22% of the nodes (the set IN). About similar percentage of nodes were reachable from the giant SCC (the set OUT). A significant portion of the rest of the nodes constituted, in Broder et al.'s terminology, “tendrils”, nodes reachable from IN or from which the set OUT

is reachable. All the experiments done so far point to fractal like self similar nature of the Web in the sense that structure described above is likely to be exhibited in any non-trivial crawl carried out at any time.

Kumar et al. [22] also carried out experiments to measure the number of bipartite cores in the Web. In a cleaned up version of the web graph consisting of 24 million nodes, they reported discovering around 135,000 bipartite cliques $K_{i,j}$ with $i \geq 3$ and $j = 3$. The number of $K_{i,j}$'s with $i, j = 3$ was approximately 90,000, and the numbers dropped exponentially with increase in j . Finding such cliques in the web graph is an algorithmically challenging problem which we will further discuss in the section on Web algorithmics.

Measurements have also been made of the “diameter” of the Web [6, 9]. If the Web has the structure as asserted in [9], then the probability of a path existing between any two random vertices is approximately 24%, and the average shortest path length is 16. Albert et al. [6] measured the average shortest path length on a directed graph generated having in and outdegree distribution satisfying the power law coefficients of 2.1 and 2.45 respectively. In a directed graph with 8×10^8 vertices the average shortest path length was determined to be approximately 19, and was a linear function of the logarithm of the number of vertices. The Web, therefore, is considered to exhibit the “small worlds” phenomenon [31].

51.3 Theoretical Growth Models

This fractal like structure of an evolving graph, whose growth processes are so organised that the degree distribution of its vertices satisfy power law, which has a large number of bipartite cliques as subgraphs, and exhibits the small world phenomenon has generated a lot of interest among computer scientists recently. Part of the reason is that the web graph does not belong to the much studied $G_{n,p}$ model which consists of all graphs with n vertices having p as the probability that there is an between any two vertices [7]. Graphs in $G_{n,p}$ are essentially sparse and are unlikely to have many bipartite cliques as subgraphs. Moreover, for large n the degree distribution function is Poisson. There is consensus that the primary reason for the web graph to be different from a traditional random graph is that edges in the Web exhibit *preferential attachment*. Hyperlinks from a new web page are more likely to be directed to popular well established website/webpage just as a new manuscript is more likely to cite papers that are already well cited. It is interesting to note that preferential attachment has been used to model evolving phenomena in fields ranging from economics [25], biology [32], to languages [33]. Simon [30] used preferential attachment to explain power law distributions already observed in phenomena like distribution of incomes, distribution of species among genera, distribution of word frequencies in documents etc. A modern exposition of Simon’s argument in the framework of its applicability to web graphs is provided by Mitzenmacher [27], and is the basis of Barabasi et al.’s “mean field theory” based model [8], as well as the “rate equation” approach of Krapivsky and Redner [20]. All three use continuous differential equations to model the dynamics of the evolving web graph. This approach is attractive because of its simplicity and the ease with which it enables one to focus on understanding the issues involved, particularly those relating to power law distributions associated with the Web.

Let us consider the following growth model (we will call this the basic model) which forms the kernel of most of the reported models in literature. At each time step a new node is added to the web graph. This node gets one edge incident at it from one of the existing nodes in the graph, and has one edge pointing out of it. Let us assume that the tail (the node from which the edge emanates) of the edge which is incident at the node just added is chosen uniformly at random from the existing nodes. The head (the node at

which the edge is incident) of the edge which emanates from the node just added is chosen with probability proportional to the indegree of the head in keeping with the preferential attachment paradigm that new web pages tend to attach themselves to popular web pages. We assume, to keep matters simple, that whole process starts with a single node with one edge emanating which is incident at the node itself.

Let $I_k(t)$ and $O_k(t)$ denote the number of nodes in the evolved web graph with indegree and outdegree equal to k respectively after t timesteps. $I_k(t)$ and $O_k(t)$ are random variables for all $k, t \geq 1$. We will assume for the purposes of what follows that their expected values are concentrated around their means, and we will use $I_k(t)$ and $O_k(t)$ to denote the expected value also. Note that, at time t , one node and two directed edges are added to the graph. The expected increase in the value of $I_k(t)$ is controlled by two processes. One is the expected increase in the value of $I_{k-1}(t)$, and the other is the expected decrease in the value of $I_k(t)$. The expected increase is $(k-1)I_{k-1}(t)/t$. This is because of our assumption that the probability is proportional to the indegree of the head of the node at which the edge emanating from the node just added is incident and t is the total number of nodes in the graph. Reasoning in the same way we get that the expected decrease is $kI_k(t)/t$. Since this change has taken place over one unit of time we can write

$$\frac{\Delta(I_k(t))}{\Delta(t)} = \frac{(k-1)I_{k-1}(t) - kI_k(t)}{t}$$

or, in the continuous domain

$$\frac{dI_k(t)}{dt} = \frac{(k-1)I_{k-1}(t) - kI_k(t)}{t}. \quad (51.1)$$

We can solve eq. (51.1) for different values of k starting with 1. For $k=1$ we note that the growth process at each time instance introduces a node of indegree 1. Therefore, eq. (51.1) takes the form

$$\frac{dI_1(t)}{dt} = 1 - \frac{I_1(t)}{t}$$

whose solution has the form $I_1(t) = i_1 t$. Substituting this in the above equation we get $i_1 = 1/2$. Working in the same way we can show that $I_2(t) = t/6$. It is not too difficult to see that $I_k(t)$ are a linear function of t . Therefore, substituting $I_k(t) = i_k t$ in eq. (51.1) we get the recurrence equation

$$i_k = \frac{k-1}{k+1} i_{k-1}$$

whose solution is

$$i_k = \frac{1}{k(k+1)}. \quad (51.2)$$

What interpretation do we put to the solution $I_k(t) = i_k t$? It essentially means that in the steady state (i.e. $t \rightarrow \infty$) the number of nodes in the graph with indegree k is proportional to i_k . Equation (51.2) implies $I_k \sim k^{-2}$, i.e. indegree distribution satisfies power law with $\alpha = 2$.

Let us now develop an estimate for O_k . The counterpart of eq. (51.1) in this case is

$$\frac{dO_k(t)}{dt} = \frac{O_{k-1}(t) - O_k(t)}{t}. \quad (51.3)$$

Assuming that in the steady state $O_k(t) = o_k t$, we can show that $o_1 = 1/2$, and for $k > 1$

$$o_k = \frac{o_{k-1}}{2}.$$

This implies $O_k \sim 2^{-k}$. That the outdegree distribution is exponential and not power law should not come as a surprise if we recall that the process that affected outdegree was uniform random and had no component of preferential attachment associated with it.

It would be instructive to note that very simple changes in the growth model affect the degree distribution functions substantially. Consider the following variation on the growth model analysed above. The edge emanating out of the node added does not just attach itself to another node on the basis of its indegree. With probability β the edge points to a node chosen uniformly at random, and with probability $1 - \beta$ the edge is directed to a node chosen proportionally to its indegree. The eq. (51.1) now takes the form

$$\frac{dI_k(t)}{dt} = \frac{(\beta I_{k-1}(t) + (1 - \beta)(k - 1)I_{k-1}(t)) - (\beta I_k(t) + (1 - \beta)kI_k(t))}{t}$$

Note that the first term of the first part of the r.h.s. corresponds to increase due to the uniform process and the second term increase due to the preferential attachment process. The recurrence equation now takes the form [27]

$$i_k(1 + \beta + k(1 - \beta)) = i_{k-1}(\beta + (k - 1)(1 - \beta))$$

or,

$$\begin{aligned} \frac{i_k}{i_{k-1}} &= 1 - \frac{2 - \beta}{1 + \beta + k(1 - \beta)} \\ &\sim 1 - \left(\frac{2 - \beta}{1 - \beta}\right)\left(\frac{1}{k}\right) \end{aligned}$$

for large k . It can be verified by substitution that

$$i_k \sim k^{-\frac{2-\beta}{1-\beta}} \tag{51.4}$$

satisfies the above recurrence. It should be noted that in this case the power law coefficient can be any number larger than 2 depending upon the value of β .

How do we get power law distribution in the outdegree of the nodes of the graph? Simplest modification to the basic model to ensure that would be to choose the tail of the edge incident at the added node to be chosen with probability proportional to the outdegree of the nodes. Aiello et al. [4], and Cooper and Frieze [13] have both given analysis of the version of the basic model in which, at any time epoch, apart from edges incident and emanating out of the added node being chosen randomly and according to the out and in degree distributions of the existing nodes, edges are added between existing nodes of the graph. Both in and out degree distributions show power law behaviour. From Web modeling perspective this is not particularly satisfying because there is no natural analogue of preferential attachment to explain the process that controls the number of hyperlinks in a web page. Never-the-less all models that enable power law distribution in outdegree of nodes in literature resort to it in one form or the other. We will now summarize the other significant variations of the basic model that have been reported in literature.

Kumar et al. [21] categorise evolutionary web graph models according to the rate of growth enabled by them. *Linear growth* models allow one node to be added at one time epoch along with a fixed number of edges to the nodes already in the graph. *Exponential growth* models allow the graph to grow by a fixed fraction of the current size at each time epoch. The models discussed above are linear growth models. Kumar et al. in [21] introduce the notion of *copying* in which the head of the edge emanating out of the added node is chosen to be the head of an edge emanating out of a “designated” node (chosen randomly).

The intuition for copying is derived from the insight that links out of a new page are more likely to be directed to pages that deal with the “topic” associated with the page. The designated node represents the choice of the topic and the links out of it are very likely links to “other” pages relating to the topic. Copying from the designated node is done with probability $1 - \beta$. With probability β the head is chosen uniformly at random. The exponential growth model that they have analyzed does not involve the copying rule for distribution of the new edges to be added. The tail of a new edge is chosen to be among the new nodes with some probability factor. If the tail is to be among the old nodes, then the old node is chosen with probability proportional to its out degree. The analysis, as in [4, 13], is carried out totally within the discrete domain using martingale theory to establish that the expected values are sharply clustered. For the linear growth model the power law coefficient is the same as in eq. (51.4). The copying model is particularly interesting because estimates of the distribution of bipartite cliques in graphs generated using copying match those found experimentally.

Another approach that has been used to model evolutionary graphs is the so called *Master Equation Approach* introduced by Dorogovtsev et al. [16] which focuses on the probability that at time t a node introduced at time i , has degree k . If we denote this quantity by $p(k, i, t)$, then the equation controlling this quantity for indegree in the basic model becomes

$$p(k, i, t + 1) = \frac{k - 1}{t + 1} p(k - 1, i, t) + \left(1 - \frac{k}{t + 1}\right) p(k, i, t). \quad (51.5)$$

First term on the r.h.s. corresponds to the probability with which the node increases its indegree. The second term is the complementary probability with which the node remains in its former state. The over all degree distribution of nodes of indegree k in the graph is

$$P(k, t) = \frac{1}{t + 1} \sum_{i=0}^{i=t} p(k, i, t).$$

Using this definition over eq. (51.5) we get

$$(t + 1)P(k, t + 1) = (k - 1)P(k - 1, t) + (t - k)P(k, t).$$

For extremely large networks the stationary form of this equation, i.e. at $t \rightarrow \infty$ is

$$P(k) = \frac{k - 1}{k + 1} P(k - 1). \quad (51.6)$$

Notice that the solution to eq. (51.6) with appropriate initial conditions is of the form $P(k) \sim k^{-2}$ which is the same as that obtained by working in the continuous domain.

Rigorous analysis of the stochastic processes done so far in [4, 13, 21] has so far taken into account growth, i.e. birth, process only. The combinatorics of a web graph model that involves death processes also has still to be worked out. It must be pointed out that using less rigorous techniques a series of results dealing with issues like non-linear preferential attachment and growth rates, growth rates that change with time (aging and decay), and death processes in the form of edge removals have appeared in literature. This work is primarily being done by physicists. Albert and Barabasi [5], and Dorogovtsev and Mendes [15] are two comprehensive surveys written for physicists which the computer scientists would do well to go through. However, with all this work we are still far away from having a comprehensive model that takes into account all that we understand of the Web. All models view web growth as a global process. But we know that a web page to propagate Esperanto in India is more likely to have hyperlinks to and from pages of Esperanto enthusiasts in rest

of the world. From modeling perspective every new node added during the growth process may be associated with the topic of the node chosen, let us say by preferential attachment, to which the added node first points to. This immediately reduces the set of nodes from which links can point to it. In effect the probability for links to be added between two nodes which are associated with some topics will be a function of how related the topics are. That there is some underlying structure to the web graph that is defined by the topics associated with the nodes in the graph has been in the modeling horizon for some time. Search engines that use web directories organised hierarchically as graphs and trees of topics have been designed [11]. Experiments to discover the topic related underlying structures have been carried out [10, 12, 18]. A model that takes into account the implicit topic based structure and models web growth as a number of simultaneously taking place local processes [2] is as step in this direction. All that is known about the Web and the models that have been developed so far are pointers to the realisation that development mathematically tractable models that model the phenomena faithfully is a challenging problem which will excite the imagination and the creative energies of researchers for some time.

51.4 Properties of Web Graphs and Web Algorithmics

The properties of web graphs that have been studied analytically are average shortest path lengths between two vertices, size of giant components and their distributions. All these properties have been studied extensively for $G_{n,p}$ class of random graphs. Seminal work in this area for graphs whose degrees were given was done by Malloy and Reed [23] who came up with a precise condition under which phase transition would take place and giant components as large as the graph itself would start to show up. In what follows we will discuss these issues using generating functions as done by Newman, Strogatz, and Watts [28] primarily because of the simplicity with which these reasonably complex issues can be handled in an integrated framework.

51.4.1 Generating Function Framework

Following [28] we define for a large undirected graph with N nodes the generating function

$$G_0(x) = \sum_{k=0}^{k=\infty} p_k x^k, \quad (51.7)$$

where p_k is the probability that a randomly chosen vertex has degree k . We assume that the probability distribution is correctly normalised, i.e. $G_0(1) = 1$. $G_0(x)$ can also represent graphs where we know the exact number n_k of vertices of degree k by defining

$$G_0(x) = \frac{\sum_{k=0}^{k=\infty} n_k x^k}{\sum_{k=0}^{k=\infty} n_k}.$$

The denominator is required to ensure that the generating function is properly normalised. Consider the function $[G_0(x)]^2$. Note that coefficient of the power of x^n in $[G_0(x)]^2$ is given by $\sum_{i+j=n} p_i p_j$ which is nothing but the probability of choosing two vertices such that the sum of their degrees is n . The product of two or more generating functions representing different degree distributions can be interpreted in the same way to represent probability distributions reflecting independent choices from the two or more distributions involved.

Consider the problem of estimating the average degree of a vertex chosen at random. The average degree is given by $\sum_k k p_k$ which is also equal to $G'_0(1)$. Interestingly the

average degree of a vertex chosen at random and the average degree of a vertex pointed to by a random edge are different. A random edge will point to a vertex with probability proportional to the degree of the vertex which is of the order of kp_k . The appropriately normalised distribution is given by the generating function

$$\frac{\sum_k kp_k x^k}{\sum_k kp_k} = x \frac{G'_0(x)}{G'_0(1)}. \quad (51.8)$$

The generating function given above in eq. (51.8) will have to be divided by x if we wanted to consider the distribution of degree of immediate neighbors excluding the edges by which one reached them. We will denote that generating function by $G_1(x)$. The neighbors of these immediate neighbors are the *second* neighbors of the original node. The probability that any of the second neighbors connect to any of the immediate neighbors or one another is no more than order of N^{-1} and hence can be neglected when N is large. Under this assumption the distribution of the second neighbors of the originally randomly chosen node is

$$\sum_k p_k [G_1(x)]^k = G_0(G_1(x)).$$

The average number of second neighbors, therefore, is

$$z_2 = \left[\frac{d}{dx} G_0(G_1(x)) \right]_{x=1} = G'_0(1)G'_1(1), \quad (51.9)$$

using the fact that $G_1(1) = 1$.

51.4.2 Average Path Length

We can extend this reasoning to estimate the distributions for the m th neighbors of a randomly chosen node. The generating function for the distribution for the m th neighbor, denoted by $G^m(x)$, is given by

$$G^m(x) = \begin{cases} G_0(x) & m = 1 \\ G^{(m-1)}(G_1(x)) & m \geq 2. \end{cases}$$

Let z_m denote the average number of m th nearest neighbors. We have

$$z_m = \left. \frac{dG^m(x)}{dx} \right|_{x=1} = G'_1(1)G^{(m-1)'}(1) = [G'_1(1)]^{m-1} z_1 = \left[\frac{z_2}{z_1} \right]^{m-1} z_1. \quad (51.10)$$

Let l be the smallest integer such that

$$1 + \sum_{i=1}^{i=l} z_i \geq N.$$

The average shortest path length between two random vertices can be estimated to be of the order l . Using eq. (51.10) we get

$$l \sim \frac{\log[(N-1)(z_2 - z_1) + z_1^2] - \log z_1^2}{\log(z_2/z_1)}.$$

When $N \gg z_1$ and $z_2 \gg z_1$ the above simplifies to

$$l \sim \frac{\log(N/z_1)}{\log(z_2/z_1)} + 1.$$

There do exist more rigorous proofs of this result for special classes of random graphs. The most interesting observation made in [28] is that only estimates of nearest and second nearest neighbors are necessary for calculation of average shortest path length, and that making these purely local measurements one can get a fairly good measure of the average shortest distance which is a global property. One, of course, is assuming that the graph is connected or one is limiting the calculation to the giant connected component.

51.4.3 Emergence of Giant Components

Consider the process of choosing a random edge and determining the component(s) of which one of its end nodes is a part. If there are no other edges incident at that node then that node is a component by itself. Otherwise the end node could be connected to one component, or two components and so on. Therefore, the probability of a component attached to the end of a random edge is the sum of the probability of the end node by itself, the end node connected to one other component, or two other components and so on. If $H_1(x)$ is the generating function for the distribution of the sizes of the components which are attached to one of the ends of the edge, then $H_1(x)$ satisfies the recursive equation

$$H_1(x) = xG_1(H_1(x)). \quad (51.11)$$

Note that each such component is associated with the end of an edge. Therefore, the component associated with a random vertex is a collection of such components associated with the ends of the edges leaving the vertex, and so, $H_0(x)$ the generating function associated with size of the whole component is given by

$$H_0(x) = xG_0(H_1(x)). \quad (51.12)$$

We can use eqs. (51.11) and (51.12) to compute the average component size which, in an analogous manner to computing the average degree of a node, is nothing but

$$H'_0(1) = 1 + G'_0(1)H'_1(1). \quad (51.13)$$

Similarly using eq. (51.11) we get

$$H'_1(1) = 1 + G'_1(1)H'_1(1),$$

which gives the average component size as

$$1 + \frac{G'_0(1)}{1 - G'_1(1)}. \quad (51.14)$$

The giant component first appears when $G'_1(1) = 1$. This condition is equivalent to

$$G'_0(1) = G''_0(1). \quad (51.15)$$

Using eq. (51.7) the condition implied by eq. (51.15) can also be written as

$$\sum_k k(k-2)p_k = 0. \quad (51.16)$$

This condition is the same as that obtained by Molloy and Reed in [23]. The sum on the l.h.s. of eq. (51.16) increases monotonically as edges are added to the graph. Therefore, the giant component comes into existence when the sum on the l.h.s. of eq. (51.16) becomes

positive. Newman et al. state in [28] that once there is a giant component in the graph, then $H_0(x)$ generates the probability distribution of the sizes of the components excluding the giant component. Molloy and Reed have shown in [24] that the giant component almost surely has $cN + o(N)$ nodes. It should be remembered that the results in [23, 24, 28] are all for graphs with specified degree sequences. As such they are applicable to graphs with power law degree distributions. However, web graphs are directed and a model that explains adequately the structure consistent with all that is known experimentally is still to be developed.

51.4.4 Search on Web Graphs

Perhaps the most important algorithmic problem on the Web is mining of data. We will, however, have not much to say about it partly because it has been addressed in [Chapter 50](#), but also because we want to focus on issues that become specially relevant in an evolving web graph. Consider the problem of finding a path between two nodes in a graph. On the Web this forms the core of the peer to peer (P2P) search problem defined in the context of locating a particular file on the Web when there is no information available in a global directory about the node on which the file resides. The basic operation available is to pass messages to neighbors in a totally decentralised manner. Normally a distributed message passing flooding technique on an arbitrary random network would be suspect suspect because of the very large number of messages that may be so generated. Under the assumption that a node knows about the identities of its neighbors and perhaps neighbors' neighbors, Adamic et al. [1] claim, experimentally through simulations, that in power law graphs with N nodes the average search time is of the order of $N^{0.79}$ (graphs are undirected and power law coefficients for the generated graphs is 2.1) when the search is totally random. The intuitive explanation is that even in a random search the search process tends to gravitate towards nodes of high degree. When the strategy is to choose to move to the highest degree neighbor the exponent comes down to 0.70. Adamic et al. [1] have come up a with a fairly simple analysis to explain why random as well as degree directed search algorithms need not have time complexity more than rootic in N (reported exponent is 0.1). In what follows we develop the argument along the lines done by Mehta [26] whose analysis, done assuming that the richest node is selected on the basis of looking at the neighbors and neighbors' neighbors, has resulted in a much sharper bound than the one in [1]. We will assume that the cut off degree, m , beyond which the the probability distribution is small enough to be ignored is given by $m = N^{1/\alpha}$ [3].

Using

$$p_k = p(k) = \frac{k^{-\alpha}}{\sum k^{-\alpha}}$$

as the probability distribution function, the expected degree z of a random node is

$$z = G'_0(1) \approx \frac{\int_1^m k^{1-\alpha} dk}{\int_1^m k^{-\alpha} dk} = \frac{(\alpha - 1)(m^{2-\alpha} - 1)}{(\alpha - 2)(m^{1-\alpha} - 1)} \sim \ln m, \quad (51.17)$$

under the assumption that α tends to 2 and $m^{1-\alpha}$ is small enough to be ignored. Similarly the PDF for $G_1(x)$ is

$$p_1(k) = \frac{p_k k}{\sum p_k k} = \frac{k^{1-\alpha}}{(\sum k^{1-\alpha})} = c k^{1-\alpha}, \quad (51.18)$$

where

$$c = \frac{1}{\sum k^{1-\alpha}} = \frac{\alpha - 2}{(m^{2-\alpha} - 1)} \approx \frac{1}{\ln m} = \frac{\alpha}{\ln N}. \quad (51.19)$$

The CDF for $G_1(x)$ using (51.18) and (51.19) is

$$P(x) = \int_1^x p_1(k)dk = \frac{\ln x}{\ln m}. \quad (51.20)$$

The search proceeds by choosing at each step the highest degree node among the n neighbors of the current node. The total time taken can be estimated to be the sum of the number of nodes traversed at each step. The number of steps itself can be bounded by noting that the sum of the number of steps at each step can not exceed the size of the graph. Let $p_{max}(x, n)$ be the distribution of the degree of the richest node (largest degree node) among the n neighbors of a node. Determining the richest node is equivalent to taking the maximum of n independent random variables and in terms of the CDF $P(x)$,

$$p_{max}(x, n) = \begin{cases} 0 & x = 0 \\ (P(x) - P(x-1))^n & 0 < x \leq m \\ 0 & x > m \end{cases}. \quad (51.21)$$

This can be approximated as follows

$$p_{max}(x, n) = \frac{d(P(x)^n)}{dx} = nP(x)^{n-1} \frac{dP(x)}{dx} = n(\log_m x)^{n-1} c x^{-1}. \quad (51.22)$$

We can now calculate the expected degree of the richest node among the n nodes as

$$f(n) = E[x_{max}(n)] = \sum_1^m x p_{max}(x, n) = nc \sum_1^m (\log_m x)^{n-1}. \quad (51.23)$$

Note that if the number of neighbors of every node on the average is z , then the number of second neighbors seen when one is at the current richest node is $f(z)(z-1)$. this is because one of the neighbors of every other node is the richest node. At the next step, the expected degree of the node whose current degree is $f(z)$ is given by $E(f(z)) = f(f(z))$ and the number of second neighbors seen at this stage correspondingly is $f(f(z))(z-1)$. The overlap between two successive computations of neighbors takes place only when there is an edge which makes a node a neighbor as well as the second neighbor. However, the probability of this happening is of the order of N^{-1} and can be ignored in the limit for large N . Therefore we can assume that at every stage new nodes are scanned and the number of steps, l , is controlled by

$$(z-1) \sum_{i=0}^{i=l} f^i(z) = N. \quad (51.24)$$

Assuming for the sake of simplicity $f(f(z)) = f(z)$ (simulations indicate that value of $f^i(z)$ increases with i and then stabilises) we can set $E[n] = n$ or

$$E(n) = n = nc \sum_1^m (\log_m(x))^{n-1},$$

or

$$(\ln m)^n = \sum_1^m (\ln x)^{n-1} \sim \int_1^m (\ln x)^{n-1} dx. \quad (51.25)$$

Substituting e^t for x we get

$$\begin{aligned}
 (\ln m)^n &= \int_1^{\ln m} e^t t^{n-1} dt \\
 &\sim \int_1^{\ln m} t^{n-1} \left(1 + \sum_0^{\infty} \frac{t^i}{i!}\right) dt \\
 &= \frac{(\ln m)^n}{n} + \left| \sum_0^{\infty} \frac{t^{n+i}}{i!(n+i)} \right|_0^{\ln m} \\
 &\leq \frac{(\ln m)^n}{n} + \frac{(\ln m)^n}{n+c} e^{\ln m},
 \end{aligned}$$

where c is a constant. The above implies

$$1 = \frac{1}{n} + \frac{m}{n+c},$$

which gives

$$n = O(m).$$

Substituting this in eq. (51.24) we get

$$(\ln m - 1) \sum_0^l O(m) = N,$$

or

$$l \sim \frac{N}{m \ln m} = \frac{N^{1-1/\alpha}}{\ln m}.$$

Taking α to be 2.1 the number of steps would be of the order of $N^{0.52}/\ln m$. Mehta [26] also reports results of simulations carried out on graphs generated with α equal to 2.1 using the method given in [3]. The simulations were carried out on the giant components. He reports the number of steps to be growing proportional to $N^{0.34}$. The simulation results are very preliminary as the giant components were not very large (largest was of the order of 20 thousand nodes).

51.4.5 Crawling and Trawling

Crawling can be looked upon as a process where an agent moves along the nodes and edges of a randomly evolving graph. Off-line versions of the Web which are the basis of much of what is known experimentally about the Web have been obtained essentially through this process. Cooper and Frieze [14] have attempted to study the expected performance of a crawl process where the agent makes a fixed number of moves between the two successive time steps involving addition of nodes and edges to the graph. The results are fairly pessimistic. Expected proportion of unvisited nodes is of order of 0.57 of the graph size when the edges are added uniformly at random. The situation is even worse when the edges are distributed in proportion to the degree of nodes. The proportion of the unvisited vertices increases to 0.59.

Trawling, on the other hand, involves analyzing the web graph obtained through a crawl for subgraphs which satisfy some particular structure. Since the web graph representation may run into Tera bytes of data, the traditional random access model for computing the

algorithmic complexity loses relevance. Even the standard external memory models may not apply as data may be on tapes and may not fit totally on the disks available. In these environments it may even be of interest to develop algorithms where the figure of merit may be the number of passes made over the graph represented on tape [29]. In any case the issue in trawling is to design algorithms that efficiently stream data between secondary and main memory.

Kumar et al. discuss in [22] the design of trawling algorithms to determine bipartite cores in web graphs. We will focus only on the core issue here which is that the size of the web graph is huge in comparison to the cores that are to be detected. This requires pruning of the web graph before algorithms for core detection are run. If (i, j) cores (i represents outdegree and j indegree) have to be detected then all nodes with outdegree less than i and indegree less than j have to be pruned out. This can be done with repeated sorting of the web graph by in and out degrees and keeping track of only those nodes that satisfy the pruning criteria. If an index is kept in memory of all the pruned in vertices then repeated sorting may also be avoided.

The other pruning strategy discussed in [22] is the so called *inclusion exclusion* pruning in which the focus at every step is to either discover an (i, j) core or exclude a node from further contention. Consider a node x with outdegree equal to i (these will be termed fans) and let $\Gamma(x)$ be the set of nodes that are potentially those with which x could form an (i, j) core (called centers). An (i, j) core will be formed if and only if there are $i - 1$ other nodes all pointing to each node in $\Gamma(x)$. While this condition is easy to check if there are two indices in memory, the whole process can be done in two passes. In the first identify all the fans with outdegree i . Output for each such fan the set of i centers adjacent to it. In the second pass use an index on the destination id to generate the set of fans pointing to each of the i centers and compute the intersection of these sets. Kumar et al. [22] mention that this process can be batched with index only being maintained of the centers that result out of the fan pruning process. If we maintain the set of fans corresponding to the centers that have been indexed, then using the dual condition that x is a part of the core if and only if the intersection of the sets $\Gamma^{-1}(x)$ has size at least j . If this process results in identification of a core then a core is outputted other wise the node x is pruned out. Kumar et al. [22] claim that this process does not result in any not yet identified cores to be eliminated.

The area of trawling is in its infancy and techniques that will be developed will depend primarily on the structural properties being discovered. It is likely to be influenced by the underlying web model and the whether any semantic information is also used in defining the structure. This semantic information may be inferred by structural analysis or may be available in some other way. Development of future web models will depend largely on our understanding of the web, and they themselves will influence the algorithmic techniques developed.

51.5 Conclusions

The modeling and analysis of web as a dynamic graph is very much in its infancy. Continuous mathematical models, which has been the focus of this write up provides good intuitive understanding at the expense of rigour. Discrete combinatorial models that do not brush the problems of proving concentration bounds under the rug are available for very simple growth models. These growth models do not incorporate death processes, or the issues relating to aging (newly created pages are more likely to be involved in link generation processes) or for that matter that in and out degree distributions on the web are not independent and may depend upon the underlying community structure. The process

of crawling which can visit only those vertices that have at least one edge pointing to it gives a very limited understanding of degree distributions. There are reasons to believe that a significantly large number of pages on the web have only out hyperlinks. All this calls for extensive experimental investigations and development of mathematical models that are tractable and help both in development of new analytical as well as algorithmic techniques.

The author would like to acknowledge Amit Agarwal who provided a willing sounding board and whose insights have significantly influenced the author's approach on these issues.

References

- [1] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman. Search in Power Law Networks. *Physical Review E*. 64:46135-46143, 2001.
- [2] A. Agarwal, S. N. Maheshwari, B. Mehta. An Evolutionary web graph Model. Technical Report, Department of Computer Science and Engineering, IIT Delhi, 2002.
- [3] W. Aiello, F. Chung, and L. Lu. A Random Graph Model for Massive Graphs. In *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*. 2000.
- [4] W. Aiello, F. Chung, and L. Lu. Random Evolution in Massive Graphs. In *Handbook on Massive Data Sets*, (Eds. J. Abello et al.)
- [5] R. Albert, and A. L. Barabasi. Statistical Mechanics of Complex Networks. cond-mat/0106096.
- [6] R. Albert, H. Jeong, and A. L. Barabasi. Diameter of the World Wide Web. *Nature* 401:130-131, 1999.
- [7] B. Bollobas. *Modern Graph Theory*. Springer-Verlag, New York, 1998.
- [8] A. L. Barabasi, R. Albert, and H. Jeong. Mean Field Theory for Scale-free Random Networks. *Physica A*. 272:173-189, 1999.
- [9] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph Structure in the Web: Experiments and Models. In *Proceedings of the 9th World Wide Web Conference*, 2000.
- [10] S. Chakrabarti, M. M. Joshi, K. Punera, and D. V. Pennock. The Structure of Broad Topics on the Web. In *Proceedings of the 11th World Wide Web Conference*, 2002.
- [11] S. Chakrabarti, M. Van den Berg, and B. Dom. A New Approach to Topic Specific Web Resource Discovery. In *Proceedings of the 8th World Wide Web Conference*, 1999.
- [12] C. Chekuri, M. Goldwasser, P. Raghavan, E. Upfal. Web Search using Automatic Classification. In *Proceedings of the 6th World Wide Web Conference*, 1997.
- [13] C. Cooper, and A. Frieze. A General model of Web Graphs. In *Proceedings of the 9th Annual European Symposium on Algorithms*, 2001.
- [14] C. Cooper, and A. Frieze. Crawling on Web Graphs. In *Proceedings of the 43rd Annual Symposium on Theory of Computing*, 2002.
- [15] S. N. Dorogovtsev, and J. F. F. Mendes. Evolution of Networks. cond-mat/0106144.
- [16] S. N. Dorogovtsev, J. F. F. Mendes, and A. N. Samukhin. Structure of Growing Networks with Preferential Linking. *Phys. Rev. Lett.* 85:4633, 2000.
- [17] L. Egghe, and R. Rousseau. *Introduction to Infometrics: Quantitative Methods in Library, Documentation and Information Science*. Elsevier, 1990.
- [18] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring Web Communities from Link Topology. In *Proceedings of the 9th ACM Conference on Hypertext and Hypermedia*, 1998.
- [19] J. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *Journal of the ACM*, 2000.
- [20] P. L. Krapivsky, and S. Redner. Organisation of Growing Random Networks. *Physical*

- Review E.* 63:066123001-066123014, 2001.
- [21] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Stochastic Models for the web graph. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, 2000.
 - [22] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Trawling Emerging Cybercommunities Automatically. In *Proceedings of the 8th World Wide Web Conference*, 1999.
 - [23] M. Malloy, and B. Reed. A Critical Point for Random Graphs with a given Degree Sequence. *Random Structures and Algorithms*. 6:161-179, 1995.
 - [24] M. Malloy, and B. Reed. The Size of a Giant Component of a Random Graph with a given Degree Sequence. *Combinatorics Probability and Computing*. 7:295-305, 1998.
 - [25] B. Mandelbrot. *Fractals and Scaling in Finance*. Springer-Verlag, 1997.
 - [26] B. Mehta. *Search in web graphs*. M. Tech. Thesis, Department of Computer Science and Engineering I.I.T. Delhi, 2002.
 - [27] M. Mitzenmacher. A Brief History of Generative Models for Power Law and Lognormal Distributions. To appear in *Internet Mathematics*.
 - [28] M. E. J. Newman, S.H. Strogatz, and D.J. Watts. Random Graphs with Arbitrary Degree Distribution and their Applications. *Physical Review E*.
 - [29] S. Rajagopalan. Private Communication.
 - [30] H. A. Simon. On a Class of Skew Distribution Functions. *Biometrika*. 42:425-440, 1955.
 - [31] D. J. Watts. *Small Worlds: The Dynamics of Networks between Order and Randomness*. Princeton University Press, 1999.
 - [32] G. U. Yule. A Mathematical Theory of Evolution based on the Conclusions of Dr J.C. Willis, F.R.S. *Philosophical Transaction of the Royal Society of London (Series B)*, 213:21-87, 1925.
 - [33] G. K. Zipf. *Selective Studies and the Principle of Relative Frequency in Language*. Harvard University Press, 1932.

52

Layout Data Structures

52.1	Introduction.....	52-1
52.2	VLSI Technology	52-1
52.3	Layout Data Structures: an Overview	52-4
52.4	Corner Stitching.....	52-4
	Point Finding • Tile Insertion • Storage Requirements of the Corner Stitching Data Structure	
52.5	Corner Stitching Extensions	52-8
	Expanded Rectangles • Trapezoidal Tiles • Curved Tiles • L-shaped Tiles	
52.6	Quad Trees and Variants.....	52-12
	Bisector List Quad Trees • k -d Trees • Multiple Storage Quad Trees • Quad List Quad Trees • Bounded Quad Trees • HV Trees • Hinted Quad Trees	
52.7	Concluding Remarks.....	52-18

Dinesh P. Mehta
Colorado School of Mines

52.1 Introduction

VLSI (Very Large Scale Integration) is a technology that has enabled the manufacture of large circuits in silicon. It is not uncommon to have circuits containing millions of transistors, and this quantity continues to increase very rapidly. Designing a VLSI circuit is itself a very complex task and has spawned the area of VLSI design automation. The purpose of VLSI design automation is to develop software that is used to design VLSI circuits. The VLSI design process is sufficiently complex that it consists of the four steps shown in [Figure 52.1](#). Architectural design is carried out by expert human engineers with some assistance from tools such as simulators. Logic design is concerned with the boolean logic required to implement a circuit. Physical design is concerned with the implementation of logic on a three dimensional physical structure: the VLSI chip. VLSI physical design consists of steps such as floorplanning, partitioning, placement, routing, circuit extraction, etc. Details about VLSI physical design automation may be found in [1–3]. [Chapter 53](#) describes the rich area of data structures for floorplanning. In this chapter, our concern will be with the representation of a circuit in its “physical” form. In order to proceed with this endeavor, it is necessary to first understand the basics of VLSI technology.

52.2 VLSI Technology

We begin with the caveat that our presentation here only seeks to convey the basics of VLSI technology. Detailed knowledge about this area may be obtained from texts such as [4]. The

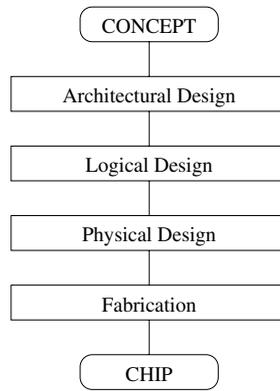


FIGURE 52.1: The VLSI Design stages.

transistor is the fundamental device in VLSI technology and may be viewed as a switch. It consists of a gate, a source, and a drain. The voltage on the gate controls the passage of current between the source and the drain. Thus, the gate can be used to switch the transistor “on” (current flows between the source and the drain) and “off” (no current flows). Basic logic elements such as the inverter (the NOT gate), the NAND gate, and the NOR gate are built using transistors. Transistors and logic gates can be manufactured in layers on a silicon disk called a wafer. Pure silicon is a semiconductor whose electrical resistance is between that of a conductor and an insulator. Its conductivity can be significantly improved by introducing “impurities” called dopants. N-type dopants such as phosphorus supply free electrons, while p-type dopants like boron supply holes. Dopants are diffused into the silicon wafer. This layer of the chip is called the diffusion layer and is further classified into n-type and p-type depending on the type of dopant used. The source and drain of a transistor are formed by separating two n-type regions with a p-type region (or vice versa). A gate is formed by sandwiching a silicon dioxide (an insulator) layer between the p-type region and a layer of polycrystalline silicon (a conductor). Figure 52.2 illustrates these concepts. Since polycrystalline silicon (poly) is a conductor, it is also used for short interconnections

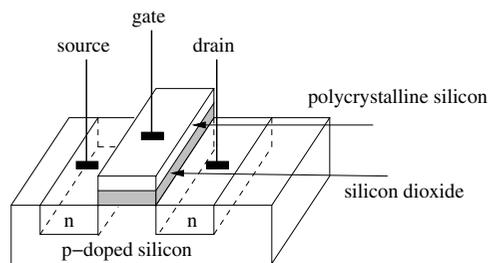


FIGURE 52.2: A transistor.

(wires). Up to this point, we have described the two layers (diff and poly) that are used to make all electronic devices. Although poly conducts electricity, it is not sufficient to complete all the interconnections using one layer. Modern chips usually have several layers of aluminum (“metal”), a conductor, separated from each other by insulators on top of the poly layer. These make it possible for the gates to be interconnected as specified in

the design. Note that a layer of material X (e.g., poly) does not mean that there is a monolithic slab of poly over the entire chip area. The poly is only deposited where gates or wires are needed. The remaining areas are filled with insulating materials and for our purposes may be viewed as being empty. In addition to the layers as described above, it is necessary to have a mechanism for signals to pass between layers. This is achieved by contacts (to connect poly with diffusion or metal) and vias (to connect metal on different layers). Figure 52.3 shows the layout and a schematic of an nMOS inverter. We briefly describe the functioning of the inverter. If the input gate voltage is “0”, the transistor is switched off and there is no connection between the ground signal and the output. The voltage at the output is identical to that of the power source, which is a “1”. If the gate is at “1”, the transistor is switched on and there is a connection between the ground signal “0” and the output, making the output “0”.

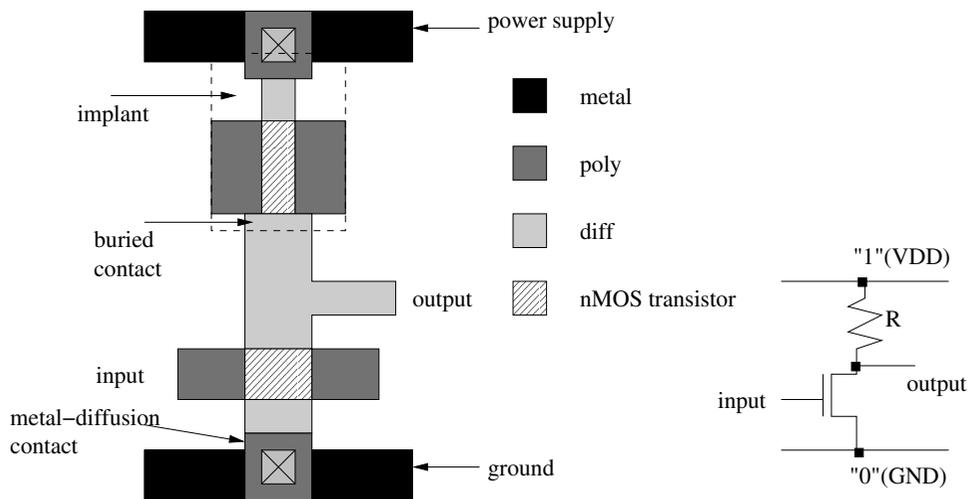


FIGURE 52.3: An inverter.

The purpose of a layout data structure is to store and manipulate the rectangles on each layer. Some important high-level operations that a layout data structure must support are design-rule checking, layout compaction, and parasitic extraction.

Design Rule Checking (DRC): Design rules are the interface between the circuit designer and the process engineer. They specify geometric constraints on the layout so that the patterns on the processed wafer preserve the topology of the designs. An example of a design rule is that the width of a wire must be greater than a specified minimum. If this constraint is violated, it is possible that for the wire to be discontinuous because of errors in the fabrication process. Similarly, if two wires are placed too close to each other, they could touch each other. The DRC step verifies that all design rules have been met. Additional design rules for CMOS technology may be found in [4, page 142]

Parasitic Extraction: Each layer of the chip has a resistance and a capacitance that are critical to the estimation of circuit performance. Inductance is usually less important on the chip, but has greater impact on the I/O components of the chip. Capacitance, resistance, and inductance are commonly referred to as “parasitics”. After a layout has been created, the parasitics must be computed in order to verify that the circuit will meet its performance

goals. (Performance is usually measured by clock cycle times and power dissipation.) The parasitics are computed from the geometry of the layout. For example, the resistance of a rectangular slab of metal is $\frac{\rho l}{tw}$, where ρ is the resistivity of the metal and l , w , and t are the slab's length, width, and thickness, respectively. See [4, Chapter 4] for more examples.

Compaction: The compaction step, as its name suggests, tries to make the layout as small as possible without violating any design rules. This reduces the area of the chip, which could result in more chips being manufactured from a single wafer, which significantly reduces cost per chip. Interestingly, the cost of a chip could grow as a *power of five* of its area [5] making it imperative that area be minimized! Two-dimensional compaction is NP-hard, but one-dimensional compaction can be carried out in polynomial time. Heuristics for 2D compaction often iteratively interleave one-dimensional compactions in the x - and y -directions. For more details, see [6].

52.3 Layout Data Structures: an Overview

There are two types of layout data structures that are based on differing philosophies. The first is in the context of a layout editor. Here, the idea is that a user manually designs the layout, for example, by inserting rectangles of the appropriate dimensions at the appropriate layer. This customized approach is used for library cells. The MAGIC system [7] developed at U.C. Berkeley is an example of a system that included a layout editor. MAGIC was in the public domain and was used to support classes on VLSI design in many universities. The layout editor context is especially important here because it permitted the developers of MAGIC to assume locality of reference; i.e., a user is likely to perform several editing operations in the same area of the layout over a short period of time. The second philosophy is that the layout process is completely automated. This has the advantage that some user-interaction operations do not need to be supported and run time is critical. This approach is more common in industrial software, where automatic translation techniques convert electronic circuits into physical layouts. This philosophy is supported by the quad-tree and variants that were designed specifically for VLSI layout.

52.4 Corner Stitching

The corner stitching data structure was proposed by Ousterhout [8] to store non-overlapping rectilinear circuit components in MAGIC. The data structure is obtained by partitioning the layout area into horizontally maximal rectangular tiles. There are two types of tiles: solid and vacant, both of which are explicitly stored in the corner-stitching data structure. Tiles are obtained by extending horizontal lines from corners of all solid tiles until another solid tile or a boundary of the layout region is encountered. The set of solid and vacant tiles so obtained is unique for a given input. The partitioning scheme ensures that no two vacant or solid tiles share a vertical side. Each tile T is stored as a node which contains the coordinates of its bottom left corner, x_1 and y_1 , and four pointers N , E , W , and S . N (respectively, E , W , S) points to the rightmost (respectively, topmost, bottommost, leftmost) tile neighboring its north (respectively, east, west, south) boundary. The x and y coordinates of the top right corner of T are $T.E \rightarrow x_1$ and $T.N \rightarrow y_1$, respectively, and are easily obtained in $O(1)$ time. Figure 52.4 illustrates the corner stitching data structure.

The corner stitching data structure supports a rich set of operations.

1. Point Finding: given a tile T and a point $p(x, y)$, search for the tile containing p by following a sequence of stitches starting at T .

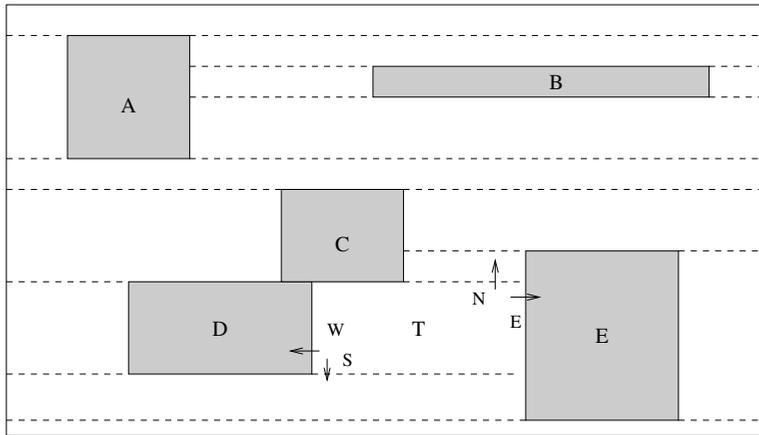


FIGURE 52.4: The corner stitching data structure. Pointers (stitches) are shown for tile T .

2. Neighbor Finding: find all solid and vacant tiles that abut a given tile T .
3. Area Searches: determine whether any solid tiles intersect a given rectangular area R . This operation is used to determine whether a new solid tile can subsequently be inserted into area R . (Recall that tiles in a layer are not permitted to overlap.)
4. Directed Area Enumeration: enumerate all the tiles contained in a given rectangular area in a specified order. This is used during the compaction operation which may require tiles to be visited and compacted in a left-to-right order.
5. Tile Creation: insert a solid tile T into the data structure at a specified location.
6. Tile Deletion: delete a specified solid tile T from the data structure.
7. Plowing: translate a large piece of a design. Move other pieces of the design that lie in its path in the same direction.
8. Compaction: this refers to one-dimensional compaction.

We describe two operations to illustrate corner stitching:

52.4.1 Point Finding

Next, we focus on the point find operation because of its effect on the performance of corner stitching. The algorithm is presented below. Given a pointer to an arbitrary tile T in the layout, the algorithm seeks the tile in the layout containing the point P .

Tile Point_Find (Tile T , Point P)

1. **begin**
2. $current = T$;
3. **while** (P is not contained in $current$)
4. **begin**
5. **while** ($P.y$ does not lie in $current$'s y-range)
6. **if** ($P.y$ is above $current$) $current = current \rightarrow N$;
7. **else** $current = current \rightarrow S$;
8. **while** ($P.x$ does not lie in $current$'s x-range)
9. **if** ($P.x$ is to the right of $current$) $current = current \rightarrow E$;
10. **else** $current = current \rightarrow W$;

```

11.   end
12.   return (current);
13. end

```

Figure 52.5 illustrates the execution of the point find operation on a pathological example. From the start tile T , the **while** loop of line 5 follows north pointers until tile A is reached. We change directions at tile A since its y -range contains P . Next, west pointers are followed until tile F is reached (whose x -range contains P). Notice that the sequence of west moves causes the algorithm to descend in the layout resulting in a vertical position that is similar to that of the start tile! As a result of this misalignment, the outer while loop of the algorithm must execute repeatedly until the point is found (note that the point will eventually be found since the point find algorithm is guaranteed to converge).

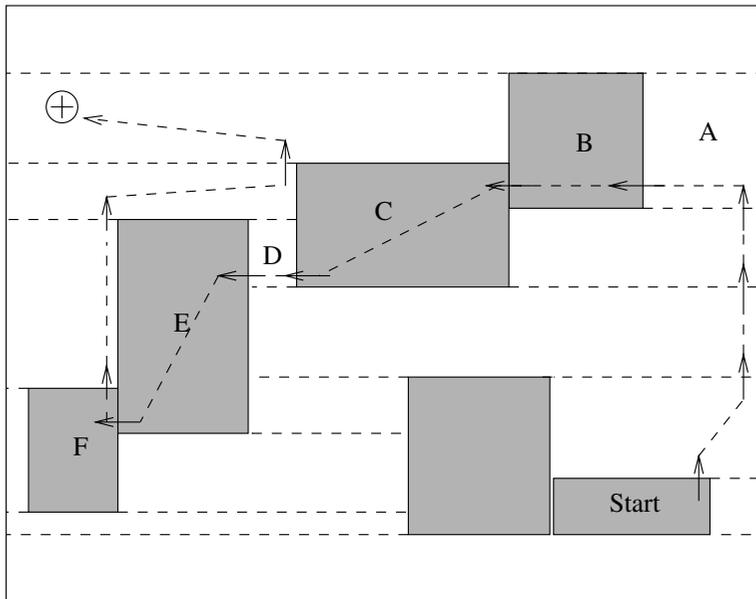


FIGURE 52.5: Illustration of point find operation and misalignment.

52.4.2 Tile Insertion

This operation creates a new solid tile and inserts it into the layout. It accomplishes this by a sequence of split and merge operations. The split operation breaks a tile into two tiles along either a vertical or a horizontal line. The merge operation combines two tiles to form a rectangular tile. Algorithm *Insert* discusses the insertion of a rectangle into the layout.

Insert(A) // (x_1, y_1) and (x_2, y_2) are the bottom left and top right corners of A .

1. **if** ($\text{!AreaSearch}(A)$) **return**; //area is not empty, abort.
2. Let $i = 0$; Split Q_i , the tile containing the north edge of A into two tiles along the line $y = y_2$; Let T be the upper tile and Q_i be the lower tile.
3. **while** (Q_i does not contain the south edge of A)

- (a) Split Q_i vertically into three tiles along $x = x_1$ and $x = x_2$; let the resulting tiles from left to right be L_i , Q_i , and R_i .
 - (b) **if** ($i > 0$)
 - i. Merge R_{i-1} and L_{i-1} into R_i and L_i , respectively, if possible.
 - ii. Merge Q_{i-1} into Q_i ;
 - (c) Let Q_{i+1} be the tile beneath Q_i .
 - (d) Increment i ;
4. Split Q_i along $y = y_1$; Let Q_i be the tile above the split and B the tile below the split; Split Q_i into L_i , Q_i , and R_i using the lines $x = x_1$ and $x = x_2$;
 5. Merge Q_i and Q_{i-1} to get Q_i . Q_i is the newly inserted solid tile;
 6. Merge R_{i-1} , L_{i-1} with neighboring tiles; if R_i (L_i) gets merged, the merged tile is called R_i (L_i). Merge R_i and L_i with neighboring tiles;

Figure 52.6 shows the various steps involved in the insertion algorithm.

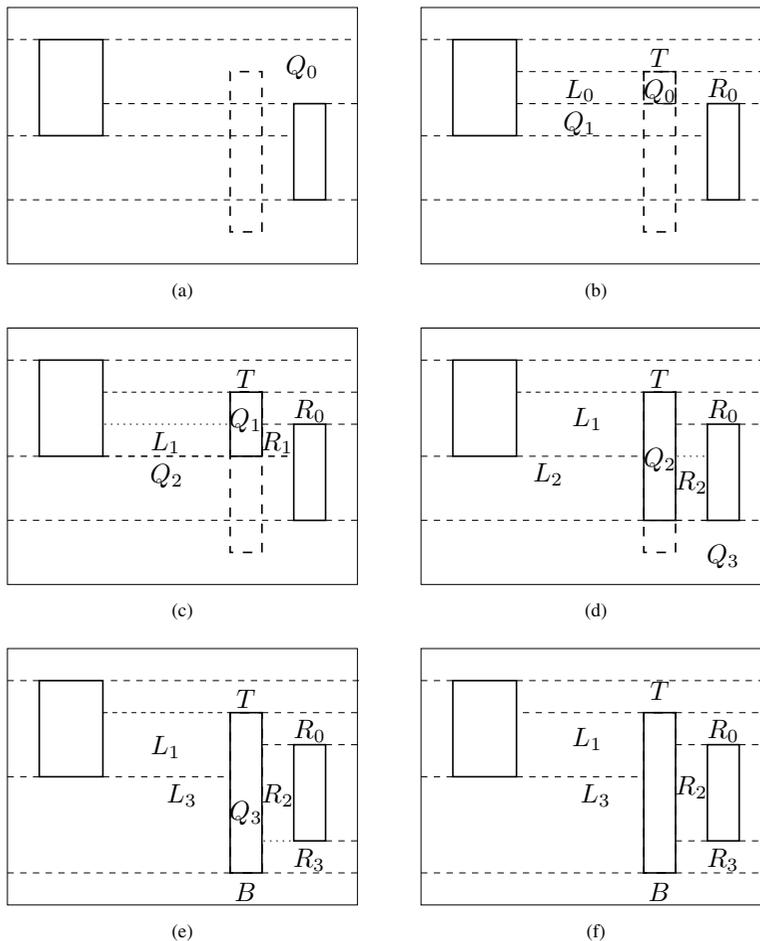


FIGURE 52.6: Illustration of insertion.

The rectangle A to be inserted is represented by thick, dashed lines in Figure 52.6(a). The coordinates of the bottom left and top right corners of A are (x_1, y_1) and (x_2, y_2) , respectively. First, Step 1 of the algorithm uses *AreaSearch* to ensure that no solid tiles intersect A . Step 2 identifies tile Q_0 as the vacant tile containing A 's north edge and splits it by the horizontal line $y = y_2$ into two tiles: T above the split-line and Q_0 below the split-line. Next, in the while loop of Step 3, Q_0 is split by vertical lines at x_1 and x_2 to form L_0, Q_0, R_0 . Tile Q_1 is the vacant tile below Q_0 . The resulting configuration is shown in Figure 52.6(b). In the next iteration, Q_1 is split to form L_1, Q_1, R_1 . L_0 merges into L_1 and Q_0 merges into Q_1 . Tile Q_2 is the vacant tile below Q_1 . The resulting configuration is shown in Figure 52.6(c). Next, Q_2 is split to form L_2, Q_2, R_2 . R_1 is merged into R_2 and Q_1 merged into Q_2 . Figure 52.6(d) shows the configuration after Tile Q_2 is processed. The vacant tile Q_3 below Q_2 contains R 's bottom edge and the while loop of Step 3 is exited. Steps 4, 5, and 6 of the algorithm result in the configuration of Figure 52.6(e). The final layout is shown in Figure 52.6(f).

52.4.3 Storage Requirements of the Corner Stitching Data Structure

Unlike simpler data structures such as arrays and linked lists, it is not trivial to manually estimate the storage requirements of a corner stitched layout. For example, if n items are inserted into a linked list, then the amount of storage required is n multiplied by the number of bytes required by a single list node. Because of vacant tiles, the total number of nodes in corner stitching is considerably more than n and depends on the relative positions of the n rectangles. In [9], a general formula for the memory requirements of the corner stitching data structure on a given layout. This formula requires knowledge about certain geometric properties of the layout called *violations* of the **CV** property and states that a corner stitching data structure representing a set of N solid, rectangular tiles with k violations contains $3N + 1 - k$ vacant tiles. Since each rectangular tile requires 28 bytes, the memory requirements are $28(4N + 1 - k)$ bytes.

52.5 Corner Stitching Extensions

52.5.1 Expanded Rectangles

Expanded rectangles [10] expands solid tiles in the corner stitching data structure so that each tile contains solid material and the empty space around it. No extra tiles are needed to represent empty space. Thus, there are fewer tiles than in corner stitching. However, each tile now requires 44 rather than 28 bytes because additional fields are needed to store the coordinates of the solid portion of the tile. It was determined that expanded rectangles required less memory than corner stitching when the ratio of vacant to solid tiles in corner stitching was greater than 0.414. Operations on the expanded rectangles data structure are similar to those in corner stitching.

52.5.2 Trapezoidal Tiles

Marple et al [11] developed a layout system called “Tailor” that was similar to MAGIC except that it allowed 45 degree layout. Thus, rectangular tiles were replaced by trapezoidal tiles. There are 9 types of trapezoidal tiles as shown in Figure 52.7. An additional field that stores the type of trapezoidal tile is used. The operations on Tailor are similar to those in MAGIC and are implemented in a similar way. It is possible to extend these techniques to arbitrary angles making it possible to describe arbitrary polygonal shapes.

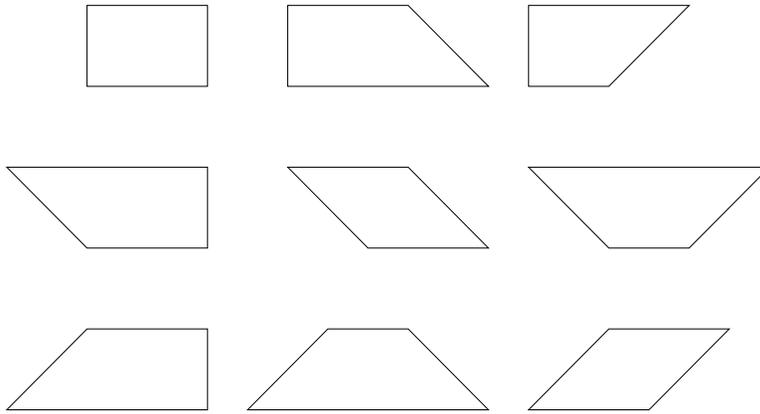


FIGURE 52.7: The different types of trapezoidal tiles.

52.5.3 Curved Tiles

Séquin and Façanha [12] proposed two generalizations to geometries including circles and arbitrary curved shapes, which arise in microelectromechanical systems (MEMS). As with its corner stitching-based predecessors, the layout area is decomposed in a horizontally maximal fashion into tiles. Consequently, tiles have upper and lower horizontal sides. Their left and right sides are represented by parameterized cubic Bezier curves or by composite paths composed of linear, circular, and spline segments. Strategies for reducing storage by minimizing the number of tiles and curve-sharing among tiles are discussed.

52.5.4 L-shaped Tiles

Mehta and Blust [13] extended Ousterhout's corner stitching data structure to directly represent L- and other simple rectilinear shapes without partitioning them into rectangles. This results in a data structure that is topologically different from the other versions of corner stitching described above. A key property of this L-shaped corner stitching (LCS) data structure is that

1. All vacant tiles are either rectangles or L- shapes.
2. No two vacant tiles in the layout can be merged to form a vacant rectangle or L-shaped tile.

Figure 52.8 shows three possible configurations for the same set of solid tiles.

There are four L-shape types (Figure 52.9), one for each orientation of the L-shape. The L-shapes are numbered according to the quadrant represented by the two lines meeting at the single concave corner of the L-shape. Figure 52.10 describes the contents of L-shapes and rectangles in LCS and rectangles in the original rectangular corner stitching (RCS) data structure. The actual memory requirements of a node in bytes (last column of the table) are obtained by assuming that pointers and coordinates, each, require 4 bytes of storage, and by placing all the remaining bits into a single 4-byte word. Note that the space required by any L-shape is less than the space required by two rectangles in RCS and that the space required by a rectangle in LCS is equal to the space required by a rectangle in RCS. The following theorem has been proved in [9]:

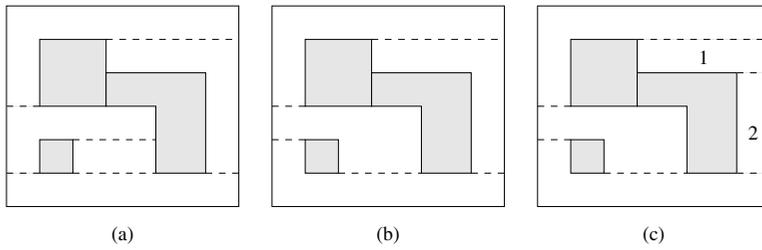


FIGURE 52.8: Layouts represented by the L-shaped corner stitching data structure: Layout (c) is invalid because the vacant tiles 1 and 2 can be merged to form an L-shaped tile. Layouts (b) and (c) are both valid, illustrating that unlike RCS, the LCS data structure does not give a unique partition for a given set of solid tiles.

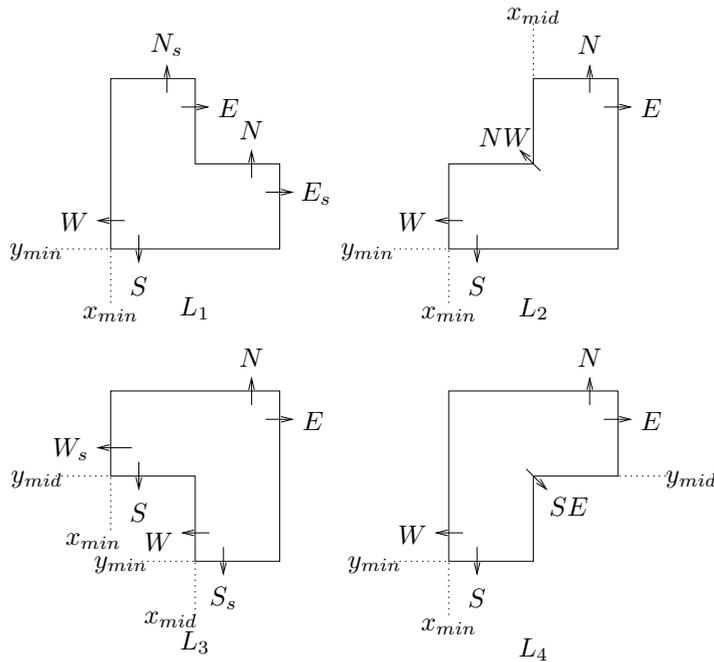


FIGURE 52.9: L-shapes and their pointers.

Tile type	Number of Coordinates	Number of Pointers	Number of N/E bits	solid/vacant tile bit	R/L bit	L type bits	Number of Bytes
L_1	2	6	4	1	1	2	36
L_2	3	5	3	1	1	2	36
L_3	4	6	2	1	1	2	44
L_4	3	5	3	1	1	2	36
R in LCS	2	4	2	1	1	0	28
R in RCS	2	4	0	1	0	0	28

FIGURE 52.10: Space requirements of tiles in LCS and RCS.

THEOREM 52.1 *The LCS data structure never requires more memory than the RCS data structure for the same set of solid rectangular tiles.*

Proof Since all solid tiles are rectangles, and since a rectangle occupies 28 bytes in both LCS and RCS, the total memory required to store solid tiles is the same in each data structure. From the definitions of RCS and LCS, there is a one-to-one correspondence between the set S_1 of vacant rectangles in RCS and the set S_2 consisting of (i) vacant rectangles in LCS and (ii) rectangles obtained by using a horizontal line to split each vacant L-shape in LCS; where each pair of related rectangles (one from S_1 , the other from S_2) have identical dimensions and positions in the layout. Vacant rectangles in LCS require the same memory as the corresponding vacant rectangles in RCS. However, a vacant L-shape requires less memory than the two corresponding rectangles in RCS. Therefore, if there is at least one vacant L-shape, LCS requires less memory than RCS.

THEOREM 52.2 *The LCS data structure requires 8.03 to 26.7 % less memory than the RCS data structure for a set of solid, rectangular tiles that satisfies the CV property.*

Rectilinear Polygons

Since, in practice, circuit components can be arbitrary rectilinear polygons, it is necessary to partition them into rectangles to enable them to be stored in the corner stitching format. MAGIC handles this by using horizontal lines to partition the polygons. This is not necessary from a theoretical standpoint, but it simplifies the implementation of the various corner stitching operations. Nahar and Sahni [15] studied this problem and presented an $O(n + k_v \log k_v)$ algorithm to decompose a polygon with n vertices and k_v vertical inversions into rectangles using horizontal cuts only. (The number of *vertical inversions* of a polygon is defined as the minimum number of changes in vertical direction during a walk around the polygon divided by 2.) The quantity k_v was observed to be small for polygons encountered in VLSI mask data. Consequently, the Nahar-Sahni algorithm outperforms an $O(n \log n)$ planesweep algorithm on VLSI mask data. We note that this problem is different from the problem of decomposing a rectilinear polygon into a minimum number of rectangles using *both* horizontal and vertical cuts, which has been studied extensively in the literature [16–19].

However, the extension is slower than the original corner stitching, and also harder to implement. Lopez and Mehta [20] presented algorithms for the problem of optimally breaking an arbitrary rectilinear polygon into L-shapes using horizontal cuts.

Parallel Corner Stitching

Mehta and Wilson [21] have studied a parallel implementation of corner stitching. Their work focuses on two batched operations (batched insert and delete). Their approach results in a significant speed-up in run times for these operations.

Comments about Corner Stitching

1. Corner stitching requires rectangles to be non-overlapping. A single layer of the chip consists of non-overlapping rectangles, but all the layers taken together will consist of overlapping rectangles. So, an instance of the corner stitching data structure can only be used for a single layer. However, corner stitching can be used to store multiple layers in the following way: consider two layers A and B. Superimpose the two layers. This can be thought of as a single layer with four types of rectangles: vacant rectangles, type A rectangles, type B rectangles, and type AB rectangles. Unfortunately, this could greatly increase the number of rectangles to be stored. It also makes it harder to perform insertion and deletion operations. Thus, in MAGIC, the

layout is represented by a number of single-layer corner stitching instances and a few multiple-layer instances when the intersection between rectangles in different layers is meaningful; for example, transistors are formed by the intersection of poly and diffusion rectangles.

2. Corner stitching is difficult to implement. While the data structure itself is quite elegant, the author's experience is that its implementation requires one to consider a lot of details that are not considered in a high-level description. This is supported by the following remark attributed to John Ousterhout [12]:

Corner-stitching is pretty straightforward at a high level, but it can become much more complicated when you actually sit down to implement things, particularly if you want the implementation to run fast

3. The *Point Find* operation can be slow. For example, *Point Find* could visit all the tiles in the data structure resulting in an $O(n)$ time complexity. On the average, it requires $O(\sqrt{n})$ time. This is expensive when compared with the $O(\log n)$ complexity that may be possible by using a tree type data structure. From a practical standpoint, the slow speed may be tolerable in an interactive environment in which a user performs one operation at a time (e.g., a *Point Find* could be performed by a mouse button click). Here, a slight difference in response time might not be noticeable by the user. Furthermore, in an interactive environment, these operations might actually be fast because they take advantage of locality of reference (i.e., the high likelihood that two successive points being searched by a user are near each other in the layout). However, in batch mode, where a number of operations are performed without user involvement, one is more likely to experience the average case complexity (unless the order of operations is chosen carefully so as to take advantage of locality of reference). The difference in time between corner stitching and a faster logarithmic technique will be significant.
4. Corner stitching requires more memory to store vacant tiles.

52.6 Quad Trees and Variants

Quad trees have been considered in [Chapters 16](#) and [19](#). These chapters demonstrate that there are different flavors of quad-trees depending on the type of the data that are to be represented. For example, there are quad trees for regions, points, rectangles, and boundaries. In this chapter, we will be concerned with quad-trees for rectangles. We also note that [Chapter 18](#) describes several data structures that can be used to store rectangles. To the best of my knowledge, the use of these structures has not been reported in the VLSI design automation literature.

The underlying principle of the quad tree is to recursively subdivide the two-dimensional layout area into four “quads” until a stopping criterion is satisfied. The resulting structure is represented by a tree with a node corresponding to each quad, with the entire layout area represented by the root. A node contains children pointers to the four nodes corresponding the quads formed by the subdivision of the node's quad. Quads that are not further subdivided are represented by leaves in the quad tree.

Ideally, each rectangle is the sole occupant of a leaf node. In general, of course, a rectangle does not fit inside any leaf quad, but rather intersects two or more leaf quads. To state this differently, it may intersect one or more of the horizontal and vertical lines (called bisectors)

used to subdivide the layout region into quads. Three strategies have been considered in the literature as to where in the quad tree these rectangles should be stored. These strategies, which have given rise to a number of quad tree variants, are listed below and are illustrated in Figure 52.11.

1. **SMALLEST:** Store a rectangle in the *smallest quad (not necessarily a leaf quad) that contains it*. Such a quad is guaranteed to exist since each rectangle must be contained in the root quad.
2. **SINGLE:** Store a rectangle in precisely *one of the leaf quads that it intersects*.
3. **MULTIPLE:** Store a rectangle in *all of the leaf quads that it intersects*.

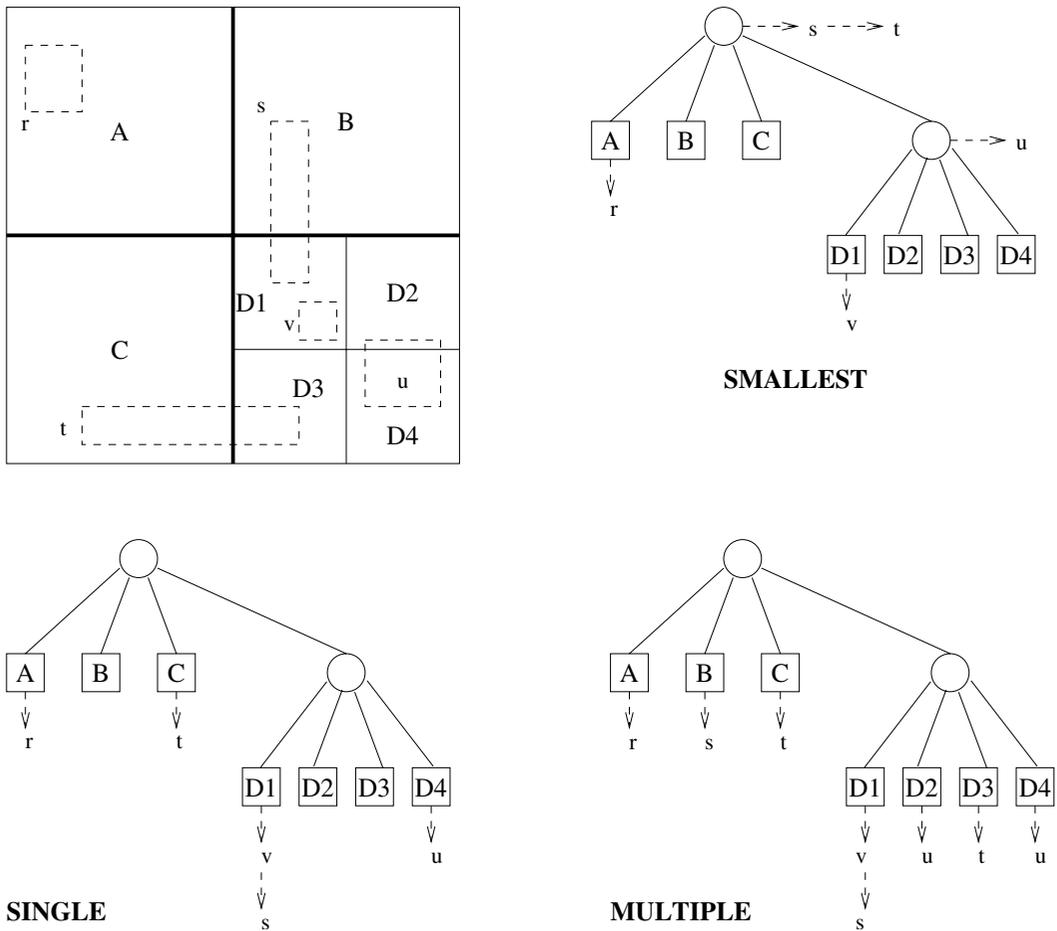


FIGURE 52.11: Quadtree variations.

Obviously, if there is only one rectangle in a quad, there is no need to further subdivide the quad. However, this is an impractical (and sometimes impossible) stopping criterion. Most of the quad tree variants discussed below have auxiliary stopping criteria. Some subdivide

a quad until it reaches a specified size related to the typical size of a small rectangle. Others stop if the number of rectangles in a quad is less than some threshold value. Figure 52.12 lists and classifies the quad tree variants.

Author	Abbreviation	Year of Publication	Strategy
Kedem	BLQT	1982	SMALLEST
Rosenberg	k -d	1985	N/A
Brown	MSQT	1986	MULTIPLE
Weyten et al	QLQT	1989	MULTIPLE
Pitaksanonkul et al	BQT	1989	SINGLE
Lai et al	HV	1993	SMALLEST
Lai et al	HQT	1996	MULTIPLE

FIGURE 52.12: Summary of Quad-tree variants.

52.6.1 Bisector List Quad Trees

Bisector List Quad Trees (BLQT) [22], which was the first quad-tree structure proposed for VLSI layouts, used the SMALLEST strategy. Here, a rectangle is associated with the smallest quad (leaf or non-leaf) that contains it. Any non-leaf quad Q is subdivided into four quads by a vertical bisector and a horizontal bisector. Any rectangle associated with this quad *must* intersect one or both of the bisectors (otherwise, it is contained in one of Q 's children, and should not be associated with Q). The set of rectangles are partitioned into two sets: V , which consists of rectangles that intersect the vertical bisector and H , which consists of rectangles that intersect the horizontal bisector. Rectangles that intersect both bisectors are arbitrarily assigned to one of V and H . These “lists” were actually implemented using binary trees. The rationale was that since most rectangles in IC layouts were small and uniformly distributed, most rectangles will be at leaf quads. A region search operation identifies all the quads that intersect a query window and checks all the rectangles in each of these quads for intersection with the query window. The BLQT (which is also called the MX-CIF quadtree) is also described in [Chapter 16](#).

52.6.2 k -d Trees

Rosenberg [23] compared BLQT with k -d trees and showed experimentally that k -d trees outperformed an implementation of BLQT. Rosenberg's implementation of the BLQT differs from the original in that linked lists rather than binary trees were used to represent bisector lists. It is hard to evaluate the impact of this on the experimental results, which showed that point-find and region-search queries visit fewer nodes when the k -d tree is used instead of BLQT. The experiments also show that k -d trees consume about 60-80% more space than BLQTs.

52.6.3 Multiple Storage Quad Trees

In 1986, Brown proposed a variation [24] called Multiple Storage Quad Trees (MSQT). Each rectangle is stored in every leaf quad it intersects. (See the quad tree labeled “MULTIPLE” in [Figure 52.11](#).) An obvious disadvantage of this approach is that it results in wasted space.

This is partly remedied by only storing a rectangle once and having all of the leaf quads that it intersects contain a pointer to the rectangle. Another problem with this approach is that queries such as Region Search may report the same rectangle more than once. This is addressed by marking a rectangle when it is reported for the first time and by not reporting rectangles that have been previously marked. At the end of the Region Search operation, all marked rectangles need to be unmarked in preparation for the next query. Experiments on VLSI mask data were used to evaluate MSQT for different threshold values and for different Region Search queries. A large threshold value results in longer lists of pointers in the leaf quads that have to be searched. On the other hand, a small threshold value results in a quad-tree with greater height and more leaf nodes as quads have to be subdivided more before they meet the stopping criterion. Consequently, a rectangle now intersects and must be pointed at by more leaf nodes. A Region Search query with a small query rectangle (window) benefits from a smaller threshold because it has to search smaller lists in a handful of leaf quads. A large window benefits from a higher threshold value because it has to search fewer quads and encounters fewer duplicates.

52.6.4 Quad List Quad Trees

In 1989, Weyten and De Pauw [25] proposed a more efficient implementation of MSQT called Quad List Quad Trees (QLQT). For Region Searches, experiments on VLSI data showed speedups ranging from 1.85-4.92 over MSQT, depending on the size of the window. In QLQT, four different lists (numbered 0-3) are associated with each leaf node. If a rectangle intersects the leaf quad, a pointer to it is stored in one of the four lists. The choice of the list is determined by the relative position of this rectangle with respect to the quad. The relative position is encoded by a pair of bits xy . x is 0 if the rectangle does not cross the lower boundary of the leaf quad and is 1, otherwise. Similarly, y is 0 if the rectangle does not cross the left boundary of the leaf quad and is 1, otherwise. The rectangle is stored in the list corresponding to the integer represented by this two bit string. Figure 52.13 illustrates the concept. Notice that each rectangle belongs to exactly one list 0. This corresponds to the quad that contains the bottom left corner of the rectangle. Observe, also, that the combination of the four lists in a leaf quad gives the same pointers as the single list in the same leaf in MSQT. The Region Search of MSQT can now be improved for QLQT by using the following procedure for each quad that intersects the query window: if the query window's left edge crosses the quad, only the quad's lists 0 and 1 need to be searched. If the window's bottom edge crosses the quad, the quad's lists 0 and 2 need to be searched. If the windows bottom left corner belongs to the quad, all four lists must be searched. For all other quads, only list 0 must be searched. Thus the advantages of the QLQT over MSQT are:

1. QLQT has to examine fewer list nodes than MSQT for a Region Search query.
2. Unlike MSQT, QLQT does not require marking and unmarking procedures to identify duplicates.

52.6.5 Bounded Quad Trees

Later, in 1989, Pitaksanonkul et al. proposed a variation of quad trees [26] that we refer to as Bounded Quad Trees (BQT). Here, a rectangle is only stored in the quad that contains its bottom left corner. (See the quad tree labeled "SINGLE" in Figure 52.11.) This may be viewed as a version of QLQT that only uses list 0. Experimental comparisons with k -d trees show that for small threshold values, quad trees search fewer nodes than k -d trees.

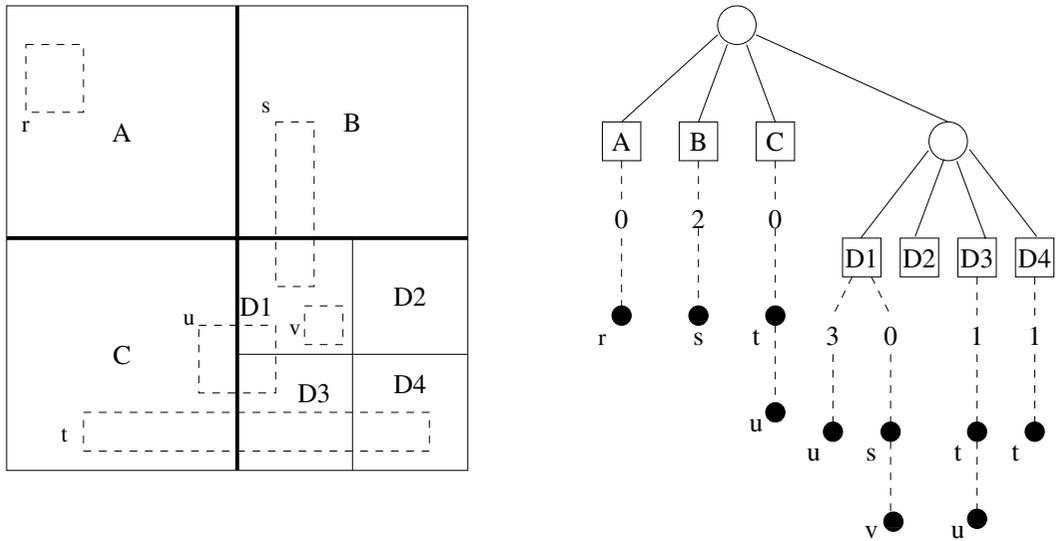


FIGURE 52.13: The leaf quads are A , B , C , $D1$, $D2$, $D3$, and $D4$. The rectangles are $r - v$. Rectangle t intersects quads C , $D3$, and $D4$ and must appear in the lists of each of the leaf nodes in the quad tree. Observe that t does not cross the lower boundaries of any of the three quads and $x = 0$ in each case. However, t does cross the left boundaries of $D3$ and $D4$ and $y = 1$ in these cases. Thus t goes into list 1 in $D3$ and $D4$. Since t does not cross the left boundary of C , it goes into list 0 in C . Note that the filled circles represent pointers to the rectangles rather than the rectangles themselves.

52.6.6 HV Trees

Next, in 1993, Lai et al. [27] presented a variation that once again uses bisector lists. It overcomes some of the inefficiencies of the original BLQT by a tighter implementation: An HV Tree consists of alternate levels of H-nodes and V-nodes. An H node splits the space assigned to it into two halves with a horizontal bisector while a V-node does the same by using a vertical bisector. A node is not split if the number of rectangles assigned to it is less than some fixed threshold.

Rectangles intersecting an H node's horizontal bisector are stored in the node's bisector list. Bisector lists are implemented using cut trees. A vertical cutline divides the horizontal bisector into two halves. All rectangles that intersect this vertical cutline are stored in the root of the cut tree. All rectangles to the left of the cutline are recursively stored in the left subtree and all rectangles to the right are recursively stored in the right subtree. So far, the data structure is identical to Kedem's binary tree implementation of the bisector list. In addition to maintaining a list of rectangles intersecting a vertical cutline at the corresponding node n , the HV tree also maintains four additional bounds which significantly improve performance of the Region Search operation. The bounds y_{upper_bound} and y_{lower_bound} are the maximum and minimum y coordinates of any of the rectangles stored in n or in any of n 's descendants. The bounds x_{lower_bound} and x_{upper_bound} are the minimum and maximum x coordinates of the rectangles stored in node n . Figure 52.14 illustrates these concepts. Comprehensive experimental results comparing HVT with BQT, kD, and QLQT showed that the data structures ordered from best to worst in terms of space requirements were HVT, BQT, kD, and QLQT. In terms of speed, the best data structures were HVT and QLQT followed by BQT and finally kD.

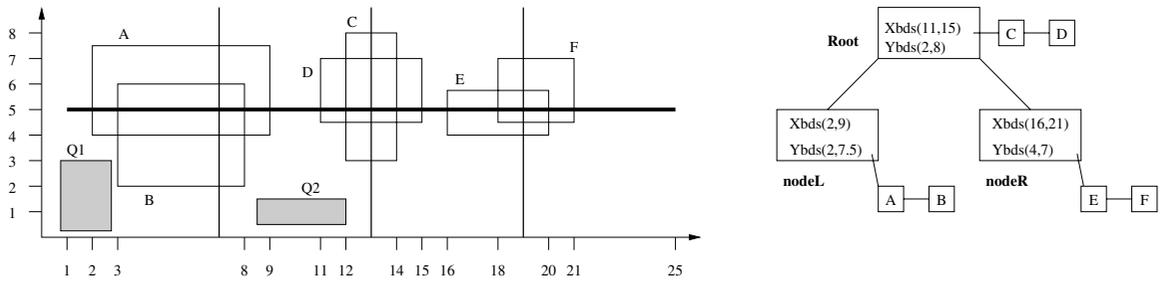


FIGURE 52.14: Bisector list implementation in HVT. All rectangles intersect the thick horizontal bisector line ($y = 5$). The first vertical cutline at $x = 13$ corresponding to the root of the tree intersects rectangles C and D . These rectangles are stored in a linked list at the root. Rectangles A and B are to the left of the vertical cutline and are stored in the left subtree. Similarly, rectangles C and D are stored in the right subtree. The X bounds associated with the root node are obtained by examining the x coordinates of rectangles C and D , while its Y bounds are obtained by examining the y coordinates of all six rectangles stored in the tree. The two shaded rectangles are query rectangles. For $Q1$, the search will start at **Root**, but will not search the linked list with C and D because $Q1$'s right side is to the left of **Root**'s lower x bound. The search will then examine **nodeL**, but not **nodeR**. For $Q2$, the search will avoid searching the bisector list entirely because its upper side is below **Root**'s lower y bound.

52.6.7 Hinted Quad Trees

In 1997, Lai et al. [28] described a variation of the QLQT that was specifically designed for design rule checking. Design-rule checking requires one to check rectangles in the vicinity of the query rectangle for possible violations. Previously, this was achieved by employing a traditional region query whose rectangle was the original query rectangle extended in all directions by a specified amount. Region searches start at the root of the tree and proceed down the tree as discussed previously. The hinted quadtree is based on the philosophy that it is wasteful to begin searching at the root, when, with an appropriate hint, the algorithm can start the search lower down in the tree. Two questions arise here: at which node should the search begin and how does the algorithm get to that node? The node at which the design rule check for rectangle r begins is called the *owner* of r . This is defined as the lowest node in the quad-tree that completely contains r expanded in all four directions. Since the type of r is known (e.g., whether it is n-type diffusion or metal), the amount by which r has to be expanded is also known in advance. Clearly, any rectangle that intersects the expanded r must be referenced by at least one leaf in the owner node's subtree. The owner node may be reached by following parent pointers from the rectangle. However, this could be expensive. Consequently, in HQT, each rectangle maintains a pointer to the owner virtually eliminating the cost of getting to that node. Although this is the main contribution of the HQT, there are additional implementation improvements over the underlying QLQT that are used to speed up the data structure. First, the HQT resolves the situation where the boundary of a rectangle stored in the data structure or a query rectangle coincides with that of a quad. Second, HQT sorts the four lists of rectangles stored in each leaf node with one of their x or y coordinates as keys. This reduces the search time at the leaves and consequently makes it possible to use a higher threshold than that used in QLQT. Experimental results showed that HQT out-performs QLQT, BQT, HVT, and k -d on neighbor-search queries by at least 20%. However, its build-time and space requirements were not as good as some of the other data structures.

52.7 Concluding Remarks

Most of the research in VLSI layout data structures was carried out in the early 80s through the mid-90s. This area has not been very active since then. One reason is that there continue to be several important challenges in VLSI physical design automation that must be addressed quickly so that physical design does not become a bottleneck to the continuing advances in semiconductors predicted by Moore's Law. These challenges require physical design tools to consider "deep sub-micron effects" into account and take priority over data structure improvements. The implication of these effects is that problems in VLSI physical design are no longer purely geometric, but rather need to merge geometry with electronics. For example, in the early 1980s, one could safely use the length of a wire to estimate the delay of a signal traveling along it. This is no longer true for the majority of designs. Thus, the delay of a signal along the wire must now be estimated by factoring in its resistance and capacitance, in addition to its geometry. On the other hand, the more detailed computations of electronic quantities like capacitance, resistance, and inductance require the underlying data structures to support more complex operations. Thus, the author believes that there remain opportunities for developing better layout data structures. Another area that, to the best of our knowledge, has been neglected in the academic literature is that VLSI layout data needs to be efficiently stored in secondary storage because of its sheer size. Thus, although there is potential for academic research in this area, we believe that the design automation industry may well have addressed these issues in their proprietary software.

Unlike corner stitching, quad-trees permit rectangles to overlap. This addresses the problem that corner stitching has with handling multiple layers. On the other hand, corner stitching was designed for use in the context of an interactive layout editor, whereas quad trees are designed in the context of batched operations. We note that, to our knowledge, relatively recent data structures such as R trees have not been used for VLSI layout. It is unclear what the outcome of this comparison will be. On one hand, R-trees are considered to be faster than quad trees. On the other, the quad-trees developed in VLSI have been improved and optimized significantly as we have seen in this chapter. Also, VLSI data has generally been considered to consist of uniformly small rectangles, which may make it particularly suitable for quad trees, although this property may not be true when wires are considered.

Acknowledgment

This work was supported, in part, by the National Science Foundation under grant CCR-9988338.

References

- [1] N. Sherwani, *Algorithms for VLSI Physical Design Automation*. Boston: Kluwer Academic Publishers, 1992.
- [2] M. Sarrafzadeh, C. K. Wong, *An Introduction to VLSI Physical Design*. New York: McGraw Hill, 1996.
- [3] S. Sait, H. Youssef, *VLSI Physical Design Automation: theory and practice*. Piscataway, NJ: IEEE Press, 1995.
- [4] N. H. E. Weste, K. Eshraghian, *Principles of CMOS VLSI Design: A Systems Perspective, Second Edition*. New York: Addison Wesley, 1993.
- [5] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach, Third Edition*. New York: Morgan Kaufmann, 2003.

- [6] D. G. Boyer, "Symbolic Layout Compaction Review," *Proceedings of 25th Design Automation Conference*, pages 383-389, 1988.
- [7] J. Ousterhout, G. Hamachi, R. Mayo, W. Scott, G. Taylor, "Magic: A VLSI layout system," in *Proc. of 21st Design Automation Conf.*, pp. 152-159, 1984.
- [8] J. K. Ousterhout, "Corner Stitching: A data structuring technique for VLSI layout tools," *IEEE Transactions on Computer-aided Design*, vol. 3, no. 1, pp. 87-100, 1984.
- [9] D. P. Mehta "Estimating the Memory Requirements of the Rectangular and L-shaped Corner Stitching Data Structures," *ACM Transactions on the Design Automation of Electronic Systems*. Vol. 3, No. 2, April 1998.
- [10] M. Quayle, J. Solworth, "Expanded Rectangles: A New VLSI Data Structure," in *ICCAD*, pp. 538-541, 1988.
- [11] D. Marple, M. Smulders, and H. Hegen, "Tailor: A Layout System Based on Trapezoidal Corner Stitching," *IEEE Transactions on Computer-aided Design*, vol. 9, no. 1, pp. 66-90, 1990.
- [12] C. H. Séquin and H. da Silva Façanha, "Corner Stitched Tiles with Curved Boundaries," *IEEE Transactions on Computer-aided Design*, vol. 12, no. 1, pp. 47-58, 1993.
- [13] D. P. Mehta, G. Blust, "Corner Stitching for Simple Rectilinear Shapes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 186-198, Feb. 1997.
- [14] D. P. Mehta "CLOTH_MEASURE: A Software Tool for Estimating the Memory Requirements of Corner Stitching Data Structures," *VLSI Design*, Vol. 7, No. 4, pp 425-436, 1998.
- [15] S. Nahar and S. Sahni, "A Fast Algorithm for Polygon Decomposition," *IEEE Transactions on Computer-aided Design*, vol. 7, pp. 478-483, Apr. 1988.
- [16] T. Ohtsuki, "Minimum Dissection of Rectilinear Regions," in *Proceedings 1982 International Symposium on Circuits and Systems (ISCAS)*, pp. 1210-1213, 1982.
- [17] H. Imai, T. Asano, "Efficient Algorithms for Geometric Graph Search Algorithms," *SIAM Journal on Computing*, vol. 15, pp. 478-494, May 1986.
- [18] W. T. Liou, J. J. M. Tan, and R. C. T. Lee, "Minimum Partitioning of Simple Rectilinear Polygons in $O(n \log \log n)$ time," in *Proceedings of the Fifth Annual Symposium on Computational Geometry*, pp. 344-353, 1989.
- [19] S. Wu and S. Sahni, "Fast Algorithms to Partition Simple Rectilinear Polygons," *International Journal on Computer Aided VLSI Design*, vol. 3, pp. 241-270, 1991.
- [20] M. Lopez, D. Mehta, "Efficient Decomposition of Polygons into L-shapes with Applications to VLSI Layouts," *ACM Transactions on Design Automation of Electronic Systems*, vol. 1, pp. 371-395, 1996.
- [21] D. Mehta, E. Wilson, "Parallel Algorithms for Corner Stitching," *Concurrency: Practice and Experience*, vol. 10, pp. 1317-1341, 1998.
- [22] G. Kedem, "The quad-CIF tree: A data structure for hierarchical on-line algorithms," in *Proceedings of the 19th Design Automation Conference*, pp. 352-357, 1982.
- [23] J. B. Rosenberg, "Geographical Data Structures Compared: A Study of Data Structures supporting Region Queries," *IEEE Transactions on Computer-Aided Design*, vol. 4, no. 1, pp. 53-67, 1985.
- [24] R. L. Brown, "Multiple Storage Quad Trees: A Simpler Faster Alternative to Bisector List Quad Trees," *IEEE Transactions on Computer-Aided Design*, vol. 5, no. 3, pp. 413-419, 1986.
- [25] L. Weyten, W. de Pauw, "Quad List Quad Trees: A Geometric Data Structure with Improved Performance for Large Region Queries," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 3, pp. 229-233, 1989.
- [26] A. Pitaksanonkul, S. Thanawastien, C. Lursinsap, "Comparison of Quad Trees and 4-D Trees: New Results," *IEEE Transactions on Computer-Aided Design*, vol. 8, no. 11,

pp. 1157–1164, 1989.

- [27] G. Lai, D. S. Fussell, D. F. Wong, “HV/VH Trees: A New Spatial Data Structure for Fast region Queries,” in *Proceedings of the 30th Design Automation Conference*, pp. 43–47, 1993.
- [28] G. Lai, D. S. Fussell, D. F. Wong, “Hinted Quad Trees for VLSI Geometry DRC Based on Efficient Searching for Neighbors,” *IEEE Transactions on Computer-Aided Design*, vol. 15, no. 3, pp. 317–324, 1996.

Floorplan Representation in VLSI

	53.1	Introduction.....	53-1
		Statement of Floorplanning Problem • Motivation of the Representation • Combinations and Complexities of the Various Representations • Slicing, Mosaic, LB Compact, and General Floorplans	
	53.2	Graph Based Representations	53-8
		Constraint Graphs • Corner Stitching • Twin Binary Tree • Single Tree Representations	
	53.3	Placement Based Representations	53-16
		Sequence-Pair • Bounded-Sliceline Grid • Corner Block List • Slicing Tree	
	53.4	Relationships of the Representations	53-24
		Summary of the Relationships • A Mosaic Floorplan Example • A General Floorplan Example	
	53.5	Rectilinear Shape Handling	53-27
	53.6	Conclusions	53-28
	53.7	Acknowledgment	53-28

Zhou Feng
Fudan University

Bo Yao
University of California, San Diego

Chung-Kuan Cheng
University of California, San Diego

53.1 Introduction

There are two main data models that can be used for representing floorplans: graph-based and placement based.

The graph-based approach includes constraint graphs, corner stitching, twin binary tree, and O-tree. They utilize constraint graphs or their simplified versions directly for the encoding. Constraint graphs are basic representations. The corner stitching simplifies the constraint graph by recording only the four neighboring blocks to each block. The twin binary tree then reduces the recorded information to only two neighbors of each block, and organizes the neighborhood relations in a pair of binary trees. The O-tree is a further simplification to the twin binary tree. It keeps only one tree for encoding.

The placement-based representations use the relative positions between blocks in a placement for encoding. This category includes sequence pair, bounded-sliceline grid, corner block list and slicing trees. The sequence pair and bounded-sliceline grid can be applied to general floorplan. The corner block list records only the relative position of adjacent blocks, and is available to mosaic floorplan only. The slicing trees are for slicing floorplan, which is a type of mosaic floorplan. The slicing floorplan can be constructed by hierarchical horizontal or vertical merges and thus can be captured by a binary tree structure known as the slicing tree.

The rest of this chapter is organized as follows. In Section 53.1, we give the introduction and the problem statement of floorplanning. In Section 53.2, we discuss the graph-based

representations. In Section 53.3, we introduce the placement-based representations. We describe the relationship between different representations in Section 53.4. We illustrate the shape handling of rectilinear blocks in Section 53.5 and summarize the chapter in Section 53.6.

53.1.1 Statement of Floorplanning Problem

Today's complexity in circuitry design wants a hierarchical approach [26]. The entire circuit is partitioned into several sub-circuits, and the sub-circuits are further partitioned into smaller sub-circuits, until they are small enough to be handled. The relationship between the sub-circuits can be represented with a tree as shown in Fig. 53.1. Here every sub-circuit is called a block, and hence the entire circuit is called the top block. From the layout point of view, every block corresponds to a rectangle, which contains sub-blocks or directly standard cells. Among the decisions to be made is the determination of shape (aspect ratio) and the pin positions on the blocks. In the top-down hierarchical methodology, blocks are designed from the top block (the entire circuit) to the leaf blocks (small modules). The minimizations of chip area and wire length are the basic targets for any layout algorithm. In addition, there are case-dependent constraints that will influence the layouts, such as the performance, the upper or/and lower boundaries of aspect ratio, the directions of pins, etc.

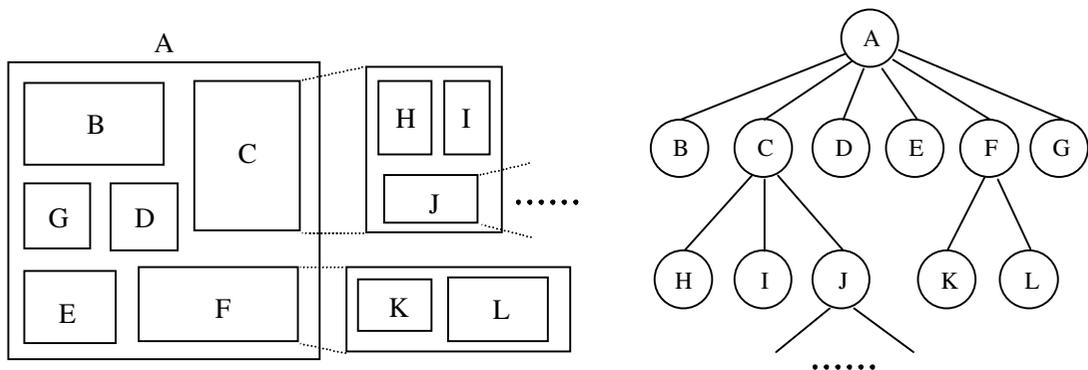


FIGURE 53.1: Hierarchical structure of blocks.

Now we give the definition of floorplanning problem:

Inputs:

1. the net-lists of the sub-circuits;
2. the area estimation of blocks and, if any, the aspect ratio constraints on the blocks;
3. the target area and shape of the entire chip.

Outputs:

1. the shapes and positions of blocks;
2. the pin positions on the blocks.

The objective functions involve: the chip area, the total wire length and, if any, the performances.

53.1.2 Motivation of the Representation

Floorplan representation becomes an important issue in floorplanning for the following reasons.

(1) The blocks may have arbitrary shapes and locations, while the size of memory used to represent a two-dimensional floorplan should be $O(n)$.

(2) For the general floorplanning problem, iterative improvement is the commonly used approach. The search for the best solution has proven to be NP-complete, so many heuristic optimizing algorithms, such as dynamic programming, simulated annealing, zone refinement, cluster refinement, have been adopted. The representation should also facilitate the operations called by those optimizing algorithms.

(3) The storage resources requirement, the redundancy of the representation and the complexity of translating the representation into floorplan are always the concerns in floorplanning. Here redundancy refers to the situation that several different expressions actually correspond to the same physical layout. Essentially, a heuristic algorithm searches part of the solution space to find the local optimal solution, which is hopefully very close to the global optimal solution. Redundancy causes the optimizing algorithm evaluate the same floorplan repeatedly.

53.1.3 Combinations and Complexities of the Various Representations

The number of possible floorplan representations describes how large the searching space is. It also discloses the redundancy of the representation. For general floorplans with n blocks, the combinations of the various representations are listed in Table 53.1. Note that the twin binary tree representation has a one to one relation with the mosaic floorplan, and the slicing tree has a one to one relation with the slicing floorplan [15]. In other words, for these two representations, the number of combinations is equal to the number of possible floorplan configurations and there is no redundancy.

TABLE 53.1 Combinations of the representations.

Representation	Combinations
Twin binary tree	$n! * B(n)$, where $B(n) = \binom{n+1}{1}^{-1} \binom{n+1}{2}^{-1} \sum_{k=1}^n \binom{n+1}{k-1} \binom{n+1}{k} \binom{n+1}{k+1}$
O-Tree	$O(n!2^{2n-2}/n^{1.5})$
Sequence pair	$(n!)^2$
Bounded-sliceline grid	$n! \binom{n^2}{n}$
Corner block list	$O(n!2^{3n-3}/n^{1.5})$
Slicing tree	$n! * A(n-1)$, where $A(n)$ is the super-Catalan number with the following definition: $A(0) = A(1) = 1$ and $A(n) = (3(2n-3)A(n-1) - (n-3)A(n-2))/n$

The combination numbers of sequence pairs, mosaic floorplans, slicing floorplans, and O-trees are illustrated on a log scale in Fig. 53.2. The combination numbers are normalized by $n!$, which is the number of permutations of n blocks. The slopes of the lines for mosaic floorplans, slicing floorplans, and O-tree structures are the constants 0.89, 0.76, and 0.59, respectively. On the other hand, the slope of the line for sequence pair increases with a rate of $\log n$. Table 53.2 provides the exact numbers of the combinations for the floorplans or representations with the block number ranging from 1 to 17.

The time complexities of the operations transforming a representation to a floorplan are

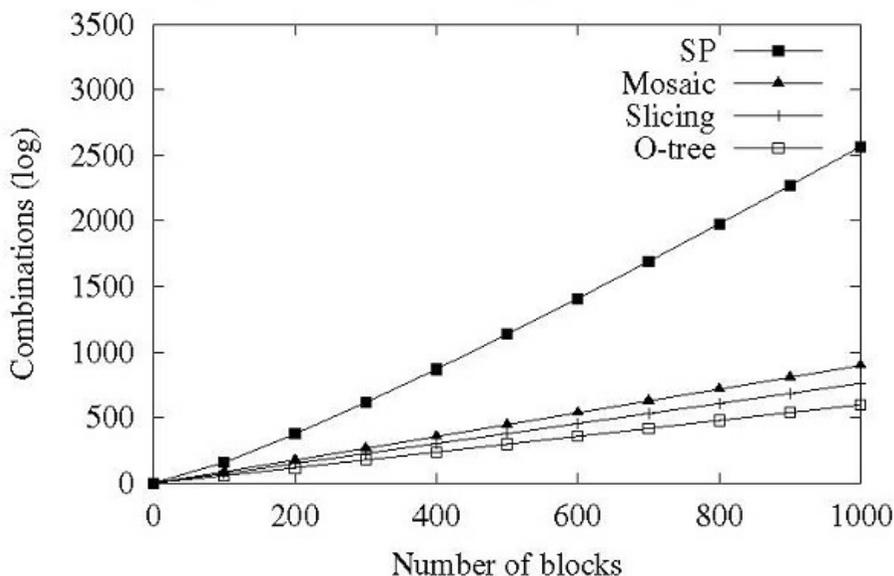


FIGURE 53.2: Combination of floorplans and representations.

TABLE 53.2 Exact number of combinations of different floorplan configurations and representations.

Number of blocks	Combinations of O-tree	Combinations of slicing floorplan	Combinations of mosaic floorplan	Combinations of sequence pairs
1	1	1	1	1
2	2	2	2	2
3	5	6	6	6
4	14	22	22	24
5	42	90	92	120
6	132	394	422	720
7	429	1,806	2,074	5,040
8	1,430	8,558	10,754	40,320
9	4,862	41,586	58,202	362,880
10	16,796	206,098	326,240	3,628,800
11	58,786	1,037,718	1,882,690	39,916,800
12	208,012	5,293,446	11,140,560	479,001,600
13	742,900	27,297,738	67,329,992	6,227,020,800
14	2,674,440	142,078,746	414,499,438	87,178,291,200
15	9,694,845	745,387,038	2,593,341,586	1,307,674,368,000
16	35,357,670	3,937,603,038	16,458,756,586	20,922,789,888,000
17	129,644,790	20,927,156,706	105,791,986,682	355,687,428,096,000

TABLE 53.3 Time Complexity comparison of the representations.

Representation	From a representation to a floorplan
Constraint graph	$O(n)$
Corner stitching	$O(n)$
Twin binary tree	$O(n)$
O-Tree	$O(n)$
Sequence pair	$O(n \log n)$
Bounded-sliceline grid	$O(n^2)$
Corner block list	$O(n)$
Slicing tree	$O(n)$

very important, because they determine the efficiency of the floorplan optimizations. The complexities of the representations covered in this chapter are compared in Table 53.3, where n is the number of blocks in a floorplan.

For sequence-pair, the time complexity to derive a floorplan is $O(n \log \log n)$ due to a fast algorithm proposed in [8]. We will discuss more on this in Section 53.3.1 For bounded slicing grid, there is a trade off between the solution space and the time complexity of deriving a floorplan. To ensure that the solution space covers all the optimal solutions, we need the grid size to be at least n by n . This results in an $O(n^2)$ time complexity in deriving a floorplan [6]. For the rest of the representations, there are algorithms with $O(n)$ time complexity to convert them into constraint graphs. The time complexity to derive a floorplan is thus $O(n)$.

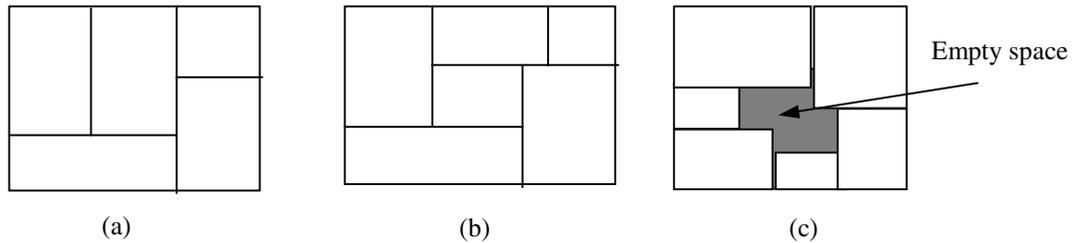


FIGURE 53.3: (a) Slicing floorplan; (b) Mosaic floorplan; (c) General floorplan.

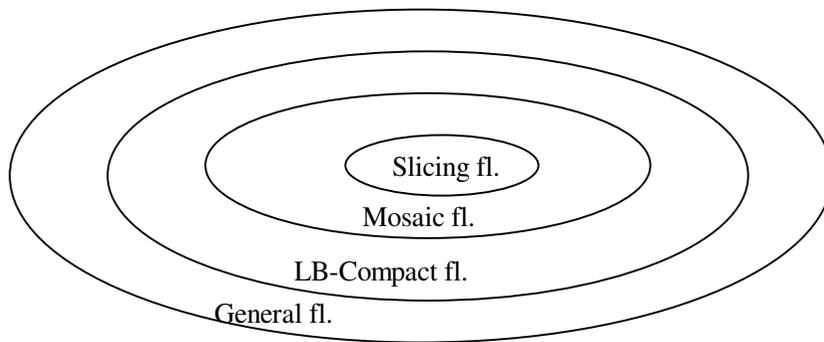


FIGURE 53.4: Set coverage of slicing floorplan, mosaic floorplans, LB-compact floorplans and general floorplans.

53.1.4 Slicing, Mosaic, LB Compact, and General Floorplans

A layout can be classified as a slicing floorplan if we can partition the chip with recursive horizontal or vertical cut lines (Fig. 53.3(a)). In a mosaic floorplan, the chip is partitioned by horizontal and vertical segments into rectangular regions and each region corresponds to exactly one block (Fig. 53.3(b)). For a general floorplan, we may find empty space outside rectangular block regions (Fig. 53.3(c)). An LB-compact floorplan is a restricted general floorplan in that all blocks are shifted to their left and bottom limits. In summary, the set

of general floorplans covers the set of LB-compact floorplans, which covers the set of mosaic floorplans, and which covers the set of slicing floorplans (Fig. 53.4).

For slicing and mosaic floorplans, the vertical segments define the left-right relation among the separated regions, and the horizontal segments define the above-below relation. Suppose that we shift the segments to change the sizes of the regions, we view the new floorplan to be equivalent to the original floorplan in terms of their topologies[4][23][24]. Therefore, we can devise representations to define the topologies of slicing and mosaic floorplans independent of the sizes of the blocks (Fig. 53.3(a) and (b)). In contrast, for a general floorplan, it is rather difficult to draw a meaningful layout (Fig. 53.3(c)) without the knowledge of the block dimensions. One approach is to extend the mosaic floorplans to general floorplans by adding empty blocks [16]. It is shown that to convert a mosaic floorplan with n blocks into a general floorplan, the upper bound on the number of empty blocks inserted is $n - 2\sqrt{n} + 1$. [16]

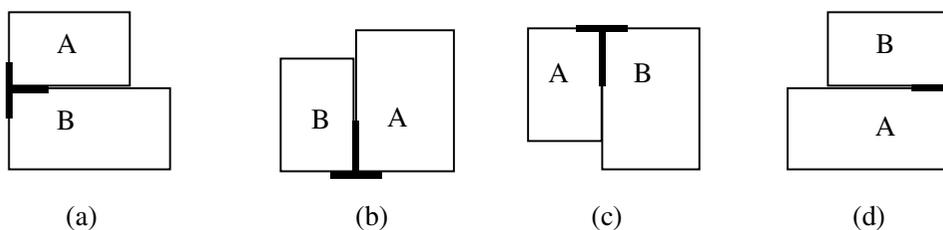


FIGURE 53.5: Four directions of T-junctions.

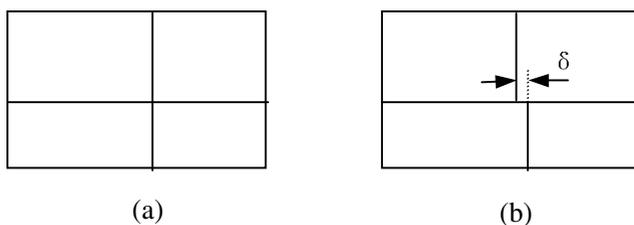


FIGURE 53.6: Degeneration.

In a mosaic floorplan, two adjacent blocks meet at a T-junction by sharing the non-crossing edge of the junction. There are four directions of T-junctions as is illustrated in Fig. 53.5. In the case of Fig. 53.5(a) and (b), block B is termed the C-neighbor at the lower left corner of block A. In Fig. 53.5(c) and (d), block B is the C-neighbor at the upper right corner of block A. The C-neighbor is used to construct twin binary tree representation.

The degeneration of a mosaic floorplan refers to the phenomenon that two T-junctions meet together to make up a cross-junction, as illustrated in Fig. 53.6(a). Some representations forbid the occurrence of degeneration. One scheme to solve the problem is to break one of the intersecting lines and assume a slight shift between the two segments, as shown in Fig. 53.6(b). Thus the degeneration disappears.

We generate an LB compact floorplan by compacting all blocks toward left and bottom. For a placement, suppose no block can be moved left, the placement is called L-compact. Similarly, if no block can be moved down, the placement is called B-compact. A floorplan is LB-compact if it is both L-compact and B-compact (Fig. 53.7).

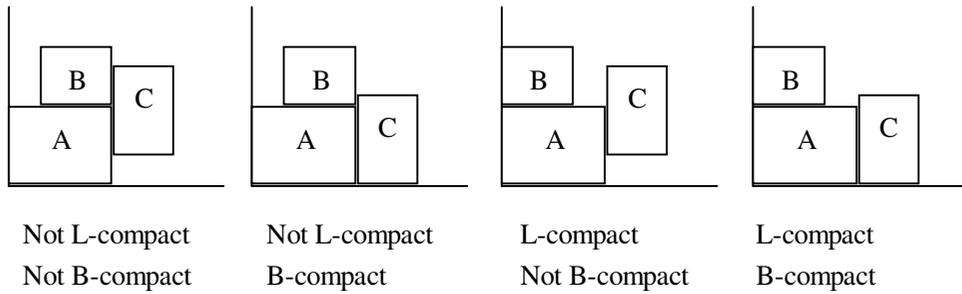


FIGURE 53.7: Examples of L-compact and B-compact.

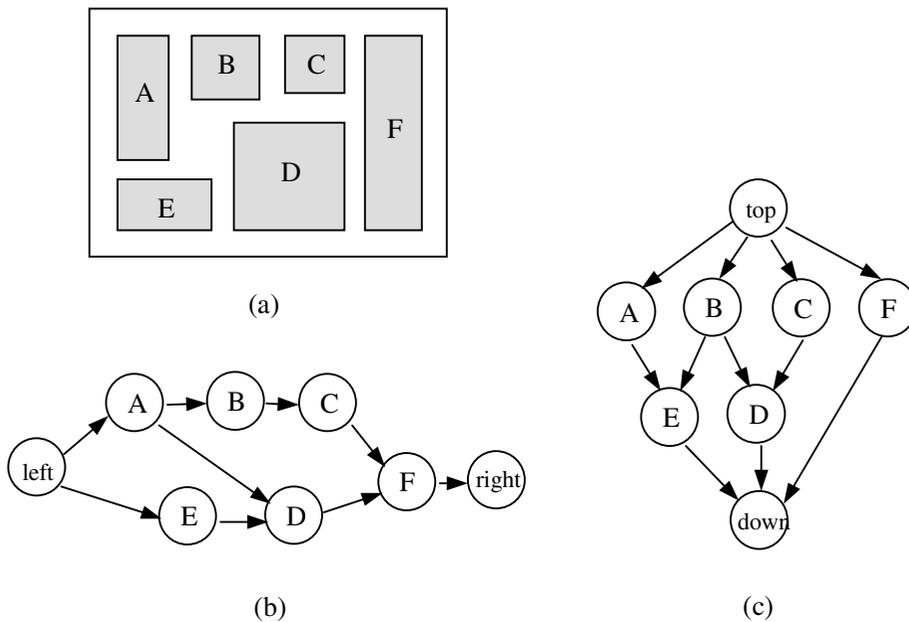


FIGURE 53.8: Constraint graphs for a general floorplan.

53.2 Graph Based Representations

Graph based representations include constraint graphs, corner stitching, twin binary trees, and O-tree. They all utilize the constraint graphs or their simplified version for floorplan encoding.

53.2.1 Constraint Graphs

Constraint graphs are directed acyclic graphs representing the adjacency relations between the blocks in a floorplan. In this subsection, we first define the constraint graphs for general floorplans. We then show that for mosaic floorplan, the constraint graphs have nice properties of triangulation and duality.

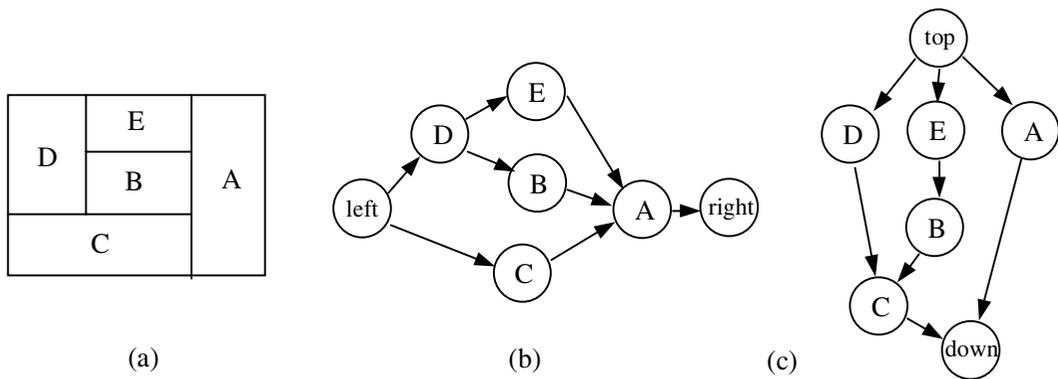


FIGURE 53.9: Constraint graphs for a mosaic floorplan.

The generation of constraint graphs

Constraint graphs reflect the relative positions between blocks [12]. Constraint graphs can be applied to general floorplans, as is shown in Fig. 53.8. A node in the constraint graph represents a block. A directed edge denotes the location relationship between two blocks. For example, $A \rightarrow B$ means block A is to the left of B in a horizontal constraint graph (Fig. 53.8(b)). $A \rightarrow E$ means block A is on top of E in a vertical constraint graph (Fig. 53.8(c)). Here we imply that if $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$. Thus even though block A stands to the left of C, the edge between A and C is not necessarily shown. To mark the four sides of the chip, we add the nodes labeled with “left”, “right”, “top” and “down”. A pair of horizontal and vertical constraints graphs can represent a floorplan. Every constraint graph, whether it is a horizontal one or a vertical one, is planar and acyclic. Fig. 53.9 shows an example of the constraint graphs to a mosaic floorplan. Fig 53.9(a) is the mosaic floorplan. Fig 53.9(b) and (c) are the corresponding horizontal and vertical constraint graphs, respectively.

Triangulation

For a mosaic floorplan without degeneration, if its horizontal and vertical constraint graphs are merged together, then we have the following conclusions [12]:

1. Every face is a triangle.
2. All internal nodes have a degree of at least 4.
3. All cycles, if not faces, have the length of at least 4.

Shown in Fig. 53.10(a) is the result of merging the pair of constraint graphs in Fig. 53.9. In fact, the merged constraint graph is a dual graph of its original floorplan (Fig. 53.9(b)).

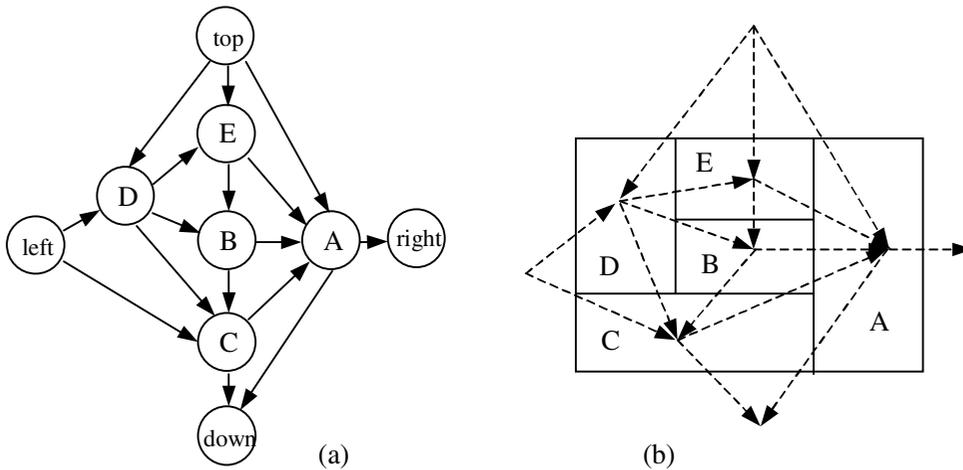


FIGURE 53.10: Triangulation.

Tutte’s duality[27]

We can also build an edge-based constraint graph for a mosaic floorplan, where the nodes denote the lines separating the blocks while the edges denote the blocks. Labeling the lines with numbers (Fig. 53.11(a)), we build a vertical constraint graph (Fig. 53.11 (b)) and a horizontal constraint graph (Fig. 53.11(c)). Fig. 53.11(d) demonstrates the result of merging the vertical and horizontal constraint graphs. Here, to make the merged graph clear, the edges representing horizontal constraints are drawn with dotted lines, and a letter at the intersection of a solid edge and a dotted edge denotes the two edges simultaneously. It is very interesting that, for mosaic floorplans, the vertical and horizontal constraint graphs are dual, as is called Tutte’s duality.

Let’s see how Tutte’s duality is used to solve the sizing problem in floorplanning. We map the constraint graphs into circuit diagrams by replacing the edges in the vertical constraint graph with resistors, as illustrated in Fig. 53.12.

The circuit is subject to Kirchoff voltage law. As a result, taking Fig. 53.12 as an

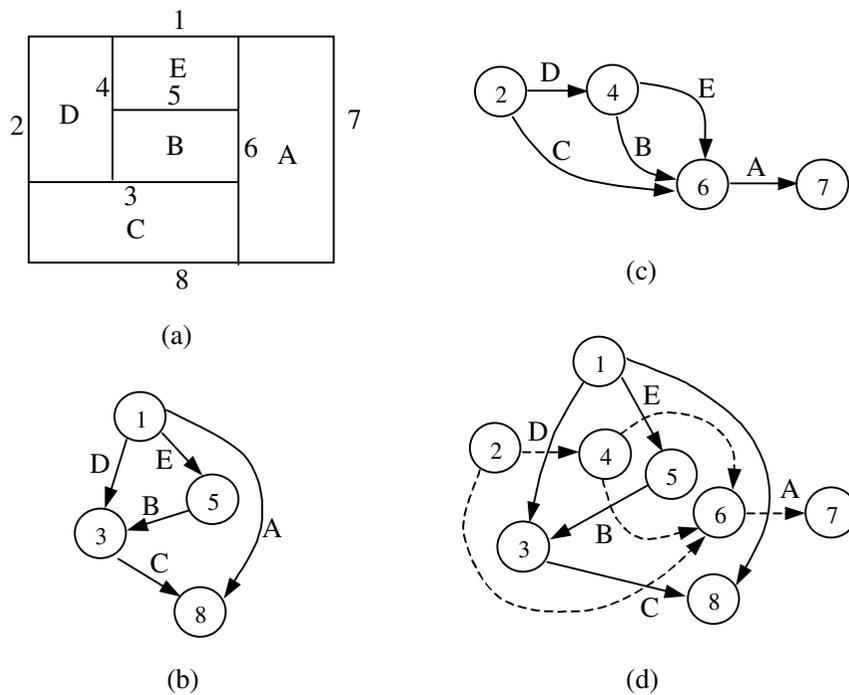


FIGURE 53.11: Line-based constraint graphs.

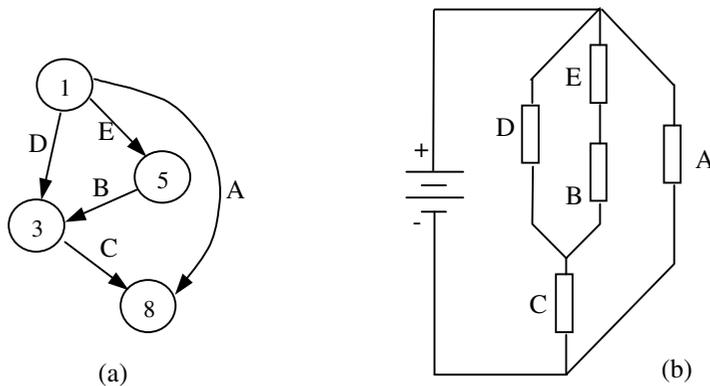


FIGURE 53.12: Mapping a constraint graph to a circuit.

example, we have:

$$\begin{aligned}
 V_E + V_B &= V_D & I_E &= I_B \\
 V_D + V_C &= V_A & I_E + I_D &= I_C \\
 V_A &= V_{source} & I_C + I_A &= I_{source} \\
 R_i &= \frac{V_i}{I_i}, i \in \{A, B, C, D, E\}
 \end{aligned}$$

Note that, if we denote the height and the width of block i to be H_i and W_i , we have (refer

to Fig. 53.12(a)):

$$\begin{aligned} H_E + H_B &= H_D & W_E &= W_B \\ H_D + H_C &= H_A & W_E + W_D &= W_C \\ H_A &= H_{chip} & W_C + W_A &= W_{chip} \end{aligned}$$

$$AspectRatio_i = \frac{H_i}{W_i}, i \in \{A, B, C, D, E\}$$

By comparing the above two equation arrays, we can find that there is a perfect correspondence between the solutions of the circuit and the floorplan. Let us set the following assumptions: (only two of the three equations are independent)

$$\begin{aligned} R_i &= AspectRatio_i \\ V_{source} &= H_{chip} \\ I_{source} &= W_{chip} \end{aligned}$$

Then we have:

$$\begin{aligned} V_i &= H_i \\ I_i &= W_i \end{aligned}$$

Thus the theories dealing with large circuits can be borrowed to solve the sizing problem in floorplanning.

Constraint graphs have the advantages of being able to cope with any types of floorplans. However, it would be rather difficult to shift to a new floorplan with just a few simple operations on the graph. The efficiency is greatly discounted for the iteratively improving algorithms. Thus in [25], a transitive closure graph (TCG) was proposed to simplify the rules of the operations. The relation of each pair of blocks is prescribed in either horizontal or vertical constraint graph via transitive closure, but not in both graphs.

53.2.2 Corner Stitching

Corner stitching is used to represent the topology of a mosaic floorplan. Simplified from constraint graphs, corner stitching [17] keeps only four pointers at the two opposite corners for every block. All the operations on a constraint graph can also be fulfilled on a corner stitching representation with acceptable increases in time complexity, while the storage for corner stitching becomes easier since the number of pointers attached to every block is fixed (equals to 4). Readers are referred to Chapter 52 for detailed descriptions and analyses of corner stitching.

53.2.3 Twin Binary Tree

Twin binary tree [15] representation applies to mosaic floorplans without degeneration. The twin binary tree is constructed as follows, every block takes its C-neighbor (Fig. 53.5) as its parent. In the first tree, only the C-neighbors in lower left corners (Fig. 53.5 (a) and (b)) are taken into account. If the related T-junction is of type (a), then the block is a left child of its parent, and if the T-junction is of type (b), then the block is a right child. The most bottom-left block in the floorplan acts as the root of the tree. Similarly, in the second tree, the C-neighbors in upper right corners (Fig. 53.12(c) and (d)) are used, and the most upper-right block becomes the tree's root. Fig. 53.13 gives an example of a twin binary tree.

The pointers of twin binary trees are in fact a subset of those in corner stitching. Besides, It has been proved that twin binary tree is a non-redundant representation. In other words, every pair of trees corresponds to a unique mosaic floorplan.

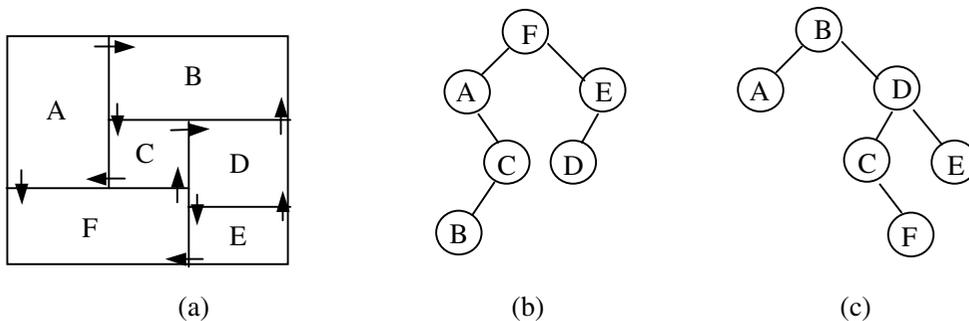


FIGURE 53.13: Twin binary tree.

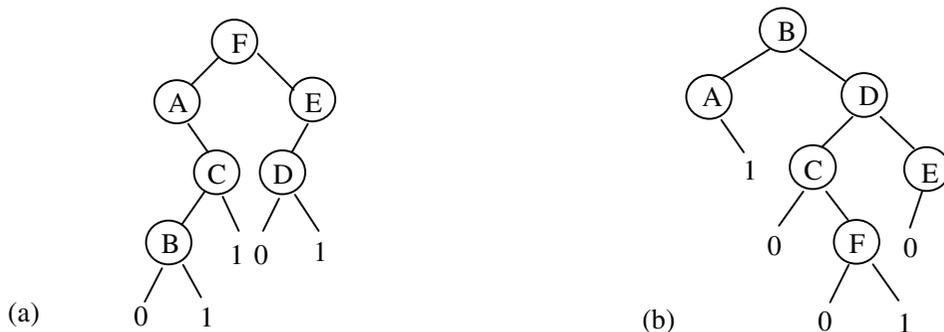


FIGURE 53.14: Extended trees.

The twin properties of binary trees can be illustrated with Fig. 53.14. Consider the trees shown in Fig. 53.13, we add a left child labeled ‘0’ to every node without left child except the most left node, and a right child labeled ‘1’ to every node without right child except the most right node. The resultant trees are the so-called extended trees (Fig. 53.14). The sequences of the in-order traverse of the two extended trees shown in Fig. 53.14 are “A0B1C1F0D1E” and “A1B0C0F1D0E” respectively. If we separate them into the label parts and the bit parts, we have $\pi_1=ABCFDE$, $\alpha_1=01101$ $\pi_2=ABCFDE$ and $\alpha_2=10010$. It is interesting to find that $\pi_1=\pi_2$ and $\alpha_1=\overline{\alpha_2}$. So rather than store the two binary trees, we only keep the linear lists π and α in memory.

However, it is insufficient to recover the two binary trees just from π and α , so we need two more lists, β and β' . If the i -th element in π is the left child of its parent in the first tree or the root of the tree, we set the i -th element in β to be ‘0’, otherwise we set it ‘1’. In the similar way β' is constructed according to the second tree. Thus we use a group of linear lists $\{\pi, \alpha, \beta, \beta'\}$, called the twin binary sequence [16], to express the twin binary tree, and equivalently the floorplan.

Finally, we give the main properties of twin binary tree representation:

1. The memory required for storing the twin binary sequence is $\log_2 n + 3n - 1$, because $|\pi| = \log_2 n$, $|\alpha| = n - 1$, and $|\beta| = |\beta'| = n$.
2. Twin binary tree is a non-redundancy floorplan representation.
3. The complexity of the translation between twin binary sequence and floorplan is $O(n)$.

53.2.4 Single Tree Representations

An ordered-tree (O-tree) [2][3], or equivalently the B* tree [1] representation uses a spanning tree of the constraint graph and the dimensions of the blocks. Because the widths and heights of the blocks are given, the representation can describe one kind of general floorplan termed LB-compact. With a proper encoding, a great enhancement on the storage efficiency and the perturbation easiness is obtained.

A horizontal O-tree is derived with the following rules:

1. If block A lies adjacent to the left side of block B, or, $X_a + W_a = X_b$ (here X_a is the coordinate of block A on the X-axis and W_a is the width of block A), then on the O-tree, B is A's child. If there happens to exist more than one block adjacent to the left side of block B (satisfying the requirement $X_i + W_i = X_b$), one of them is assigned to be the parent of block B.
2. If block A lies on top of block B and the two blocks have the same parents on the O-tree, then B is A's elder brother.
3. A virtual block is presumed to have the left most position, and therefore serves as the root of the O-tree.

Fig. 53.15 shows an example of an O-tree representation for the same mosaic floorplan shown in Fig. 53.13. If we show the pointer of every block pointing to its parent in the O-tree (Fig. 53.15(b)), we can find that the pointers are in fact a subset of those in twin binary tree. Similarly a vertical O-tree can be built up. Without the loss of generality, hereafter we only discuss the horizontal O-tree.

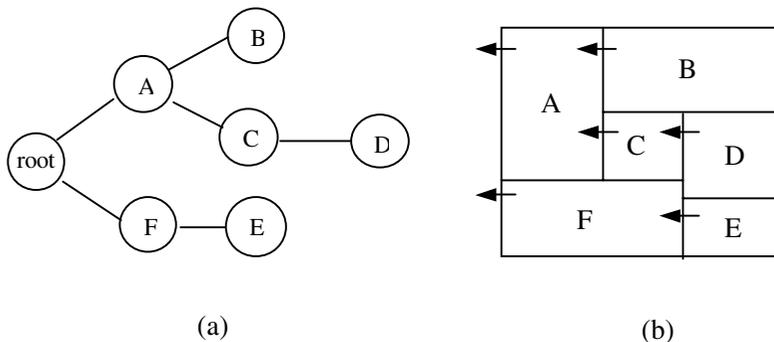
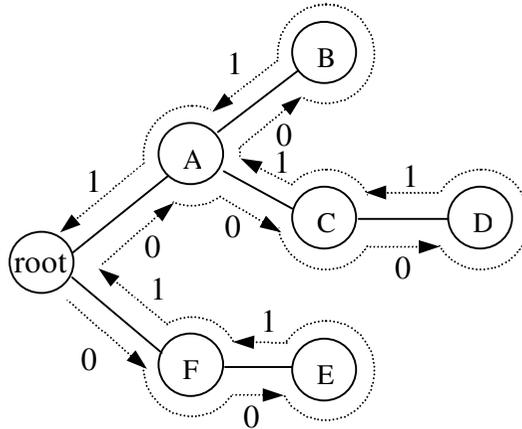


FIGURE 53.15: Building an O-tree.

Next, let's describe the O-tree with linear lists: a bit string (denoted with T) and a label string (denoted with π). The bit string records the structure of an O-tree. We make a depth-first traverse on the O-tree. A "0" is inserted into T for descending an edge while a "1" for ascending an edge. For example, the O-tree in Fig. 53.16 corresponds to the bit string $T = "001100011011"$. Inversely, given the bit string, the structure of the O-tree is solely determined. To get the complete description of an O-tree, we need a second linear list, the label string, to record the labels of the tree's nodes. Not only should the label string include all the labels of the tree's nodes, but also reflect the labels' positions on the tree. A depth-first traverse generates such a linear list. For the example in Fig. 53.16, we have $\pi = 'FEACDB'$.

FIGURE 53.16: Derivation of T, π .

Given a horizontal O-tree, or $\{T, \pi\}$, and the dimensions of blocks, the positions of blocks can be derived with the following rules:

1. Each block has to be placed to the left of its children.
2. If two blocks overlap along their x-coordinate projection, then the block having a higher index in π must be placed on top of the other one.

In fact the second rule applies to the situation in which none of the two blocks is a descendant of the other, like blocks ‘ f ’ and ‘ a ’, or ‘ b ’ and ‘ d ’, in Fig. 53.16. The way we derive π indicates that the block with higher index always stands “higher”. To get to a placement, we need to locate all the blocks. The following operation is executed on blocks according to their orders in π . Here B_i refers to the i -th block, and (x_i, y_i) refers to the coordinate of B_i ’s left-bottom corner.

1. Find B_i ’s parent, B_j , and then we have $x_i = x_j + w_j$, here w_j is the width of block B_j .
2. Let ψ be a set of blocks who have a lower index than B_i in π and have an projection overlap with B_i in the X-axis, find the maximum $y_k + w_k$ for $B_k \in \psi$, then we have $y_i = \max(y_k + w_k)$.

Now we analyze the performance of O-tree:

1. The bit string has a length of $2n$ for an O-tree of n blocks, because each node except the root has an edge towards its parents and each edge is traversed twice during the construction of the bit string. The label string takes $n \log_2 n$ bits for we have to use $\log_2 n$ bits to distinguish the n blocks. So the total memory occupied by an O-tree is $n(2 + \log_2 n)$. By comparison, a sequence pair takes $2n \log_2 n$ bits and a slicing structure takes $n(6 + \log_2 n)$ bits.
2. The time needed to transform $\{T, \pi\}$ to a placement is linear to the number of blocks, or we can say the complexity of transforming $\{T, \pi\}$ to a placement is $O(n)$. For a sequence pair or a slicing structure, the complexity is $O(n \log_2 \log_2 n)$ or $O(n)$ respectively. Upon this point, O-tree has the same performance as the slicing structure but is more powerful for representing a placement.

- The number of combinations is $O(n! \cdot 2^{2n-2} / n^{1.5})$, which is smaller than any other representation that has ever been discussed.

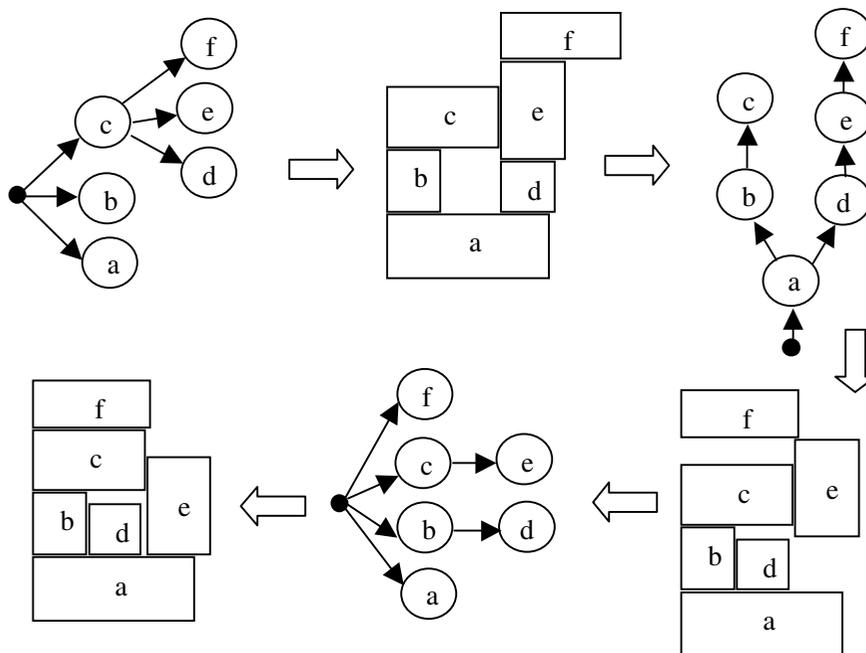


FIGURE 53.17: Transformation between horizontal and vertical O-trees.

The floorplan of an arbitrarily given O-tree is not necessarily LB-compact. Yet in this case, we can compact the floorplan to reduce the chip area. Notice that an O-tree-to-placement transform tightens the blocks, with one direction having the priority. For example, in a placement transformed from a horizontal O-tree, the blocks are placed tighter along the X-axis than along the Y-axis. It would be undoubted that a placement transformed from a vertical O-tree will give Y-axis the priority. Thus, by iteratively making the transforms between horizontal O-trees and vertical O-trees via placements, we can finally reach an LB-compact floorplan (Fig. 53.17).

On the other hand, given an LB-compact floorplan, we can always derive an O-tree representation. For example, Figure 53.18 illustrates a general floorplan with seven blocks and the corresponding O-tree. O-tree is able to cover LB-compact floorplan with the smallest number of combinations in Table 53.1 and Fig. 53.2 because the O-tree structure avoids redundancy and screens improper floorplans by taking advantage of the knowledge of the block dimensions. For example, given an O-tree, we can convert it to a binary tree and find many other possible trees to pair up as twin binary trees, which correspond to many different mosaic floorplans. In the perspective of O-tree representation, these floorplans are the variations due to the differences of the block sizes.

The B* tree and the O-tree are equivalent because the transformation between the two is one to one mapping. The merit of the B tree is a different data structure and implementation.

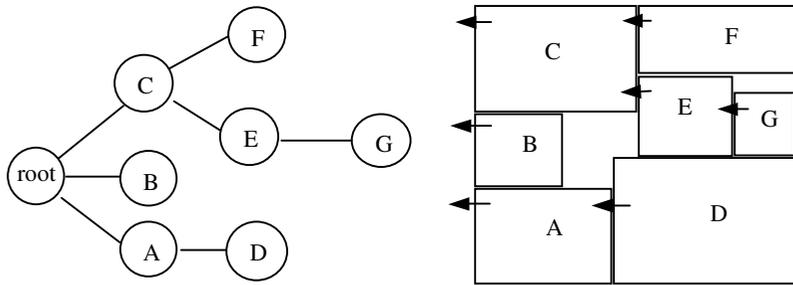


FIGURE 53.18: An O-tree representation of a general floorplan.

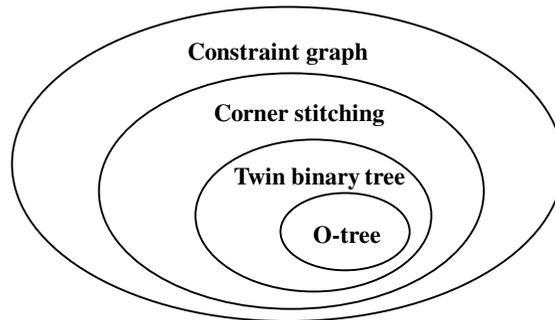


FIGURE 53.19: Relationship between the constraint graph based representation.

In summary, among the constraint-graphs based representations, from the point of view of the pointers necessary for representing the floorplans, O-tree is a subset of twin binary tree; twin binary is a subset of corner stitching; and corner stitching is a subset of constraint graphs, as demonstrated in Fig. 53.19.

53.3 Placement Based Representations

The placement-based representations include sequence pair, bounded-sliceline grid, corner block list and slicing tree. With the dimensions of all the blocks known, the sequence pair and bounded-sliceline grid can be applied to general floorplans. The corner block list records the relative position of adjacent blocks, and is applicable to mosaic floorplan only. The slicing tree is for slicing floorplan, which is a special case of mosaic floorplan.

53.3.1 Sequence-Pair

Sequence-pair expresses the topological positions of blocks with a pair of block lists [5][18]. For each pair of blocks, there are four possible relations, left-right, right-left, above-below, and below-above. We use a pair of sequences $\{\pi_1, \pi_2\}$ to record the order of the two blocks. We can record two blocks A, B in four different orders: (AB, AB), (AB, BA), (BA, AB), (BA, BA). Hence, we use the four orders to represent the four placement relations. Fig. 53.20 shows the four possible relations between a pair of blocks A and B and the corresponding sequence pair for each relation.

The two sequences can be viewed as two coordinate systems that define a grid placement

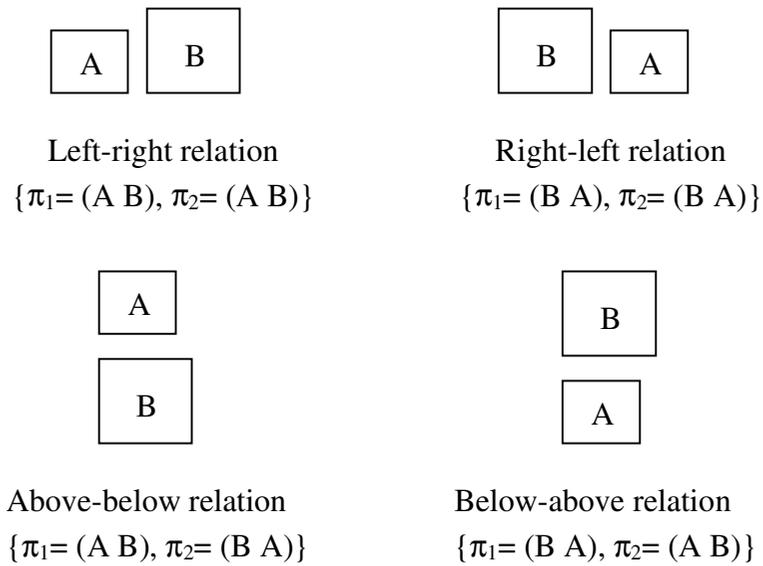


FIGURE 53.20: Four possible relations between a pair of blocks.

of the blocks. To see it more clearly, we construct a grid system illustrated in Fig. 53.21. The slopes are denoted with the labels of blocks according to their orders in the two sequences. The intersections of two perpendicular lines that have the same label indicate the topological positions of the blocks.

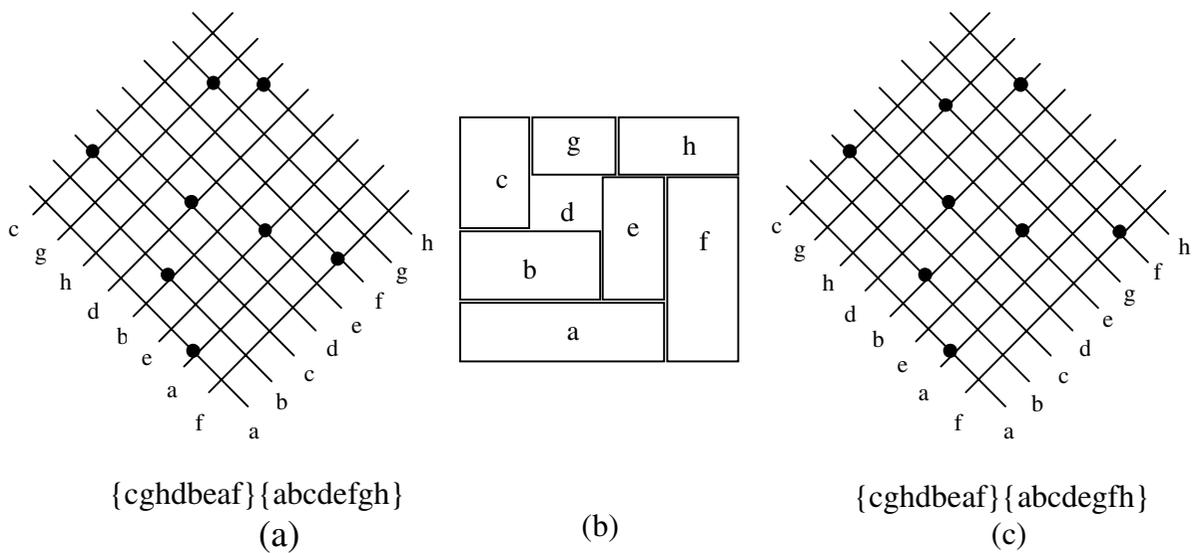


FIGURE 53.21: The grid system and the redundancy.

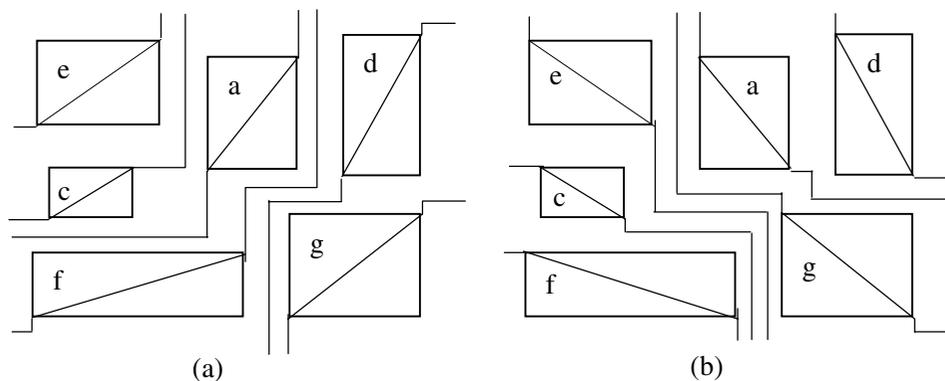


FIGURE 53.22: Derivation of sequence-pair.

The grid placement defines the floorplan relation between blocks. For each node in the grid placement, we divide the whole space into four quadrants, quadrant I: -45 to 45 degrees, quadrant II: 45 to 135 , quadrant III: 135 to 225 , quadrant IV: 225 to 315 . Block B 's floorplan relation with block A can be derived from block B 's location in block A 's quadrants.

1. Quadrant I: block B is on the right of block A
2. Quadrant II: block B is above block A
3. Quadrant III: block B is on the left of block A
4. Quadrant IV: block B is below block A

For example, from the grid placement in Fig. 53.22, using node d as a reference, blocks e and f are on the right of block d ; blocks g and h are above; blocks c is on the left; and blocks a and b are below.

We can derive the sequence-pair from a floorplan by tracing the blocks through two directions. We first extend the boundary of each block by drawing lines from its four corners (Fig. 53.22). In Fig. 53.22(a), we draw a line from the upper-right corner of each block that goes either rightward or upward. When the line is obstructed with a block or an already existing line, it changes its direction. The extension of the line ends when it reaches the boundary of the chip. We also draw a line from the lower-left corner of each block that goes either downwards or leftwards. The extended lines partition the chip into zones. Following the order of the zones, we get a label sequence. For instance, we have $\pi_1 = 'ecafdg'$ for Fig. 53.22(a). With similar operations, the second sequence is derived in the orthogonal direction ($\pi_2 = 'fcedag'$ in Fig. 53.22(b)). π_1 and π_2 consist of a sequence-pair, which involves all the information of the topological positions of blocks.

For translating a sequence-pair into a topological floorplan with n blocks, a brute force implementation requires time complexity $O(n^2)$, since there are $O(n^2)$ pairs of blocks and we need to scan all the pairs to get the constraint graph and then derive the floorplan by performing a longest path computation. In [8], a fast algorithm with $O(n \log \log n)$ time complexity is proposed to complete the transformation of a sequence pair to the floorplan. The fast algorithm takes advantage of the longest common sequence of the two sequences in a sequence-pair. They show that the longest common sequence calculation is equivalent to the longest path computation in the constraint graph for a sequence-pair. The authors use a priority queue data structure to record the longest common sequence, thus reduce the

time for calculating the position of a single block to $O(\log\log n)$ by amortizing.

The representation of a sequence-pair has the following properties:

1. The number of bits for storing a sequence-pair is $2n \log_2 n$, where n is the number of blocks in the placement.
2. The time complexity of translating a sequence-pair into a topological floorplan is $n \log_2 \log_2 n$ by the newest report.
3. There are totally $(n!)^2$ possible sequence-pairs for n blocks. However, there exists the redundancy. For example, the sequence-pairs in Fig. 53.21(a) and (c) actually correspond to the same placement (Fig. 53.21(b)).

53.3.2 Bounded-Sliceline Grid

Another method rising from the same idea as sequence-pair is BSG [6], where, instead of the intersections of grid lines, the squares surrounded by the grid lines, called rooms, are used to assign the blocks (Fig. 53.23). Although put forward aiming at the packing problem, BSG can also act as a compacting algorithm to determine the accurate positions of blocks.

In BSG, grid lines are separated into vertical and horizontal segments (Fig. 53.23). Two constraint graphs are set up with their vertexes corresponding to the grid segments and their directed edges corresponding to the relative positions of the grid segments (Fig. 53.24). The vertexes ‘source’ and ‘sink’ are added to make the operation on the graph easy. Every room is crossed by one edge of the constraint graph (respectively in the vertical and horizontal constraint graphs). If the room is not empty, or there is a block assigned into the room, the crossing edge has a weight equal to the width (in the horizontal graph) or the height (in the vertical graph) of the assigned block, otherwise the crossing edge has a weight of zero. Fig. 53.24 shows the weights derived from the example in Fig. 53.23. By calculating the longest path lengths between the source and the other vertexes in the constraint graphs the real coordinates of the grid segments can be determined and in fact the translation to the placement is implemented. Table 53.4 gives the final results of the example in Fig. 53.23 and Fig. 53.24. Notice that segment (1, 3) and (2, 2) have the same position, for the edge between the two vertexes in the horizontal constraint graph has the weight of zero.

TABLE 53.4 Physical positions of segments.

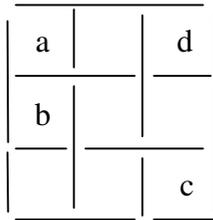
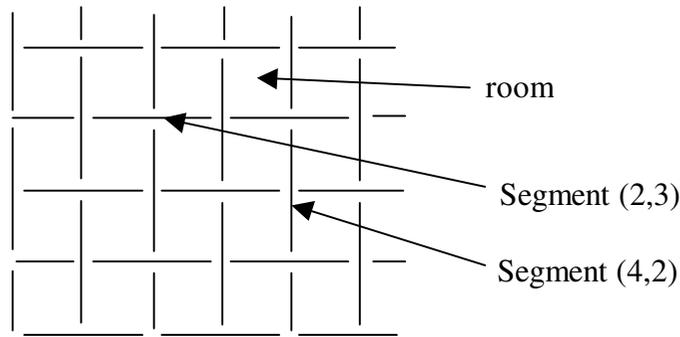
Segment	0,1	1,1	0,2	1,2	1,3	0,3	2,2	2,3	3,3	3,2	2,1	2,0	3,1	3,0
Position (X or Y)	0	6	0	9	8	12	8	12	17	4	4	6	17	0
	Y	X	X	Y	X	Y	X	Y	X	Y	Y	X	X	Y

Due to the homology, BSG has the similar performance as sequence-pair, except that the complexity of the translation into placement, or finding the longest path lengths for all the vertexes in the constraint graphs, is $O(pq)$, provided that the grid array has p columns and q rows.

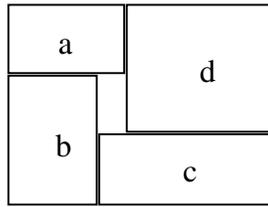
53.3.3 Corner Block List

Corner block list is a third method of representing non-slicing structures [4][19][20]. Refer to Fig. 53.25. The floorplan is required to be mosaic and without degeneration.

For every block, the T-junction at the bottom-left corner can be in either of the two directions. Accordingly we say the orientation of the block is 1 or 0 respectively. Now



(a)



(b)

	width	height
a	8	3
b	6	9
c	11	4
d	9	8

FIGURE 53.23: Basic structure of BSG.

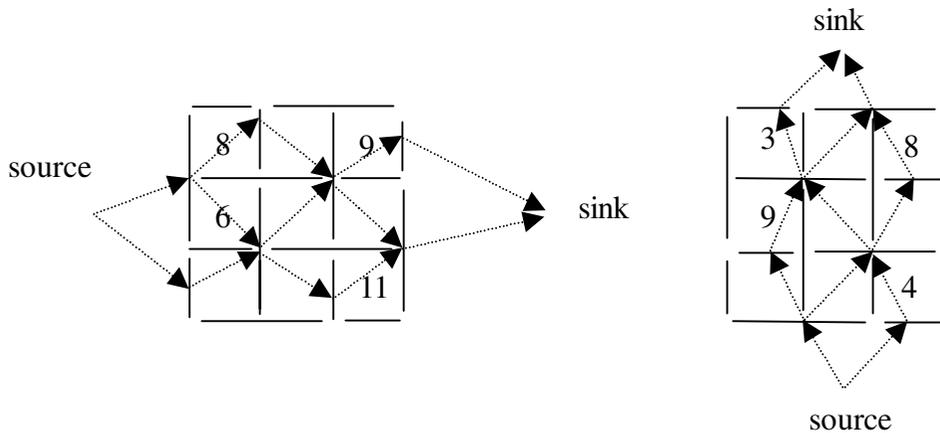


FIGURE 53.24: Constraint graphs.

let's observe the most upper-right blocks in Fig. 53.25(a) and (b), both denoted with 'd'. If we push the bottom boundary of block 'd' in (a) upwards, or the left boundary in (b) rightwards, then block 'd' is squeezed and finally deleted (Fig. 53.25(c) and (d)), and thereafter block 'a' becomes the most upper-right block. The operation of squeezing and deleting can be repeatedly performed until there is only one block left in the placement.

According to the order in which the blocks are squeezed out, we get two lists. The labels of blocks are stored in list 'S' while the orientations of the blocks in list 'L'. For example,

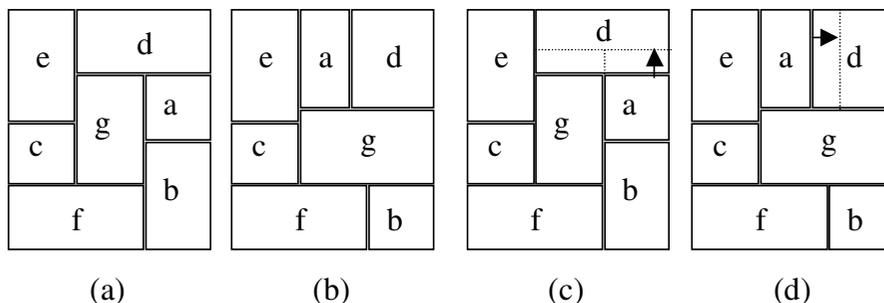


FIGURE 53.25: Corner block list.

for Fig. 53.25(a), $S = 'fcegbad'$ and $L = '001100'$. S and L are not sufficient to recover the placement, so we need the third list, T . Each time we squeeze a block, we record the number of T-junctions on the moving boundary, excluding the two at its ends, then add the same number of '1's and one '0' into T . For example, in Fig. 53.25(a), when squeezing block d , we have $T = '01'$, for there is only one T-junction on the bottom boundary, which separates the blocks a , d and g . Next, while squeezing block a , only a '0' is added to T , for there is no T-junction on the bottom boundary of block a . Consequently, we have $T = '001'$ after the deletion of block a , and $T = '00101001'$ after the deletion of block c . With an inverse process, the placement can be recovered from S, L and T .

The performance of corner block list is:

1. The storage of the 3 lists needs at most $n(3 + \log n)$ bits.
2. The number of combinations is $TRIALRESTRICTION$, which is the same as that of a slicing tree. However, slicing structure only covers part of all possible combinations, therefore has larger redundancy than corner block list.
3. The complexity of the translation between placement and lists is $TRIALRESTRICTION$.

53.3.4 Slicing Tree

A floorplan is called a slicing structure [7][29] if it can be iteratively partitioned into two parts with vertical or horizontal cut lines throughout the separated parts of the area, until the individual blocks. Fig. 53.26(a) gives an example. Line 1 splits the whole layout into two parts. The left half involves only one block but the right half can be further split by line 2. Then follows line 3, line 4, and so on. On the contrary, Fig. 53.26(b) is a non-slicing structure. We can't find a cut line throughout the floorplan to split it into two parts.

Now that a slicing floorplan can be iteratively split into two parts with vertical or horizontal cut lines, a binary tree can be structured according to the splitting process.[13]

Fig. 53.27 gives an example of a slicing floorplan and its binary tree. The internal nodes on the tree denoted with "H" or "V" indicate the direction of the line splitting the areas, while the leaf nodes correspond to the blocks. We call the binary tree a slicing tree.

Just as in a binary tree, we hope to find a simple data structure to express the slicing tree. Polish Expression is a good solution. It is the result of a post-order traversal on the slicing tree. As an example, the Polish Expression of Fig. 53.27 is: $123V56V8H47HVHV$. We regard "H" and "V" as operators and the blocks operands.

Formally, we have the following definition for a Polish expression:

A sequence $b_1b_2...b_{2n-1}$ of elements from $\{1, 2, \dots, n, V, H\}$ is a Polish expression of

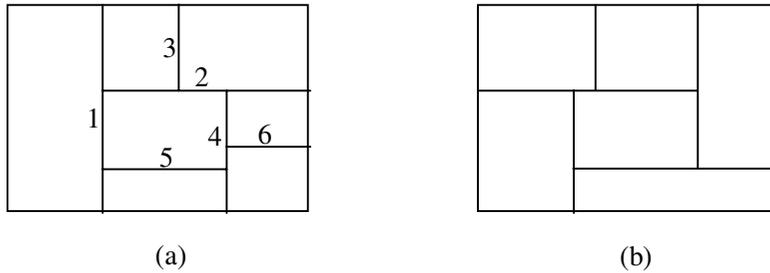


FIGURE 53.26: Slicing structure and non-slicing structure.

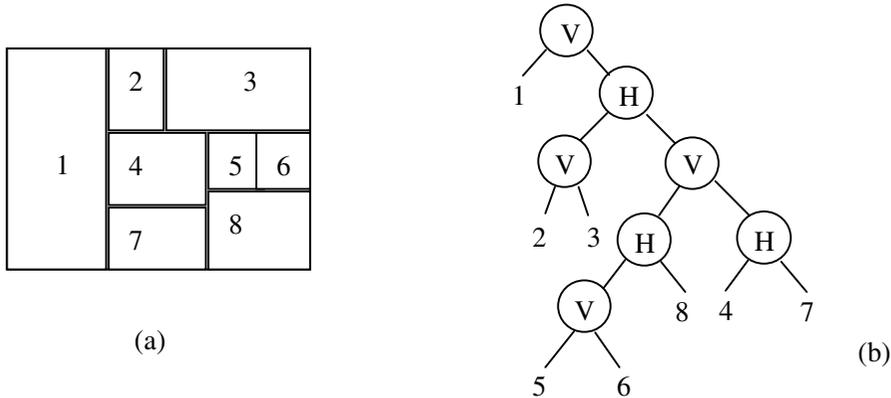


FIGURE 53.27: Slicing structure and its binary tree representation.

length $2n - 1$ iff:

1. Every subscript i appears exactly once in the sequence, $1 \leq i \leq 2n-1$.
2. The number of operators is less than the number of the components for any prefix sub-string.

So needless to re-construct the slicing tree, we are able to judge a legal Polish Expression. For example, $21H57V43H6VHV$ and $215H7V43H6VHV$ are legal Polish Expressions while $215HVHV7436HV$ and $2H15437VVVHH$ are not. An illegal Polish Expression can't be converted to a binary tree.

It is obvious that for a given slicing tree, we have a unique Polish Expression. Inversely, we can easily conduct the slicing tree according to its Polish Expression. So we conclude that there is a one-to-one correspondence between the slicing tree and Polish Expression. However, there may be more than one slicing tree corresponding to one floorplan, as shown in Fig. 53.28. Thus there exists redundancy. To compact the solution space, we define the skewed slicing tree to be a slicing tree without a node that has the same operator as its right son. Correspondingly, a normalized Polish expression is defined:

A Polish expression $b_1b_2 \dots b_{2n-1}$ is called to be normalized iff there is no consecutive "V" or "H" in the sequence.

In Fig. 53.28, (b) is a normalized Polish expression while (c) is not. There is a one-to-one correspondence between the set of normalized Polish expressions and the set of slicing structures.

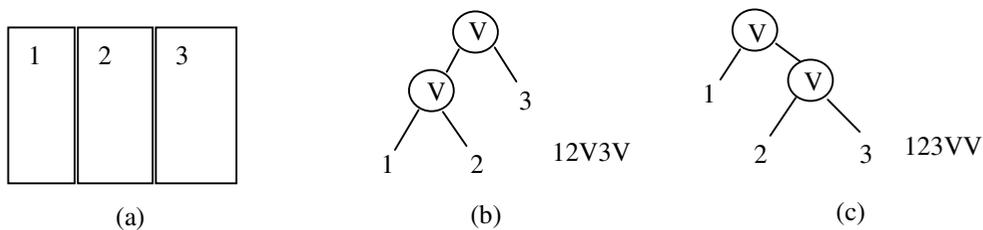


FIGURE 53.28: Redundancy of slicing trees.

To perturb the blocks to get new solutions, we define three types of perturbations on polish expressions:

- M1: Swap two adjacent operands. For example, $12V3H \rightarrow 13V2H$;
- M2: Complement some chain of operators. For example, $12V3H \rightarrow 12V3V$;
- M3: Swap two adjacent operand and operator. For example, $12V3H \rightarrow 123VH$;

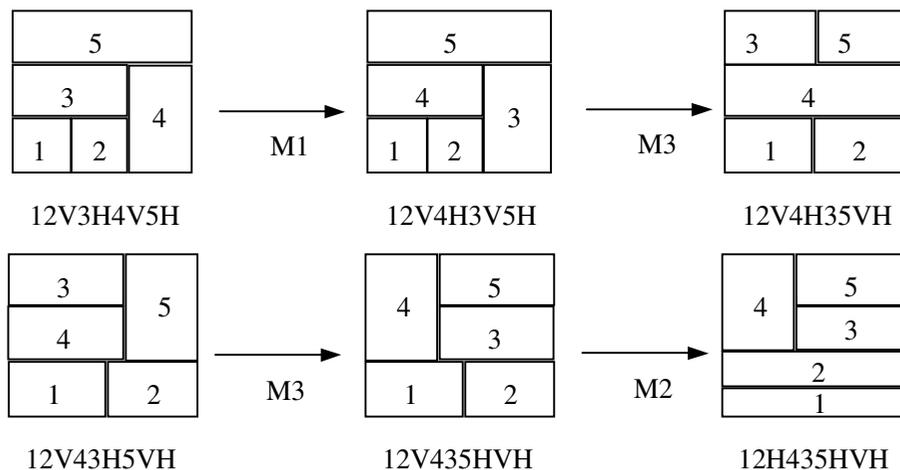


FIGURE 53.29: Slicing structure moves.

The operation of M3 may result in a non-normalized Polish Expression. So a check is necessary after an operation of M3 to guarantee that a normalized expression is generated.

Fig. 53.29 illustrates two sequences of polish expression transformations as well as their corresponding floorplans. We can see that a slight modification on the Polish Expression can result in a completely different floorplan topology. This is just what we want.

Finally, we analyze the performance of slicing tree.

1. Given a floorplan of n blocks, the slicing tree contains n leaves and $n - 1$ internal nodes.
2. The number of possible configurations for the tree is $O(n! \cdot 2^{3n-3} \cdot n^{1.5})$.
3. The storage of a Polish Expression needs a $3n$ -bit stream plus a permutation of n blocks.

4. It will take linear time to transform a floorplan into a slicing tree, and vice versa.

53.4 Relationships of the Representations

We summarize the relationships between the representations in this section. A mosaic floorplan example and a general floorplan example are also discussed in detail to show the relationships.

53.4.1 Summary of the Relationships

According to the definitions of the representations, we have four relationships between the sequence-pair (SP), the corner block list (CBL), the twin binary tree (TBT) and the O-Tree representations.

For further discussions, we first define the 90° rotation of a floorplan as follows:

Definition 90° rotation of a floorplan: We use F^{90} to denote the floorplan obtained by rotating floorplan F , 90 degrees counterclockwise.

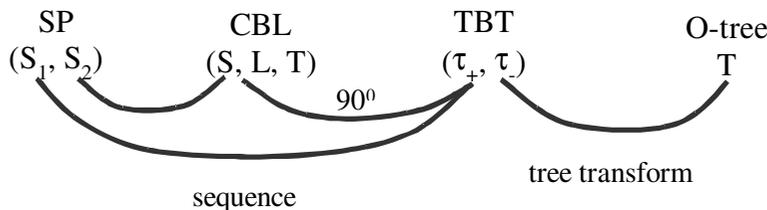
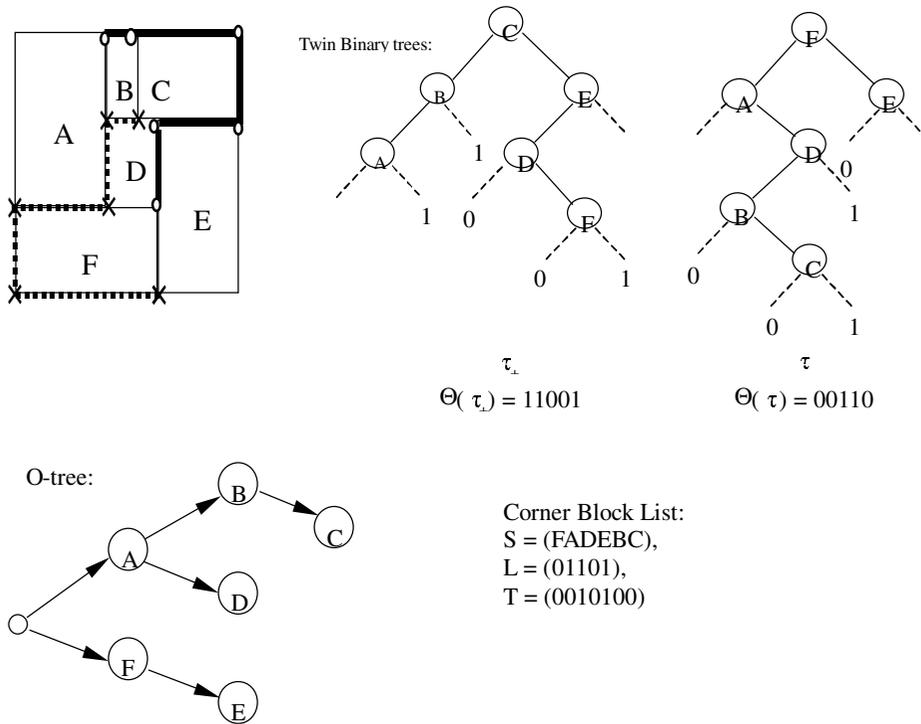


FIGURE 53.30: Summary of the relationships between floorplan representations.

The four relationships are as follows and summarized in Fig. 53.30.

1. Given a mosaic floorplan and its corresponding corner block list $CB=(S, L, T)$, there exists a sequence pair $SP = (S_1, S_2)$ corresponding to the same floorplan such that the second sequence of SP is same as the sequence S of the corner block list.
2. Given a mosaic floorplan and its corresponding twin binary trees (τ_+, τ_-) , there exists a sequence pair $SP = (S_1, S_2)$ corresponding to the same floorplan such that the first sequence of SP is the same as the node sequence of in-order traversal in tree τ_+ .
3. Given a mosaic floorplan and its corresponding twin binary trees $TBT (\tau_+, \tau_-)$, there exists an O-tree corresponding to the same floorplan such that τ_- is identical to the O-tree after the tree conversion from a binary tree to an ordered tree.
4. Given a mosaic floorplan F , and its corresponding twin binary trees $TBT (\tau_+, \tau_-)$, the node sequence of in-order traversal in tree τ_+ is identical to the sequence S in the corner block list $CB=(S, L, T)$ of the floorplan F^{90} .



Sequence Pair: $SP_1 = (S_1, S_2) = (ABCD FE, FADEBC)$. Also $SP_2 = (S_1, S_2) = (ABCD FE, FAD BEC)$ refers to the same floorplan

FIGURE 53.31: A mosaic floorplan and its different representations.

53.4.2 A Mosaic Floorplan Example

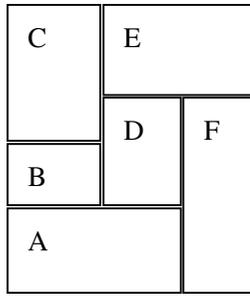
Fig. 53.31 describes an example of a mosaic floorplan. We illustrate the four representations. The twin binary trees are marked with circles and crosses as shown in Fig. 53.31. Two SP s, SP_1 and SP_2 , out of many possible choices, are described in the figure.

The in-order traversal of the τ_+ in the twin binary trees representation produces the sequence $\pi(\tau_+) = ABCDFE$, which is same as the first sequence of SP_1 and SP_2 . In Fig. 53.31, an O-tree representation is also given. Its binary tree representation is identical to the τ_- of the twin binary trees after tree conversion. The corner block list representation is next to the O-tree representation in Fig. 53.31. Its sequence $S=FADEBC$ is same as the second sequence of SP_1 .

Fig. 53.32 shows the 90° rotation of the mosaic floorplan in Fig. 53.31. The $CB(S, L, T)$ representation of F^{90} has $S = ABCDFE$ (Fig. 53.32.), which is identical to $\pi(\tau)$, the order of the twin binary trees representation (τ_+, τ_-) of the floorplan in Fig. 53.31.

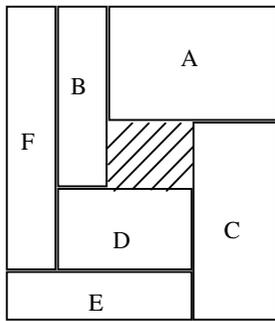
53.4.3 A General Floorplan Example

Fig. 53.33 illustrates a general floorplan. Only the O-tree and the sequence pair are capable of representing this general floorplan. The O-tree and SP representations are shown in the figure.

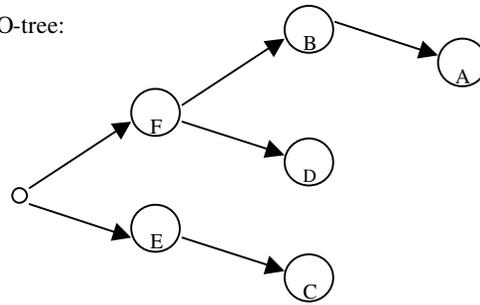


Corner Block List
 Representation of F^{90}
 $S=(ABCDFE);$
 $L=(00110);$
 $T=(00010101);$

FIGURE 53.32: 90° rotation of the floorplan in Fig. 58.31.



O-tree:



Sequence Pair: $SP1=(\Gamma_+, \Gamma_-) = (FBADEC, EFDBCA),$
 Also, $SP2=(\Gamma_+, \Gamma_-) = (FBADEC, EDFBCA)$

FIGURE 53.33: A general floorplan with its O-tree and SP representation.

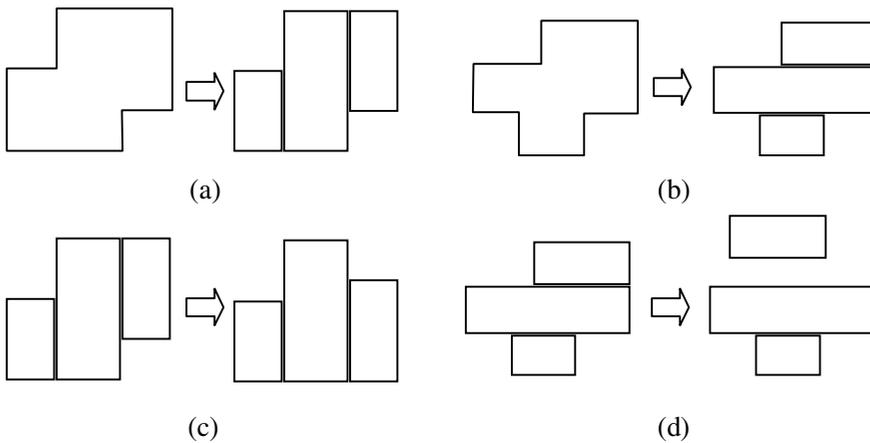


FIGURE 53.34: Shape handling.

53.5 Rectilinear Shape Handling

Shape handling makes feasible the treating of blocks with arbitrary shapes. It makes sense especially in deep sub-micron technology, where, with the number of routing layers increasing, wires are placed in layers other than the device layer, and the placement becomes more like a packing problem: a set of macro blocks are packed together without overlaps, with some objective functions minimized.

The basic idea to handle the arbitrary-shaped blocks is to partition them into rectangles, and then the representations and the optimizing strategies described above can be used. Fig. 53.34(a) and (b) demonstrate two schemes of partitioning, the horizontal one and the vertical one. However, if the generated rectangle blocks were treated as individual ones, it would be quite impossible to recover the original blocks. The generated rectangle blocks may be shifted (Fig. 53.34(c)) or separated (Fig. 53.34(d)). So extra constraints have to be imposed on the generated rectangle blocks in order to keep the original blocks' shapes.

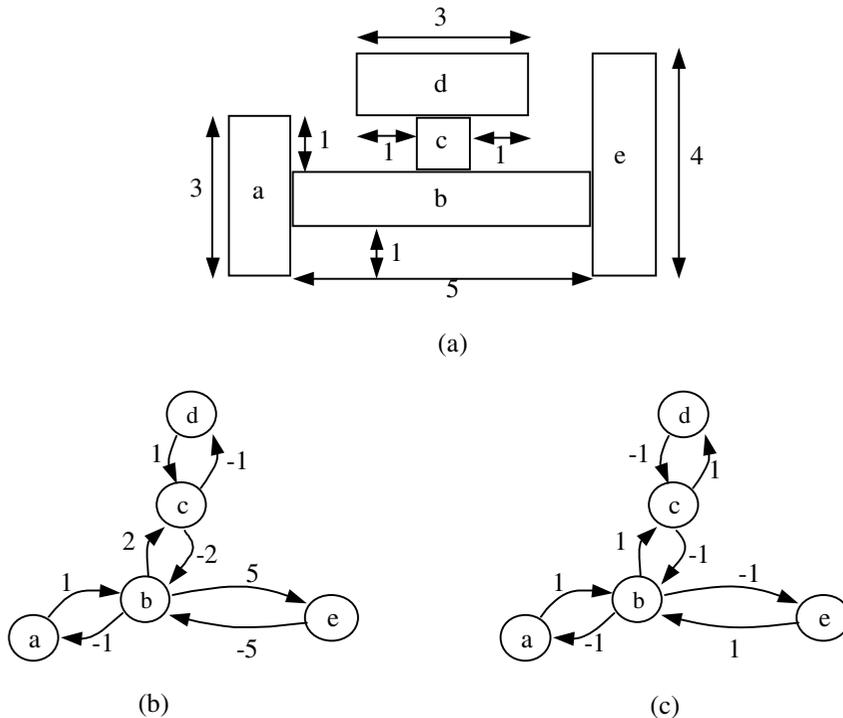


FIGURE 53.35: An example of adding extra constraints.

One proposed method of adding extra constraints is illustrated with Fig. 53.35(b) for horizontal and (c) for vertical. The vertices in the constraint graph correspond to the generated rectangle blocks, while the direct edges denote the relative positions of the blocks. For example, an edge from 'a' to 'b' with a weight of 1 means that block 'b' lies to the right of 'a', the distance between their left boundaries is 1, which is exactly the width of 'a'.

Because the relative-position constraints are always represented with the format such as “larger than or equal to” during optimization, inversely directed edges, such as the edge from ‘b’ to ‘a’ with a negative weight, helps to determine the position relationships solely.

53.6 Conclusions

Floorplan representation is an important issue for floorplan optimization. We classify the representations into graph based and placement based methods based on the different strategies used in deriving the representations. The graph-based methods are based on the constraint graphs, which include the constraint graphs, the corner stitching, the twin binary tree and the O-tree. The placement-based methods are based on the local topology relations in deriving the floorplans, which include the sequence pair, the bounded sliceline grid, the corner block list and the slicing tree.

The floorplans can also be classified into different categories: general, mosaic and slicing floorplans. Slicing floorplan is a subset of mosaic floorplan, which is again a subset of the general floorplan. We have different representations for different types of floorplans. The constraint graphs, the corner stitching, the sequence pair, the bounded sliceline grid and the O-trees are for general floorplans. The twin binary tree and the corner block list are for mosaic floorplans. The slicing tree is for slicing floorplans. Different representations have different solution spaces and time complexities to derive a floorplan.

53.7 Acknowledgment

The authors are grateful to Mr. Simon Chu of Sun Microsystems for his thorough review of the chapter.

References

- [1] Y. Chang, G. Wu, and S. Wu, “B*-Trees: A New Representation for Non-Slicing Floorplans”, in *Proc. 37th DAC*, 2000, pp. 458-463.
- [2] P. Guo, C.K. Cheng, and T. Yoshimura, “An O-Tree Representation of Non-Slicing Floorplan and Its Applications,” in *Proc. 36th DAC*, 1999, pp. 268-273.
- [3] Y. Pang, C.K. Cheng, K. Lampaert, W. Xie, “Rectilinear block packing using O-tree representation,” in *Proc. ISPD 2001*, pp. 156-161.
- [4] X. Hong, G. Huang, Y. Cai, J. Gu, S. Dong, C.K. Cheng, J. Gu, “Conner Block List: An Effective and Efficient Topological Representation of Non-Slicing Floorplan,” in *Proceeding of ICCAD-2000*, 2000, pp. 8-12.
- [5] H. Murata, K.Fujiyoshi, S. Nakatake and Y. Kajitani, “Rectangle-packing-based Module Placement”, in *Proc. of International Conference on Computer Aided Design*, 1995, pp. 472-479.
- [6] S. Nakatake, K.Fujiyoshi, H. Murata, and Y. Kajitani, “Module Packing Based on the BSG-Structure and IC Layout Applications”, in *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 17, No. 6, June 1998, pp. 519-530.
- [7] R.H.J.M.Otten, “Automatic Floorplan Design”, in *Proc. ACM/IEEE Design Automation Conf.*, 1982, pp. 261-267.
- [8] X. Tang, M. Wong, “FAST-SP A Fast Algorithm for Block Placement based on Sequence Pair”, in *Proc. of Asia and South Pacific Design Automation Conference*, Yokohama, Japan, Jan. 2001, pp. 521-526.
- [9] D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley Press, 1997.

- [10] R.H.J.M. Otten, "What is a Floorplan?" in *Proc. of International Symposium on Physical Design*, 2000, pp. 212-217.
- [11] J. Grason, "A Dual Linear Graph Representation for Space-filling Location Problems of the Floor-planning Type," MIT Press, Cambridge (Mass), USA, 1970.
- [12] K. Kozminsky and E. Kinnen, "Rectangular Duals of Planar Graph," in *Networks*, 15, 1985, pp. 145-157.
- [13] A.A. Szepieniec and R.H.J.M. Otten, "The Genealogical Approach to the Layout Problem," in *Proc. of the 17th Design Automation Conference*, 1980, pp. 535-542.
- [14] K. Sakanushi, and Y. Kajitani, "The Quarter-State Sequence (Q-sequence) to Represent the Floorplan and Applications to Layout Optimization," in *Proc. IEEE APC-CAS*, 2000, pp. 829-832.
- [15] B. Yao, H. Chen, C.K. Cheng, and R. Graham, "Revisiting Floorplan Representations," in *Proc. International Symposium on Physical Design*, 2001, pp. 138-143.
- [16] F.Y. Young, C.N. Chu and Z.C. Shen, "Twin Binary Sequences: A Non-redundant Representation for General Non-slicing Floorplan," in *Proc International Symposium on Physical Design*, 2002, pp.196-201.
- [17] J. Ousterhous, "Corner stitching: A data-structuring technique for VLSI layout tools," *IEEE Transactions on Computer-Aided Design*, CAD-3, January 1984, pp. 87-100.
- [18] J. Xu, P. Guo, C.K. Cheng, "Sequence-Pair Approach for Rectilinear Module Placement," *IEEE Trans. on CAD*, April 1999, pp. 484-493.
- [19] S. Zhou, S. Dong, X. Hong, Y. Cai, J. Gu, and C.K. Cheng, "ECBL: An Extended Corner Block List with O(n) Complexity and Solution Space Including Optimum Placement," *Int. Symp. on Physical Design*, 2001, pp. 150-155.
- [20] Y. Ma, X. Hong, S. Dong, Y. Cai, C.K. Cheng, J. Gu, "Floorplanning with Abutment Constraints and L-Shaped /T-Shaped Blocks Based on Corner Block List," *ACM/IEEE Design Automation Conf.*, 2001, pp. 770-775.
- [21] J. Xu, P. Guo, C.K. Cheng, "Cluster Refinement for Block Placement", *Proc. ACM/IEEE Design Automation Conf.*, 1997, pp. 762-765.
- [22] J. Xu, P. Guo, C.K. Cheng, "Empirical Study of Block Placement by Cluster Refinement," *VLSI Design*, vol.10, no.1, pp. 71-86.
- [23] M. Abe, "Covering the square by squares without overlapping (in Japanese)," *J. Japan Math. Phys.*, vol. 4, no. 4, 1930, pp. 359-366.
- [24] K. Sakanushi, Y. Kajitani, D. P. Mehta, "The Quarter-State-Sequence Floorplan Representation," in *IEEE Trans. on Circuits and Systems—I*, vol. 50, no. 3, 2003, pp. 376-385.
- [25] J.M. Lin, Y.W. Chang, "TCG: A Transitive Closure Graph-Based Representation for Non-Slicing Floorplans," in *Proc. DAC*, 2001, pp. 764-76
- [26] W. Dai, and E.S. Kuh, "Hierarchical Floor Planning for Building Block Layout," *IEEE Int. Conf. on Computer-Aided Design*, pp.454-457, 1986.
- [27] R.L. Brooks, C.A.B. Smith, A.H. Stone, and W.T. Tutte, "The Dissection of Rectangle into Squares," *Duke Math. Journal*, vol. 7, pp. 312-340, 1940.
- [28] C. Zhuang, K. Sakanushi, J. Liyan, Y. Kajitani, "An Enhanced Q-Sequence Augmented with Empty-Room-Insertion and Parenthesis Trees," *Design Automation and Test in Europe Conference and Exhibition*, pp.61-68, 2002.
- [29] D.F. Wong and C.L. Liu, "A New Algorithm for Floorplan Design," *ACM/IEEE Design Automation Conf.*, pp.101-107, 1986.

Computer Graphics

Dale McMullin
Colorado School of Mines

Alyn Rockwood
Colorado School of Mines

54.1	Introduction	54-1
	Hardware and Pipeline	
54.2	Basic Applications	54-2
	Meshes • CAD/CAM Drawings • Fonts • Bitmaps • Texture Mapping	
54.3	Data Structures	54-7
	Vertices, Edges, and Faces • Vertex, Normal, and Face Lists • Winged Edge • Tiled, Multidimensional Array • Linear Interpolation and Bezier Curves	
54.4	Applications of Previously Discussed Structures	54-15
	Hidden Surface Removal: An Application of the BSP Tree • Proximity and Collision: Other Applications of the BSP Tree • More With Trees: CSG Modeling	

54.1 Introduction

Like all major applications within the vast arenas of computer science and technology, the computer graphics industry depends on the efficient synergy of hardware and software to deliver to the growing demands of computer users. From computer gaming to engineering to medicine, computer graphics applications are pervasive. As hardware capabilities grow, the potential for new feasible uses for computer graphics emerge.

In recent years, the exponential growth of chipset speeds and memory capacities in personal computers have made commonplace the applications that were once only available to individuals and companies with specialized graphical needs. One excellent example is flight simulators. As recently as twenty years ago, an aviation company would have purchased a flight simulator, capable of rendering one thousand shaded polygons per second, for ten million dollars. Even with a room full of processing hardware and rendering equipment, it would be primitive by today's graphics standards. With today's graphics cards and software, one renders tens of millions of shaded polygons every second for two hundred dollars.

As the needs for graphics applications grow and change, research takes the industry in many different directions. However, even though the applications may evolve, what happens under the scenes is much more static; the way graphics primitives are represented, or stored in computer memory, have stayed relatively constant. This can be mainly attributed to the continued use of many standard, stable data structures, algorithms, and models. As this chapter will illustrate, data and algorithmic standards familiar to computer science lend themselves quite well to turning the mathematics and geometries of computer graphics into impressive images.

54.1.1 Hardware and Pipeline

Graphics hardware plays an important role in nearly all applications of computer graphics. The ability for systems to map 3-D vertices to 2-D locations on a monitor screen is critical. Once the object or “model” is interpreted by the CPU, the hardware draws and shades the object according to a user’s viewpoint. The “pipeline” [2], or order in which the computer turns mathematical expressions into a graphics scene, governs this process. Several complex sub-processes define the pipeline. Figure 54.1 illustrates a typical example.

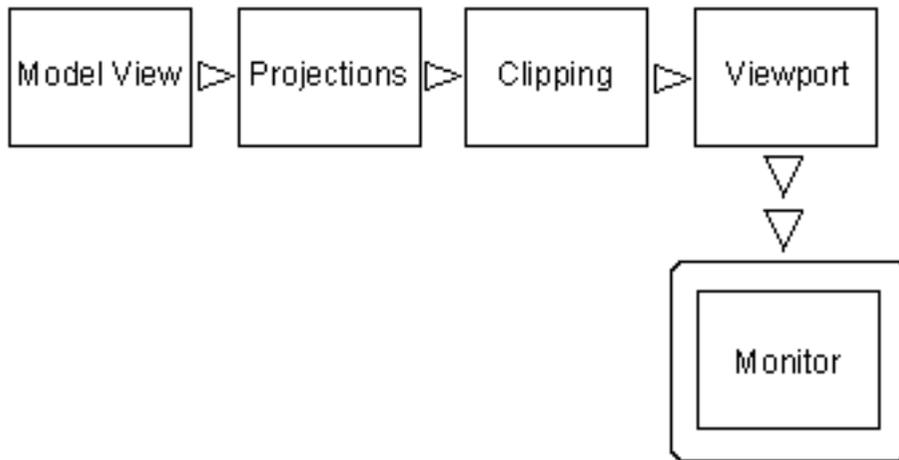


FIGURE 54.1: Graphics pipeline.

The Model View is the point where the models are created, constructed as a combination of meshes and mapped textures. The Projection point is where the models are transformed (scaled, rotated, and moved) through a series of affine transformations to their final position. Clipping involves invoking algorithms to determine perspective, and what objects are visible in the Viewport. The Viewport reads the scene for display on the computer’s monitor. The final scene is “rasterized” to the monitor [2].

Standard data structures and algorithms apply to all steps in the pipeline process. Since the speed of most rendering hardware (and hence the pipeline) is directly dependent on the number of models to be drawn, it becomes important to utilize structures that are as efficient as possible. In fact, graphics hardware is often designed and engineered to cater to a specific, standard form of data representation.

54.2 Basic Applications

54.2.1 Meshes

In both 2-D and 3-D applications, objects are modeled with polygons and polygon meshes. The basic elements of a polygon mesh include the vertex, the edge, and the face. An edge is composed of the line segment between two vertices, and a face is defined by the closed polygon of three or more edges. A mesh is formed when two or more faces are connected by shared vertices and edges. A typical polygon (triangle) mesh is shown in [Fig. 54.2](#).

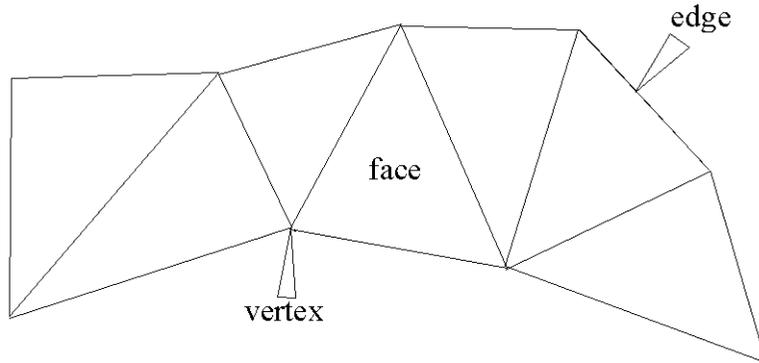


FIGURE 54.2: A triangle mesh.

Specific examples of meshes include “triangle meshes, polygon meshes, and polygon meshes with holes.” [1] However, the triangle is currently the most popular face geometry in standard meshes. This is mainly because of its simplicity, direct application to trigonometric calculations, and ease of mesh storage (as we will see).

54.2.2 CAD/CAM Drawings

In Computer Aided Design (CAD), Computer Aided Manufacturing (CAM), and other channels of engineering design, we see the same basic elements. During the design process of everything from automobiles to telephones, applications are used to “see” the parts before they are manufactured. The physical interactions and assemblies of parts can be tested before any steel is milled or plastic is poured.

Figure 54.3 illustrates a wire-frame (and fully rendered) model of a drill bit used in the oil and gas industry. Every modeled element of this design requires a definition of vertices and edges. These vertices define single-line edges, polyline edges, and the curved edges. The proximity of vertices to edges, edges to edges, and vertices to vertices may be tested for tolerances and potential problems. The software makes the stored model visible to the user within the current coordinate system and viewpoint. It is then up to the user to interface the visual medium with his or her engineering (or design) training to create a useful design.

Even in wire frame models, we see the application of more sophisticated graphics practices. Note how many of the contours of the drill bit parts are not connected line segments, but actual smooth curves. This is an example of where vertices have been used not to define edges, but the “control points” of a Bezier Curve. How curves like the B-Spline and Bezier utilize vertices in their structures is discussed later in this chapter. For now it is sufficient to mention that these types of curves are present in nearly all corners of the graphics industry.

54.2.3 Fonts

Fonts are another example of where the vertex is used, in two dimensions, as the basis of a data structure in order to define the appearance of each character and number in various languages. Several examples of interesting Greek characters, each defined by a different font definition file, and at different sizes, are shown in Figure 54.4.

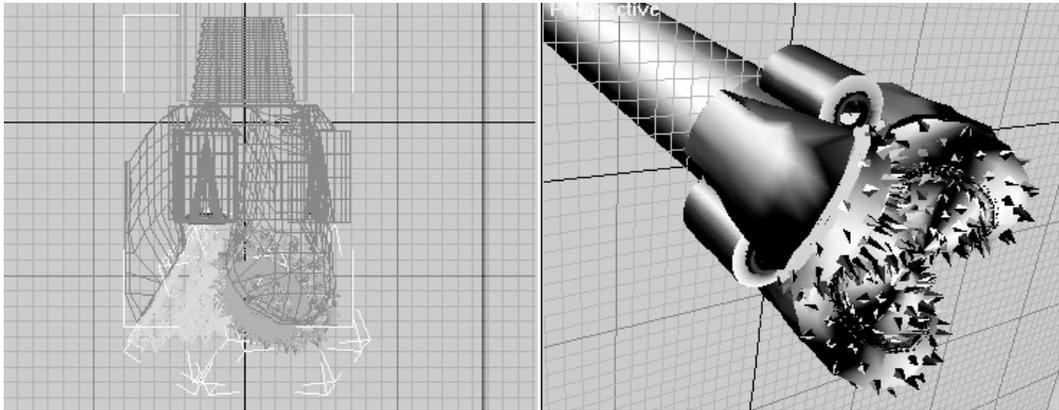


FIGURE 54.3: Models of a drill bit.

M β γ Δ δ ϕ **T**

FIGURE 54.4: Fonts.

The typical Postscript font is defined by a number of points, read in as control points for each curve (or line) in the font character. The “Postscript” driver software is then used to interpret the control points and render the characters with Bezier curves (see [section 3.5](#)).

Perhaps the most important concept is that each pixel of the font character does not have to be stored, only a small number of vertex definitions. As the control points are transformed through operations like italicizing, the curves remain aligned in the proper proportion to the original character. Note the “M” character in the above figure has been italicized. The “Postscript” font representation serves as a very effective example of how a single standardized data structure of specific size can be utilized to define a potentially infinite number of characters.

54.2.4 Bitmaps

Bitmaps are a cornerstone to computer graphics. In fact, the name “bitmap” has become a commodity to most computer users in the way that it describes a computer generated picture. A bitmap is a very simple, yet versatile, way to store a graphic image as a binary file. The file header structure of a bitmap [6] in itself is a standard data structure, as the following illustrates:

```
struct _BITMAP
{
  UInt8   bmType;
  UInt8   bmBitsPerPixel;
  UInt8   bmBytesPerPixel;
  UInt8   bmReserved;
```

```
Uint32  bmColorKey;

Uint32  bmWidth;
Uint32  bmHeight;
Uint32  bmPitch;

void*   bmBits;
};
```

The bitmap is initially defined by a number of parameters that define the type, width, height, etc. of the graphic image. These properties are stored in the header of the file as shown. The actual “pixels” required to fill that space is then defined in the “bmBits” pointer. As a rule, the total number of pixels in the image memory will equal the width times the height divided by the “bits per pixel” property. The “bytes per pixel” property determines how the “bits per pixel” are divided among the individual color components of each pixel. For instance, a bitmap with RGB color map is commonly defined by twenty four (24) bits per pixel and three (3) bytes per pixel. Each of the three bytes for each pixel use 8 of the twenty four bits to define red, green, and blue values respectively.

Early bitmaps, when color monitors were first being used, were defined with 4-bit color. In other words, each color was defined by a 4-bit (or half-byte) word. Since a half-byte has a maximum value of 24 or 16, 4-bit bitmaps were capable of supporting sixteen (16) different colors. The required disk space to store a 100 x 100 4-bit bitmap would then be $10000 * .5$ bytes or 5000 Bytes (5kB).

In the past fifteen years, the need for more realistic graphics has driven the increase in the memory used for color. Today 24-bit (also called true color) and 32-bit color, which both represent 16.7 million colors (32-bit adds an extra byte for the alpha channel), are the most commonly supported formats in today’s hardware. The alpha channel refers to a single byte used to store the transparency of the color. Bitmap files in these formats now require 3 and 4 bytes per pixel. Additionally, current monitor resolutions require over 1000 pixels in width or height, and thus graphic files are growing even larger. [Figure 54.5](#) is a photograph that has been scanned in and stored as a bitmap file.

A typical 5x7 photograph, at 100 dpi (dots per inch), and 24-bit color, would require 500 x 700 x 3 or 1.05 megabytes (MB) to store in a conventional bitmap format. Because of the increased size requirements in bitmaps, compression algorithms have become commonplace. File formats such as JPG (jpeg) and PNG (ping) are examples of widely used formats. However, there is a tradeoff. When a compression algorithm is applied to a bitmap, a degree of image quality is inevitably lost. Consequently, in applications like publishing and graphic art, uncompressed bitmaps are still required where high image quality is expected.

54.2.5 Texture Mapping

The processes of texture and surface mapping represent an application where bitmaps and polygonal meshes are combined to create more realistic models. Texture mapping has become a cornerstone of graphics applications in recent years because of its versatility, ease of implementation, and speed in graphical environments with a high number of objects and polygons. In fact, today’s graphics hardware ships with the tools necessary to implement the texture and surface mapping processes on the local chipset, and the data structures used in rendering environments are largely standardized.

Prior to texture mapping, 3-D polygonal meshes were processed through shading models such as Gouraud and Phong to provide realism. Each shading model provides the means



FIGURE 54.5: Bitmap file.

for providing color, shadows, and even reflection (glare) to individual components of the model. This was accomplished through a mathematical model of how light behaves within specular, diffuse, reflective, and refractive contexts. The general shading equation for light intensity, I , based on the individual ambient, diffuse, and specular components is shown below.

$$I = I_a p_a + (I_d p_d \times \text{lambert}) + (I_s p_s \times \text{phong})$$

where:

$$\text{lambert} = \max\left(0, \frac{s \cdot m}{|s||m|}\right)$$

$$\text{phong} = \max\left(0, \frac{h \cdot m}{|h||m|}\right)$$

p_a = ambient coefficient $0 \leq p_a \leq 1$

p_d = diffuse coefficient $0 \leq p_d \leq 1$

p_s = specular coefficient $0 \leq p_s \leq 1$

The primary difference between Gouraud and Phong is in that Phong provides the additional component for specular reflection, which gives objects a more realistic glare when desired. Also, because the Phong model requires more detail, the intensity values are calculated (interpolated) at each pixel rather than each polygon. The vectors m , s , and h represent the normal, reflected, and diffuse light from a given polygonal surface (or face). Linear interpolation, a widely used algorithm in graphics, is discussed in Section 3.5.

Although still used today, these shading models have limitations when more realistic results are desired. Because color and light reflection are modeled on individual model components, the model would have to be constructed as a composite of many different components, each with different color (or material) properties, in order to achieve a more realistic effect. This requires models to be much more complex, and increasingly difficult to construct. If changes in surface appearance or continuity were desired, they would have to be physically modeled in the mesh itself in order to be viewed.

Texture and surface mapping have provided a practical solution to these model complexity dilemmas. The mapping methods involve taking the input from a bitmap file, like those described previously, and “stretching” them over a polygonal mesh. The end result is a

meshed object, which takes on the texture properties (or appearance) of the bitmap. The figure below illustrates a simple example of how the mesh and bitmap can be combined to create a more interesting object. Figure 54.6 shows how Figure 54.5 has been “mapped” onto a sphere.

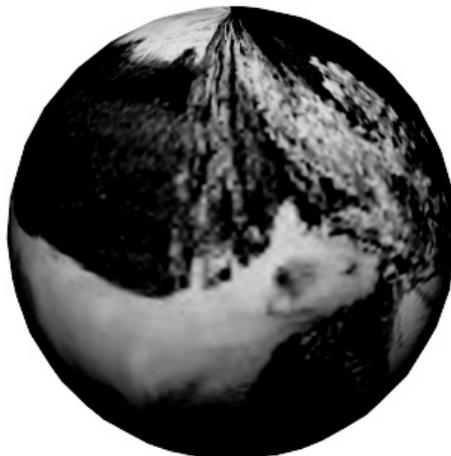


FIGURE 54.6: Combination of mesh and bitmap.

The object seems to take on a more realistic appearance even when modeled by a very simple polygonal mesh. This technology has made fast rendering of realistic environments much more feasible, especially in computer games.

Texture, or “bump”, mapping utilizes a similar process as surface mapping, where a bitmap is stretched over a polygonal mesh. However, the pixels, commonly called “texels” [2], are used to alter how the light intensity interacts with the surface. Initially, the lighting model shown above would calculate an intensity, I , for a given surface. With texture mapping, individual grayscale values at each texel are used to alter the intensity vectors across each polygon in order to produce roughness effects.

Figure 54.7 illustrates a model of a sphere that has been rendered with the traditional Gouraud model, then Phong, and then finally rendered again with a texture map. This approach to improving model realism through mapping applies also to reflection, light intensity, and others.

54.3 Data Structures

54.3.1 Vertices, Edges, and Faces

As mentioned previously, vertices, edges, and faces form the most basic elements of all polygonal representations in computer graphics. The simplest point can be represented in Cartesian coordinates in two (2D) and three dimensions (3D) as (x,y) and (x,y,z) respectively (Figure 54.8).



FIGURE 54.7: Texture mapping.

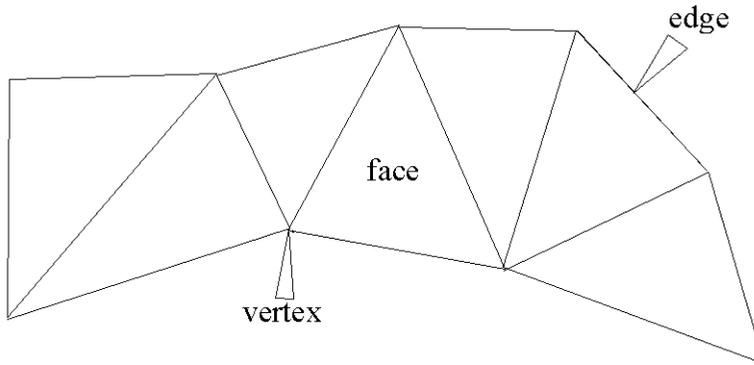


FIGURE 54.8: Vertices, edges, and faces.

As a simple data structure, each vertex may then be stored as a two or three-dimensional array. Edges may then be represented by two-dimensional arrays containing the indexes of two points. Further, a face may be dimensioned based on the number of desired sides per face, and contain the indices of those edges. At first, this approach may seem acceptable, and in basic applications it is common to model each vertex, edge, and face as a separate class. Relative relationships between classes are then governed by the software to build meshes. However, modeling objects in this manner does have disadvantages.

Firstly, important information becomes difficult to track, such as the normal at each vertex, adjacency of faces, and other properties required for blending and achieving realism. Furthermore, the number of intersecting edges at each vertex may vary throughout the model, and mesh sizes between objects may be unpredictable. The ability to manage this approach then becomes increasingly difficult, with the potential for unnecessary overhead and housekeeping of relationships within the model. It then becomes necessary to create higher level data structures that go beyond these basic elements, and provide the elements necessary for efficient graphics applications.

54.3.2 Vertex, Normal, and Face Lists

In this storage method, list structures are used to store three, inter-dependent lists of data. The first list defines the vertexes contained in the scene as follows. Each vertex is assigned an index, and a coordinate location, or (x,y,z) point. The second list defines the normals for each vertex. Again, each normal is assigned a numbered index and a 3-D coordinate point. The final list serves to integrate the first two. Each face is identified by a numbered index, an array of vertex indexes, and an array of indexed normals for each vertex. Figure 54.9 illustrates typical examples of a similar application with vertex, face, and edge lists.

vertex	x	Y	Z	face	vertex number	edge	vertex begin	vertex end
0	0	0	0	0	4, 5, 6, 7	0	0	1
1	0	1	0	1	0, 1, 2, 3	1	1	2
2	1	1	0	2	3, 2, 6, 5	2	2	3
3	1	0	0	3	4, 7, 1, 0	3	3	0
4	0	0	1	4	2, 1, 7, 6	4	4	5
5	1	0	1	5	4, 0, 3, 5	5	5	6
6	1	1	1			6	6	7
7	0	1	1			7	7	4

FIGURE 54.9: Example of vertex, normal, and face lists.

In the table, three specific lists are evident. The first list represents each vertex in the model as it is defined in 3D space. The “vertex” column defines the id, index, or label of the vertex. The x, y, and z coordinates are then defined in the adjacent columns.

In the second list, six faces are defined. Each face has a label or index similar to the vertex list. However, rather than specifying coordinates in space, the adjacent column stores the id’s of the vertexes that enclose the face. Note that each face consists of four vertices, indicating that the each face will be defined by a quadrilateral.

The final list defines edges. Again, each edge definition contains a label or index column, followed by two adjacent vertex columns. The vertices of each edge define the start point and end point of each edge. In many applications of graphics, it is important to define the direction of an edge. In right handed coordinate systems, the direction of an edge will determine the direction of the normal vector which is orthogonal to the face surrounded by the edges.

54.3.3 Winged Edge

Although one of the oldest data structures relating to polygonal structures, the Winged Edge approach is very effective, and still widely used [5]. This structure is different from the wireframe model in that edges are the primary focal point of organization.

In the structure, each edge is stored in an indexed array, with its vertices, adjacent faces, previous, and successive edges. This allows the critical information for each edge to be stored in an array of eight integer indexes; it is both consistent and scalable between applications. The structure is

Edge	Vertices		Faces		Left Traverse		Right Traverse	
	Name	Start	End	Left	Right	Previous	Succeeding	Previous
a	X	Y	1	2	b	d	e	c

FIGURE 54.10: Winged edge table.

Figure 54.11 illustrates a typical layout for a winged edge.

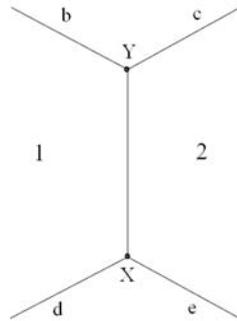


FIGURE 54.11: Winged edge.

An important aspect of the Winged Edge structure is the order in which entries are listed. The edge itself has a direction, from the start vertex to the end vertex. The other entries are then defined by their proximity to the edge. If the direction of the edge were reversed, the right and left faces would change accordingly, as would the previous and succeeding entries of both the left and right traverse.

There is a time/space trade-off in using this model. What is saved in storage space adds to the needed time to find previous and successive edges. See [Chapter 17](#) for more details.

54.3.4 Tiled, Multidimensional Array

In this section we will discuss a data structure that is important to most geometric implementations. In many ways, the tiled array behaves identically to matrix data structures. For instance, a p by q matrix is normally stored as a single dimension array. However, since the matrix has p -rows and q -columns, the array needs to be addressed by creating an “offset” of size q in order to traverse each row. The following example should illustrate this concept.

Consider a matrix with $p = 3$ rows and $q = 3$ columns:

is stored as:

$$A = [1, 3, 4, 5, 2, 7, 3, 9, 3]$$

This represents where a 3x3 matrix is stored as an array with $(3)(3) = 9$ entries. In order to find the entry at row(i) = 3 and column(j) = 2 we employ the following method on the array:

$$Entry = j + (i - 1)q$$

Or in this case, $Entry = A[2 + (3 - 1)(3)] = A[8] = 9$

We use a similar method for tiling a single bitmap into several smaller images. This is analogous to each number entry in the above array being replaced by a bitmap with n by n pixels.

“Utilizing cache hierarchy is a crucial task in designing algorithms for modern architectures.” [2] In other words, tiling is a crucial step to ensuring multi-dimensional arrays are stored in an efficient, useful manner. Indexing mechanisms are used to locate data in each dimension. For example, a p by q array is stored in a one-dimensional array of length $p \cdot q$, and indexed in row-column fashion as above.

When we wish to tile a p by q matrix into n by n tiles, the number of blocks in x is defined by q/n and the number of blocks in y is defined by p/n . Therefore, to find the “tile index” or the row and column of the tile for a value (x,y) we first calculate the tile location, or bitmap within the matrix of bitmaps. Then once the bitmap is located, the pixel location within that bitmap tile is found (sub-indexed). The entire two-step process can be simplified into a single equation that is executed once. Figure 54.12 illustrates this concept.

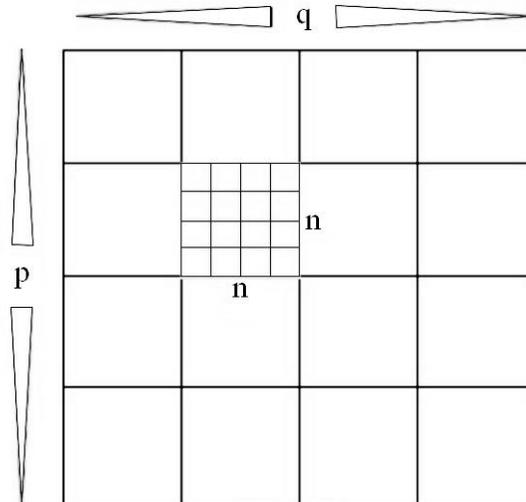


FIGURE 54.12: Tiled array.

The final formula for locating x,y in a p by q array divided into n,n tiles is:

$$n^2 \left(\left(\frac{q}{n} \right) \left(\frac{y}{n} \right) + \frac{x}{n} \right) + (y \bmod n)n + (x \bmod n)$$

When dealing with matrices and combining multiple bitmaps and/or data into a Tile Multidimensional Array, performance and speed can both improve substantially.

54.3.5 Linear Interpolation and Bezier Curves

This section will introduce one of the most significant contributions to design and graphics: the interpolated curve structure.

Linear interpolation refers to the parameterization of a straight line as a function of t , or:

$$L(t) = (1 - t)a + tb$$

where a , b are points in space. This equation represents both an affine invariant and barycentric combination of the points a and b . Affine invariance means that the point $L(t)$ will always be collinear with the straight line through the point set $\{a, b\}$, regardless of the positioning of a and b . Describing this set as barycentric simply means that for t values between 0 and 1, $L(t)$ will always occur between a and b . Another accurate description is that the equation $L(t)$ is a linear “mapping” of t into some arbitrary region of space. Figure 54.13 illustrates this concept.

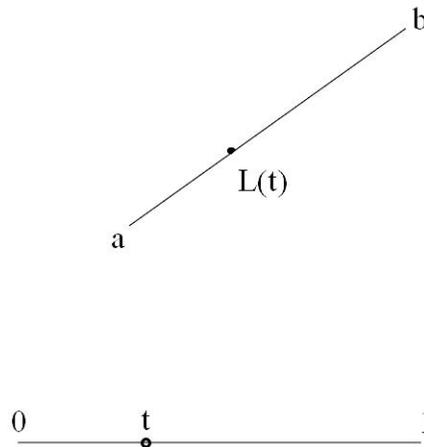


FIGURE 54.13: Linear interpolation.

Linear interpolation is an extremely powerful concept. It is not only the foundation of many curve approximation algorithms, but the method is also used in many non-geometric applications as well, including calculating individual pixel intensities in the Phong shading method mentioned previously.

The basic Bezier curve derivation is based on an expanded form of linear interpolation. This concept uses the parameterization of t to represent two connected lines. The curve is

then derived by performing a linear interpolation between the points interpolated on each line; a sort of linear interpolation in n parts, where n is the number of control points. The following example should illustrate:

Given are three points in space, $\{a, b, c\}$. These three points form the two line segments ab and bc (Figure 54.14).

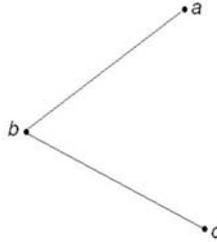


FIGURE 54.14: Expanded form of linear interpolation.

During the first “iteration” of the Bezier curve derivation, linear interpolation is performed on each line segment for a given t value. These points are then connected by an additional line segment. The resulting equations (illustrated in Figure 54.15) are:

$$\begin{aligned}x &= (1 - t)a + tb \\y &= (1 - t)b + tc\end{aligned}$$

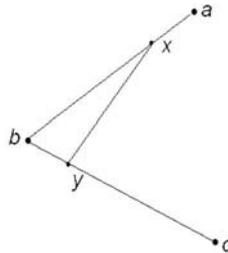


FIGURE 54.15: First iteration of Bezier curve derivation.

The linear interpolation is performed one last time, with the same t value between the new points $\{x, y\}$ (Figure 54.16):

$$z = (1 - t)x + ty$$

This final point z , after three linear interpolations, is on the curve. Following this 3-step process for several “stepped” values for t between 0 and 1 would result in a smooth curve of z -values from a to c , and is illustrated in Figure 54.17:

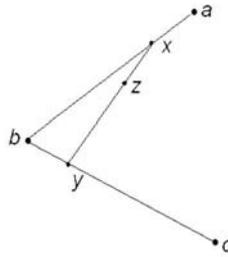


FIGURE 54.16: Result of three linear interpolations.

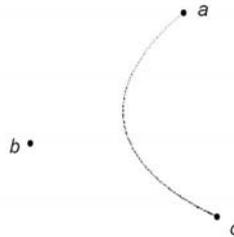


FIGURE 54.17: Smooth curve.

This is a quadratic Bezier curve, and is represented mathematically by a linear interpolation between each set of x and y points, which were also derived through linear interpolation, for every t . By substituting the equations for x and y into the basic form, we obtain:

$$z(t) = (1-t)[(1-t)a + tb] + t[(1-t)b + tc]$$

Simplified, we obtain a quadratic polynomial for a , b , and c as a function of the parameter t , or

$$z(t) = (1-t)^2a + 2(1-t)tb + t^2c$$

The “string art” algorithm described previously is also referred to as the de Causteljau algorithm. Programmatically, the algorithm performs an n -step process for each value of t , where n is the number of “control points” in the curve. In this example, $\{a, b, c\}$ are the control points of the curve.

Because of their ease of implementation and versatility, Bezier curves are invaluable in CAD, CAM, and graphic design. Also, Bezier curves require only a small number of control points for an accurate definition, so the data structure is ideal for compact storage. As mentioned previously, the Bezier form is the foundation for the Postscript font definition structure.

However, the Bezier form has limitations. When several attached curves are required in a design component, it takes considerable effort, or pre-calculation, to ensure the connected curves are “fair” or continuous with each other. Merely joining the end points of the curves may not be enough, resulting in “kinks” and other undesirable effects. For this reason, the B-spline curve provides an alternative to the Bezier where continuity is important. In fact, B-spline curves have often been referred to as a “user interface” for the Bezier form.

The goal of this section was to illustrate the versatility of linear interpolation and basic iteration structures in graphics and design. If more information is desired, numerous texts are currently available which describe the properties, mathematics, and applications of Bezier and B-spline curves, including the references listed at the end of this chapter.

54.4 Applications of Previously Discussed Structures

54.4.1 Hidden Surface Removal: An Application of the BSP Tree

In addition to other algorithms, BSP (Binary Space Partitioning) trees are one example where a specific, widely used, data structure has direct application in computer graphics. Hidden surface removal is essential to realistic 3-D graphics, and a primary application of BSP trees.

Hidden surface removal is used anywhere 3-dimensional objects must be made visible from multiple, if not infinite view points within a graphics scene. Whether a scene is being viewed in a 3-D design application, or the latest sci-fi game, the problem is the same: objects furthest away from the user's viewpoint may be hidden by closer objects. Therefore, the algorithm used to determine visibility must effectively sort the objects prior to rendering. This has been a hot topic until recent years, with researchers finding new and creative ways to tackle the problem. The z-buffer is now undeniably the most widely used algorithm. Tree algorithms are, on the other hand, also widely used where time-based rendering (animation) is not an issue, especially where the object positions are static. BSP trees are discussed in greater detail in [Chapter 20](#).

The BSP tree is an example of a “painter’s algorithm.” [1] The basic concept of this algorithm involves sorting the objects (polygons) from back to front “relative to [the] viewpoint” and then drawing them in order. The key to the BSP algorithm in hidden surface removal is in the pre-processing, and encoding of the objects into a data structure that is later used to determine visibility. In other words, the data structure does not change, only the way it is viewed.

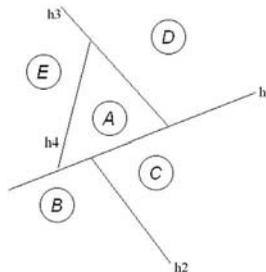


FIGURE 54.18: Objects in a plane.

For hidden surface removal, the BSP tree is built by passing a plane through each polygon in the scene. For each point p in front of the plane, $f(p) > 0$ and for each point behind the plane, $f(p) < 0$. The tree structure is created by applying this object to each polygon, and defining a “negative branch” and “positive branch” for each polygon relative to the current position in the tree. Also called the “half space” on each side of the plane, each vertex

position in the tree is dictated by its position relative to the passing planes. One plane is treated as the root of the tree, and successive branches are defined from that root.

Because the relative position of vertices to each other is defined by the tree, regardless of position, the entire BSP tree can be pre-computed. Whether or not polygons in the tree are visible is then a function of their position in the viewer's plane. Figure 54.19 demonstrates a BSP tree for vertices A through E shown in Figure 54.18.

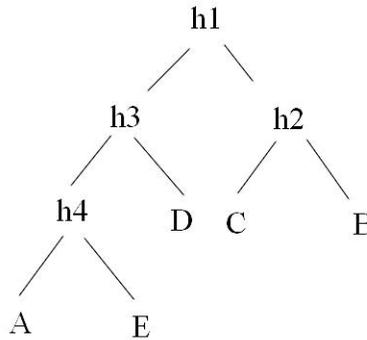


FIGURE 54.19: BSP tree for Figure 54.18.

Note how each vertex can be located relative to at least two planes. For a viewpoint along h_3 , it is immediately apparent that vertex D is on the right, while A and E are on the left. Vertices C and B are located in the half space of h_1 and h_2 , and are therefore of no interest. Vertices C and B will not be rendered. This relative method works for all positions in the BSP tree.

As mentioned, BSP trees are not a universal solution for hidden surface removal, mainly because of the pre-processing requirement. There is a major caveat; if the objects in the scene are moving, the pre-processing of the BSP tree is no longer valid as the polygons change relative position. The tree must be built every time the relative positions of objects within the tree change. Re-calculating the tree at each time step is often too slow, especially in 3-D gaming, where several million polygons must be rendered every second.

Another problem is that the BSP tree works in hidden surface removal only when “no polygon crosses the plane defined by any other polygon.” In other words, no object can be both behind and in front of another object for the BSP sorting algorithm to work. In gaming, it is common for objects to “collide” so this algorithm becomes even less desirable in these unpredictable conditions.

54.4.2 Proximity and Collision: Other Applications of the BSP Tree

Although BSP Tree structures are not as useful for determining the rendering order of moving polygons, they have other applications in graphics and gaming. For instance, trees are commonly used for collision detection, line-of sight, and other algorithms where the number of required calculations is lower. Determining between time-steps the relative positions of several (or several hundred) models, rather than several hundred million polygons, is much more feasible with today's hardware. Enhanced Tree structures are even used in many of today's more innovative artificial intelligence algorithms for gaming. These structures are used within the game space to quickly determine how an object sees, touches, and ultimately reacts with other objects.

54.4.3 More With Trees: CSG Modeling

Another widely used application of tree-based structures is application of Boolean operations to object construction. Constructive Solid Geometry, or CSG, modeling involves the construction of complex objects using only simple, primitive shapes. These shapes are added and subtracted to each other through "set operations," or "Boolean operations." The final objects are referred to as "compound," "Boolean," or "CSG" objects [4].

Figure 54.20 of a dumbbell illustrates a typical example of a CSG object.

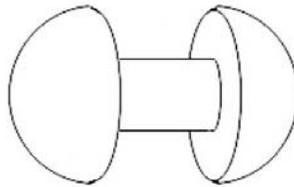


FIGURE 54.20: CSG object.

To construct this object in a CSG environment, Boolean operations are applied to primitive objects. These operations are "intersection," "union," and "difference" [4]. Each step in the tree represents a boolean combination between two objects. The resulting object at each point in the tree is then combined again with another Boolean operation at the next step. This type of progression is continued until the finished object is created. Figure 54.21 illustrates the CSG tree used to construct this dumbbell using two spheres, a cylinder, and a half-plane.

A union operation is analogous to gluing two primitives together. An intersection operation results in only the portion (or volume) that both primitives occupy. Finally, a difference operation in effect removes the intersected section of one primitive from another. Armed with these three Boolean operations, modeling and displaying very complex shapes are possible. However, attempting to cover the surface of a final shape with a continuous mesh is another problem entirely, and the subject of numerous texts and papers. Consequently, CSG objects are largely used for initial visualization of a complete model. This solid model is then sent to another application that converts the model to a polygonal mesh.

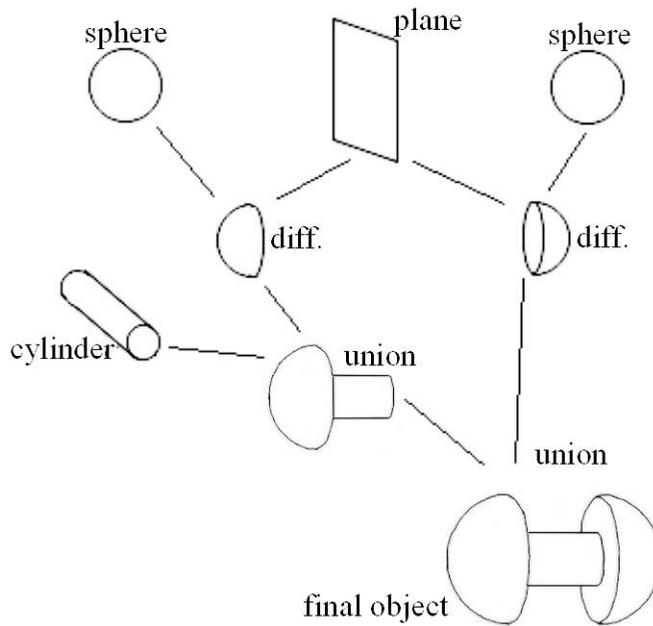


FIGURE 54.21: CSG tree corresponding to [Figure 54.20](#).

From a data structure and algorithm standpoint, CSG trees are quite useful. Firstly, and most obvious, the method utilizes the tree structure already mentioned throughout the text. Secondly, the final objects do not require individual mesh models to define them. Rather, each object is simply defined by its tree, where each node of the tree references primitives, such as the sphere, cone, cylinder, cube, and plane. With the basic models for the sphere, cone, etc. preprocessed and stored in memory, the overhead of CSG applications is kept to a minimum.

Many commercially available solid modeling and animation packages still provide CSG as a standard user-interface.

References

- [1] Shirley, *Fundamentals of Computer Graphics*, 2002, School of Computing University of Utah
- [2] Hill, *Computer Graphics Using Open GL*, 1990, Macmillan Publishing Company
- [3] Farin, *Curves and Surfaces for CAGD*, A Practical Guide, Fifth Edition, 2002, Academic Press.
- [4] Rockwood, *Geometric Primitives*, 1997, CRC Press Inc.
- [5] Arvo, *Graphics Gems II*, Chapter IV.6, Academic Press, Inc, 1991.
- [6] Pappas and Murray, *Visual Basic Programming with the Windows API*, Chapter 15, Prentice Hall, 1998.

Geographic Information Systems

55.1	Geographic Information Systems: What They Are All About	55-1
	Geometric Objects • Topological versus Metric Data • Geometric Operations • Geometric Data Structures • Applications of Geographic Information	
55.2	Space Filling Curves: Order in Many Dimensions	55-5
	Recursively Defined Space Filling Curves • Range Queries for Space Filling Curve Data Structures • Are All Space Filling Curves Created Equal? • Many Curve Pieces for a Query Range • One Curve Piece for a Query Range	
55.3	Spatial Join	55-11
	External Algorithms • Advanced Issues	
55.4	Models, Toolboxes and Systems for Geographic Information	55-15
	Standardized Data Models • Commercial Systems • Research Prototypes	

Bernhard Seeger
University of Marburg

Peter Widmayer
ETH Zurich

55.1 Geographic Information Systems: What They Are All About

Geographic Information Systems (GISs) serve the purpose of maintaining, analyzing and visualizing spatial data that represent geographic objects, such as mountains, lakes, houses, roads, tunnels. For spatial data, geometric (spatial) attributes play a key role, representing e.g. points, lines, and regions in the plane or volumes in 3-dimensional space. They model geographical features of the real world, such as geodesic measurement points, boundary lines between adjacent pieces of land (in a cadastral database), lakes or recreational park regions (in a tourist information system). In three dimensions, spatial data describe tunnels, underground pipe systems in cities, mountain ranges, or quarries. In addition, spatial data in a GIS possess non-geometric, so-called thematic attributes, such as the time when a geodesic measurement was taken, the name of the owner of a piece of land in a cadastral database, the usage history of a park.

This chapter aims to highlight some of the data structures and algorithms aspects of GISs that define challenging research problems, and some that show interesting solutions. More background information and deeper technical expositions can be found in books such as [38, 64, 66, 75].

55.1.1 Geometric Objects

Our technical exposition will be limited to geometric objects with a vector representation. Here, a point is described by its coordinates in the Euclidean plane with a Cartesian coordinate system. We deliberately ignore the geographic reality that the earth is (almost) spherical, to keep things simple. A line segment is defined by its two end points. A polygonal line is a sequence of line segments with coinciding endpoints along the way. A (simple) polygonal region is described by its corner points, in clockwise (or counterclockwise) order around its interior. In contrast, in a raster representation of a region, each point in the region, discretized at some resolution, has an individual representation, just like a pixel in a raster image. Satellites take raster images at an amazing rate, and hence raster data abound in GISs, challenging current storage technology with terabytes of incoming data per day. Nevertheless, we are not concerned with raster images in this chapter, even though some of the techniques that we describe have implications for raster data [58]. The reason for our choice is the different flavor that operations with raster images have, as compared with vector data, requiring a chapter of their own.

55.1.2 Topological versus Metric Data

For some purposes, not even metric information is needed; it is sufficient to model the topology of a spatial dataset. *How many states share a border with Iowa?* is an example of a question of this topological type. In this chapter, however, we will not specifically study the implications that this limitation has. There is a risk of confusing the limitation to topological aspects only with the explicit representation of topology in the data structure. Here, the term explicit refers to the fact that a topological relationship need not be computed with substantial effort. As an example, assume that a partition of the plane into polygons is stored so that each polygon individually is a separate clockwise sequence of points around its interior. In this case, it is not easy to find the polygons that are neighbors of a given polygon, that is, share some boundary. If, however, the partition is stored so that each edge of the polygon explicitly references both adjacent polygons (just like the *doubly connected edge list* in computational geometry [62]), then a simple traversal around the given polygon will reveal its neighbors. It will always be an important design decision for a data structure which representation to pick.

55.1.3 Geometric Operations

Given this range of applications and geometric objects, it is no surprise that a GIS is expected to support a large variety of operations. We will discuss a few of them now, and then proceed to explain in detail how to perform two fundamental types of operations in the remainder of the chapter, spatial searching and spatial join. Spatial searching refers to rather elementary geometric operations without which no GIS can function. Here are a few examples, always working on a collection of geometric objects, such as points, lines, polygonal lines, or polygons. A *nearest neighbor query* for a query point asks for an object in the collection that is closest to the query point, among all objects in the collection. A *distance query* for a query point and a certain distance asks for all objects in the collection that are within the given distance from the query point. A *range query* (or *window query*) for a query range asks for all objects in the collection that intersect the given orthogonal window. A *ray shooting query* for a point and a direction asks for the object in the collection that is hit first if the ray starts at the given point and shoots in the given direction. A *point-in-polygon query* for a query point asks for all polygons in the collection in which the query

point lies. These five types of queries are illustrations only; many more query types are just as relevant. For a more extensive discussion on geometric operations, see the chapters on geometric data structures in this Handbook. In particular, it is well understood that great care must be taken in geometric computations to avoid numeric problems, because already tiny numerical inaccuracies can have catastrophic effects on computational results. Practically all books on geometric computation devote some attention to this problem [13, 49, 62], and geometric software libraries such as CGAL [11] take care of the problem by offering exact geometric computation.

55.1.4 Geometric Data Structures

Naturally, we can only hope to respond to queries of this nature quickly, if we devise and make use of appropriate data structures. An extra complication arises here due to the fact that GISs maintain data sets too large for internal memory. Data maintenance and analysis operations can therefore be efficient only if they take external memory properties into account, as discussed also in other chapters in this Handbook. We will limit ourselves here to external storage media with direct access to storage blocks, such as disks (for raster data, we would need to include tertiary storage media such as tapes). A block access to a random block on disk takes time to move the read-write-head to the proper position (the latency), and then to read or write the data in the block (the transfer). With today's disks, where block sizes are on the order of several kBytes, latency is a few milliseconds, and transfer time is less. Therefore, it pays to read a number of blocks in consecution, because they require the head movement only once, and in this way amortize its cost over more than one block. We will discuss in detail how to make use of this cost savings possibility.

All operations on an external memory geometric data structure follow the general *filter-refinement* pattern [54] that first, all relevant blocks are read from disk. This step is a first (potentially rough) *filter* that makes a superset of the relevant set of geometric objects available for processing in main memory. In a second step, a *refinement* identifies the exact set of relevant objects. Even though complicated geometric operators can make this refinement step quite time consuming, in this chapter we limit our considerations to the filter step. Because queries are the dominant operations in GISs by far, we do not explicitly discuss updates (see the chapters on [external memory spatial data structures](#) in this Handbook for more information).

55.1.5 Applications of Geographic Information

Before we go into technical detail, let us mention a few of the applications that make GISs a challenging research area up until today, with more fascinating problems to expect than what we can solve.

Map Overlay

Maps are the most well-known visualizations of geographical data. In its simplest form, a map is a partition of the plane into simple polygons. Each polygon may represent, for instance, an area with a specific thematic attribute value. For the attribute *land use*, polygons can stand for *forest*, *savanna*, *lake* areas in a simplistic example, whereas for the attribute *state*, polygons represent *Arizona*, *New Mexico*, *Texas*. In a GIS, each separable aspect of the data (such as the planar polygonal partitions just mentioned) is said to define a *layer*. This makes it easy to think about certain analysis and viewing operations, by just superimposing (overlying) layers or, more generally, by applying Boolean operations on

sets of layers. In our example, an overlay of a *land use* map with a *state* map defines a new map, where each new polygon is (some part of) the intersection of two given polygons, one from each layer. In *map overlay* in general, a Boolean combination of all involved thematic attributes together defines polygons of the resulting map, and one resulting attribute value in our example are the *savannas* of *Texas*. Map overlay has been studied in many different contexts, ranging from the special case of convex polygons in the partition and an internal memory plane-sweep computation [50] to the general case that we will describe in the context of spatial join processing later in this chapter.

Map Labeling

Map visualization is an entire field of its own (traditionally called cartography), with the general task to layout a map in such a way that it provides just the information that is desired, no more and no less; one might simply say, the map *looks right*. What that means in general is hard to say. For maps with texts that label cities, rivers, and the like, *looking right* implies that the labels are in proper position and size, that they do not run into each other or into important geometric features, and that it is obvious to which geometric object a label refers. Many simply stated problems in map labeling turn out to be NP-hard to solve exactly, and as a whole, map labeling is an active research area with a variety of unresolved questions (see [47] for a tutorial introduction).

Cartographic Generalization

If cartographers believe that automatically labeled maps will never look really good, they believe even more that another aspect that plays a role in map visualization will always need human intervention, namely map generalization. Generalization of a map is the process of reducing the complexity and contents of a map by discarding less important information and retaining the more essential characteristics. This is most prominently used in producing a map at a low resolution, given a map at a high resolution. Generalization ensures that the reader of the produced low resolution map is not overwhelmed with all the details from the high resolution map, displayed in small size in a densely filled area. Generalization is viewed to belong to the art of map making, with a whole body of rules of its own that can guide the artist [9, 46]. Nevertheless, computational solutions of some subproblem help a lot, such as the simplification of a high resolution polygonal line to a polygonal line with fewer corner points that does not deviate too much from the given line. For line simplification, old algorithmic ideas [16] have seen efficient implementations [28] recently. Maps on demand, with a selected viewing window, to be shown on a screen with given resolution, imply the choice of a corresponding scale and therefore need the support of data structures that allow the retrieval up to a desired degree of detail [4]. Apart from the simplest aspects, automatic map generalization and access support are open problems.

Road Maps

Maps have been used for ages to plan trips. Hence, we want networks of roads, railways, and the like to be represented in a GIS, in addition to the data described above. This fits naturally with the geometric objects that are present in a GIS in any case, such as polygonal lines. A point where polygonal lines meet (a node) can then represent an intersection of roads (edges), with a choice which road to take as we move along. The specialty in storing roads comes from the fact that we want to be able to find paths between nodes efficiently, for instance in car navigation systems, while we are driving. The fact that not all roads are equally important can be expressed by weights on the edges. Because a shortest path

computation is carried out as a breadth first search on the edge weighted graph, in one way or another (e.g. bidirectional), it makes sense to partition the graph into pages so as to minimize the weight of edges that cross the cuts induced by the partition. Whenever we want to maintain road data together with other thematic data, such as land use data, it also makes sense to store all the data in one structure, instead of using an extra structure for the road network. It may come as no surprise that for some data structures, partitioning the graph and partitioning the other thematic aspects go together very well (compromising a little on both sides), while for others this is not easily the case. The compromise in partitioning the graph does almost no harm, because it is NP-complete to find the optimum partition, and hence a suboptimal solution of some sort is all we can get anyway. Even though this type of heuristic approaches for maintaining road networks in GIS are useful [69], it is by no means clear whether this is the best that can be achieved.

Spatiotemporal Data

Just like for many other database applications, a *time* component brings a new dimension to spatial data (even in the mathematical sense of the word, if you wish). *How did the forest areas in New Mexico develop over the last 20 years?* Questions like this one demonstrate that for environmental information systems, a specific branch of GISs, keeping track of developments over time is a must. Spatiotemporal database research is concerned with all problems that the combination of space with time raises, from models and languages, all the way through data structures and query algorithms, to architectures and implementations of systems [36]. In this chapter, we refrain from the temptation to discuss spatiotemporal data structures in detail; see [Chapter 22](#) for an introduction into this lively field.

Data Mining

The development of spatial data over time is interesting not only for explicit queries, but also for data mining. Here, one tries to find relevant patterns in the data, without knowing beforehand the character of the pattern (for an introduction to the field of data mining, see [27]). Let us briefly look at a historic example for spatial data mining: A London epidemiologist identified a water pump as the centroid of the locations of cholera cases, and after the water pump was shut down, the cholera subsided. This and other examples are described in [68]. If we want to find patterns in quite some generality, we need a large data store that keeps track of data extracted from different data sources over time, a so-called data warehouse. It remains as an important, challenging open problem to efficiently run a spatial data warehouse and mine the spatial data. The spatial nature of the data seems to add the extra complexity that comes from the high autocorrelation present in typical spatial data sets, with the effect that most knowledge discovery techniques today perform poorly. This omnipresent tendency for data to cluster in space has been stated nicely [74]: *Everything is related to everything else but nearby things are more related than distant things*. For a survey on the state of the art and the challenges of spatial data mining, consult [70].

55.2 Space Filling Curves: Order in Many Dimensions

As explained above, our interest in space filling curves (SFCs) comes from two sources. The first one is the fact that we aim at exploiting the typical database support mechanisms that a conventional database management system (DBMS) offers, such as support for transactions and recovery. On the data structures level, this support is automatic if

we resort to a data structure that is inherently supported by a DBMS. These days, this is the case for a number of one dimensional data structures, such as those of the B-tree family (see [Chapter 15](#)). In addition, spatial database management systems support one of a small number of multidimensional data structures, such as those of the R-tree family (see [Chapter 21](#)) or of a grid based structure. The second reason for our interest in space filling curves is the general curiosity in the gap between one dimension and many: In what way and to what degree can we bridge this gap for the sake of supporting spatial operations of the kind described above? In our setting, the gap between one dimension and many goes back to the lack of a linear order in many dimensions that is useful for all purposes. A space filling curve tries to overcome this problem at least to some degree, by defining an artificial linear order on a regular grid that is as useful as possible for the intended purpose. Limiting ourselves in this section to two dimensional space, we define a space filling curve more formally as a bijective mapping p from an index pair of a grid cell to its number in the linear order:

$$p: N \times N \longrightarrow \{1, \dots, N^2\}.$$

For the sake of simplicity, we limit ourselves to numbers $N = 2^n$ for some positive integer n .

Our main attention is on the choice of the linear order in such a way that range queries are efficient. In the GIS setting, it is not worst case efficiency that counts, but efficiency in expectation. It is, unfortunately, hard to say what queries can be expected. Therefore, a number of criteria are conceivable according to which the quality of a space filling curve should be measured. Before entering the discussion about these criteria, let us present some of the most prominent space filling curves that have been investigated for GIS data structures.

55.2.1 Recursively Defined Space Filling Curves

Two space filling curves have been investigated most closely for the purpose of producing a linear ordering suitable for data structures for GIS, the z -curve, also called Peano curve or Morton encoding, and the Hilbert curve. We depict them in [Figs. 55.1](#) and [55.2](#). They can both be defined recursively with a simple refinement rule: To obtain a $2^{n+1} \times 2^{n+1}$ grid from a $2^n \times 2^n$ grid, replace each cell by the elementary pattern of four cells as in [Fig. 55.1](#), with the appropriate rotation for the Hilbert curve, as indicated in [Fig. 55.2](#) for the four by four grid (i.e., for $n = 2$). In the same way, a slightly less popular space filling curve can be defined, the Gray code curve (see [Fig. 55.3](#)).

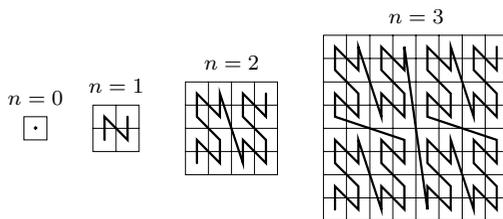


FIGURE 55.1: The z -curve for a $2^n \times 2^n$ grid, for $n = 0, \dots, 3$.

For all recursively defined space filling curves, there is an obvious way to compute the mapping p (and also its inverse), namely just along the recursive definition. Any such

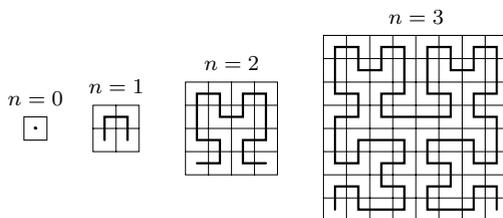


FIGURE 55.2: The Hilbert-curve for a $2^n \times 2^n$ grid, for $n = 0, \dots, 3$.

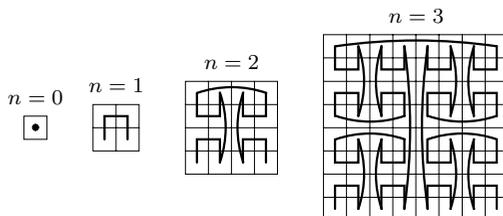


FIGURE 55.3: The Gray-curve for a $2^n \times 2^n$ grid, for $n = 0, \dots, 3$.

computation therefore takes time proportional to the logarithm of the grid size. Without going to great lengths in explaining how this computation is carried out in detail (see the beautiful book [65] for this and other mathematical aspects of space filling curves), let us just mention that there is a particularly nice way of viewing it for the z -curve: Here, p simply alternately interleaves the bits of both of its arguments, when these are expressed in binary. This may have made the z -curve popular among geographers at an early stage, even though our subsequent discussion will reveal that it is not necessarily the best choice.

55.2.2 Range Queries for Space Filling Curve Data Structures

A space filling curve defines a linear order that is used to store the geographical data objects. We distinguish between two extreme storage paradigms that can be found in spatial data structures, namely a partition of the data space according to the objects present in the data set (such as in R-trees, or in B-trees for a single dimension), or a partition of the space only, regardless of the objects (such as in regular cell partitions). The latter makes sense if the distribution of objects is somewhat uniform (see the [spatial data structures](#) chapters in this Handbook), and in this case achieves considerable efficiency. Naturally, a multitude of data structures that operate in between both extremes have been proposed, such as the grid file (see the [spatial data structures](#) chapters in this Handbook). As to the objects, let us limit ourselves to points in the plane, so that we can focus on a single partition of the plane into grid cells and need not worry about objects that cross cell boundaries (these are dealt with elsewhere in this Handbook, see e.g. the chapter on R-trees, or the survey chapter). For simplicity, let us restrict our attention to partitions of the data space into a $2^n \times 2^n$ grid, for some integer n . Partitions into other numbers of grid cells (i.e., not powers of four) can be achieved dynamically for some data structures (see e.g. z -Hashing, [32]), but are not always easy to obtain; we will ignore that aspect for now.

Corresponding to both extreme data structuring principles, there are two ways to associate data with external storage blocks for space filling curves. The simplest way is to identify a grid cell with a disk block. This is the usual setting. Grid cell numbers correspond to physical disk block addresses, and a range query translates into the need to access the

corresponding set of disk blocks. In the example of Fig. 55.4, for instance, the query range shown in bold results in the need to read disk blocks corresponding to cells with numbers 2-3, 6, 8-12, 14, that is, cell numbers come in four consecutive sequences. In the best case, the cells to be read have consecutive numbers, and hence a single disk seek operation will suffice, followed by a number of successive block read operations. In the worst case, the sequence of disk cell numbers to be read breaks into a large number of consecutive pieces. It is one concern in the choice of a space filling curve to bound this number, or some measure related to it (see the next subsection).

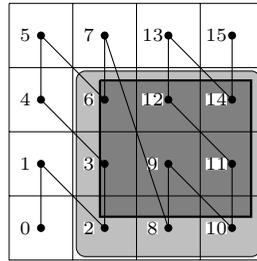


FIGURE 55.4: Range queries and disk seek operations.

The association of a grid cell with exactly one disk block gives a data structure based on a space filling curve very little flexibility to adapt to a not so uniform data set. A different and more adaptive way of associating disk blocks with grid cells is based on a partition of the data space into cells so that for each cell, the objects in the cell fit on a disk block (as before), but a disk block stores the union of the sets of objects in a consecutive number of cells. This method has the advantage that relatively sparsely populated cells can go together in one disk block, but has the potential disadvantage that disk block maintenance becomes more complex. Not too complex, though, because a disk block can simply be maintained as a node of a B-tree (or, more specifically, a leaf, depending on the B-tree variety), with the content of a cell limiting the granularity of the data in the node. Under this assumption, the adaptation of the data structure to a dynamically changing population of data objects simply translates to split and merge operations of B-tree nodes. In this setting, the measure of efficiency for range queries may well be different from the above: One might be interested in running a range query on the B-tree representation, and ignoring (skipping) the contents of retrieved cells that do not contribute to the result. Hence, for a range query to be efficient, the consecutive single piece of the space filling curve that includes all cells of the query range should not run outside the query range for too long, i.e., for too high a number of cells. We will discuss the effect of a requirement of this type on the design of a space filling curve in more detail below.

Obviously, there are many other ways in which a space filling curve can be the basis for a spatial data structure, but we will refrain from a more detailed discussion here and limit ourselves to the two extremes described so far.

55.2.3 Are All Space Filling Curves Created Equal?

Consider a space filling curve that visits the cells of a grid by visiting next an orthogonal neighbor of the cell just visited. The Hilbert curve is of this *orthogonal neighbor* type, but the *z-curve* is not. Now consider a square query region of some fixed (but arbitrary) size,

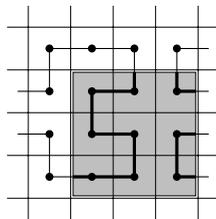


FIGURE 55.5: Disk seeks for an orthogonal curve.

say k by k grid cells, and study the number of consecutive pieces of the space filling curve that this query region defines (see Fig. 55.5 for a range query region of 3 by 3 grid cells that defines 3 consecutive pieces of a Hilbert curve). We are interested in the average number of curve pieces over all locations of the query range.

For a fixed location of the query range, the number of curve pieces that the range defines is half the number of orthogonal neighbor links that cross the range boundary (ignoring the starting or ending cell of the entire curve). The reason is that when we start at the first cell of the entire curve and follow the curve, it is an orthogonal neighbor on the curve that leads into the query range and another one that leads out, repeatedly until the query range is exhausted. To obtain the average number of curve pieces per query range, we sum the number of boundary crossing orthogonal neighbors over all query range locations (and then divide that number by twice the number of locations, but this step is of no relevance in our argument).

This sum, however, amounts to the same as summing up for every orthogonal neighbor link the number of range query locations in which it is a crossing link. We content ourselves for the sake of simplicity with an approximate average, by ignoring cells close to the boundary of the data space, an assumption that will introduce only a small inaccuracy whenever k is much less than N (and this is the only interesting case). Hence, each orthogonal link will be a crossing link for $2k$ query range positions, namely k positions for the query range on each of both sides. The interesting observation now is that this summation disregards the position of the orthogonal link completely (apart from the fact that we ignore an area of size $O(kN)$ along the boundary of the universe). Hence, for any square range query, the average number of pieces of a space filling curve in the query range is the same across all space filling curves of the orthogonal neighbor type.

In this sense, therefore, all space filling curves of the orthogonal neighbor type have the same quality. This includes snake like curves such as row-major zigzag snakes and spiral snakes. Orthogonal neighbors are not a painful limitation here: If we allow for other than orthogonal neighbors, this measure of quality can become only worse, because a non-orthogonal neighbor is a crossing link for more query range locations. In rough terms, this indicates that the choice of space filling curve may not really matter that much. Nevertheless, it also shows that the above measure of quality is not perfectly chosen, since it captures the situation only for square query ranges, and that is not fully adequate for GISs.

55.2.4 Many Curve Pieces for a Query Range

The properties that recursively definable space filling curves entail for spatial data structure design have been studied from a theoretical perspective [3, 8]. The performance of square range queries has been studied [3] on the basis that range queries in a GIS can have any size, but are most likely to not deviate from a square shape by too much. Based on the

external storage access model in which disk seek and latency time by far dominates the time to read a block, they allow a range query to skip blocks for efficiency's sake. Skipping a block amounts to reading it in a sequence of blocks, without making use of the information thus obtained. In the example of Fig. 55.4, skipping the blocks of cells 4, 5, 7, and 13 leads to a single sequence of the consecutive blocks 2-14. This can obviously be preferable to just reading consecutive sequences of relevant cell blocks only and therefore performing more disk seek operations, provided that the number of skipped blocks is not too large. In an attempt to quantify one against the other, consider a range query algorithm that is allowed to read a linear number of additional cells [3], as compared with those in the query range. It turns out that for square range query regions of arbitrary size and the permission to read at most a linear number of extra cells, each recursive space filling curve needs at least three disk seek operations in the worst case. While no recursive space filling curve needs more than four disk seeks in the worst case, none of the very popular recursive space filling curves, including the *z*-curve, the Hilbert curve, and the Gray code curve, can cope with less than four. One can define a recursive space filling curve that needs only three disk seeks in the worst case [3], and hence the lower and upper bounds match. This result is only a basis for data structure design, though, because its quality criterion is still too far from GIS reality to guide the design of practical space filling curves.

Along the same lines, one can refine the cost measure and account explicitly for disk seek cost [8]: For a sufficiently short sequence of cells, it is cheaper to skip them, but as soon as the sequence length exceeds a constant (that is defined by the disk seek cost, relative to the cost of reading a block), it is cheaper to stop reading and perform a disk seek. A refinement of the simple observation from above leads to performance formulas expressed in the numbers of links of the various types that the space filling curve uses, taking the relative disk seek cost into consideration [8]. It turns out that the local behavior of space filling curves can be modeled well by random walks, again perhaps an indication that at least locally, the choice of a particular space filling curve is not crucial for the performance.

55.2.5 One Curve Piece for a Query Range

The second approach described above reads a single consecutive piece of the space filling curve to respond to a range query, from the lowest numbered cell in the range to the highest numbered. In general, this implies that a high number of irrelevant cells will be read. As we explained above, such an approach can be attractive, because it allows immediately to make use of well established data structures such as the B-tree, including all access algorithms. We need to make sure, however, that the inefficiency that results from the extra accessed cells remains tolerable. Let us therefore now calculate this inefficiency approximately. To this end, we change our perspective and calculate for any consecutive sequence of cells along the space filling curve its shape in two-dimensional space. The shape itself may be quite complicated, but for our purpose a simple rectangular bounding box of all grid cells in question will suffice. The example in Fig. 55.6 shows a set of 3 and a set of 4 consecutive cells along the curve, together with their bounding boxes (shaded), residing in a corner of a Hilbert curve. Now, a simple measure of quality suggests itself: The fewer useless cells we get in the bounding box, the better. A quick thought reveals that this measure is all too crude, because it does not take into consideration that square ranges are more important than skinny ranges. This bias can be taken into account by placing a piece of the space filling curve into its smallest enclosing square (see the dotted outline of a 3 by 3 square in Fig. 55.6 for a bounding square of the 3 cells), and by taking the occupancy (i.e., percentage of relevant cells) of that square as the measure of quality. The question we now face is to bound the occupancy for a given space filling curve, across all possible pieces of the curve,

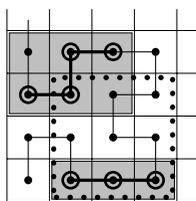


FIGURE 55.6: Bounding a piece of a curve.

and further, to find a best possible curve with respect to this bound. For simplicity's sake, let us again limit ourselves to curves with orthogonal links.

Let us first argue that the lower bound on the occupancy, for any space filling curve, cannot be higher than one third. This is easy to see by contradiction. Assume to this end that there is a space filling curve that guarantees an occupancy of more than one third. In particular, this implies that there cannot be two vertical links immediately after each other, nor can there be two consecutive horizontal links (the reason is that the three cells defining these two consecutive links define a smallest enclosing square of size 3 by 3 cells, with only 3 of these 9 cells being useful, and hence with an occupancy of only one third). But this finishes the argument, because no space filling curve can always alternate between vertical and horizontal links (the reason is that a corner of the space that is traversed by the curve makes two consecutive links of the same type unavoidable, see Fig. 55.7).

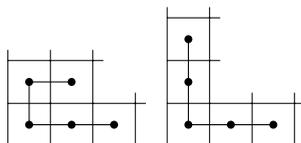


FIGURE 55.7: All possible corner cell traversals (excluding symmetry).

On the positive side, the Hilbert curve guarantees an occupancy of one third. The reason is that the Hilbert curve gives this guarantee for the 4 by 4 grid, and that this property is preserved in the recursive refinement.

This leads to the conclusion that in terms of the worst case occupancy guarantee, the Hilbert curve is a best possible basis for a spatial data structure.

55.3 Spatial Join

In order to compute map overlays, a GIS provides an operator called spatial join that allows flexible combinations of multiple inputs according to a spatial predicate. The spatial join computes a subset of the Cartesian product of the inputs and therefore, it is closely related to the join operator of a database management system (DBMS). A (binary) join on two sets of spatial objects, say R and S , according to a binary spatial predicate P is given by

$$SJ_P(R, S) = \{(r, s) \mid r \in R \wedge s \in S \wedge P(r, s)\}$$

The join is called a spatial join if the binary predicate P refers to spatial attributes of the objects. Among the most important ones are the following:

- Intersection predicate: $P_{\cap}(r, s) = (r \cap s \neq \emptyset)$.
- Distance predicate: Let DF be a distance function and $\epsilon > 0$. Then, $P_{DF}(r, s) = (DF(r, s) < \epsilon)$.

In the following we assume that R and S consist of N two-dimensional spatial objects and that the join returns a total of T pairs of objects. N is assumed being sufficiently large such that the problem cannot be solved entirely in main memory of size M . Therefore, we are particularly interested in external memory algorithms. Due to today's large main memory, the reader might question this assumption. However, remember that a GIS is a resource-intensive multi-user system where multiple complex queries are running concurrently. Thus, the memory assigned to a specific join algorithm might be substantially lower than the total physically available. Given a block size of B , we use the following notations: $n = N/B$, $m = M/B$ and $t = T/B$. Without loss of generality, we assume that B is a divisor of N , M and T .

A naive approach to processing a spatial join is simply to perform a so-called nested-loop algorithm, which checks the spatial predicate for each pair of objects. The nested-loop algorithm is generally applicable, but requires $O(N^2)$ time and $O(n^2)$ I/O operations. For special predicates, however, we are able to design new algorithms that provide substantial improvements in runtime compared to the naive approach. Among those is the intersection predicate, which is also the most frequently used predicate in GIS. We therefore will restrict our discussion to the intersection predicate to which the term spatial join will refer to by default. We postpone the discussion of other predicates to the end of the section.

The processing of spatial joins follows the general paradigm of multi-step query processing [55] that consists at least of the following two processing steps. In the filter step, a spatial join is performed using conservative approximations of the spatial objects like their minimum bounding rectangles (MBRs). In the refinement step, the candidates of the filter step are checked against the join predicate using their exact geometry. In this section, we limit our discussion on the filter step that utilizes the MBR of the spatial objects. The reader is referred to [6, 31] for a detailed discussion of the refinement step and intermediate processing steps that are additionally introduced.

If main memory is large enough to keep R and S entirely in memory, the filter step is (almost) equivalent to the rectangle intersection problem, one of the elementary problems in computational geometry. The problem can be solved in $O(N \log N + T)$ runtime where T denotes the size of the output. This can be accomplished by using either a sweep-line algorithm [72] or a divide-and-conquer algorithm [25]. However, the disadvantage of these algorithms is their high overhead that results in a high constant factor. For real-life spatial data, it is generally advisable, see [1] for example, to employ an algorithm that is not optimal in the worst-case.

55.3.1 External Algorithms

In this subsection, we assume that the spatial relations are larger than main memory. According to the availability of indices on both relations, on one of the relations, or on none of the relations, we obtain three different classes of spatial join algorithms for the filter step.

Index on both spatial relations

In [5, 7], spatial join algorithms were presented where each of the relations is indexed by an R-tree. Starting at the roots, this algorithm synchronously traverses the trees node by node and joins all pairs of overlapping MBRs. If the nodes are leaves, the associated

spatial objects are further examined in the refinement step. Otherwise, the algorithm is called recursively for each qualifying pair of entries. As pointed out in [20], this algorithm is not limited to R-trees, but can be applied to a broad class of index structures. Important to both, I/O and CPU cost, is the traversal strategy. In [7], a depth-first strategy was proposed where the qualifying pairs of nodes are visited according to a spatial ordering. If large buffers are available, a breadth-first traversal strategy was shown to be superior [30] in an experimental comparison. Though experimental studies have shown that these algorithms provide fast runtime in practice, the worst-case complexity of the algorithm is $O(n^2)$. The general problem of computing an optimal schedule for reading pages from disk is shown [48] to be NP-hard for spatial joins. For specific situations where two arbitrary rectangles from R and S do not share boundaries, the optimal solution can be computed in linear time. This is generally satisfied for bounding boxes of the leaves of R-trees.

Index on one spatial relation

Next, we assume that only one of the relations, say R , is equipped with a spatial index. We distinguish between the following three approaches. The first approach is to issue a range query against the index on R for each MBR of S . By using worst-case optimal spatial index structures, this already results in algorithms with subquadratic runtime. When a page buffer is available, it is also beneficial to sort the MBRs of S according to a criterion that preserves locality, e.g. Hilbert-value of the center of the MBRs. Then, two consecutive queries will access the same pages with high probability and therefore, the overall number of disk accesses decreases. The second approach as proposed in [40] first creates a spatial index on S , the relation without spatial index. The basic idea for the creation of the index is to use the upper levels of the available index on R as a skeleton. Thereafter, one of the algorithms is applied that requires an index on both of the relations. In [43] an improvement of the algorithm is presented that makes better use of the available main memory. A third approach is presented in [2, 22] where the spatial index is used for sorting the data according to one of the minimum boundaries of the rectangles. The sorted sequence then serves as input to an external plane-sweep algorithm.

Index on none of the inputs

There are two early proposals on spatial join processing that require no index. The first one [23] suggests using an external version of a computational geometry algorithm. This is basically achieved by employing an external segment tree. The other [54] can be viewed as a sweep-line algorithm where the ordering is derived from z -order. Though the algorithm was originally not designed for rectangles, this can be accomplished in a straightforward manner [15, 37]. Like any other sweep-line algorithm, R and S are first sorted, where the z -order serves as criterion in the following way. A rectangle receives the z -value of the smallest cell that still covers the entire rectangle. Let x be a z -value and $b(x) = (x_0, x_1, \dots, x_k)$ its binary representation where $k = l(x)$ denotes the level of the corresponding cell. Then, $x <_z y$, if $b(x) <_{lexi} b(y)$ where $<_{lexi}$ denotes the lexicographical order on strings. Note that if x is a prefix of y , x will precede y in lexicographical order. After sorting the input, the processing continues while maintaining a stack for each input. The stacks satisfy the following invariant: The z -value of each element (except the last) is a prefix of its successor. The algorithm simply takes the next element from the sorted inputs, say x from input R . Before x is pushed to the stack, all elements from both stacks are removed that are not prefix of $b(x)$. Thereafter, the entire stack of S is checked for overlap. The worst-case of the join occurs when each of the rectangles belongs to the cell that represents the entire data space. Then, the join runs like a nested loop and requires $O(n^2)$ I/O operations. This

can practically be improved by introducing redundancy [15,57]. However, the worst-case bound remains the same.

Other methods like the Partition Based Spatial-Merge Join (PBSM) [61] and the Spatial Hash-Join [41] are based on the principles of divide-and-conquer where spatial relations are partitioned into buckets that fit into main memory and the join is computed for each pair of corresponding buckets. Let us discuss PBSM in more detail. PBSM performs in four phases. In the first phase, the number of partitions p is computed such that the join for each pair of partitions is likely to be processed in main memory. Then, a grid is introduced with g cells, $g \geq p$ and each of the cells is then associated with a partition. A rectangle is then assigned to a partition if it intersects with at least one of its cells. In the second phase, pairs of partitions have to be processed that still contain too much data. This usually requires repartitioning of the data into smaller partitions. In the third phase, the join is processed in main memory for every pair of related partitions. The fourth phase consists of sorting, in order to get rid of duplicates in the response set. This however can be replaced by applying an inexpensive check of the result whenever it is produced [14]. Overall, the worst-case is $O(n^2)$ I/O operations for PBSM.

Spatial Hash-Joins [41] differ from PBSM in the following way. One of the relations is first partitioned using a spatial index structure like an R-tree which uniquely assigns rectangles to leaves, where each of them corresponds to a partition. The other relation is then partitioned using the index of the first relation. Each rectangle has to be stored in all partitions where an overlap with the MBR of the partition exists. Overall, this guarantees the avoidance of duplicate results.

At the end of the nineties, [1] proposed the first spatial join algorithm that meets the lower I/O bound $O(n \log_m n + t)$. The method is an external sweep-line algorithm that is also related to the divide-and-conquer algorithm of [23]. Rather than partitioning the problem recursively into two, it is proposed to partition the input recursively into $k = \sqrt{m}$ strips of almost equal size until the problem is small enough for being solved in memory. This results in a recursion tree of height $O(\log_m n)$. At each level $O(m)$ sorted lists of size $\Theta(B)$ are maintained where simultaneously an interval join is performed. Since each of the interval joins runs in $O(n' + t')$ (n' and t' denotes the input size and result size of the join, respectively) and at most $O(N)$ intervals are maintained on each level of the recursion, it follows that at most $O(n)$ accesses are required for each level.

Instead of using the optimal algorithm, [1] employs a plane-sweep algorithm in their experiments where the sweep-line is organized by a dynamic interval tree. This is justified by the observation that the sweep-line is small as it holds only a small fraction of the entire spatial relations. If the algorithm really runs out of memory, it is recommended invoking the optimal algorithm for the entire problem. A different overflow strategy has been presented in [34] where multiple iterations over the spatial relations might be necessary. The advantage of this strategy is that each answer is computed exactly once.

55.3.2 Advanced Issues

There are various extensions of the processing of spatial joins which will be discussed in the following. We first discuss the processing of spatial joins on multiple inputs. Next we discuss the processing of distance joins. Eventually, we conclude the section with a brief summary of requirements on the implementation of algorithms within a system.

The problem of joining more than two spatial relations according to a set of binary spatial predicates has been addressed in [42,59]. Such a join can be represented by a connected graph where the nodes correspond to the spatial relations and edges to binary join predicates. A first approach, called pairwise join method (PJM), is to decompose the

multi-way join into an operator tree of binary joins. The graphs with cycles therefore need to be transformed into a tree by ignoring the edges with lowest selectivity. Depending on the availability of spatial indexes, it is then advantageous to use from each class of spatial join algorithms the most efficient one. A different approach generalizes synchronous traversal to multiple inputs. The most important problem is not related to I/O, but to the processing cost of checking the join predicates. In [42] different strategies are examined for an efficient processing of the join predicates. Results of an experimental study revealed that neither PJM nor synchronous traversal performs best in all situations. Therefore, an algorithm is presented for computing a hybrid of these approaches by using dynamic programming.

In addition to the intersection predicate, there are many other spatial predicates that are of practical relevance for spatial joins in a GIS, see [60] for a survey. Among those, the distance predicate has received most attention. The distance join of R and S computes all pairs within a given distance. This problem has been even more extended in the following directions. First, pairs should be reported in an increasing order of the distances of the spatial objects. Second, only a fixed number of pairs should be reported. Third, answers should be produced on demand one at a time (without any limitations on the total number of answers). This problem has been addressed in [29, 71] where R-trees are assumed to be available on the spatial relations. The synchronized traversal is then controlled by two priority queues, where one maintains pairs of nodes according to their minimum distance and the other is primarily used for pruning irrelevant pairs of entries. In [71], it was recognized that there are many overlapping nodes which are not distinguishable in the priority queues. In order to break ties, a secondary ordering has been introduced that assigns a high priority to such pairs that are likely to contribute to the final result.

The design of algorithms for processing spatial joins largely depends on the specific system requirements. Similar to DBMSs, a complex query in a GIS is translated into an operator tree where nodes may correspond to spatial joins or other operators. There are two important problems that arise in this setting. First, an operator for processing spatial joins should have the ability to produce an estimation of the processing cost before the actual processing starts. Therefore, cost formulas are required that are inexpensive to compute, depend on only a few parameters of the input and produce sufficiently accurate estimations. This problem has recently attracted research attention [73]. Second, a demand-driven implementation of operators is generally required [19] where answers are lazily produced. This allows a pipelined processing of chains of operators where answers can continuously be delivered to the user without materializing them in advance. Therefore, join algorithms should be non-blocking, i.e., first answers should be produced without having consumed the entire input.

55.4 Models, Toolboxes and Systems for Geographic Information

GISs differ substantially with respect to their specific functionality, which makes a comparison of the different systems quite difficult. We restrict our evaluation to the core functionality of a GIS related to manipulating vector data. Moreover, we stress the following three criteria in our comparison:

Data Model: The spatial data model offered by the system is very important to a user since it provides the geometric data types and the operations.

Spatial indexing: Spatial index structures are important for efficiently supporting the most important spatial queries. It is therefore important what kind of index-

structures are actually available, particularly for applications that deal with very large databases.

Spatial Join: Since spatial joins are the most important operation for combining different maps, system performance depends on the efficiency of the underlying algorithm.

These criteria are among the most important ones for spatial query processing in a GIS [24].

Though we already have restricted our considerations to specific aspects, we limit our comparison to a few important systems and libraries. We actually start our discussion with introducing two common standardized data models that are implemented by many commercial systems. Thereafter, we will discuss a few commercial systems that are used in the context of GIS in industry. Next, we present a few prototype systems that mainly serve as research platforms.

55.4.1 Standardized Data Models

The most important standard for GIS [51] is published by the OpenGIS Consortium. It provides an object-oriented vector model and basic geometric data types. The actual implementations of commercial vendors like Oracle are closely related to the OpenGIS standard. All of the geometric data types are subclasses of the class *Geometry* that provides an attribute that specifies the spatial reference system. One of the methods of *Geometry* delivers the so-called envelope of an object that is called MBR in our terminology. The data model distinguishes between atomic geometric types like points, curves and surfaces and the corresponding collection types. The most complex atomic type is a polygonal surface that consists of an exterior polygonal ring and a set of internal polygonal rings where each of them represents a hole in the surface. Certain assertions have to be obeyed for such a polygonal surface to be in a consistent state.

The topological relationships of two spatial objects are expressed by using the nine-intersection model [17]. This model distinguishes between the exterior, interior and boundary of an object. Spatial predicates like overlaps are then defined by specifying which of the assertions has to be satisfied. In addition to predicates, the OpenGIS specification defines different constructive methods like the one for computing the convex hull of a spatial object. Another important function allows to compute the buffer object which contains those points that are within distance ε of a given object. Moreover, there are methods for computing the intersection (and other set operations) on two objects.

The OpenGIS standard has also largely influenced other standards for geographic data like the standard for storing, retrieving and processing spatial data using SQL [33] and the standard for the Geography Markup Language (GML) that is based on XML. The recently published version of the GML standard [52] additionally provides functionality to support three-dimensional objects and spatio-temporal applications.

55.4.2 Commercial Systems

In this subsection, we give a brief overview on the geographic query processing features of database systems, geographic information systems and data structures libraries.

Oracle

Among the three big database vendors, Oracle offers the richest support for the management of spatial data. The data model of Oracle [53] is similar to the simple feature model of the OpenGIS Consortium. Oracle additionally offers curves where the arcs might be circular. The spatial functionality is implemented on top of Oracle's DBMS and therefore, it is not fully integrated. This is most notably when SQL is used for specifying spatial queries where the declarative flavor of SQL is in contrast to the imperative procedural calls of the spatial functionality.

The processing of spatial queries is performed by using the filter-refinement approach. Moreover, intermediate filters might be employed where a kernel approximation of the object is used. This kind of processing is applied to spatial selection queries and to spatial joins.

There are R-trees and quadtrees in Oracle for indexing spatial data. In contrast to R-trees, (linear) quadtrees are based on a grid decomposition of the data space into tiles, each of them keep the list of intersecting objects. The linear quadtree is implemented within Oracle's B+-tree. In case of fixed indexing, the tiles are all of the same size. Oracle provides a function to enable users to determine a good setting of the tile size. In the case of hybrid indexing, tiles may vary in size. This is accomplished by locally increasing the grid resolution if the number of tiles is still below a given threshold. A comparison of Oracle's spatial index-structures [35] shows that the query performance of the R-tree is superior compared to the quadtree.

SpatialWare

SpatialWare [44] provides a set of functions that allow to manage spatial data within Microsoft SQL Server. The implementation is based on the extensibility features of SQL Server. Again, the spatial data types are similar to the simple features of OpenGIS.

The query processing functionality consists of spatial selection queries as well as spatial joins. Most notable is the fact that SpatialWare provides R-trees for spatial indexing.

LEDA and CGAL

LEDA [45] and CGAL [11] are C++-libraries (see [Chapter 41](#) for more on LEDA) that offer a rich collection of data structures and algorithms. Among the more advanced structures are spatial data types suitable for being used for the implementation of GIS. Most interesting to GIS is LEDA's and CGAL's ability to compute the geometry exactly by using a so-called rational kernel, i.e., spatial data types whose coordinates are rational numbers. LEDA provides the most important two-dimensional data types like points, iso-oriented rectangles, polygons and planar subdivisions. Moreover, LEDA provides efficient implementations of important geometric algorithms like convex hull, triangulations and line intersection. AlgoComs, a companion product of LEDA, also provides a richer functionality for polygons that is closely related to a map overlay. In contrast to LEDA, the focus of CGAL is limited to computational geometry algorithms where CGAL's functionality is generally richer in comparison to LEDA. CGAL contains kd-trees for indexing multidimensional point data and supports incremental nearest neighbor queries.

Both, LEDA and CGAL, do not support external algorithms and index-structures and therefore, they only partly cover the functionality required for a GIS. There has been an extension of LEDA, called LEDA-SM [12], that supports the most important external data structures.

JTS Topology Suite

The JTS Topology Suite (JTS) [76] is a Java class library providing fundamental geometric functions according to the geometry model defined by the OpenGIS Consortium [51]. Hence, it provides the basic spatial data types like polygonal surfaces and spatial predicates and operations like buffer and convex hull. The library also supports a user-definable precision model and contains code for robust geometric computation. There are also a few classes available for indexing MBRs (envelopes). The one structure is the MX-CIF quadtree [67] that is a specialized quadtree for organizing a dynamic set of rectangles. The other structure is a static R-tree that is created by using a bulk-loading technique [39]. Currently, there is no support for managing data on disk efficiently. JTS is published under an open source licensing agreement (the GNU LGPL).

55.4.3 Research Prototypes

SAND

The SAND System [18] gives the full query processing power of a spatial data base system and additionally, contains a browser for displaying the results of a spatial query. It provides the common folklore of spatial data types like point, rectangle, polygon and polygonal surface (termed polyregion). SAND offers three kinds of query predicates that refer to topological, metric and distance predicates, respectively. A user might ask for the sequence of objects within a given region ranked according to their distance to a given query point.

SAND is very powerful with respect to its indexing. SAND offers the PMR-quadtree [67] as well as the R-tree [26]. Both of these spatial index-structures support ranking queries by controlling the traversal of the index through a priority queue. Note that SAND delivers the answers of a query as an iterator where the next answer is produced on demand. SAND offers a rich source of spatial joins that are based on the principle of synchronized traversal of spatial indexes. A special feature of SAND is its query optimizer as well as its script language that serves as a glue between the different system components.

XXL

XXL (eXtensible and fleXible Library) [10] is not a system, but a pure Java library that does not support spatial data types, but points and rectangles. XXL provides powerful support for various kinds of (spatial) indexes. There are different kinds of implementations of R-trees as well as B-trees that might be combined with space-filling curves (e.g. *z*-order and Hilbert order). The concept of containers is introduced in XXL to provide an abstract view on external memory. Implementations of containers exist that are based on main memory, files and raw disks. XXL offers a rich source of different kinds of spatial joins like [14, 56] that are based on using space-filling curves and the sort-merge paradigm. XXL is equipped with an object-relational algebra of query operators and a query optimizer that is able to rewrite Java programs. Query operators are iterators that deliver the answers of a query on demand, one by one. XXL is available under an open source licensing agreement (GNU LGPL).

Dedale

The Dedale System [63] is unique in the sense that its underlying data model is based on constraints [64]. Rather than using a boundary representation, a set of constraints describes spatial objects of arbitrary dimensions. This also allows the usage of constrained-based languages for expressing spatial queries.

The latest version of Dedale is implemented on BASIS [21] that consists of the R*-tree as its spatial index structure and different spatial join algorithms that are able to exploit spatial indexes [22].

Acknowledgment

We gratefully acknowledge the assistance of Michael Cammert. Only with his help were we able to deliver the document (almost) in time.

References

- [1] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J.S. Vitter, *Scalable sweeping-based spatial join*, VLDB Conf., 1998, pp. 570–581.
- [2] ———, *A unified approach for indexed and non-indexed spatial joins*, Int. Conf. on Extending Database Tech., 2000, pp. 413–429.
- [3] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer, *Space filling curves and their use in the design of geometric data structures*, Theoretical Computer Science **18** (1997), no. 1, 3–15.
- [4] B. Becker, H.-W. Six, and P. Widmayer, *Spatial priority search: an access technique for scaleless maps*, ACM SIGMOD Conf., 1991, pp. 128–137.
- [5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, *The R*-tree: An efficient and robust access method for points and rectangles*, ACM SIGMOD Conf., 1990, pp. 322–331.
- [6] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger, *Multi-step processing of spatial joins*, ACM SIGMOD Conf., 1994, pp. 197–208.
- [7] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, *Efficient processing of spatial joins using R-trees*, ACM SIGMOD Conf., 1993, pp. 237–246.
- [8] E. Bugnion, T. Roos, R. Wattenhofer, and P. Widmayer, *Space filling curves versus random walks*, in Proc. Algorithmic Foundations of Geographic Information Systems (van Kreveld, Nievergelt, Roos, and Widmayer, eds.), Lecture Notes in Computer Science, vol. 1340, Springer, 1997.
- [9] B.P. Buttenfield and R.B. McMaster, *Map generalization: Making rules for knowledge representation*, Longman, 1991.
- [10] M. Cammert, C. Heinz, J. Krämer, M. Schneider, and B. Seeger, *A status report on XXL - a software infrastructure for efficient query processing*, IEEE Data Eng. Bull. **26** (2003), no. 2, 12–18.
- [11] CGAL, *Basic library, release 3.0*, <http://www.cgal.org/Manual>, 2003.
- [12] A. Crauser and K. Mehlhorn, *LEDA-SM: Extending LEDA to secondary memory*, Lecture Notes in Computer Science **1668** (1999), 228–242.
- [13] M.T. de Berg, M.J. van Kreveld, M.H. Overmars, and O. Schwarzkopf, *Computational geometry: Algorithms and applications (second edition)*, Springer, 2000.
- [14] J.-P. Dittrich and B. Seeger, *Data redundancy and duplicate detection in spatial join processing*, IEEE Int. Conf. on Data Eng., 2000, pp. 535–546.
- [15] J.-P. Dittrich and B. Seeger, *GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces*, ACM SIGKDD Conf., 2001, pp. 47–56.
- [16] D.H. Douglas and T.K. Peucker, *Algorithms for the reduction of the number of points required to represent a digitized line or its caricature*, Canadian Cartographer **10** (1973), no. 2, 112–122.
- [17] M.J. Egenhofer, *Reasoning about binary topological relations*, Symp. on Spatial Databases, Lecture Notes in Computer Science, vol. 525, Springer, 1991, pp. 143–160.

- [18] C. Esperanca and H. Samet, *Experience with SAND-Tcl: A scripting tool for spatial databases*, Journal of Visual Languages and Computing **13** (2002), no. 2, 229–255.
- [19] G. Graefe, *Query evaluation techniques for large databases*, ACM Computing Surveys **25** (1993), no. 2, 73–170.
- [20] O. Günther, *Efficient computation of spatial joins*, IEEE Int. Conf. on Data Eng., 1993, pp. 50–59.
- [21] C. Gurret, Y. Manolopoulos, A. Papadopoulos, and P. Rigaux, *The BASIS system: A benchmarking approach for spatial index structures*, Proceedings Spatio-Temporal Database Management Conference (STDBM'99), 1999, pp. 152–170.
- [22] C. Gurret and P. Rigaux, *The sort/sweep algorithm: A new method for R-Tree based spatial joins*, Statistical and Scientific Database Management, 2000, pp. 153–165.
- [23] R. H. Güting and W. Schilling, *A practical divide-and-conquer algorithm for the rectangle intersection problem*, Information Sciences **42** (1987), 95–112.
- [24] R.H. Güting, *An introduction to spatial database systems*, VLDB Journal **3** (1994), no. 4, 357–399.
- [25] R.H. Güting and D. Wood, *Finding rectangle intersections by divide-and-conquer*, IEEE Trans. on Computers **33** (1984), no. 7, 671–675.
- [26] A. Guttman, *R-trees: A dynamic index structure for spatial searching*, ACM SIGMOD Conf., 1984, pp. 47–57.
- [27] J. Han and M. Kamber, *Data mining: Concepts and techniques*, Morgan Kaufmann, 2001.
- [28] J. Hershberger and J. Snoeyink, *Cartographic line simplification and polygon CSG formulae in $O(n \log n)$ time*, Computational Geometry, vol. 11, 1998, pp. 175–185.
- [29] G.R. Hjaltason and H. Samet, *Incremental distance join algorithms for spatial databases*, ACM SIGMOD Conf., 1998, pp. 237–248.
- [30] Y.-W. Huang, N. Jing, and E.A. Rundensteiner, *Spatial joins using R-trees: Breadth-first traversal with global optimizations*, The VLDB Journal, 1997, pp. 396–405.
- [31] Y.-W. Huang, M.C. Jones, and E.A. Rundensteiner, *Improving spatial intersect joins using symbolic intersect detection*, SSD, Lecture Notes in Computer Science, vol. 1262, 1997, pp. 165–177.
- [32] A. Hutflesz, H.-W. Six, and P. Widmayer, *Globally order preserving multidimensional linear hashing*, IEEE Int. Conf. on Data Eng., 1988, pp. 572–579.
- [33] ISO/IEC 13249-3:2002 FDIS, *Information technology - database languages - SQL multimedia and application packages - part 3: Spatial, 2nd edition*, 2002.
- [34] E.H. Jacox and H. Samet, *Iterative spatial join*, TODS **28** (2003), no. 3, 230–256.
- [35] R.K.V. Kothuri, S. Ravada, and D. Abugov, *Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data*, ACM SIGMOD Conf., 2002, pp. 546–557.
- [36] M. Koubarakis, T.K. Sellis, A.U. Frank, S. Grumbach, R.H. Güting, C.S. Jensen, N.A. Lorentzos, Y. Manolopoulos, E. Nardelli, B. Pernici, H.-J. Schek, M. Scholl, B. Theodoulidis, and N. Tryfona (eds.), *Spatio-temporal databases: The Chorochronos Approach*, Lecture Notes in Computer Science, vol. 2520, Springer, 2003.
- [37] N. Koudas and K.C. Sevcik, *Size separation spatial join*, ACM SIGMOD Conf., 1997, pp. 324–335.
- [38] R. Laurini and D. Thompson, *Fundamentals of spatial information systems*, Academic Press, 1992.
- [39] S. Leutenegger, M. Lopez, and J. Edgington, *A simple and efficient algorithm for R-tree packing*, IEEE Int. Conf. on Data Eng., 1997, pp. 497–506.
- [40] M.-L. Lo and C. V. Ravishankar, *Spatial joins using seeded trees*, ACM SIGMOD Conf., 1994, pp. 209–220.

- [41] ———, *Spatial hash-joins*, ACM SIGMOD Conf., 1996, pp. 247–258.
- [42] N. Mamoulis and D. Papadias, *Multiway spatial joins*, TODS **26** (2001), no. 4, 424–475.
- [43] ———, *Slot index spatial join*, IEEE Trans. Knowl. Data Eng. **15** (2003), no. 1, 211–231.
- [44] MapInfo, *MapInfo SpatialWare for Microsoft SQL Server v4.8 user guide*, http://www.mapinfo.com/common/docs/spatialware/sw48_userguide.pdf, 2003.
- [45] K. Mehlhorn and S. Näher, *LEDA : A platform for combinatorial and geometric computing*, Cambridge University Press, 1999.
- [46] J.-C. Müller, J.-P. Lagrange, and R. Weibel, *GIS and generalization: Methodology and practice*, Taylor & Francis, London, 1995.
- [47] G. Neyer, *Map labeling with application to graph drawing*, Drawing Graphs (M. Kaufmann and D. Wagner, eds.), Lecture Notes in Computer Science, vol. 2025, Springer, 2001, pp. 247–273.
- [48] G. Neyer and P. Widmayer, *Singularities make spatial join scheduling hard*, Int. Symp. on Algorithms and Computation, Lecture Notes in Computer Science, vol. 1350, Springer, 1997, pp. 293–302.
- [49] J. Nievergelt and K. Hinrichs, *Algorithms and data structures: With applications to graphics and geometry*, Prentice Hall, 1993.
- [50] J. Nievergelt and F. Preparata, *Plane-sweep algorithms for intersecting geometric figures*, CACM **25** (1982), no. 10, 739–747.
- [51] Open GIS Consortium, *OpenGIS simple features specification for SQL, revision 1.1*, 1999.
- [52] ———, *OpenGIS Geography Markup Language (GML) Implementation Specification, Version 3.0*, 2003.
- [53] Oracle, *Oracle spatial users guide and reference release 9.2*, http://otn.oracle.com/products/spatial/spatial_doc_index.html, 2002.
- [54] J.A. Orenstein, *Spatial query processing in an object-oriented database system*, ACM SIGMOD Conf., 1986, pp. 326–336.
- [55] ———, *Redundancy in spatial databases*, ACM SIGMOD Conf., 1989, pp. 294–305.
- [56] ———, *Strategies for optimizing the use of redundancy in spatial databases*, Symp. on Spatial Databases, Lecture Notes in Computer Science, vol. 409, 1989, pp. 115–136.
- [57] ———, *An algorithm for computing the overlay of k-dimensional spaces*, Symp. on Spatial Databases, Lecture Notes in Computer Science, vol. 525, 1991, pp. 381–400.
- [58] R. Pajarola and P. Widmayer, *An image compression method for spatial search*, IEEE Transactions on Image Processing, vol. 9, 2000, pp. 357–365.
- [59] D. Papadias, N. Mamoulis, and Y. Theodoridis, *Constraint-based processing of multiway spatial joins*, Algorithmica **30** (2001), no. 2, 188–215.
- [60] D. Papadias, Y. Theodoridis, T. Sellis, and M. Egenhofer, *Topological relations in the world of minimum bounding rectangles*, ACM SIGMOD Conf., 1995, pp. 92–103.
- [61] J.M. Patel and D.J. DeWitt, *Partition based spatial-merge join*, ACM SIGMOD Conf., 1996, pp. 259–270.
- [62] F.P. Preparata and M.I. Shamos, *Computational geometry: An introduction*, Springer, 1985.
- [63] P. Rigaux, M. Scholl, L. Segoufin, and S. Grumbach, *Building a constraint-based spatial database system: Model, languages, and implementation*, Information Systems **28** (2003), no. 6, 563–595.
- [64] P. Rigaux, M.O. Scholl, and A. Voisard, *Spatial databases: With application to GIS*, Morgan Kaufmann, 2001.
- [65] H. Sagan, *Space filling curves*, Springer, 1994.

- [66] H. Samet, *Applications of spatial data structures*, Addison-Wesley, 1989.
- [67] ———, *The design and analysis of spatial data structures*, Addison-Wesley, 1989.
- [68] S. Shekhar and S. Chawla, *Spatial databases: A tour*, Prentice Hall, 2003.
- [69] S. Shekhar and D.-R. Liu, *CCAM: A connectivity-clustered access method for networks and network computations*, IEEE Trans. Knowl. Data Eng. **9** (1997), no. 1, 102–119.
- [70] S. Shekhar, P. Zhang, Y. Huang, and R. Vatsavai, *Trends in spatial data mining, to appear in: Data mining: Next generation challenges and future directions*, AAAI/MIT Press, 2004.
- [71] H. Shin, B. Moon, and S. Lee, *Tie-breaking strategies for fast distance join processing*, Data and Knowledge Engineering **41** (2002), no. 1, 67–83.
- [72] H.-W. Six and D. Wood, *The rectangle intersection problem revisited*, BIT **20** (1980), no. 4, 426–433.
- [73] Y. Theodoridis, E. Stefanakis, and T.K. Sellis, *Efficient cost models for spatial queries using R-Trees*, IEEE Trans. Knowl. Data Eng. **12** (2000), no. 1, 19–32.
- [74] W. Tobler, *Cellular geography*, Philosophy in Geography, 1979, pp. 379–386.
- [75] M.J. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer (eds.), *Algorithmic foundations of geographic information systems*, Lecture Notes in Computer Science, vol. 1340, Springer, 1997.
- [76] Vivid Solutions, *JTS topology suite technical specifications version 1.4*, <http://www.vividsolutions.com/jts/>, 2003.

Collision Detection

56.1	Introduction.....	56-1
56.2	Convex Polytopes	56-3
	Linear Programming • Voronoi-Based Marching Algorithm • Minkowski Sums and Convex Optimization	
56.3	General Polygonal Models	56-6
	Interference Detection using Trees of Oriented Bounding Boxes • Performance of Bounding Volume Hierarchies	
56.4	Penetration Depth Computation.....	56-11
	Convex Polytopes • Incremental Penetration Depth Computation • Non-Convex Models	
56.5	Large Environments	56-14
	Multiple-Object Collision Detection • Two-Dimensional Intersection Tests	

Ming C. Lin

University of North Carolina, Chapel Hill

Dinesh Manocha

University of North Carolina, Chapel Hill

56.1 Introduction

In a geometric context, collision detection refers to checking the relative configuration of two or more objects. The goal of collision detection, also known as interference detection or contact determination, is to automatically report a geometric contact when it is about to occur or has actually occurred. The objects may be represented as polygonal objects, spline or algebraic surfaces, deformable models, etc. Moreover, the objects may be static or dynamic.

Collision detection is a fundamental problem in computational geometry and frequently arises in different applications. These include:

1. **Physically-based Modeling and Dynamic Simulation:** The goal is to simulate dynamical systems and the physical behavior of objects subject to dynamic constraints. The mathematical model of the system is specified using geometric representations of the objects and the differential equations that govern the dynamics. The objects may undergo rigid or non-rigid motion. The contact interactions and trajectories of the objects are affected by collisions. It is important to model object interactions precisely and compute all the contacts accurately [1].
2. **Motion Planning:** The goal of motion planning is to compute a collision free path for a robot from a start configuration to a goal configuration. Motion planning is a fundamental problem in algorithmic robotics [2]. Most of the practical algorithms for motion planning compute different configurations of the robot and

check whether these configurations are collision-free, i.e. no collision between the robot and the objects in the environment. For example, probabilistic roadmap planners can spend up to 90% of the running time in collision checking.

3. **Virtual Environments and Walkthroughs:** A large-scale virtual environment, like a walkthrough, creates a computer-generated world, filled with real, simulated or virtual entities. Such an environment should give the user a feeling of presence, which includes making the images of both the user and the surrounding objects feel solid. For example, the objects should not pass through each other, and things should move as expected when pushed, pulled or grasped. Such actions require accurate and interactive collision detection. Moreover, runtime performance is critical in a virtual environment and all collision computations need to be performed at less than 1/30th of a second to give a sense of presence [3].
4. **Haptic Rendering:** Haptic interfaces, or force feedback devices, improve the quality of human-computer interaction by accommodating the sense of touch. In order to maintain a stable haptic system while displaying smooth and realistic forces and torques, haptic update rates must be as high as 1000 Hz. This involves accurately computing all contacts between the object attached to the probe and the simulated environment, as well as the restoring forces and torques – all in less than one millisecond [4].

In each of these applications, collision detection is one of the major computational bottlenecks.

Collision detection has been extensively studied in the literature for more than four decades. Hundreds of papers have been published on different aspects in computational geometry and related areas like robotics, computer graphics, virtual environments and computer-aided design. Most of the algorithms are designed to check whether a pair of objects collide. Some algorithms have been proposed for large environments composed of multiple objects and perform some form of culling or localize pairs of objects that are potentially colliding. At a broad level, different algorithms for collision detection can be classified based on the following characteristics:

- **Query Type:** The basic collision query checks whether two objects, described as set of points, polygons or other geometric primitives, overlap. This is the boolean form of the query. The enumerative version of the query yields some representations of the intersection set. Other queries compute the separation distance between two non-overlapping objects or the penetration distance between two overlapping objects.
- **Object Types:** Different algorithms have been proposed for convex polytopes, general polygonal models, curved objects described using parametric splines or implicit functions, set-theoretic combinations of objects, deformable models, etc.
- **Motion Formulation:** The collision query can be augmented by adding the element of time. If the trajectories of two objects are known, then we can determine when is the next time that a particular boolean query will become true or false. These queries are called *dynamic queries*, whereas the ones that do not use motion information are called *static queries*. In the case where the motion of an object can not be represented as a closed form function of time, the underlying application often performs static queries at specific time steps in the application.

In this chapter, we give a brief survey of different collision detection algorithms for convex polytopes, general polygonal models, penetration computations and large-scaled environments composed of multiple objects. In each category, we give a detailed description of one

of the algorithms and the underlying data structures.

56.2 Convex Polytopes

In this section, we give a brief survey of algorithms for collision detection between a pair of convex polytopes. This problem has been extensively studied and a number of algorithms with good asymptotic performance have been proposed. The best known runtime algorithm for boolean collision queries takes $O(\log^2 n)$ time, where n is the number of features [5]. It precomputes the Dobkin-Kirkpatrick hierarchy for each polytope and uses it to perform the runtime query. In practice, three classes of algorithms are commonly used for convex polytopes. These are linear programming, Minkowski sums, and tracking closest features based on Voronoi diagrams.

56.2.1 Linear Programming

The problem of checking whether two convex polytopes intersect or not can be posed as a linear programming (LP) problem. In particular, two convex polytopes do not overlap, if and only if there exists a separation plane between them. The coefficients of the separation plane equation are treated as unknowns. The linear constraints are formulated by imposing that all the vertices of the first polytope lie in one half-space of this plane and those of the other polytope lie in the other half-space. The linear programming algorithms are used to check whether there is any feasible solution to the given set of constraints. Given the fixed dimension of the problem, some of the well-known linear programming algorithms [6] can be used to perform the boolean collision query in expected linear time.

56.2.2 Voronoi-Based Marching Algorithm

An expected constant time algorithm for collision detection was proposed by Lin and Canny [7, 8]. This algorithm tracks the closest features between two convex polytopes. The features may correspond to a vertex, an edge or a face of each polytope. Variants of this algorithm have also been presented in [9, 10]. The original algorithm basically works by traversing the external Voronoi regions induced by the features of each convex polyhedron toward the pair of the closest features between the two given polytopes. The invariant is that at each step, either the inter-feature distance is reduced or the dimensionality of one or both of the features decreases by one, i.e. a move from a face to an edge or from an edge to a vertex.

The algorithm terminates when the pair of testing features contain a pair of points that lie within the Voronoi regions of the other feature. It returns the pair of closest features and the Euclidean distance between them, as well as the contact status (i.e. colliding or not). This algorithm uses a modified boundary representation to represent convex polytopes and a data structure for describing “*Voronoi regions*” of convex polytopes.

Polytope Representation

Let A be a polytope. A is partitioned into “*features*” f_1, \dots, f_n where n is the total number of features, i.e. $n = f + e + v$ where f, e, v stands for the total number of faces, edges, vertices respectively. Each feature (except vertex) is an open subset of an affine plane and does not contain its boundary.

Definition: B is in the *boundary* of F and F is in *coboundary* of B , if and only if B is in the closure of F , i.e. $B \subseteq \bar{F}$ and B has one fewer dimension than F does.

For example, the coboundary of a vertex is the set of edges touching it and the coboundary of an edge are the two faces adjacent to it. The boundary of a face is the set of edges in the closure of the face. It uses winged edge representation, commonly used for boundary evaluation of boolean combinations of solid models [11]. The edge is oriented by giving two incident vertices (the head and tail). The edge points from tail to head. It has two adjacent faces cobounding it as well. Looking from the the tail end toward the head, the adjacent face lying to the right hand side is labeled as the “right face” and similarly for the “left face”.

Each polytope’s data structure has a field for its features (faces, edges, vertices) and Voronoi cells to be described below. Each feature is described by its geometric parameters. Its data structure also includes a list of its boundary, coboundary, and *Voronoi regions*.

Definition: A *Voronoi region* associated with a *feature* is a set of points exterior to the polyhedron which are closer to that feature than any other. The Voronoi regions form a partition of space outside the polyhedron according to the closest feature. The collection of Voronoi regions of each polyhedron is the generalized Voronoi diagram of the polyhedron. Note that the Voronoi diagram of a convex polyhedron has linear size and consists of polyhedral regions.

Definition: A Voronoi *cell* is the data structure for a Voronoi region. It has a set of constraint planes that bound its Voronoi region with pointers to the neighboring cells (each of which shares a common constraint plane with the given Voronoi cell) in its data structure.

Using the geometric properties of convex sets, “applicability criteria” are established based upon the Voronoi regions. That is, if a point P on object A lies inside the Voronoi region of f_B on object B , then f_B is a closest feature to the point P . If a point lies on a constraint plane, then it is equi-distant from the two features that share this constraint plane in their Voronoi cells.

Local Walk

The algorithm incrementally traverses the features of each polytope to compute the closest features. For example, given a pair of features, *Face 1* and *vertex V_a* on objects A and B , respectively, as the pair of initial features (Fig. 56.2.2). The algorithm verifies if the vertex V_a lies within *Cell 1* of *Face 1*. However, V_a violates the constraint plane imposed by CP of *Cell 1*, i.e. V_a does not line in the half-space defined by CP which contains *Cell 1*. The constraint plane CP has a pointer to its adjacent cell *Cell 2*, so the walk proceeds to test whether V_a is contained within *Cell 2*. In similar fashion, vertex V_a has a cell of its own, and the algorithm checks whether the nearest point P_a on the edge to the vertex V_a lies within V_a ’s Voronoi cell. Basically, the algorithm checks whether a point is contained within a Voronoi region defined by the constraint planes of the region. The constraint plane, which causes this test to fail, points to the next pair of closest features. Eventually, the algorithm computes the closest pair of features.

Since the polytopes and its faces are convex, the containment test involves only the neighboring features of the current candidate features. If either feature fails the test, the algorithm steps to a neighboring feature of one or both candidates, and tries again. With some simple preprocessing, the algorithm can guarantee that every feature of a polytope has a constant number of neighboring features. As a result, it takes a constant number of operations to check whether two features are the closest features.

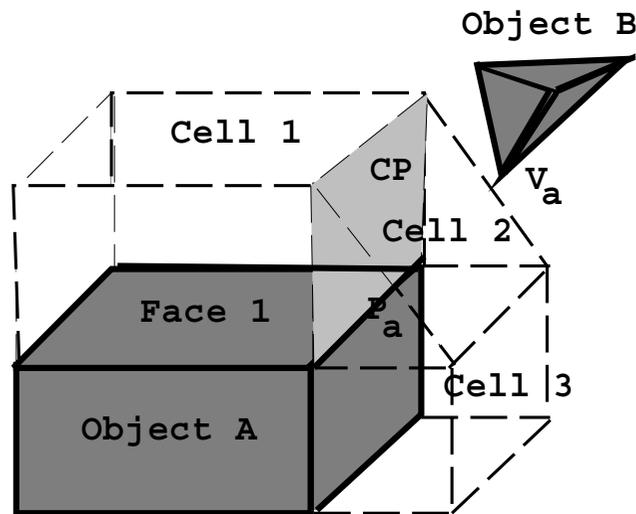


FIGURE 56.1: A walk across Voronoi cells. Initially, the algorithm checks whether the vertex V_a lies in *Cell 1*. After it fails the containment test with respect to the plane CP , it walks to *Cell 2* and checks for containment in *Cell 2*.

This approach can be used in a static environment, but is especially well-suited for dynamic environments in which objects move in a sequence of small, discrete steps. The method takes advantage of coherence within two successive static queries: i.e. the closest features change infrequently as the polytopes move along finely discretized paths. The closest features computed from the previous positions of the polytopes are used as the initial features for the current positions. The algorithm runs in *expected constant time* if the polytopes are not moving quickly. Even when a closest feature pair is changing rapidly, the algorithm takes only slightly longer. The running time is proportional to the number of feature pairs traversed, which is a function of the relative motion the polytopes undergo.

Implementation and Application

The Lin-Canny algorithm has been implemented as part of several public-domain libraries, include I-COLLIDE and SWIFT [12]. It has been used for different applications including dynamic simulation [13], interactive walkthrough of architectural models [3] and haptic display (including force and torque computation) of polyhedral models [14].

56.2.3 Minkowski Sums and Convex Optimization

The collision and distance queries can be performed based on the Minkowski sum of two objects. Given two sets of points, P and Q , their Minkowski sum is the set of points:

$$\{\mathbf{p} + \mathbf{q} \mid \mathbf{p} \in P, \mathbf{q} \in Q.\}$$

It has been shown in [15], that the minimum separation distance between two objects is the same as the minimum distance from the origin of the Minkowski sums of A and $-B$ to the surface of the sums. The Minkowski sum is also referred to as the *translational C-space obstacle* (TCSO) If we take the TCSO of two convex polyhedra, A and B , then the TCSO is another convex polyhedra, and each vertex of the TCSO correspond to the vector difference

of a vertex from A and a vertex from B . While the Minkowski sum of two convex polytopes can have $O(n^2)$ features [16], a fast algorithm for separation distance computation based on convex optimization that exhibits linear-time performance in practice has been proposed by Gilbert et al. [17]. It is also known as the GJK algorithm. It uses pairs of vertices from each object that define simplices (i.e. a point, bounded line, triangle or tetrahedron) within each polyhedra and a corresponding simplex in the TCSO. Initially the simplex is set randomly and the algorithm refines it using local optimization, till it computes the closest point on the TCSO from the origin of the Minkowski sums. The algorithm assumes that the origin is not inside the TCSO.

By taking the similar philosophy as the Lin-Canny algorithm [8], Cameron [18] presented an extension to the basic GJK algorithm by exploiting motion coherence and geometric locality in terms of connectivity between neighboring features. It keeps track of the *witness points*, a pair of points from the two objects that realize the minimum separation distance between them. As opposed to starting from a random simplex in the TCSO, the algorithm starts with the witness points from the previous iteration and performs hill climbing to compute a new set of witness points for the current configuration. The running time of this algorithm is a function of the number of refinement steps that the algorithm has to perform.

56.3 General Polygonal Models

Algorithms for collision and separation distance queries between general polygons models can be classified based on the fact whether they are closed polyhedral models or represented as a collection of polygons. The latter, also referred to as “polygon soups”, make no assumption related to the connectivity among different faces or whether they represent a closed set.

Some of the commonly known algorithms for collision detection and separation distance computation use spatial partitioning or bounding volume hierarchies (BVHs). The spatial subdivisions are a recursive partitioning of the embedding space, whereas bounding volume hierarchies are based on a recursive partitioning of the primitives of an object. These algorithms are based on the divide-and-conquer paradigm. Examples of spatial partitioning hierarchies include k-D trees and octrees [19], R-trees and their variants [20], cone trees, BSPs [21] and their extensions to multi-space partitions [22]. The BVHs use bounding volumes (BVs) to bound or contain sets of geometric primitives, such as triangles, polygons, curved surfaces, etc. In a BVH, BVs are stored at the internal nodes of a tree structure. The root BV contains all the primitives of a model, and children BVs each contain separate partitions of the primitives enclosed by the parent. Each of the leaf node BVs typically contains one primitive. In some variations, one may place several primitives at a leaf node, or use several volumes to contain a single primitive. The BVHs are used to perform collision and separation distance queries. These include sphere-trees [23, 24], AABB-trees [20, 25, 26], OBB-trees [27–29], spherical shell-trees [30, 31], k -DOP-trees [32, 33], SSV-trees [34], and convex hull-trees [35]. Different BVHs can be classified based on:

- **Choice of BV:** The AABB-tree uses an axis-aligned bounding box (AABB) as the underlying BV. The AABB for a set of primitives can be easily computed from the extremal points along the X , Y and Z direction. The sphere tree uses a sphere as the underlying BV. Algorithms to compute a minimal bounding sphere for a set of points in 3D are well known in computational geometry. The k -DOP-tree is an extension of the AABB-tree, where each BV is computed from extremal points along k fixed directions, as opposed to the 3 orthogonal axes.

A spherical shell is a subset of the volume contained between two concentric spheres and is a tight fitting BV. A SSV (swept sphere volume) is defined by taking the Minkowski sum of a point, line or a rectangle in 3D with a sphere. The SSV-tree corresponds to a hybrid hierarchy, where the BVs may correspond to a point-swept sphere (PSS), a line-swept sphere (LSS) or a rectangle-swept sphere (RSS). Finally, the BV in the convex-hull tree is a convex polytope.

- **Hierarchy generation:** Most of the algorithms for building hierarchies fall into two categories: bottom-up and top-down. Bottom-up methods begin with a BV for each primitive and merge volumes into larger volumes until the tree is complete. Top-down methods begin with a group of all primitive, and recursively subdivide until all leaf nodes are indivisible. In practice, top-down algorithms are easier to implement and typically take $O(n \lg n)$ time, where n is the number of primitives. On the other hand, the bottom-up methods use clustering techniques to group the primitives at each level and can lead to tighter-fitting hierarchies.

The collision detection queries are performed by traversing the BVHs. Two models are compared by recursively traversing their BVHs in tandem. Each recursive step tests whether BVs A and B , one from each hierarchy, overlap. If A and B do not overlap, the recursion branch is terminated. But if A and B overlap, the enclosed primitives may overlap and the algorithm is applied recursively to their children. If A and B are both leaf nodes, the primitives within them are compared directly. The running time of the algorithm is dominated by the overlap tests between two BVs and a BV and a primitive. It is relatively simple to check whether two AABBs overlap or two spheres overlap or two k-DOPs overlap. Specialized algorithms have also been proposed to check whether two OBBs, two SSVs, two spherical shells or two convex polytopes overlap. Next, we described a commonly used interference detection algorithm that uses hierarchies of oriented bounding boxes.

56.3.1 Interference Detection using Trees of Oriented Bounding Boxes

In this section we describe an algorithm for building a BVH of OBBs (called OBBTree) and using them to perform fast interference queries between polygonal models. More details about the algorithm are available in [27, 29].

The underlying algorithm is applicable to all triangulated models. They need not represent a compact set or have a manifold boundary representation. As part of a preprocess, the algorithm computes a hierarchy of oriented bounding boxes (OBBs) for each object. At runtime, it traverses the hierarchies to check whether the primitives overlap.

OBBTree Construction

An OBBTree is a bounding volume tree of OBBs. Given a collection of triangles, the algorithm initially approximates them with an OBB of similar dimensions and orientation. Next, it computes a hierarchy of OBBs.

The OBB computation algorithm makes use of first and second order statistics summarizing the vertex coordinates. They are the mean, μ , and the covariance matrix, \mathbf{C} , respectively [36]. If the vertices of the i 'th triangle are the points \mathbf{p}^i , \mathbf{q}^i , and \mathbf{r}^i , then the mean and covariance matrix can be expressed in vector notation as:

$$\mu = \frac{1}{3n} \sum_{i=0}^n (\mathbf{p}^i + \mathbf{q}^i + \mathbf{r}^i),$$

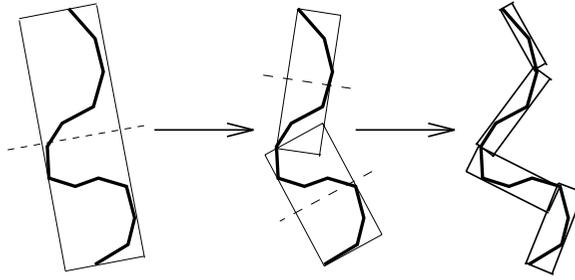


FIGURE 56.2: Building the OBBTree: recursively partition the bounded polygons and bound the resulting groups. This figure shows three levels of an OBBTree of a polygonal chain.

$$\mathbf{C}_{jk} = \frac{1}{3n} \sum_{i=0}^n (\bar{\mathbf{p}}_j^i \bar{\mathbf{p}}_k^i + \bar{\mathbf{q}}_j^i \bar{\mathbf{q}}_k^i + \bar{\mathbf{r}}_j^i \bar{\mathbf{r}}_k^i), \quad 1 \leq j, k \leq 3$$

where n is the number of triangles, $\bar{\mathbf{p}}^i = \mathbf{p}^i - \mu$, $\bar{\mathbf{q}}^i = \mathbf{q}^i - \mu$, and $\bar{\mathbf{r}}^i = \mathbf{r}^i - \mu$. Each of them is a 3×1 vector, e.g. $\bar{\mathbf{p}}^i = (\bar{p}_1^i, \bar{p}_2^i, \bar{p}_3^i)^T$ and \mathbf{C}_{jk} are the elements of the 3 by 3 covariance matrix.

The eigenvectors of a symmetric matrix, such as \mathbf{C} , are mutually orthogonal. After normalizing them, they are used as a basis. The algorithm finds the extremal vertices along each axis of this basis. Two of the three eigenvectors of the covariance matrix are the axes of maximum and of minimum variance, so they will tend to align the box with the geometry of a tube or a flat surface patch.

The algorithm's performance can be improved by using the convex hull of the vertices of the triangles. To get a better fit, we can sample the surface of the convex hull densely, taking the mean and covariance of the sample points. The uniform sampling of the convex hull surface normalizes for triangle size and distribution.

One can sample the convex hull "infinitely densely" by integrating over the surface of each triangle, and allowing each differential patch to contribute to the covariance matrix. The resulting integral has a closed form solution. Let the area of the i 'th triangle in the convex hull be denoted by

$$A^i = \frac{1}{2} |(\mathbf{p}^i - \mathbf{q}^i) \times (\mathbf{p}^i - \mathbf{r}^i)|$$

Let the surface area of the entire convex hull be denoted by

$$A^H = \sum_i A^i$$

Let the centroid of the i 'th convex hull triangle be denoted by

$$\mathbf{c}^i = (\mathbf{p}^i + \mathbf{q}^i + \mathbf{r}^i)/3$$

Let the centroid of the convex hull, which is a weighted average of the triangle centroids (the weights are the areas of the triangles), be denoted by

$$\mathbf{c}^H = \frac{\sum_i A^i \mathbf{c}^i}{\sum_i A^i} = \frac{\sum_i A^i \mathbf{c}^i}{A^H}$$

The elements of the covariance matrix \mathbf{C} have the following closed-form,

$$\mathbf{C}_{jk} = \sum_{i=1}^n \frac{A^i}{12A^H} (9\mathbf{c}_j^i \mathbf{c}_k^i + \mathbf{p}_j^i \mathbf{p}_k^i + \mathbf{q}_j^i \mathbf{q}_k^i + \mathbf{r}_j^i \mathbf{r}_k^i) - \mathbf{c}_j^H \mathbf{c}_k^H$$

Given an algorithm to compute tight-fitting OBBs around a group of polygons, we need to represent them hierarchically. The simplest algorithm for OBBTree computation uses a top-down method. It is based on a subdivision rule that splits the longest axis of a box with a plane orthogonal to one of its axes, partitioning the polygons according to which side of the plane their center point lies on (a 2-D analog is shown in Figure 56.3.1). The subdivision coordinate along that axis is chosen to be that of the mean point, μ , of the vertices. If the longest axis cannot be subdivided, the second longest axis is chosen. Otherwise, the shortest one is used. If the group of polygons cannot be partitioned along any axis by this criterion, then the group is considered indivisible.

Given a model with n triangles, the overall time to build the tree is $O(n \lg^2 n)$ if we use convex hulls, and $O(n \lg n)$ if we don't. The recursion is similar to that of quicksort. Fitting a box to a group of n triangles and partitioning them into two subgroups takes $O(n \lg n)$ with a convex hull and $O(n)$ without it. Applying the process recursively creates a tree with leaf nodes $O(\lg n)$ levels deep.

Interference Detection

Given OBBTrees of two objects, the interference algorithm typically spends most of its time testing pairs of OBBs for overlap. The algorithm computes axial projections of the bounding boxes and check for disjointness along those axes. Under this projection, each box forms an interval on the axis (a line in 3D). If the intervals don't overlap, then the axis is called a 'separating axis' for the boxes, and the boxes must then be disjoint.

It has been shown that we need to perform at most 15 axial projections in 3D to check whether two OBBs overlap or not [29]. These 15 directions correspond to the face normals of each OBB, as well as 9 pairwise combinations obtained by taking the cross-product of the vectors representing their edges.

To perform the test, the algorithm projects the centers of the boxes onto the axis, and also to compute the radii of the intervals. If the distance between the box centers as projected onto the axis is greater than the sum of the radii, then the intervals (and the boxes as well) are disjoint. This is shown in 2D in Fig. 56.3.1.

OBB Representation and Overlap Test

We assume we are given two OBBs, A and B , with B placed relative to A by rotation matrix \vec{R} and translation vector \vec{T} . The half-dimensions (or 'radii') of A and B are a_i and b_i , where $i = 1, 2, 3$. We will denote the axes of A and B as the unit vectors \vec{A}^i and \vec{B}^i , for $i = 1, 2, 3$. These will be referred to as the 6 box axes. Note that if we use the box axes of A as a basis, then the three columns of \vec{R} are the same as the three \vec{B}^i vectors.

The centers of each box projects onto the midpoint of its interval. By projecting the box radii onto the axis, and summing the length of their images, we obtain the radius of the interval. If the axis is parallel to the unit vector \vec{L} , then the radius of box A 's interval is

$$r_A = \sum_i |a_i \vec{A}^i \cdot \vec{L}|$$

A similar expression is used to compute r_B .

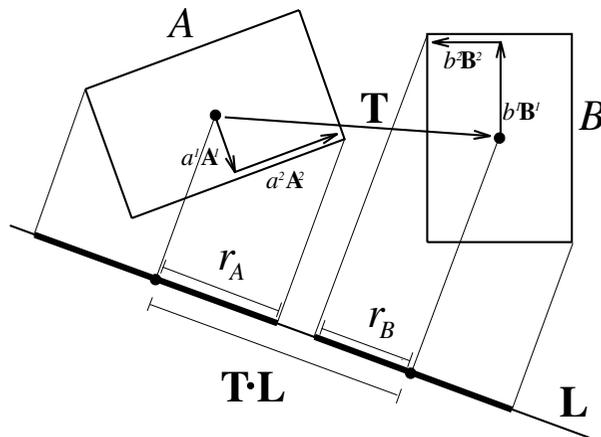


FIGURE 56.3: \vec{L} is a separating axis for OBBs A and B because A and B become disjoint intervals under projection (shown as r_A and r_B , respectively) onto \vec{L} .

The placement of the axis is immaterial, so we assume it passes through the center of box A . The distance between the midpoints of the intervals is $|\vec{T} \cdot \vec{L}|$. So, the intervals are disjoint if and only if

$$|\vec{T} \cdot \vec{L}| > \sum_i |a_i \vec{A}^i \cdot \vec{L}| + \sum_i |b_i \vec{B}^i \cdot \vec{L}|$$

This simplifies when \vec{L} is a box axis or cross product of box axes. For example, consider $\vec{L} = \vec{A}^1 \times \vec{B}^2$. The second term in the first summation is

$$\begin{aligned} |a_2 \vec{A}^2 \cdot (\vec{A}^1 \times \vec{B}^2)| &= |a_2 \vec{B}^2 \cdot (\vec{A}^2 \times \vec{A}^1)| \\ &= |a_2 \vec{B}^2 \cdot \vec{A}^3| \\ &= |a_2 \vec{B}_3^2| \\ &= a_2 |\vec{R}_{32}| \end{aligned}$$

The last step is due to the fact that the columns of the rotation matrix are also the axes of the frame of B . The original term consisted of a dot product and cross product, but reduced to a multiplication and an absolute value. Some terms reduce to zero and are eliminated. After simplifying all the terms, this axis test looks like:

$$|\vec{T}_3 \vec{R}_{22} - \vec{T}_2 \vec{R}_{32}| > a_2 |\vec{R}_{32}| + a_3 |\vec{R}_{22}| + b_1 |\vec{R}_{13}| + b_3 |\vec{R}_{11}|$$

All 15 axis tests simplify in similar fashion. Among all the tests, the absolute value of each element of \vec{R} is used four times, so those expressions can be computed once before beginning the axis tests. If any one of the expressions is satisfied, the boxes are known to be disjoint, and the remainder of the 15 axis tests are unnecessary. This permits early exit from the series of tests. In practice, it takes up to 200 arithmetic operations in the worst case to check whether two OBBs overlap.

Implementation and Application

The OBBTree interference detection algorithm has been implemented and used as part of the following packages: RAPID, V-COLLIDE and PQP [12]. These implementations have

been used for robot motion planning, dynamic simulation and virtual prototyping.

56.3.2 Performance of Bounding Volume Hierarchies

The performance of BVHs on proximity queries is governed by a number of design parameters. These include techniques to build the trees, number of children each node can have, and the choice of BV type. An additional design choice is the descent rule. This is the policy for generating recursive calls when a comparison of two BVs does not prune the recursion branch. For instance, if BVs A and B failed to prune, one may recursively compare A with each of the children of B , B with each of the children of A , or each of the children of A with each the children of B . This choice does not affect the correctness of the algorithm, but may impact the performance. Some of the commonly used algorithms assume that the BVHs are binary trees and each primitive is a single triangle or a polygon. The cost of performing the collision query is given as [27, 34]:

$$T = N_{bv} \times C_{bv} + N_p \times C_p,$$

where T is the total cost function for collision queries, N_{bv} is the number of bounding volume pair operations, and C_{bv} is the total cost of a BV pair operation, including the cost of transforming and updating (including resizing) each BV for use in a given configuration of the models, and other per BV-operation overhead. N_p is the number of primitive pairs tested for proximity, and C_p is the cost of testing a pair of primitives for proximity (e.g. overlaps or distance computation).

Typically for tight fitting bounding volumes, e.g., oriented bounding boxes (OBBs), N_{bv} and N_p are relatively low, whereas C_{bv} is relatively high. In contrast, C_{bv} is low while N_{bv} and N_p may be higher for simple BV types like spheres and axis-aligned bounding boxes (AABBs). Due to these opposing trends, no single BV yields optimum performance for collision detection in all possible cases.

56.4 Penetration Depth Computation

In this section, we give a brief overview of penetration depth (PD) computation algorithms between convex polytopes and general polyhedral models. The PD of two inter-penetrating objects A and B is defined as the minimum translation distance that one object undergoes to make the interiors of A and B disjoint. It can be also defined in terms of the TCSO. When two objects are overlapping, the origin of the Minkowski sum of A and $-B$ is contained inside the TCSO. The penetration depth corresponds to the minimum distance from the origin to the surface of TCSO [18]. PD computation is often used in motion planning [37], contact resolution for dynamic simulation [38, 39] and force computation in haptic rendering [40]. For example, computation of dynamic response in penalty-based methods often needs to perform PD queries for imposing the non-penetration constraint for rigid body simulation. In addition, many applications, such as motion planning and dynamic simulation, require a continuous distance measure when two (non-convex) objects collide, in order to have a well-posed computation.

Some of the algorithms for PD computation involve computing the Minkowski sums and computing the closest point on its surface from the origin. The worst case complexity of the overall PD algorithm is governed by the complexity of computing Minkowski sums, which can be $O(n^2)$ for convex polytopes and $O(n^6)$ for general (or non-convex) polyhedral models [16]. Given the complexity of Minkowski sums, many approximation algorithms have been proposed in the literature for fast PD estimation.

56.4.1 Convex Polytopes

Dobkin et al. [16] have proposed a hierarchical algorithm to compute the directional PD using Dobkin and Kirkpatrick polyhedral hierarchy. For any direction d , it computes the directional penetration depth in $O(\log n \log m)$ time for polytopes with m and n vertices. Agarwal et al. [41] have presented a randomized approach to compute the PD values [41]. It runs in $O(m^{\frac{3}{4}+\epsilon}n^{\frac{3}{4}+\epsilon} + m^{1+\epsilon} + n^{1+\epsilon})$ expected time for any positive constant ϵ . Cameron [18] has presented an extension to the GJK algorithm [17] to compute upper and lower bounds on the PD between convex polytopes. Bergen has further elaborated this idea in an expanding polytope algorithm [42]. The algorithm iteratively improves the result of the PD computation by expanding a polyhedral approximation of the Minkowski sums of two polytopes.

56.4.2 Incremental Penetration Depth Computation

Kim et al. [43] have presented an incremental penetration depth (PD) algorithm that marches towards a “locally optimal” solution by walking on the surface of the Minkowski sum. The surface of the TCSO is implicitly computed by constructing a local Gauss map and performing a local walk on the polytopes.

This algorithm uses the concept of width computation from computational geometry. Given a set of points $P = \{p_1, p_2, \dots, p_n\}$ in 3D, the *width* of P , $W(P)$, is defined as the minimum distance between parallel planes supporting P . The width $W(P)$ of convex polytopes A and B is closely related to the penetration depth $PD(A, B)$, since it is easy to show that $W(P) = PD(P, P)$. It can be shown that width and penetration depth computation can be reduced to searching only the VF and EE antipodal pairs (where V, E and F denote a vertex, edge and face, respectively, of the polytopes). This is accomplished by using the standard dual mapping on the Gauss map (or normal diagram). The mapping is defined from object space to the surface of a unit sphere \mathbb{S}^2 as: a vertex is mapped to a region, a face to a point, and an edge to a great arc. The algorithm finds the antipodal pairs by overlaying the upper hemisphere of the Gauss map on the lower hemisphere and computing the intersections between them.

Local Walk

The incremental PD computation algorithm does not compute the entire Gauss map for each polytope or the entire boundary of the Minkowski sum. Rather it computes them in a lazy manner based on local walking and optimization. Starting from some feature on the surface of the Minkowski sum, the algorithm computes the direction in which it can decrease the PD value and proceeds towards that direction by extending the surface of the Minkowski sum.

At each iteration of the algorithm, a vertex is chosen from each polytope to form a pair. It is called a *vertex hub pair* and the algorithm uses it as a hub for the expansion of the local Minkowski sum. The vertex hub pair is chosen in such a way that there exists a plane supporting each polytope, and is incident on each vertex. It turns out that the vertex hub pair corresponds to two intersected convex regions on a Gauss map, which later become intersecting convex polygons on the plane after *central projection*. The intersection of convex polygons corresponds to the VF or EE antipodal pairs that are used to reconstruct the local surface of the Minkowski sum around the vertex hub pair. Given these pairs, the algorithm chooses the one that corresponds to the shortest distance from the origin of the Minkowski sum to their surface. If this pair decreases the estimated PD value, the algorithm updates

the current vertex hub pair to the new adjacent pair. This procedure is repeated until the algorithm can not decrease the current PD value and converges to a local minima.

Initialization and Refinement

The algorithm starts with an initial guess on the vertex hub pair. A good estimate to the penetration direction can be obtained by taking the centroid difference between the objects, and computing an extremal vertex pair for the difference direction. In other cases, the penetrating features (for overlapping polytopes) or the closest features (from non-overlapping polytopes) from the previous instance can also suggest a good initial guess.

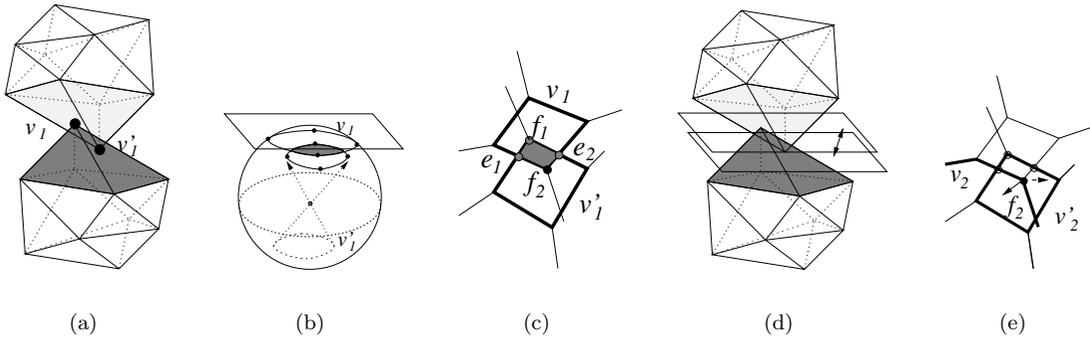


FIGURE 56.4: Iterative optimization for incremental PD computation: (a) The current \mathbf{VV} pair is $v_1 v'_1$ and a shaded region represents edges and faces incident to $v_1 v'_1$. (b) shows local Gauss maps and their overlay for $v_1 v'_1$. (c) shows the result of the overlay after central projection onto a plane. Here, f_1 , e_1 , f_2 and e_2 comprise vertices (candidate PD features) of the overlay. (d) illustrates how to compute the PD for the candidate PD features in object space. (e) f_2 is chosen as the next PD feature, thus $v_2 v'_2$ is determined as the next vertex hub pair.

After the algorithm obtains a initial guess for a \mathbf{VV} pair, it iteratively seeks to improve the PD estimate by jumping from one \mathbf{VV} pair to an adjacent \mathbf{VV} pair. This is accomplished by looking around the neighborhood of the current \mathbf{VV} pair and walking to a pair which provides the greatest improvement in the PD value. Let the current vertex hub pair be $v_1 v'_1$. The next vertex hub pair $v_2 v'_2$ is computed as follows:

1. Construct a local Gauss map each for v_1 and v'_1 ,
2. Project the Gauss maps onto $z = 1$ plane, and label them as G and G' , respectively. G and G' correspond to convex polygons in 2D.
3. Compute the intersection between G and G' using a linear time algorithm such as [44]. The result is a convex polygon and let u_i be a vertex of the intersection set. If u_i is an original vertex of G or G' , it corresponds to the VF antipodal pair in object space. Otherwise, it corresponds to an EE antipodal pair.
4. In object space, determine which u_i corresponds to the best local improvement in PD, and set an adjacent vertex pair (adjacent to u_i) to $v_2 v'_2$.

This iteration is repeated until either there is no more improvement in the PD value or number of iterations reach some maximum value. At step 4 of the iteration, the next vertex

hub pair is selected in the following manner. If u_i corresponds to VF, then the algorithm chooses one of the two vertices adjacent to F assuming that the model is triangulated. The same reasoning is also applied to when u_i corresponds to EE. As a result, the algorithm needs to perform one more iteration in order to actually decide which vertex hub pair should be selected. A snapshot of a typical step during the iteration is illustrated in [Figure 56.4](#). Eventually the algorithm computes a local minima.

Implementation and Application

The incremental algorithm is implemented as part of DEEP [12]. It works quite well in practice and is also able to compute the global penetration depth in most cases. It has been used for 6-DOF haptic rendering (force and torque display), dynamic simulation and virtual prototyping.

56.4.3 Non-Convex Models

Algorithms for penetration depth estimation between general polygonal models are based on discretization of the object space containing the objects or use of digital geometric algorithms that perform computations on a finite resolution grid. Fisher and Lin [45] have presented a PD estimation algorithm based on the distance field computation using the fast marching level-set method. It is applicable to all polyhedral objects as well as deformable models, and it can also check for self-penetration. Hoff et al. [46, 47] have proposed an approach based on performing discretized computations on the graphics rasterization hardware. It uses multi-pass rendering techniques for different proximity queries between general rigid and deformable models, including penetration depth estimation. Kim et al. [43] have presented a fast approximation algorithm for general polyhedral models using a combination of object-space as well discretized computations. Given the global nature of the PD problem, it decomposes the boundary of each polyhedron into convex pieces, computes the pairwise Minkowski sums of the resulting convex polytopes and uses graphics rasterization hardware to perform the closest point query up to a given discretized resolution. The results obtained are refined using a local walking algorithm. To further speed up this computation and improve the estimate, the algorithm uses a hierarchical refinement technique that takes advantage of geometry culling, model simplification, accelerated ray-shooting, and local refinement with greedy walking. The overall approach combines discretized closest point queries with geometry culling and refinement at each level of the hierarchy. Its accuracy can vary as a function of the discretization error. It has been applied to haptic rendering and dynamic simulation.

56.5 Large Environments

Large environments are composed of multiple moving objects. Different methods have been proposed to overcome the computational bottleneck of $O(n^2)$ pairwise tests in an environment composed of n objects. The problem of performing proximity queries in large environments, is typically divided into two parts [3, 23]: the *broad phase*, in which we identify the pair of objects on which we need to perform different proximity queries, and the *narrow phase*, in which we perform the exact pairwise queries. In this section, we present a brief overview of algorithms used in the broad phase.

The simplest algorithms for large environments are based on spatial subdivisions. The space is divided into cells of equal volume, and at each instance the objects are assigned to one or more cells. Collisions are checked between all object pairs belonging to each

Architecture for Multi-body Collision Detection

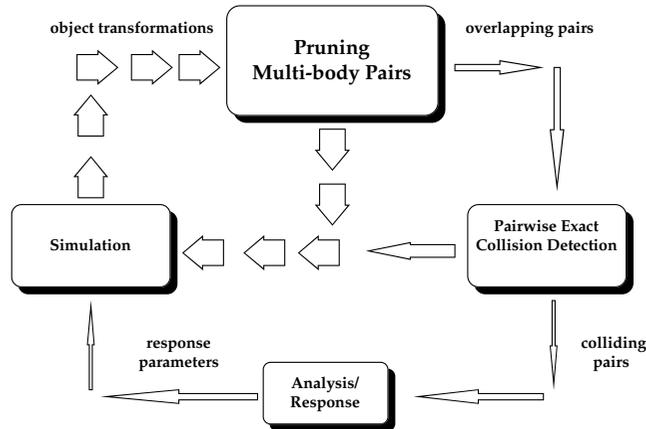


FIGURE 56.5: Architecture for multiple body collision detection algorithm.

cell. In fact, Overmars has presented an efficient algorithm based on hash table to efficiently perform point location queries in fat subdivisions [48]. This approach works well for sparse environments in which the objects are uniformly distributed through the space. Another approach operates directly on four-dimensional volumes swept out by object motion over time [49].

56.5.1 Multiple-Object Collision Detection

Large-scale environments consist of stationary as well as moving objects. Let there be N moving objects and M stationary objects. Each of the N moving objects can collide with the other moving objects, as well as with the stationary ones. Keeping track of $\binom{N}{2} + NM$ pairs of objects at every time step can become time consuming as N and M get large. To achieve interactive rates, we must reduce this number before performing pairwise collision tests. In this section, we give an overview of sweep-and-prune algorithm used to perform multiple-object collision detection [3]. The overall architecture of the multiple object collision detection algorithm is shown in Fig. 56.5.1.

The algorithm uses sorting to prune the number of pairs. Each object is surrounded by a 3-dimensional bounding volume. These bounding volumes are sorted in 3-space to determine which pairs are overlapping. The algorithm only needs to perform exact pairwise collision tests on these remaining pairs.

However, it is not intuitively obvious how to sort objects in 3-space. The algorithm uses a *dimension reduction* approach. If two bodies collide in a 3-dimensional space, their orthogonal projections onto the xy , yz , and xz -planes and x , y , and z -axes must overlap. Based on this observation, the algorithm uses axis-aligned bounding boxes and efficiently project them onto a lower dimension, and perform sorting on these lower-dimensional structures.

The algorithm computes a rectangular bounding box to be the tightest axis-aligned box

containing each object at a particular orientation. It is defined by its minimum and maximum x , y , and z -coordinates. As an object moves, the algorithm recomputes its minima and maxima, taking into account the object's orientation.

As a precomputation, the algorithm computes each object's initial minima and maxima along each axis. It is assumed that the objects are convex. For non-convex polyhedral models, the following algorithm is applied to their convex hulls. As an object moves, its minima and maxima are recomputed in the following manner:

1. Check to see if the current minimum (or maximum) vertex for the x , y , or z -coordinate still has the smallest (or largest) value in comparison to its neighboring vertices. If so the algorithm terminates.
2. Update the vertex for that extreme direction by replacing it with the neighboring vertex with the smallest (or largest) value of all neighboring vertices. Repeat the entire process as necessary.

This algorithm recomputes the bounding boxes at an expected constant rate. It exploits the temporal and geometric coherence.

The algorithm does not transform all the vertices as the objects undergo motion. As it is updating the bounding boxes, new positions are computed for current vertices using matrix-vector multiplications. This approach is optimized based on the fact that the algorithm is interested in *one* coordinate value of each extremal vertex, say the x coordinate while updating the minimum or maximum value along the x -axis. Therefore, there is no need to transform the other coordinates in order to compare neighboring vertices. This reduces the number of arithmetic operations by two-thirds, as we only compute a dot-product of two vectors and do not perform matrix-vector multiplication.

One-Dimensional Sweep and Prune

The one-dimensional sweep and prune algorithm begins by projecting each three-dimensional bounding box onto the x , y , and z axes. Because the bounding boxes are axis-aligned, projecting them onto the coordinate axes results in intervals (see Fig. 56.5.1). The goal is to compute the overlaps among these intervals, because a pair of bounding boxes can overlap if and only if their intervals overlap in all three dimensions.

The algorithm constructs three lists, one for each dimension. Each list contains the values of the endpoints of the intervals corresponding to that dimension. By sorting these lists, it determines which intervals overlap. In the general case, such a sort would take $O(n \log n)$ time, where n is the number of objects. This time bound is reduced by keeping the sorted lists from the previous frame, changing only the values of the interval endpoints. In environments where the objects make relatively small movements between frames, the lists will be nearly sorted, so we can sort in expected $O(n)$ time, as shown in [50, 51]. In practice, insertion sort works well for almost sorted lists.

In addition to sorting, the algorithm keeps track of changes in the overlap status of interval pairs (i.e. from overlapping in the last time step to non-overlapping in the current time step, and vice-versa). This can be done in $O(n + e_x + e_y + e_z)$ time, where e_x , e_y , and e_z are the number of exchanges along the x , y , and z -axes. This also runs in expected linear time due to coherence, but in the worst case e_x , e_y , and e_z can each be $O(n^2)$ with an extremely small constant.

This algorithm is suitable for dynamic environments where coherence between successive static queries is preserved. In computational geometry literature several algorithms exist that solve the static version of determining 3-D bounding box overlaps in $O(n \log^2 n + s)$

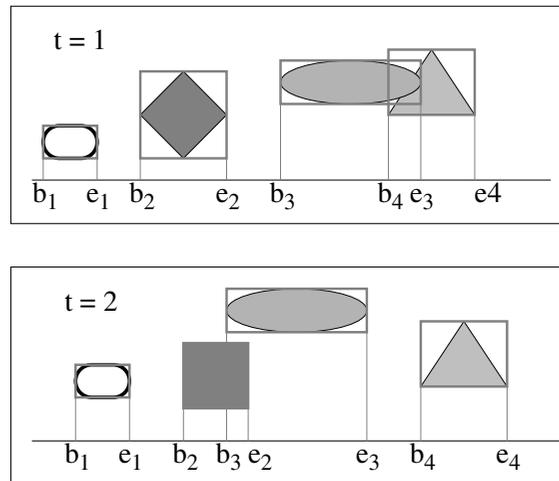


FIGURE 56.6: Bounding box behavior between successive instances. Notice the coherence between the 1D list obtained after projection onto the X-axis.

time, where s is the number of pairwise overlaps [52, 53]. It has been reduced to $O(n + s)$ by exploiting coherence.

Implementation and Application

The sweep-and-prune has been used in some of the widely used collision detection systems, including I-COLLIDE, V-COLLIDE, SWIFT and SWIFT++ [12]. It has been used for multi-body simulations and interactive walkthroughs of complex environments.

56.5.2 Two-Dimensional Intersection Tests

The two-dimensional intersection algorithm begins by projecting each three-dimensional axis-aligned bounding box onto any two of the x - y , x - z , and y - z planes. Each of these projections is a rectangle in 2-space. Typically there are fewer overlaps of these 2-D rectangles than of the 1-D intervals used by the sweep and prune technique. This results in fewer swaps as the objects move. In situations where the projections onto one-dimension result in densely clustered intervals, the two-dimensional technique is more efficient. The interval tree is a common data structure for performing such two-dimensional range queries [54].

Each query of an interval intersection takes $O(\log n + k)$ time where k is the number of reported intersections and n is the number of intervals. Therefore, reporting intersections among n rectangles can be done in $O(n \log n + K)$ where K is the total number of intersecting rectangles [55].

References

- [1] D. Baraff and A. Witkin, *Physically-Based Modeling*. ACM SIGGRAPH Course Notes, 2001.
- [2] J. Latombe, *Robot Motion Planning*. Kluwer Academic Publishers, 1991.
- [3] J. Cohen, M. Lin, D. Manocha, and M. Ponamgi, "I-collide: An interactive and exact

- collision detection system for large-scale environments,” in *Proc. of ACM Interactive 3D Graphics Conference*, pp. 189–196, 1995.
- [4] M. A. Otaduy and M. C. Lin, “Sensation preserving simplification for haptic rendering,” *Proc. of ACM SIGGRAPH*, 2003.
 - [5] D. P. Dobkin and D. G. Kirkpatrick, “Determining the separation of preprocessed polyhedra – a unified approach,” in *Proc. 17th Internat. Colloq. Automata Lang. Program.*, vol. 443 of *Lecture Notes in Computer Science*, pp. 400–413, Springer-Verlag, 1990.
 - [6] R. Seidel, “Linear programming and convex hulls made easy,” in *Proc. 6th Ann. ACM Conf. on Computational Geometry*, (Berkeley, California), pp. 211–215, 1990.
 - [7] M. Lin, *Efficient Collision Detection for Animation and Robotics*. PhD thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, December 1993.
 - [8] M. Lin and J. F. Canny, “Efficient algorithms for incremental distance computation,” in *IEEE Conference on Robotics and Automation*, pp. 1008–1014, 1991.
 - [9] S. Ehmann and M. C. Lin, “Accelerated proximity queries between convex polyhedra using multi-level voronoi marching,” *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2101–2106, 2000.
 - [10] B. Mirtich, “V-Clip: Fast and robust polyhedral collision detection,” *ACM Transactions on Graphics*, vol. 17, pp. 177–208, July 1998.
 - [11] C. Hoffmann, *Geometric and Solid Modeling*. San Mateo, California: Morgan Kaufmann, 1989.
 - [12] “Collision detection systems.” <http://gamma.cs.unc.edu/collide/packages.shtml>, 2002.
 - [13] B. Mirtich and J. Canny, “Impulse-based simulation of rigid bodies,” in *Proc. of ACM Interactive 3D Graphics*, (Monterey, CA), 1995.
 - [14] A. Gregory, A. Mascarenhas, S. Ehmann, M. C. Lin, and D. Manocha, “6-dof haptic display of polygonal models,” *Proc. of IEEE Visualization Conference*, 2000.
 - [15] S. Cameron and R. K. Culley, “Determining the minimum translational distance between two convex polyhedra,” *Proceedings of International Conference on Robotics and Automation*, pp. 591–596, 1986.
 - [16] D. Dobkin, J. Hershberger, D. Kirkpatrick, and S. Suri, “Computing the intersection-depth of polyhedra,” *Algorithmica*, vol. 9, pp. 518–533, 1993.
 - [17] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi, “A fast procedure for computing the distance between objects in three-dimensional space,” *IEEE J. Robotics and Automation*, vol. vol RA-4, pp. 193–203, 1988.
 - [18] S. Cameron, “Enhancing GJK: Computing minimum and penetration distance between convex polyhedra,” *Proceedings of International Conference on Robotics and Automation*, pp. 3112–3117, 1997.
 - [19] H. Samet, *Spatial Data Structures: Quadtree, Octrees and Other Hierarchical Methods*. Addison Wesley, 1989.
 - [20] M. Held, J. Klosowski, and J. Mitchell, “Evaluation of collision detection methods for virtual reality fly-throughs,” in *Canadian Conference on Computational Geometry*, 1995.
 - [21] B. Naylor, J. Amanatides, and W. Thibault, “Merging bsp trees yield polyhedral modeling results,” in *Proc. of ACM SIGGRAPH*, pp. 115–124, 1990.
 - [22] W. Bouma and G. Vanecek, “Collision detection and analysis in a physically based simulation,” *Proceedings Eurographics workshop on animation and simulation*, pp. 191–203, 1991.
 - [23] P. M. Hubbard, “Interactive collision detection,” in *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993.

- [24] S. Quinlan, "Efficient distance computation between non-convex objects," in *Proceedings of International Conference on Robotics and Automation*, pp. 3324–3329, 1994.
- [25] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," *Proc. SIGMOD Conf. on Management of Data*, pp. 322–331, 1990.
- [26] M. Ponamgi, D. Manocha, and M. Lin, "Incremental algorithms for collision detection between solid models," *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, no. 1, pp. 51–67, 1997.
- [27] S. Gottschalk, M. Lin, and D. Manocha, "OBB-Tree: A hierarchical structure for rapid interference detection," *Proc. of ACM Siggraph'96*, pp. 171–180, 1996.
- [28] G. Barequet, B. Chazelle, L. Guibas, J. Mitchell, and A. Tal, "Boxtree: A hierarchical representation of surfaces in 3D," in *Proc. of Eurographics'96*, 1996.
- [29] S. Gottschalk, *Collision Queries using Oriented Bounding Boxes*. PhD thesis, University of North Carolina. Department of Computer Science, 2000.
- [30] S. Krishnan, A. Pattekar, M. Lin, and D. Manocha, "Spherical shell: A higher order bounding volume for fast proximity queries," in *Proc. of Third International Workshop on Algorithmic Foundations of Robotics*, pp. 122–136, 1998.
- [31] S. Krishnan, M. Gopi, M. Lin, D. Manocha, and A. Pattekar, "Rapid and accurate contact determination between spline models using shelltrees," *Computer Graphics Forum, Proceedings of Eurographics*, vol. 17, no. 3, pp. C315–C326, 1998.
- [32] M. Held, J. Klosowski, and J. S. B. Mitchell, "Real-time collision detection for motion simulation within complex environments," in *Proc. ACM SIGGRAPH'96 Visual Proceedings*, p. 151, 1996.
- [33] J. Klosowski, M. Held, J. Mitchell, H. Sowizral, and K. Zikan, "Efficient collision detection using bounding volume hierarchies of k-dops," *IEEE Trans. on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21–37, 1998.
- [34] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha, "Fast proximity queries with swept sphere volumes," Tech. Rep. TR99-018, Department of Computer Science, University of North Carolina, 1999. 32 pages.
- [35] S. Ehmann and M. C. Lin, "Accurate and fast proximity queries between polyhedra using convex surface decomposition," *Computer Graphics Forum (Proc. of Eurographics'2001)*, vol. 20, no. 3, pp. 500–510, 2001.
- [36] R. Duda and P. Hart, *Pattern Classification and Scene Analysis*. John Wiley and Sons, 1973.
- [37] D. Hsu, L. Kavraki, J. Latombe, R. Motwani, and S. Sorkin, "On finding narrow passages with probabilistic roadmap planners," *Proc. of 3rd Workshop on Algorithmic Foundations of Robotics*, pp. 25–32, 1998.
- [38] M. McKenna and D. Zeltzer, "Dynamic simulation of autonomous legged locomotion," in *Computer Graphics (SIGGRAPH '90 Proceedings)* (F. Baskett, ed.), vol. 24, pp. 29–38, Aug. 1990.
- [39] D. E. Stewart and J. C. Trinkle, "An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction," *International Journal of Numerical Methods in Engineering*, vol. 39, pp. 2673–2691, 1996.
- [40] Y. Kim, M. Otaduy, M. Lin, and D. Manocha, "6-dof haptic display using localized contact computations," *Proc. of Haptics Symposium*, pp. 209–216, 2002.
- [41] P. Agarwal, L. Guibas, S. Har-Peled, A. Rabinovitch, and M. Sharir, "Penetration depth of two convex polytopes in 3d," *Nordic J. Computing*, vol. 7, pp. 227–240, 2000.
- [42] G. Bergen, "Proximity queries and penetration depth computation on 3d game objects," *Game Developers Conference*, 2001.

- [43] Y. Kim, M. Lin, and D. Manocha, "Deep: An incremental algorithm for penetration depth computation between convex polytopes," *Proc. of IEEE Conference on Robotics and Automation*, pp. 921–926, 2002.
- [44] J. O'Rourke, C.-B. Chien, T. Olson, and D. Naddor, "A new linear algorithm for intersecting convex polygons," *Comput. Graph. Image Process.*, vol. 19, pp. 384–391, 1982.
- [45] S. Fisher and M. C. Lin, "Deformed distance fields for simulation of non-penetrating flexible bodies," *Proc. of EG Workshop on Computer Animation and Simulation*, pp. 99–111, 2001.
- [46] K. Hoff, A. Zaferakis, M. Lin, and D. Manocha, "Fast and simple 2d geometric proximity queries using graphics hardware," *Proc. of ACM Symposium on Interactive 3D Graphics*, pp. 145–148, 2001.
- [47] K. Hoff, A. Zaferakis, M. Lin, and D. Manocha, "Fast 3d geometric proximity queries between rigid and deformable models using graphics hardware acceleration," Tech. Rep. TR02-004, Department of Computer Science, University of North Carolina, 2002.
- [48] M. H. Overmars, "Point location in fat subdivisions," *Inform. Proc. Lett.*, vol. 44, pp. 261–265, 1992.
- [49] S. Cameron, "Collision detection by four-dimensional intersection testing," *Proceedings of International Conference on Robotics and Automation*, pp. 291–302, 1990.
- [50] D. Baraff, *Dynamic simulation of non-penetrating rigid body simulation*. PhD thesis, Cornell University, 1992.
- [51] M. Shamos and D. Hoey, "Geometric intersection problems," *Proc. 17th An. IEEE Symp. Found. on Comput. Science*, pp. 208–215, 1976.
- [52] J. Hopcroft, J. Schwartz, and M. Sharir, "Efficient detection of intersections among spheres," *The International Journal of Robotics Research*, vol. 2, no. 4, pp. 77–80, 1983.
- [53] H. Six and D. Wood, "Counting and reporting intersections of D -ranges," *IEEE Transactions on Computers*, pp. 46–55, 1982.
- [54] F. Preparata and M. I. Shamos, *Computational Geometry*. New York: Springer-Verlag, 1985.
- [55] H. Edelsbrunner, "A new approach to rectangle intersections, Part I," *Internat. J. Comput. Math.*, vol. 13, pp. 209–219, 1983.

Image Data Structures

	57.1	Introduction.....	57-1
	57.2	What is Image Data?	57-2
	57.3	Quadtrees	57-3
		What is a Quadtree? • Variants of Quadtrees	
	57.4	Virtual Quadtrees	57-6
		Compact Quadtrees • Forest of Quadtrees (FQT)	
	57.5	Quadtrees and R-trees.....	57-11
	57.6	Octrees.....	57-12
	57.7	Translation Invariant Data Structure (TID) ...	57-13
	57.8	Content-Based Image Retrieval System.....	57-15
		What is CBIR? • An Example of CBIR System	
	57.9	Summary	57-16
	57.10	Acknowledgments	57-17

S. S. Iyengar
Louisiana State University

V. K. Vaishnavi
Georgia State University

S. Gunasekaran
Louisiana State University

57.1 Introduction

Image has been an integral part of our communication. Visual information aids us in understanding our surroundings better. Image processing, the science of manipulating digital images, is one of the methods used for digitally interpreting images. Image processing generally comprises three main steps:

1. Image acquisition: Obtaining the image by scanning it or by capturing it through some sensors.
2. Image manipulation/analysis: Enhancing and/or compressing the image for its transfer or storage.
3. Display of the processed image.

Image processing has been classified into two levels: low-level image processing and high-level image processing. Low-level image processing needs little information about the content or the semantics of the image. It is mainly concerned with retrieving low-level descriptions of the image and processing them. Low-level data include matrix representation of the actual image. Image calibration and image enhancement are examples of low-level image processing.

High-level image processing is basically concerned with segmenting an image into objects or regions. It makes decisions according to the information contained in the image. High-level data are represented in symbolic form. The data include features of the image such as object size, shape and its relation with other objects in the image. These image-processing techniques depend significantly on the underlying image data structure.

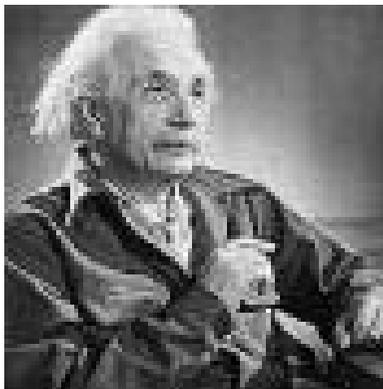


FIGURE 57.1: Example of an image.

Efficient data structures for region representation are important for use in manipulating pictorial information.

Many techniques have been developed for representing pictorial information in image processing [20]. These techniques include data structures such as quadtrees, linear quadtrees, Forest of Quadtrees, and Translation Invariant Data structure (TID). We will discuss these data structures in the following sections. Research on quadtrees has produced several interesting results in different areas of image processing [2] [6] [7] [11] [12] [14]. In 1981, Jones and Iyengar [8] proposed methods of refining quadtrees. A good tracing of the history of the evolution of quadtrees is provided by Klinger and Dyer [13]. See also [Chapter 19](#) of this handbook.

57.2 What is Image Data?

An image is a visual reproduction of an object using an optical or electronic device. Image data include pictures taken by satellites, scanned images, aerial photographs and other digital photographs. In the computer, image is represented as a data file that consists of a rectangular array of picture elements called pixels. This rectangular array of pixels is also called a raster image. The pixels are the smallest programmable visual unit. The size of the pixel depends on the resolution of the monitor. The resolution can be defined as the number of pixels present on the horizontal axis and vertical axis of a display monitor. When the resolution is set to maximum the pixel size is equal to a dot on the monitor. The pixel size increases with the decrease of the resolution. Figure 57.1 shows an example of an image.

In a monochrome image each pixel has its own brightness value ranging from 0 (black) to 255 (white). For a color image each pixel has a brightness value and a RGB color value. RGB is an additive color state that has separate values for red, green, and blue. Hence each pixel has independent values (0 to 255) for red, green, and blue colors. If the values for red, green, and blue components of the pixel are the same then the resulting pixel color is gray. Different shades of gray pixels constitute a gray-scale image. If pixels of an image have only two states, black or white, then the image is called a binary image.

Image data can be classified into raster graphics and vector graphics. Raster graphics, also known as bitmap graphics, represents an image using x and y coordinates (for 2d-images) of display space; this grid of x and y coordinates is called the raster. [Figure 57.2](#) shows how a circle will appear in raster graphics. In raster graphics an image is divided up

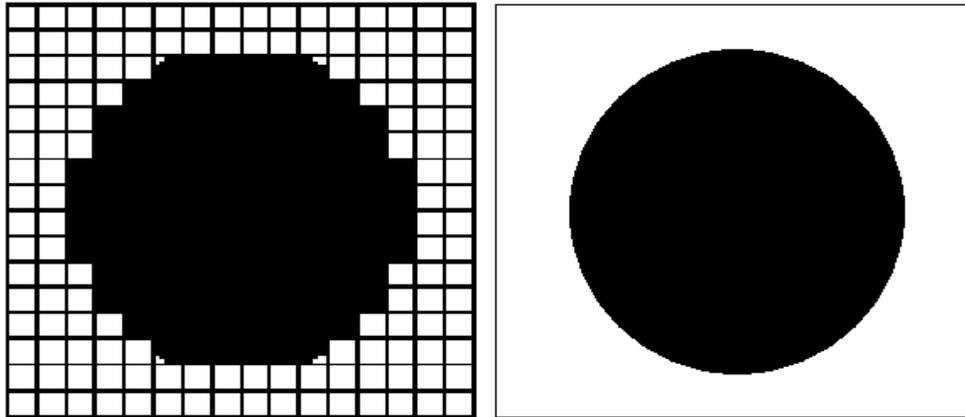


FIGURE 57.2: A circle in raster graphics.

in raster. All raster dots that are more than half full are displayed as black dots and the rest as white dots. This results in step like edges as shown in the figure. The appearance of jagged edges can be minimized by reducing the size of the raster dots. Reducing the size of the dots will increase the number of pixels needed but increases the size of the storage space.

Vector graphics uses mathematical formulas to define the image in a two-dimensional or three-dimensional space. In a vector graphics file an image is represented as a sequence of vector statements instead of bits, as in bitmap files. Thus it needs just minimal amount of information to draw shapes and therefore the files are smaller in size compared to raster graphics files.

Vector graphics does not consist of black and white pixels but is made of objects like line, circle, and rectangle. The other advantage of vector graphics is that it is flexible, so it can be resized without loss of information. Vector graphics is typically used for simple shapes. CorelDraw images, PDF, and PostScript are all in vector image format. The main drawbacks of vector graphics is that it needs longer computational time and has very limited choice of shapes compared to raster graphics.

Raster file, on the other hand, is usually larger than vector graphics file and is difficult to modify without loss of information. Scaling a raster file will result in loss of quality whereas vector graphics can be scaled to the quality of the device on which it is rendered. Raster graphics is used for representing continuous numeric values unlike vector graphics, which is used for discrete features. Examples of raster graphic formats are GIF, JPEG, and PNG.

57.3 Quadrees

Considerable research on quadtrees has produced several interesting results in different areas of image processing. The basic relationship between a region and its quadtree representation is presented in [6]. In 1981, Jones and Iyengar [8] proposed methods for refining quadtrees. The new refinements were called virtual quadtrees, which include compact quadtrees and forests of quadtrees. We will discuss virtual quadtrees in a later section of this chapter. Much work has been done on the quadtree properties and algorithms for manipulations and translations have been developed by Samet [16], [19], Dyer [2] and others [6], [9], [11].

57.3.1 What is a Quadtree?

A quadtree is a class of hierarchical data structures used for image representation. The fundamental principle of quadtrees is based on successive regular decomposition of the image data until each part contains data that is sufficiently simple so that it can be represented by some other simpler data structure. A quadtree is formed by dividing an image into four quadrants, each quadrant may further be divided into four sub quadrants and so on until the whole image has been broken down to small sections of uniform color.

In a quadtree the root represents the whole image and the non-root nodes represent sub quadrants of the image. Each node of the quadtree represents a distinct section of the image; no two nodes can represent the same part of the image. In other words, the sub quadrants of the image do not overlap. The node without any child is called a leaf node and it represents a region with uniform color. Each non-leaf node in the tree has four children, each of which represents one of the four sub regions, referred to as NW, NE, SW, and SE, that the region represented by the parent node is divided into.

The leaf node of a quadtree has the color of the pixel (black or white) it is representing. The nodes with uniform colored children have the color of their children and all their child nodes are removed from the tree. All the other nodes with non-uniform colored children have the gray color. [Figure 57.3](#) shows the concept of quadtrees.

The image can be retrieved from the quadtree by using a recursive procedure that visits each leaf node of the tree and displays its color at an appropriate position. The procedure starts with visiting the root node. In general, if the visited node is not a leaf then the procedure is recursively called for each child of the node, in order from left to right.

The main advantage of the quadtree data structure is that images can be compactly represented by it. The data structure combines data having similar values and hence reduces storage size. An image having large areas of uniform color will have very small storage size. The quadtree can be used to compress bitmap images, by dividing the image until each section has the same color. It can also be used to quickly locate any object of interest.

The drawback of the quadtree is that it can have totally different representation for images that differ only in rotation or translation. Though the quadtree has better storage efficiency compared to other data structures such as the array, it also has considerable storage overheads. For a given image it stores many white leaf nodes and intermediate gray nodes, which are not required information.

57.3.2 Variants of Quadtrees

The numerous quadtree variants that have been developed so far can be differentiated by the type of data they are designed to represent [15]. The many variants of the quadtree include region quadtrees, point quadtrees, line quadtrees, and edge quadtrees. Region quadtrees are meant for representing regions in images, while point quadtrees, edge quadtrees, and line quadtrees are used to represent point features, edge features, and line features, respectively. Thus there is no single quadtree data structure which is capable of representing a mixture of features of images like regions and lines.

Region quadtrees

In the region quadtree, a region in a binary image is a subimage that contains either all 1's or all 0's. If the given region does not consist entirely of 1's or 0's, then the region is divided into four quadrants. This process is continued until each divided section consists entirely of 1's or 0's; such regions are called final regions. The final regions (either black or white) are represented by leaf nodes. The intermediate nodes are called gray nodes. A region quadtree

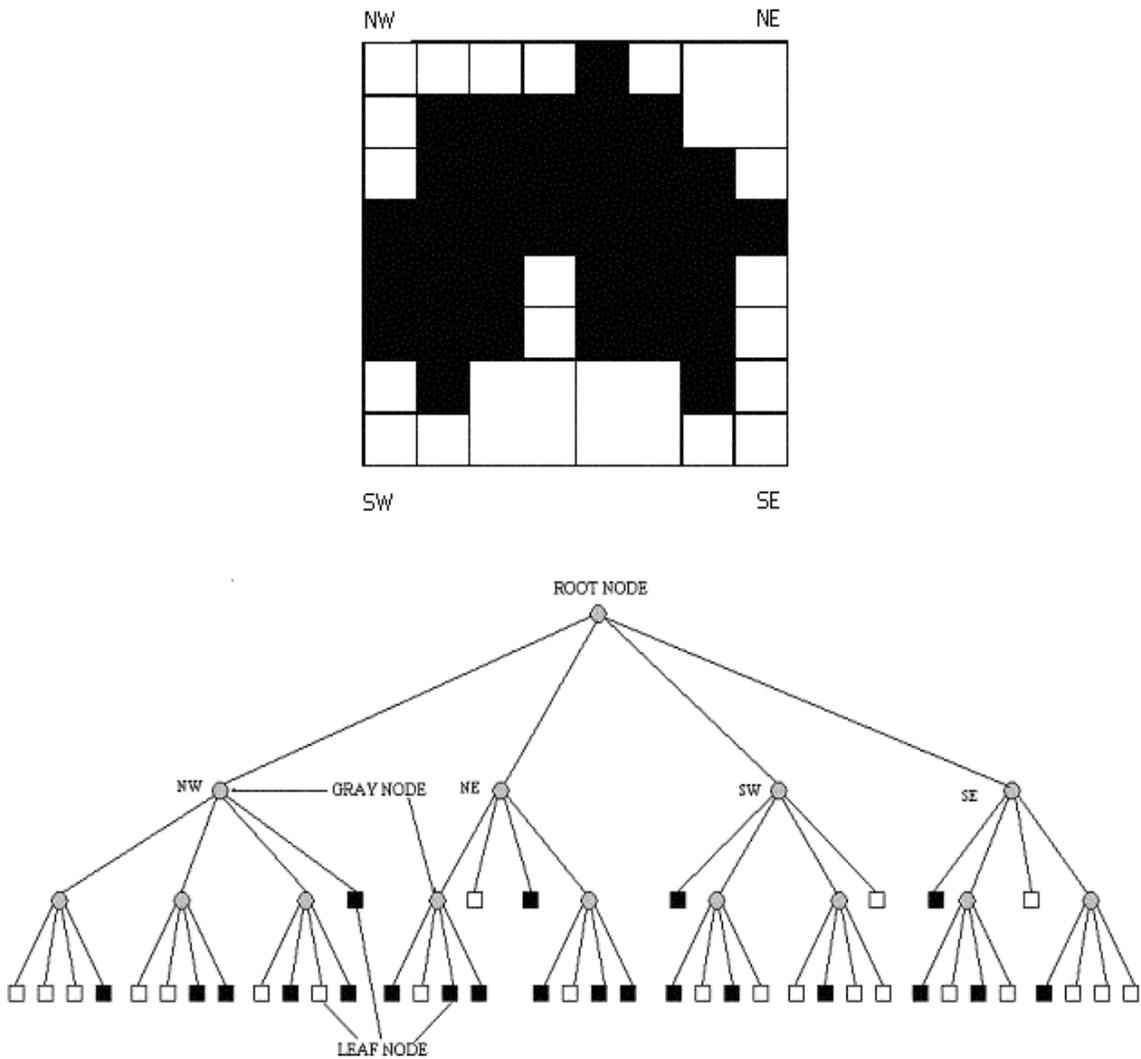


FIGURE 57.3: The concept of quadtrees

is space efficient for images containing square-like regions. It is very inefficient for regions that are elongated or for representing line features.

Line quadtrees

The line quadtree proposed by Samet [21] is used for representing the boundaries of a region. The given region is represented as in the region quadtree with additional information associated with nodes representing the boundary of the region. The data structure is used for representing curves that are closed. The boundary following algorithm using the line quadtree data structure is at least twice as good as the one using the region quadtree in terms of execution time; the map superposition algorithm has execution time proportional to number of nodes in the line quadtree [15]. The main disadvantage of the line quadtree is that it cannot represent independent linear features.

Edge quadtrees

Shneier [23] formulated the edge quadtree data structure for storing linear features. The main principle used in the data structure is to approximate the curve being represented by a number of straight line segments. The edge quadtree is constructed using a recursive procedure. If the sub quadrant represented by a node does not contain any edge or line, it is not further subdivided and is represented by a leaf. If it does contain one then an approximate equation is fitted to the line. The error caused by approximation is calculated using a measure such as least squares. When the error is less than the predetermined value, the node becomes a leaf; otherwise the region represented by the node is further subdivided. If an edge terminates (within the region represented by) a node then a special flag is set. Each node contains magnitude, direction, intercept, and error information about the edge passing through the node. The main drawback of the edge quadtree is that it cannot efficiently handle two or more intersecting lines.

Template quadtrees

An image can have regions and line features together. The above quadtree representational schemas are not efficient for representing such an image in one quadtree data structure. The template quadtree is an attempt toward development of such a quadtree data structure. It was proposed by Manohar, Rao, and Iyengar [15] to represent regions and curvilinear data present in images.

A template is a $2^k \times 2^k$ sub image, which contains either a region of uniform color or straight run of black pixels in horizontal, vertical, left diagonal, or right diagonal directions spanning the entire sub image. All the possible templates for a 2×2 sub image are shown in Figure 57.4.

A template quadtree can be constructed by comparing a quadrant with any one of the templates, if they match then the quadrant is represented by a node with the information about the type of template it matches. Otherwise the quadrant is further divided into four sub quadrants and each one of them is compared with any of the templates of the next lower size. This process is recursively followed until the entire image is broken down into maximal blocks corresponding to the templates. The template quadtree representation of an image is shown in Figure 57.5.

Here the leaf is defined as a template of variable size, therefore it does not need any approximation for representing curves present in the images. The advantage of template quadtree is that it is very accurate and has the capabilities for representing features like regions and curves. The main drawback is that it needs more space compared to edge quadtrees. For more information on template quadtrees see [15].

57.4 Virtual Quadtrees

The quadtree has become a major data structure in image processing. Though the quadtree has better storage efficiency compared to other data structures such as the array, it also has considerable storage overhead. Jones and Iyengar [10] have proposed two ways in which quadtrees may be efficiently stored: as “forest of quadtrees” and as “compact quadtrees”. They called these new data structures virtual quadtrees because the basic operations performed in quadtrees can also be performed on the new representations. The virtual quadtree is a space-efficient way of representing a quadtree. It is a structure that simulates quadtrees in such a way that we can,

1. Determine the color of any node in the quadtree.

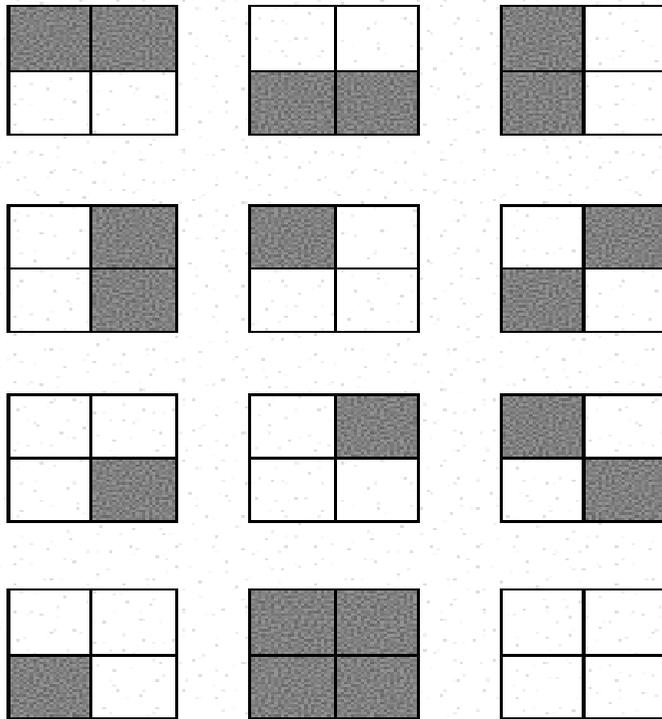


FIGURE 57.4: Template sets.

2. Find the child node of any node, in any direction in the quadtree.
3. Find the parent node of any node in the quadtree.

The two types of virtual quadtrees, which we are going to discuss, are the compact quadtree and the forest of quadtrees.

57.4.1 Compact Quadtrees

The compact quadtree has all the information contained in a quadtree but needs less space. It is represented as $C(T)$, where T is the quadtree it is associated with. Each set of four sibling nodes in the quadtree is represented as a single node called metanode in the corresponding compact quadtree.

The metanode M has four fields, which are explained as follows:

- $M\text{COLOR}(M, D)$ – The colors of the nodes included in M . Where $D \in \{NW, NE, SW, SE\}$.
- $M\text{SONS}(M)$ – Points to the first metanode that represents offsprings of a node represented in M ; NIL if no offspring exists.
- $M\text{FATHER}(M)$ – Points to the metanode that holds the representation of the parent of the nodes that M represents.
- $M\text{CHAIN}(M)$ – If there are more than one metanode that represent offspring of nodes represented by a given metanode M , then these are linked by $M\text{CHAIN}$ field.

The compact quadtree for the quadtree shown in the [Figure 57.6](#) is given in [Figure 57.7](#).

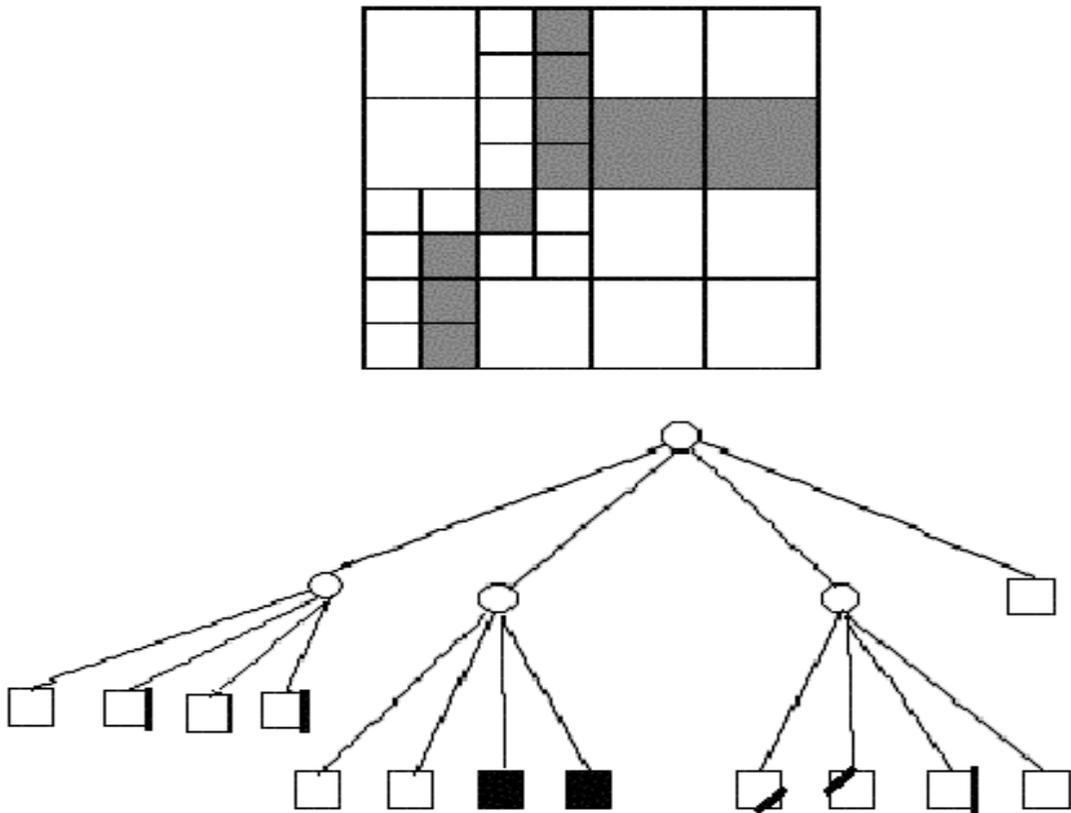


FIGURE 57.5: Template quadtree representation.

In Figure 57.7 downward links are MSON links, horizontal links are MCHAIN links and upward links are MFATHER links.

The compact quadtree uses the same amount of space for the storage of color but it uses very less space for storage of pointers. In Figure 57.7, the compact quadtree has 10 metanodes, whereas the quadtree has 41 nodes. Thus it saves about 85 percent of the storage space. Since the number of nodes in a compact quadtree is less a simple recursive tree traversal can be done more efficiently.

57.4.2 Forest of Quadtrees (FQT)

Jones and Iyengar [8] proposed a variant of quadtrees called forest of quadtrees to improve the space efficiency of quadtrees. A forest of quadtrees is represented by $F(T)$ where T is the quadtree it represents. A forest of quadtrees, $F(T)$ that represents T consists of a table of triples of the form (L, K, P) , and a collection of quadtrees where,

1. Each triple (L, K, P) consists of the coordinates, (L, K) , of a node in T , and a pointer, P , to a quadtree that is identical to the subtree rooted at position (L, K) in T .
2. If (L, K) and (M, N) are coordinates of nodes recorded in $F(T)$, then neither node is a descendant of the other.
3. Every black leaf in T is represented by a black leaf in $F(T)$.

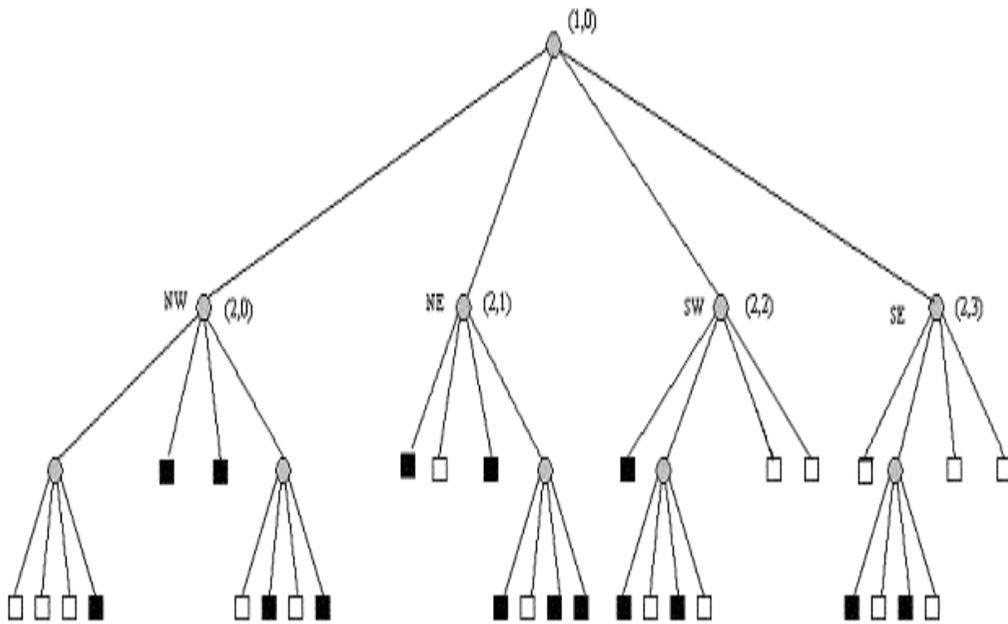


FIGURE 57.6: Quadtree with coordinates.

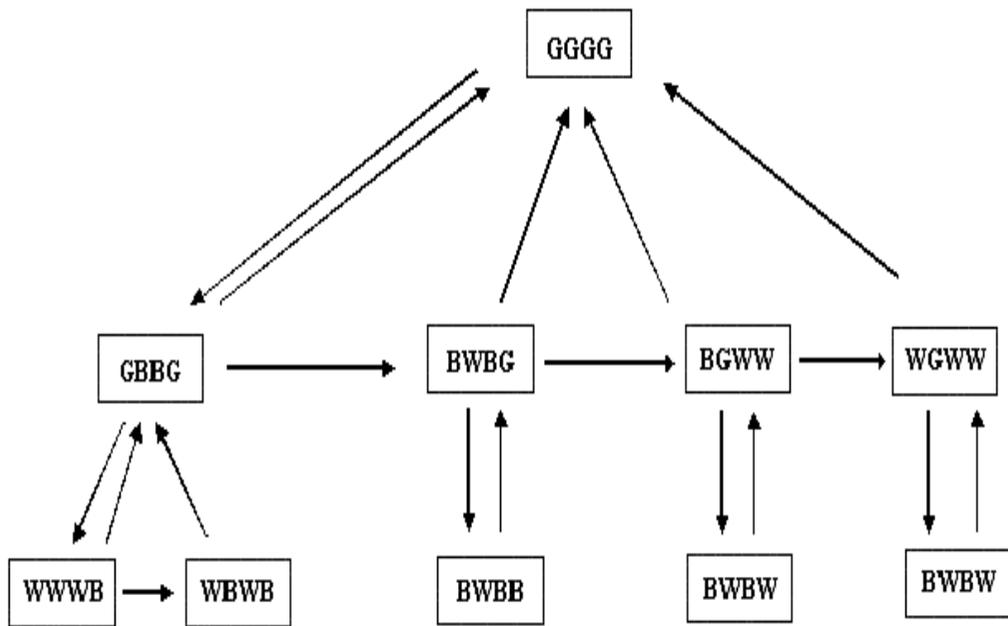


FIGURE 57.7: Compact quadtree $C(T)$.

To obtain the corresponding forest of quadtrees, the nodes in a quadtree need to be classified as good and bad nodes. A leaf node with black color or an intermediate node that has two or more black child nodes are called good nodes, the rest are called bad nodes. Only the good nodes are stored in the forest of quadtrees thereby saving lot of storage space.

Table in the FQT

2	0	Pointer to A
2	1	Pointer to B
3	8	Pointer to C
3	9	Pointer to D
4	3	Pointer to E
3	13	Pointer to F

Trees in the FQT

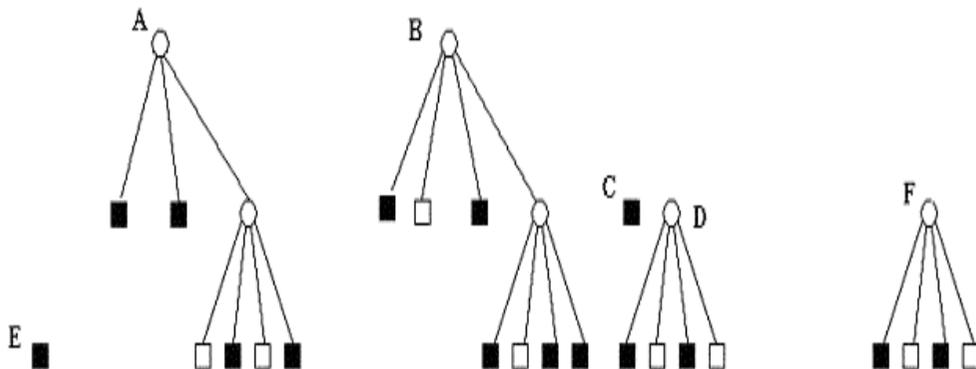


FIGURE 57.8: Forest of quadrees F(T).

Figure 57.8 contains a forest of quadrees that represents the quadtree shown in Figure 57.6.

To reduce a quadtree to a forest of quadrees, we first need to identify the good nodes and the bad nodes. We then break down the quadtree into smaller quadrees in such a way that each of them has a good root node and none of them is a subtree of another and the bad nodes encountered by forest are freed. This collection of quadrees is called as forest of quadrees. The time required for the execution of the conversion is obviously linear in the number of nodes in the quadtree [10].

Theorem: The maximum number of trees in a forest of quadrees derived from a quadtree that represents a square of dimension $2^k \times 2^k$ is 4^{k-1} , i.e., one-fourth the area of the square. For proof see [10].

The corresponding quadtree can be easily reconstructed from a forest of quadrees F. The reconstructed quadtree R(F) consists of real nodes and virtual nodes (nodes corresponding to the bad nodes that are deleted while creating the forest). Since the virtual nodes require no storage they are located by giving their coordinates. The virtual nodes are represented as $v(L, K)$.

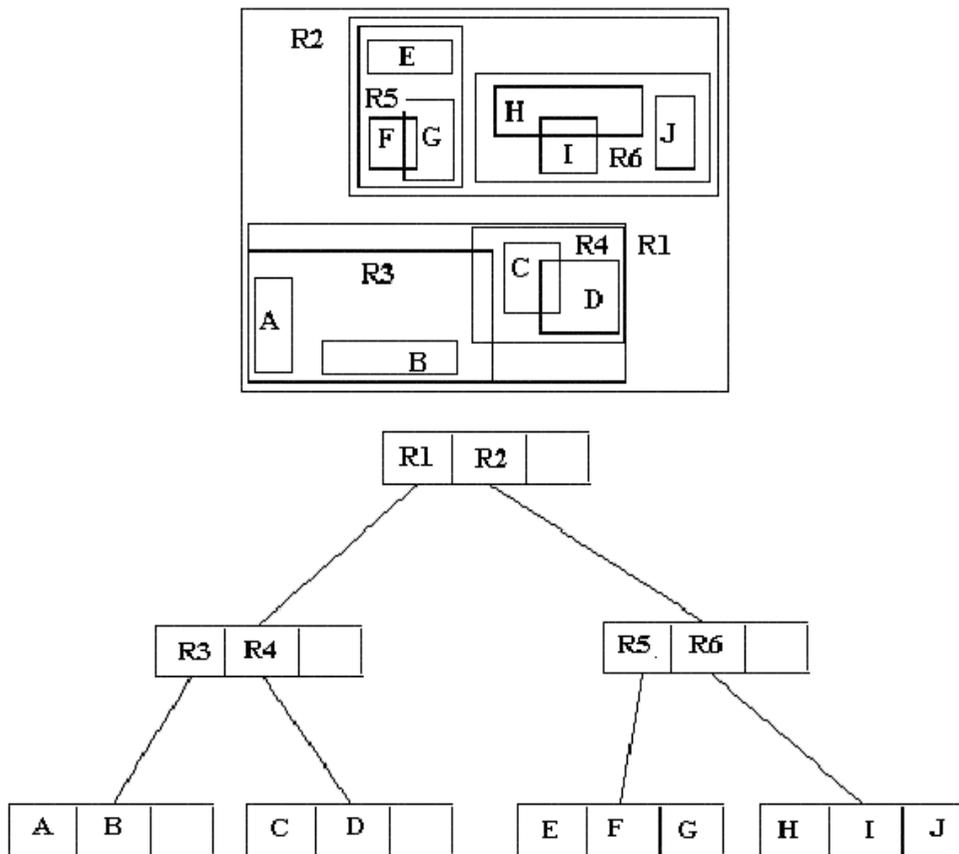


FIGURE 57.9: The concept of R-tree.

57.5 Quadrees and R-trees

Quadrees and R-trees [3] are two commonly used spatial data structures to locate or organize objects in a given image. Both quadrees and R-trees use bounding boxes to depict the outline of the objects. Therefore, the data structure only needs to keep track of the boundaries instead of the actual shape and size of the objects in the image. The size of the bounding boxes usually depends on the size of the object we are trying to locate.

The R-tree (see also [Chapter 21](#)) is a hierarchical, height balanced data structure, similar to the B^+ -tree that aggregates objects based on their spatial proximity. It is a multidimensional generalization of the B^+ -tree. The R-tree indexes multidimensional data objects by enclosing data objects in bounding rectangles, which may further be enclosed in bounding rectangles; these bounding rectangles can overlap each other. Each node except the leaf node has a pointer and the corresponding bounding rectangle. The pointer points to the subtree with all the objects enclosed in its corresponding rectangle. The leaf node has a bounding rectangle and the pointer pointing to the actual data object. The root node has a minimum of two children unless it is a leaf node and all the leaf nodes appear at the same level. Figure 57.9 Shows the R-tree representation of an image.

The main difference between quadrees and R-trees is that unlike R-trees the bounding rectangle of quadrees do not overlap. In real world objects overlap; in such cases more than

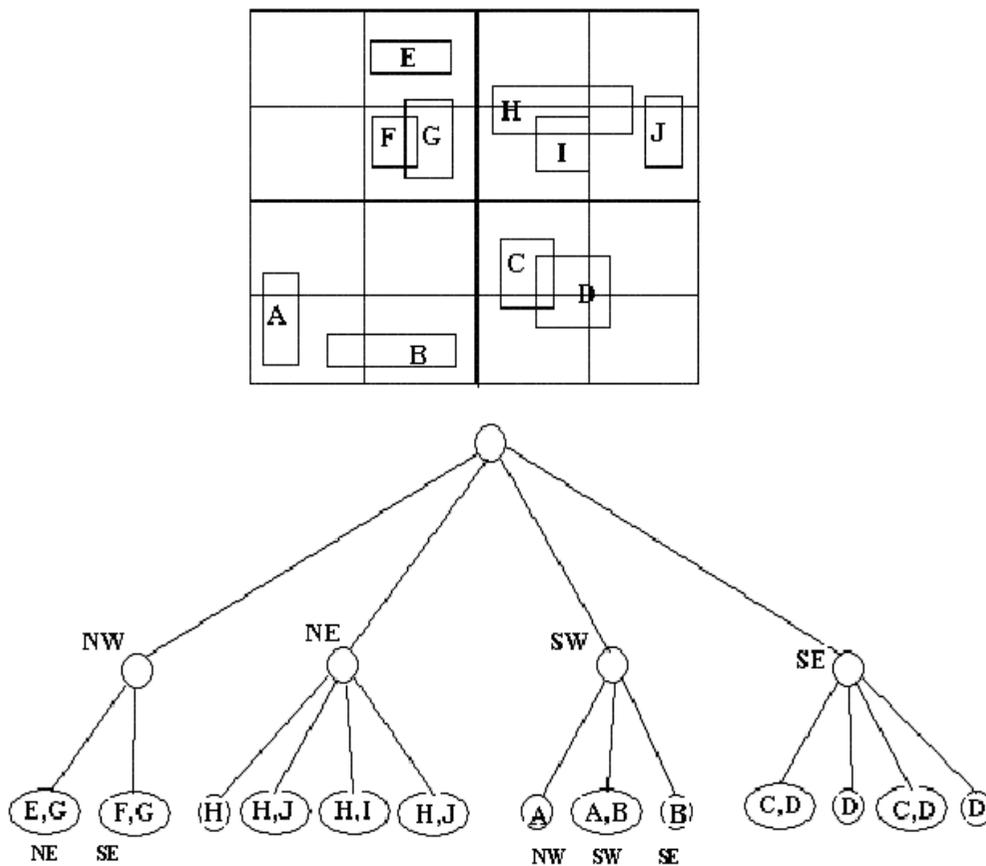


FIGURE 57.10: Quadtree representation of the image in Figure 57.9.

one node in a quadtree can point to the same object. This is a serious disadvantage for the quadtree. Figure 57.10 shows the quadtree representation of the image in Figure 57.9.

As you can see from the Figure 57.10 objects like ‘C’ and ‘D’ are represented by more than one node. This makes it difficult to find the origin of the quadtree.

The R-tree is a dynamic structure, so its contents can be modified without having to reconstruct the entire tree. It can be used to determine which objects intersect a given query region. The R-tree representation of an image is not unique; size and placement of rectangles in the tree depends on the sequence of insertions and deletions resulting in the tree starting from the empty R-tree.

57.6 Octrees

Octrees are 3D equivalent of quadtrees and are used for representing 3D images. An octree is formed by dividing a 3D space into 8 sub-cubes called cells. This process of dividing the cube into cells is carried on until the (image) objects lie entirely inside or outside the cells. The root node of an octree represents a cube, which encompasses the objects of interest.

In the octree each node except the leaf node has eight child nodes, which represent the sub-cubes of the parent node. The node stores information like pointer to the child nodes, pointer to the parent node, and pointers to the contents of the cube. An example of an

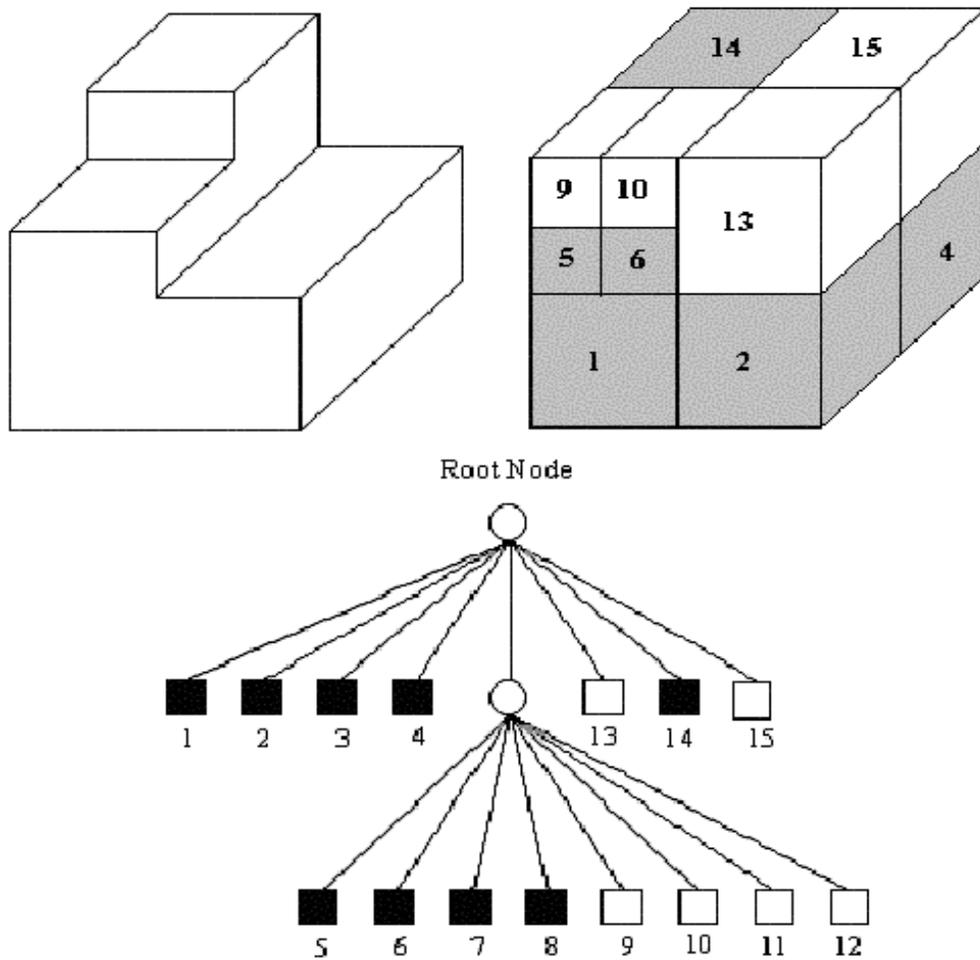


FIGURE 57.11: Octree representation.

octree representation is shown in Figure 57.11.

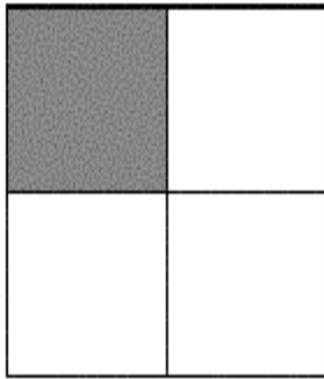
The Octree become very ineffective if the objects in the image that it is representing are not uniformly distributed, as in that case many nodes in the octree will not contain any objects. Such an octree is also very costly to compute as a large amount of data has to be examined.

57.7 Translation Invariant Data Structure (TID)

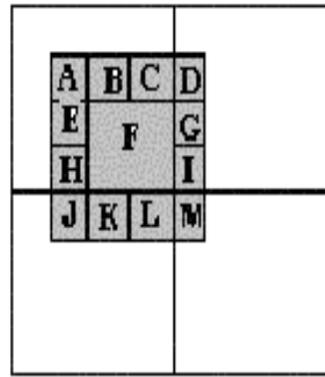
Methods for the region representation using quadtrees exist in the literature [2], [16], [17]. Much work has been done on quadtree properties, and algorithms for translations and manipulations have been derived by Dyer [2], Samet [18], [19], Shneier [24] and others.

Various improvements to quadtrees have been suggested including forests of quadtrees, hybrid quadtrees, linear quadtrees, and optimal quadtrees for image segments. All of these methods try to optimize quadtrees by removing some are all of the gray and white nodes. All of them maintain the same number of black nodes.

All these methods are sensitive to the placement of the origin. An image, which has been



Example 1



Example 2

FIGURE 57.12: Sample Regions.

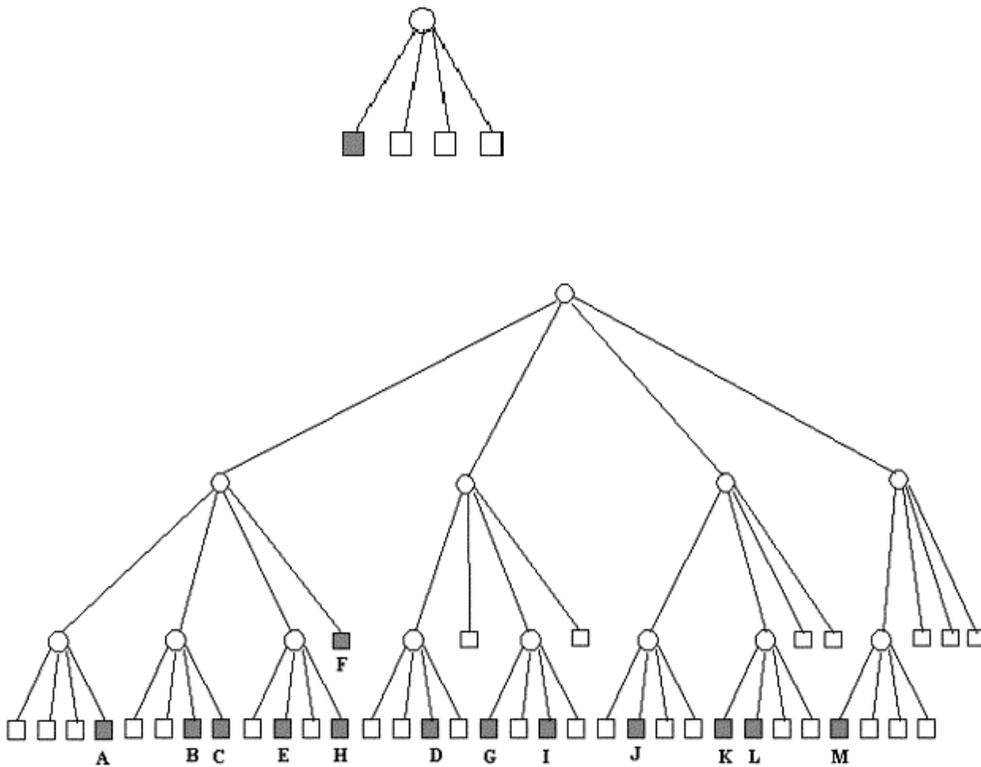


FIGURE 57.13: Quadtrees for Example 1 and 2 of Figure 57.12.

translated from its original position, can have a very different looking structure [22]. We explain this phenomenon by using the example given in Figure 57.12. In Example 1, the black square is in the upper left corner. In Example 2, it is translated down and right by one pixel. Figure 57.13 gives the quadtree representation for these two examples.

The shift sensitivity of the image data structure derives from the fact that the positions of the maximal blocks represented by leaf nodes are not explicitly represented in the data structure. Instead, these positions are determined by the paths leading to them from the root of the tree. Thus, when the image is shifted, the maximal blocks are formed in a different way.

For this reason Scott and Iyengar [22] have introduced a new data structure called Translation Invariant Data structure (TID), which is not sensitive to the placement of the region and is translation invariant.

A maximal square is a black square of pixels that is not included in any large square of black pixels. TID is made of such maximal squares, which are represented as (i, j, s) where (i, j) is the coordinates of the northwest vertex of the square and 's' is the length of the square. Translation made to any image can be represented as a function of these triples. For example consider a triple (i, j, s) , translating it x units to the right and y units up yields $(i+x, j+y, s)$ [22].

The rotation of the square by $\pi/2$ is only slightly more complicated due to the fact that the NW corner of the square changes upon rotation. The $\pi/2$ rotation around the origin gives $(-j, i + s, s)$.

57.8 Content-Based Image Retrieval System

Images have always been a part of human communication. Due to the increase in the use of Internet the interest in the potential of digital images has increased greatly. Therefore we need to store and retrieve images in an efficient way. Locating and retrieving a desired image from a large database can be a very tedious process and is still an active area of research. This problem can be reduced greatly by using Content-Based Image Retrieval (CBIR) systems, which retrieves images based only on the content of the image. This technique retrieves images on the basis of automatically-derived features such as color, texture, and shape.

57.8.1 What is CBIR?

Content-Based Image Retrieval is a process of retrieving desired images from a large database based on the internal features that can be obtained automatically from the images themselves. CBIR techniques are used to index and retrieve images from databases based on their pictorial content, typically defined by a set of features extracted from an image that describe the color, texture, and/or shape of the entire image or of specific objects in the image. This feature description is used to index a database through various means such as distance-based techniques, rule-based decision-making, and fuzzy inferencing [4], [5], [25]. Images can be matched in two ways. Firstly, an image can be compared with another image to check for similarity. Secondly, images similar to the given image can be retrieved by searching a large image database. The latter process is called content-based image retrieval.

General structure of CBIR systems

The general computational framework of a CBIR system as shown in [Figure 57.14](#) was proposed in [26].

At first, the image database is created, which stores the images as numerical values supplied by the feature extraction algorithms. These values are used to locate an image similar to the query image. The query image is processed by the same feature extraction algorithm that is applied to the images stored in the database.

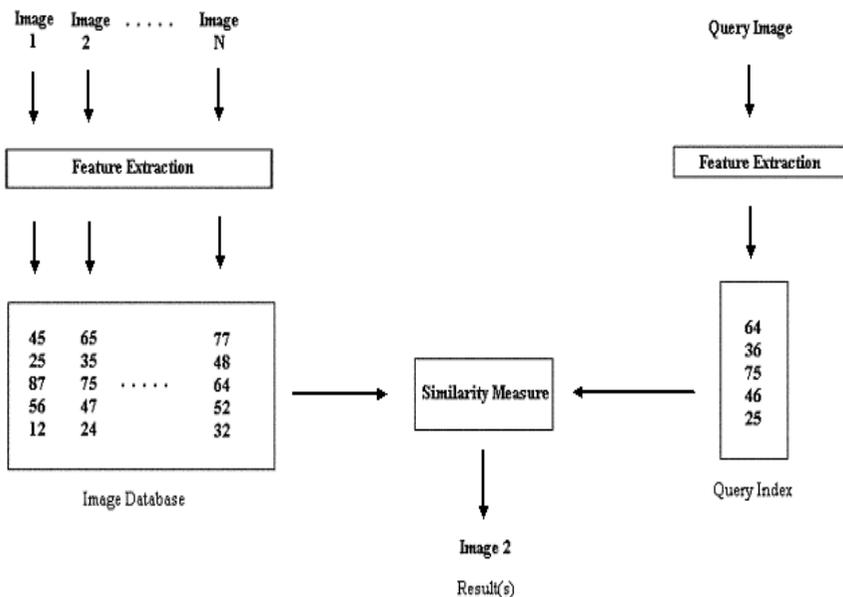


FIGURE 57.14: Computational framework of CBIR systems.

The similarity between the query image and the images stored in the database can be verified with the help of the similarity measure algorithm, which compares the results obtained from the feature extraction algorithms for both the query image and the images in the database. Thus after comparing the query image with all the images in the database the similarity measure algorithm gives a set as the result, which has all the images from the database that are similar to the query image.

57.8.2 An Example of CBIR System

An example of Content-Based Image Retrieval System is BlobWorld. The BlobWorld system, developed at the University of California, Berkeley, supports color, shape, spatial, and texture matching features. Blobworld is based on finding coherent image regions that roughly correspond to objects. The system automatically separates each image into homogeneous regions in order to improve content-based image retrieval. Querying is based on the user specifying attributes of one or two regions of interest, rather than a description of the entire image. For more information on Blobworld see [1].

CBIR techniques are likely to be of most use in restricted subject areas, where merging with other types of data like text and sound can be achieved. Content-based image retrieval provides an efficient solution to the restrictions and the problems caused by the traditional information retrieval technique. The number of active research systems is increasing, which reflects the increasing interest in the field of content-based image retrieval.

57.9 Summary

In this chapter we have explained what an image data is and how it is stored in raster graphics and vector graphics. We then discussed some of the image representation techniques like quadtrees, virtual quadtrees, octrees, R-trees and translation invariant data structures. Fi-

nally, we have given a brief introduction to Content-Based Image Retrieval (CBIR) systems. For more information on image retrieval concepts see [4], [5], [25], [26].

57.10 Acknowledgments

The authors acknowledge the contributions of Sridhar Karra and all the previous collaborators in preparing this chapter.

References

- [1] C. Carson, M. Thomas, S. Belongie, J. M. Hellerstein, and J. Malik, *Blobworld: A System for Region-based Image Indexing and Retrieval.*, Proc. Int'l Conf. Visual Information System, pp. 509-516, 1999.
- [2] C. R. Dyer, A. Rosenfeld and H. Samet, *Region representation: Boundary codes from Quadtrees*, Comm. ACM 23, 3 (March 1980), 171-179.
- [3] A. Guttman, *R-Trees: A Dynamic Index Structure for Spatial Searching.*, Proc. SIGMOD Conf., June 1984, 47-57.
- [4] C. Hsu, W. W. Chu, and R. K. Taira, *A Knowledge-Based Approach for Retrieving Images by Content.*, IEEE Trans. on Knowledge and Data Engineering, Vol. 8, No. 4, August 1996.
- [5] P. W. Huang and Y. R. Jean, *Design of Large Intelligent Image Database Systems.*, International Journal of Intelligent Systems, Vol. 11, 1996, p. 347.
- [6] G. M. Hunter and K. Steiglitz, *Operations On Images Using Quadtrees*, IEEE Trans. on Pattern Analysis and Machine Intell. 1, 2 (April 1979), 145-153.
- [7] G. M. Hunter and K. Steiglitz *Linear Transformation Of Pictures Represented By Quadtrees*, Computer, Graphics and Image Processing 10, 3(July 1979), 289-296.
- [8] L. Jones and S. S. Iyengar, *Representation Of A Region As A Forest Of Quadtrees.*, Proc. IEEE-PRIP 81 Conference, Dallas, TX, IEEE Publications, (1981), 57-59.
- [9] L. Jones and S. S. Iyengar, *Virtual Quadtrees.*, Proc. IEEE-CVIP 83 Conference, Virginia, IEEE publications (1983), 101-105.
- [10] L. P. Jones and S. S. Iyengar, *Space And Time Efficient Virtual Quadtrees.*, IEEE Transactions on Pattern Analysis and Machine Intell. 6, 2 (March 1984), 244-247.
- [11] E. Kawaguchi and T. Endo, *On a Method of Binary-picture Representation and its Application to Data Compression*, IEEE Trans. on Pattern Analysis and Machine Intell. 2, 1 (Jan.1980), 27-35.
- [12] A. Klinger, *Patterns and Search Statistics, Optimizing Methods in Statistics*, J.D. Rustagi, (Ed.), Academic Press, New York (1971).
- [13] A. Klinger and C. R. Dyer, *Experiments in Picture Representation Using Regular Decomposition.*, Computer Graphics and Image Processing, Vol. 5, No. 1 (March 1976), 68-105.
- [14] A. Klinger and M. L. Rhodes, *Organization and Access of Image Data by Areas.*, IEEE Trans. on Pattern Analysis and Machine Intell. 1, 11 (Jan. 1979), 50-60.
- [15] M. Manohar, P. Sudarsana Rao, S. Sitarama Iyengar, *Template Quadtrees for Representing Region and Line Data Present in Binary Images.*, NASA Goddard Flight Center, Greenbelt, MD, 1988.
- [16] H. Samet, *Region Representation: Quadtree from Boundary Codes.*, Comm. ACM 23, 3 (March 1980), 163-170.
- [17] H. Samet, *Region Representation: Quadtree from Binary Arrays.*, Comput. Graphics Image Process. 18, 1 1980, 88-93.

- [18] H. Samet, *An Algorithm for Converting Rasters to Quadrees.*, IEEE Trans. on Pattern Analysis and Machine Intell. 3, 1 (Jan. 1981), 93-95.
- [19] H. Samet, *Connected Component Labeling using Quadrees.*, JACM 28, 3 (July 1981), 487-501.
- [20] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley Pub. Co., 1990.
- [21] H. Samet and R. E. Webber, *On Encoding Boundaries with Quadrees.*, IEEE Trans. on Pattern Analysis and Machine Intell. 6, 1984. 365-369.
- [22] D. S. Scott and S. S. Iyengar, *TID- A Translation Invariant Data Structure For Storing Images.*, Comm. ACM 29, 5 (May 1985), 418-429.
- [23] M. Shneier, *Two Hierarchical Linear Feature Representations: Edge Pyramids and Edge Quadrees.*, Comput. Graphics Image Process. 17, 1981, 211-224.
- [24] M. Shneier, *Path Length Distances for Quadrees.*, Inform.Sci. 23,1 (1981), 49-67.
- [25] B. Tao and B. Dickinson, *Image Retrieval and Pattern Recognition.*, The International Society for Optical Engineering, Vol. 2916, 1996, p. 130.
- [26] J. Zachary, S. S. Iyengar and J. Barhen, *Content Based Image Retrieval and Information Theory: A General Approach.*, JASIS 2000.

Computational Biology

Stefan Kurtz
University of Hamburg

Stefano Lonardi
University of California, Riverside

58.1	Introduction.....	58-1
58.2	Discovering Unusual Words.....	58-1
58.3	Comparing Whole Genomes	58-9

58.1 Introduction

In the last fifteen years, biological sequence data have been accumulating at exponential rate under continuous improvement of sequencing technology, progress in computer science, and steady increase of funding. Molecular sequence databases (e.g., EMBL, Genbank, DDJB, Entrez, Swissprot, etc.) currently collect hundreds of thousand of sequences of nucleotides and amino acids from biological laboratories all over the world, reaching into the hundreds of gigabytes. Such an exponential growth makes it increasingly important to have fast and automatic methods to process, analyze, and visualize massive amounts of data.

The exploration of many computational problems arising in contemporary molecular biology has now grown to become a new field of Computer Science. A coarse selection would include sequence homology and alignment, physical and genetic mapping, protein folding and structure prediction, gene expression analysis, evolutionary trees construction, gene finding, assembly for shotgun sequencing, gene rearrangements and pattern discovery, among others.

In this chapter we focus the attention on the applications of suffix trees to computational biology. In fact, suffix trees and their variants are among the most used (and useful) data structures in computational biology.

Among their applications, we mention pattern discovery [4, 31, 36, 37], alignment of whole genomes [17, 18, 24], detection of exact tandem repeats [46], detection of approximate repeats [27], fragment assembly [15], oligo and probe design [41, 42], etc.

Here we describe two applications. In the first, the use of suffix trees and DAWGs is essential in developing a linear-time algorithm for solving a pattern discovery problem. In the second, suffix trees are used to solve a challenging algorithmic problem: the alignment of two or more complete genomes. (See [Chapters 29](#) and [30](#) for more on suffix trees and DAWGs.)

58.2 Discovering Unusual Words

In the context of computational biology, “pattern discovery” refers to the automatic identification of biologically significant patterns (or *motifs*) by statistical methods. These methods

originated from the work of R. Staden [45] and have been recently expanded in a subfield of data-mining. The underlying assumption is that biologically significant words show distinctive distributional patterns within the genomes of various organisms, and therefore they can be distinguished from the others. The reality is not too far from this hypothesis (see, for example, [11, 12, 19, 25, 35, 39, 40, 51, 52]), because during the evolutionary process, living organisms have accumulated certain biases toward or against some specific motifs in their genomes. For instance, highly recurring oligonucleotides are often found in correspondence to regulatory regions or protein binding sites of genes (see, e.g., [10, 21, 49, 50]). Vice versa, rare oligonucleotide motifs may be discriminated against due to structural constraints of genomes or specific reservations for global transcription controls (see, e.g., [20, 51]).

From a statistical viewpoint, over- and under-represented words have been studied quite rigorously (see, e.g., [44] for a review). A substantial corpus of works has been also produced by the scientific community which studies combinatorics on words (see, e.g., [29, 30] and references therein). In the application domain, however, the “success story” of pattern discovery has been its ability to find previously unknown regulatory elements in *DNA* sequences (see, e.g., [48, 49]). Regulatory elements control the expression (i.e., the amount of *mRNA* produced) of the genes over time, as the result of different external stimuli to the cell and other metabolic processes that take place internally in the cell. These regulatory elements are typically, but not always, found in the *upstream* sequence of a gene. The upstream sequence is defined as a portion of *DNA* of 1-2Kbases of length, located upstream of the site that controls the start of transcription. The regulatory elements correspond to binding sites for the factors involved in the transcriptional process. The complete characterization of these elements is a critical step in understanding the function of different genes, and the complex network of interaction between them.

The use of pattern discovery to find regulatory elements relies on a conceptual hypothesis that genes which exhibit similar expression patterns are assumed to be involved in the same biological process or functions. Although many believe that this assumption is an oversimplification, it is still a good working hypothesis. Co-expressed genes are therefore expected to share common regulatory domains in the upstream regions for the coordinated control of gene expression. In order to elucidate genes which are co-expressed as a result of a specific stress or condition, a *DNA* microarray experiment is typically designed. After collecting data from the microarray, co-expressed genes are usually obtained by clustering the time series corresponding to their expression profiles.

As said, words that occur unexpectedly often or rarely in genetic sequences have been variously linked to biological meanings and functions. With increasing availability of whole genomes, exhaustive statistical tables and global detectors of unusual words on a scale of millions, even billions of bases become conceivable. It is natural to ask how large such tables may grow with increasing length of the input sequence, and how fast they can be computed. These problems need to be regarded not only from the conventional perspective of asymptotic space and time complexities, but also in terms of the volumes of data produced and ultimately, of practical accessibility and usefulness. Tables that are too large at the outset saturate the perceptual bandwidth of the user, and might suggest approaches that sacrifice some modeling accuracy in exchange for an increased throughput.

The number of distinct substrings in a string is at worst quadratic in the length of that string. The situation does not improve if one restricts himself to computing and displaying the most *unusual* words in a given sequence. This presupposes comparing the frequency of occurrence of every word in that sequence with its expectation: a word that departs from expectation beyond some pre-set *threshold* will be labeled as *unusual* or *surprising*. Departure from expectation is assessed by a distance measure often called a *score* function. The typical format for a *z*-score is that of a difference between observed and expected

counts, usually normalized to some suitable moment. For most *a priori* models of a source, it is not difficult to come up with extremal examples of *observed* sequences in which the number of, say, over-represented substrings grows itself with the square of the sequence length.

An extensive study of probabilistic models and scores for which the population of potentially unusual words in a sequence can be described by tables of size at worst linear in the length of that sequence was carried out in [4]. That study not only leads to more palatable representations for those tables, but also supports (non-trivial) linear time and space algorithms for their constructions, as described in what follows. These results do not mean that the number of unusual words must be linear in the input, but just that their representation and detection can be made such. Specifically, it is seen that it suffices to consider as candidate surprising words only the members of an *a priori* well identified set of “representative” words, where the cardinality of that set is linear in the text length. By the representatives being identifiable *a priori* we mean that they can be known before any score is computed. By neglecting the words other than the representatives we are not ruling out that those words might be surprising. Rather, we maintain that any such word: (i) is embedded in one of the representatives, and (ii) does not have a bigger score or degree of surprise than its representative (hence, it would add no information to compute and give its score explicitly).

Statistical analysis of words

For simplicity of exposition, assume that the source can be modeled by a Bernoulli distribution, i.e., symbols are generated i.i.d., and that strings are ranked based on their number of occurrences (possibly overlapping). The results reported in the rest of this section can be extended to other models and counts (see [4]).

We use standard concepts and notation about strings, for which we refer to [5, 6]. For a substring y of a text x over an alphabet Σ , we denote by $f_x(y)$ the number of occurrences of y in x . We have $f_x(y) = |\text{pos}_x(y)| = |\text{endpos}_x(y)|$, where $\text{pos}_x(y)$, is the *start-set* of starting positions of y in x and $\text{endpos}_x(y)$ is the similarly defined *end-set*. Clearly, for any *extension* uyv of y , $f_x(uyv) \leq f_x(y)$.

Suppose now that string $x = x[0]x[1] \dots x[n-1]$ is a realization of a stationary ergodic random process and $y[0]y[1] \dots y[m-1] = y$ is an arbitrary but fixed pattern over Σ with $m < n$. We define Z_i , for all $i \in [0 \dots n-m]$, to be 1 if y occurs in x starting at position i , 0 otherwise, so that

$$Z_y = \sum_{i=0}^{n-m} Z_i$$

is the random variable for $f_x(y)$.

Expressions for the expectation and the variance for the number of occurrences in the Bernoulli model, have been given by several authors (see, e.g., [22, 26, 38, 43, 47]). Here we adopt derivations in [5, 6]. With p_a the probability of symbol $a \in \Sigma$ and $\hat{p} = \prod_{i=0}^{m-1} p_{y[i]}$, we have

$$\begin{aligned} E(Z_y) &= (n-m+1)\hat{p} \\ \text{Var}(Z_y) &= \begin{cases} (1-\hat{p})E(Z_y) - \hat{p}^2(2n-3m+2)(m-1) + 2\hat{p}B(y) & \text{if } m \leq (n+1)/2 \\ (1-\hat{p})E(Z_y) - \hat{p}^2(n-m+1)(n-m) + 2\hat{p}B(y) & \text{otherwise} \end{cases} \end{aligned}$$

where

$$B(y) = \sum_{d \in \mathcal{P}(y)} (n - m + 1 - d) \prod_{j=m-d}^{m-1} p_{y[j]} \quad (58.1)$$

is the *auto-correlation factor* of y , that depends on the set $\mathcal{P}(y)$ of the lengths of the periods* of y .

Given $f_x(y)$, $E(Z_y)$ and $Var(Z_y)$, a statistical significance score that measures the degree of “unusual-ness” of a substring y must be carefully chosen. Ideally, the score function should be independent of the structure and size of the word. That would allow one to make meaningful comparisons among substrings of various compositions and lengths based on the value of the score.

There is some general consensus that z -scores may be preferred over other types of score function [28]. For any word w , a standardized frequency called z -score, can be defined by

$$z(y) = \frac{f_x(y) - E(Z_y)}{\sqrt{Var(Z_y)}}$$

If $E(Z_y)$ and $Var(Z_y)$ are known, then under rather general conditions, the statistics $z(y)$ is asymptotically normally distributed with zero mean and unit variance as n tends to infinity. In practice $E(Z_y)$ and $Var(Z_y)$ are seldom known, but are estimated from the sequence under study.

Detecting unusual words

Consider now the problem of computing exhaustive tables reporting scores for all substrings of a sequence, or perhaps at least for the most surprising among them. While the complexity of the problem ultimately depends on the probabilistic model and type of count, a table for all words of any size would require at least quadratic space in the size of the input, not to mention that such a table would take at least quadratic time to be filled.

As seen towards the end of the section, such a limitation can be overcome by partitioning the set of all words into equivalence classes with the property, that it suffices to account for only one or two candidate surprising words in each class, while the number of classes is linear in the textstring size. More formally, given a score function z , a set of words C , and a real positive *threshold* T , we say that a word $w \in C$ is *T -over-represented* in C (resp., *T -under-represented*) if $z(w) > T$ (resp., $z(w) < -T$) and for all words $y \in C$ we have $z(w) \geq z(y)$ (resp., $z(w) \leq z(y)$). We say that a word w is *T -surprising* if $z(w) > T$ or $z(w) < -T$. We also call $\max(C)$ and $\min(C)$ respectively the longest and the shortest word in C , when $\max(C)$ and $\min(C)$ are unique.

Let now x be a textstring and $\{C_1, C_2, \dots, C_l\}$ a partition of all its substrings, where $\max(C_i)$ and $\min(C_i)$ are uniquely determined for all $1 \leq i \leq l$. For a given score z and a real positive constant T , we call \mathcal{O}_z^T the set of T -over-represented words of C_i , $1 \leq i \leq l$, with respect to that score function. Similarly, we call \mathcal{U}_z^T the set of T -under-represented words of C_i , and \mathcal{S}_z^T the set of all T -surprising words, $1 \leq i \leq l$.

For strings u and $v = suz$, a (u, v) -*path* is a sequence of words $\{w_0 = u, w_1, w_2, \dots, w_j = v\}$, $l \geq 0$, such that w_i is a unit-symbol extension of w_{i-1} ($1 \leq i \leq j$). In general a (u, v) -

*String z has a *period* w if z is a non-empty prefix of w^k for some integer $k \geq 1$.

path is not unique. If all $w \in C$ belong to some $(\min(C_i), \max(C_i))$ -path, we say that class C is *closed*.

A score function z is (u, v) -*increasing* (resp., *non-decreasing*) if given any two words w_1, w_2 belonging to a (u, v) -path, the condition $|w_1| < |w_2|$ implies $z(w_1) < z(w_2)$ (resp., $z(w_1) \leq z(w_2)$). The definitions of a (u, v) -*decreasing* and (u, v) -*non-increasing* z -scores are symmetric. We also say that a score z is (u, v) -*monotone* when specifics are unneeded or understood. The following fact and its symmetric are immediate.

FACT 58.1 *If the z -score under the chosen model is $(\min(C_i), \max(C_i))$ -increasing, and C_i is closed, $1 \leq i \leq l$, then*

$$\mathcal{O}_z^T \subseteq \bigcup_{i=1}^l \{\max(C_i)\} \quad \text{and} \quad \mathcal{U}_z^T \subseteq \bigcup_{i=1}^l \{\min(C_i)\}$$

In [4], extensive results on the monotonicity of several scores for different probabilistic models and counts are reported. For the purpose of this chapter, we just need the following result.

THEOREM 58.1 *(Apostolico et al. [4]) Let x be a text generated by a Bernoulli process, and p_{\max} be the probability of the most frequent symbol.*

If $f_x(w) = f_x(wv)$ and $p_{\max} < \min\{1/\sqrt[m]{4m}, \sqrt{2} - 1\}$ then

$$\frac{f_x(wv) - E(Z_{wv})}{\sqrt{\text{Var}(Z_{wv})}} > \frac{f_x(w) - E(Z_w)}{\sqrt{\text{Var}(Z_w)}}$$

Here, we pursue substring partitions $\{C_1, C_2, \dots, C_l\}$ in forms which would enable us to restrict the computation of the scores to a constant number of candidates in each class C_i . Specifically, we require, (1) for all $1 \leq i \leq l$, $\max(C_i)$ and $\min(C_i)$ to be unique; (2) C_i to be closed, i.e., all w in C_i belong to some $(\min(C_i), \max(C_i))$ -path; (3) all w in C_i have the same count. Of course, the partition of all substrings of x into singleton classes fulfills those properties. In practice, l should be as small as possible.

We begin by recalling a few basic facts and constructs from, e.g., [9]. We say that two strings y and w are *left-equivalent* on x if the set of starting positions of y in x matches the set of starting positions of w in x . We denote this equivalence relation by \equiv_l . It follows from the definition that if $y \equiv_l w$, then either y is a prefix of w , or vice versa. Therefore, each class has unique shortest and longest word. Also by definition, if $y \equiv_l w$ then $f_x(y) = f_x(w)$.

Example 58.1

For instance, in the string `ataatataataatataatag` the set `{ataa, ataat, ataata}` is a left-equivalent class (with position set `{1, 6, 9, 14}`) and so are `{taa, taat, taata}` and `{aa, aat, aata}`. We have 39 left-equivalent classes, much less than the total number of substrings, which is $22 \times 23/2 = 253$, and than the number of distinct substrings, in this case 61.

We similarly say that y and w are *right-equivalent* on x if the set of ending positions of y in x matches the set of ending positions of w in x . We denote this by \equiv_r . Finally, the

equivalence relation \equiv_x is defined in terms of the *implication* of a substring of x [9, 16]. Given a substring w of x , the implication $imp_x(w)$ of w in x is the longest string uvw such that every occurrence of w in x is preceded by u and followed by v . We write $y \equiv_x w$ iff $imp_x(y) = imp_x(w)$. It is not difficult to see that

LEMMA 58.1 The equivalence relation \equiv_x is the transitive closure of $\equiv_l \cup \equiv_r$.

More importantly, the size l of the partition is linear in $|x| = n$ for all three equivalence relations considered. In particular, the smallest size is attained by \equiv_x , for which the number of equivalence classes is at most $n + 1$.

Each one of the equivalence classes discussed can be mapped to the nodes of a corresponding automaton or word graph, which becomes thereby the natural support for the statistical tables. The table takes linear space, since the number of classes is linear in $|x|$. The automata themselves are built by classical algorithms, for which we refer to, e.g., [5, 9] with their quoted literature, or easy adaptations thereof. The graph for \equiv_l , for instance, is the compact subword tree T_x of x , whereas the graph for \equiv_r is the *DAWG*, or *Directed Acyclic Word Graph* D_x , for x . The graph for \equiv_x is the compact version of the *DAWG*.

These data structures are known to commute in simple ways, so that, say, an \equiv_x -class can be found on T_x as the union of some left-equivalent classes or, alternatively, as the union of some right-equivalent classes. Beginning with left-equivalent classes, that correspond one-to-one to the nodes of T_x , we can build some right-equivalent classes as follows. We use the elementary fact that whenever there is a branching node μ in T_x , corresponding to $w = ay, a \in \Sigma$, then there is also a node ν corresponding to y , and there is a special *suffix* link directed from ν to μ . Such auxiliary links induce another tree on the nodes of T_x , that we may call S_x . It is now easy to find a right-equivalent class with the help of suffix links. For this, traverse S_x bottom-up while grouping in a single class all strings such that their terminal nodes in T_x are roots of isomorphic subtrees of T_x . When a subtree that violates the isomorphism condition is encountered, we are at the end of one class and we start with a new one.

Example 58.2

For example, the three subtrees rooted at the solid nodes in [Figure 58.1](#) correspond to the end-sets of **ataata**, **taata** and **aata**, which are the same, namely, $\{6, 11, 14, 19\}$. These three words define the right-equivalent class $\{\mathbf{ataata}, \mathbf{taata}, \mathbf{aata}\}$. In fact, this class cannot be made larger because the two subtrees rooted at the end nodes of **ata** and **tataata** are not isomorphic to the subtree of the class. We leave it as an exercise for the reader to find *all* the right-equivalence classes on T_x . It turns out that there are 24 such classes in this example.

Subtree isomorphism can be checked by a classical linear-time algorithm by Aho *et al.* [3]. But on the suffix tree T_x this is done even more quickly once the f counts are available (see, e.g., [8, 23]).

LEMMA 58.2 Let T_1 and T_2 be two subtrees of T_x . T_1 and T_2 are isomorphic if and only if they have the same number of leaves and their roots are connected by a chain of suffix links.

If, during the bottom-up traversal of S_x , we collect in the same class strings such that

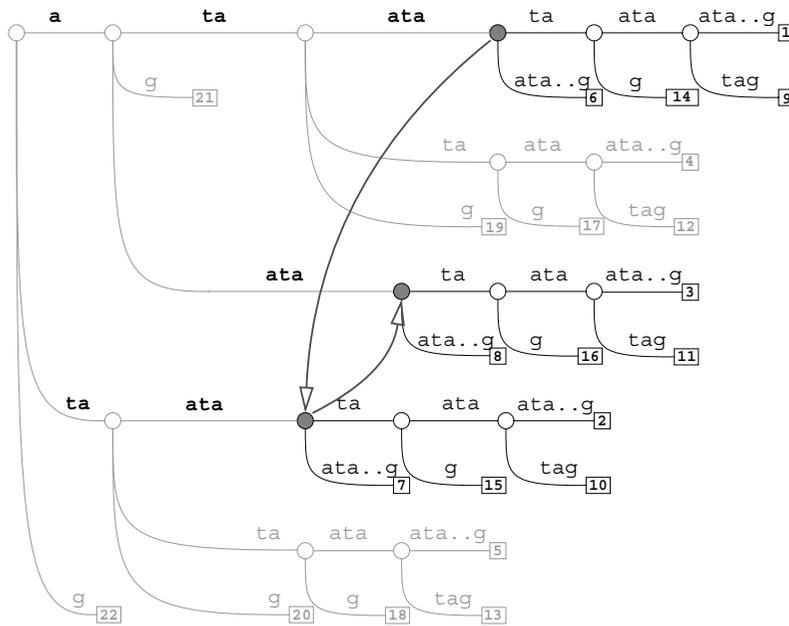


FIGURE 58.1: The tree T_x for $x = \text{ataataataataataatag}$: subtrees rooted at the solid nodes are isomorphic.

their terminal *arc* leads to nodes with the same frequency counts f , then this would identify and produce the \equiv_x -classes, i.e., the smallest substring partition.

Example 58.3

For instance, starting from the right-equivalent class $C = \{\text{ataata}, \text{taata}, \text{aata}\}$, one can augment it with of all words which are left-equivalent to the elements of C . The result is one \equiv_x -class composed by $\{\text{ataa}, \text{ataat}, \text{ataata}, \text{taa}, \text{taat}, \text{taata}, \text{aa}, \text{aat}, \text{aata}\}$. Their respective *pos* sets are $\{1,6, 9,14\}, \{1,6, 9,14\}, \{1,6, 9,14\}, \{2,7, 10,15\}, \{2,7, 10,15\}, \{2,7, 10,15\}, \{3,8, 11,16\}, \{3,8, 11,16\}, \{3,8, 11,16\}$. Their respective *endpos* sets are $\{4,9, 12,17\}, \{5,10, 13,18\}, \{6,11, 14,19\}, \{4,9, 12,17\}, \{5,10, 13,18\}, \{6,11, 14,19\}, \{4,9, 12,17\}, \{5,10, 13,18\}, \{6,11, 14,19\}$. Because of Lemma 58.1, given two words y and w in the class, either they share the start set, or they share the end set, or they share the start set by transitivity with a third word in the class, or they share the end set by transitivity with a third word in the class. It turns out that there are only seven \equiv_x -classes in this example.

Note that the longest string in this \equiv_x -class is unique (**ataata**) and that it contains all the others as substrings. The shortest string is unique as well (**aa**). As said, the number of occurrences for all the words in the same class is the same (4 in the example). Figure 58.2 illustrates the seven equivalence classes for the running example. The words in each class have been organized in a lattice, where edges correspond to extensions (or contractions) of a single symbol. In particular, horizontal edges correspond to right extensions and vertical edges to left extensions.

While the longest word in an \equiv_x -class is unique, there may be in general more than one shortest word. Consider, for example, the text $x = \mathbf{a}^k \mathbf{g}^k$, with $k > 0$. Choosing $k = 2$ yields a class which has three words of length two as minimal elements, namely, **aa**, **gg**, and **ag**. (In fact, $\text{imp}_x(\mathbf{aa}) = \text{imp}_x(\mathbf{gg}) = \text{imp}_x(\mathbf{ag}) = \mathbf{aagg}$.) Taking instead $k = 1$, all three

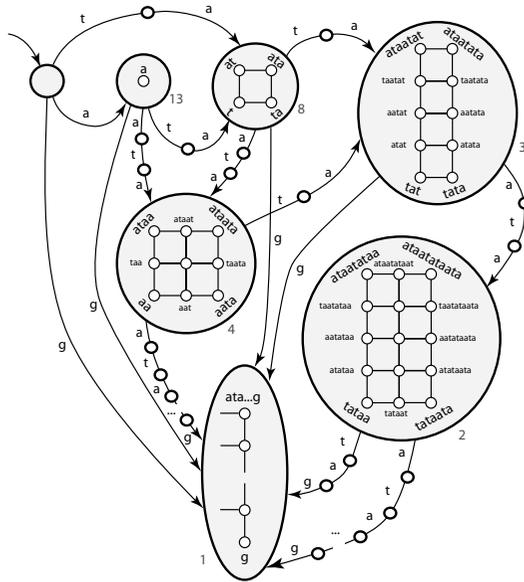


FIGURE 58.2: A representation of the seven \equiv_x -classes for $x = ataataataataataatag$. The words in each class can be organized in a lattice. Numbers refer to the number of occurrences.

substrings of $x = ag$ coalesce into a single class which has two shortest words.

Recall that by Lemma 58.1 each \equiv_x -class C can be expressed as the union of one or more left-equivalent classes. Alternatively, C can be also expressed as the union of one or more right-equivalent classes. The example above shows that there are cases in which left- or right-equivalent classes *cannot* be merge without violating the uniqueness of the shortest word. Thus, we may use the \equiv_x -classes as the C_i 's in our partition only if we are interested in detecting over-represented words. If under-represented words are also wanted, then we must represent a same \equiv_x -class once for each distinct shortest word in it.

It is not difficult to accommodate this in the subtree merge procedure. Let $p(u)$ denote the parent of u in T_x . While traversing S_x bottom-up, merge two nodes u and v with the same f count if and only if u and v are connected by a suffix link and so are $p(u)$ and $p(v)$. This results in a substring partition slightly coarser than \equiv_x . It will be denoted by $\tilde{\equiv}_x$. In conclusion, we can state the following fact.

FACT 58.2 *Let $\{C_1, C_2, \dots, C_l\}$ be the set of equivalence classes built on the equivalence relation $\tilde{\equiv}_x$ on the substrings of text x . Then, for all $1 \leq i \leq l$,*

1. $\max(C_i)$ and $\min(C_i)$ are unique
2. all $w \in C_i$ are on some $(\min(C_i), \max(C_i))$ -path
3. all $w \in C_i$ have the same number of occurrences $f_x(w)$

We are now ready to address the computational complexity of our constructions. In [5], linear-time algorithms are given to compute and store expected value $E(Z_w)$ and variance $Var(Z_w)$ for the number of occurrences under Bernoulli model of *all* prefixes of a given string. The crux of that construction rests on deriving an expression of the variance (see Expression 58.1) that can be cast within the classical linear time computation of the “failure

function” or smallest periods for all prefixes of a string (see, e.g., [3]). These computations are easily adapted to be carried out on the linked structure of graphs such as S_x or D_x , thereby yielding expectation and variance values at all nodes of T_x , D_x , or the compact variant of the latter. These constructions take time and space linear in the size of the graphs, hence linear in the length of x . Combined with the monotonicity results this yields immediately:

THEOREM 58.2 (Apostolico et al. [4]) Under the Bernoulli model, the sets \mathcal{O}_z^T and \mathcal{U}_z^T associated to the score

$$z(w) = \frac{f_x(w) - E(Z_w)}{\sqrt{\text{Var}(Z_w)}}$$

can be computed in linear time and space, provided that $p_{max} < \min\{1/\sqrt[m]{4m}, \sqrt{2} - 1\}$.

The algorithm is implemented in a software tool called VERBUMCULUS that can be found at <http://www.cs.ucr.edu/~stelo/Verbunculus/>. A description of the tool and a demonstration of its applicability to the analysis of biological datasets will appear in [7].

58.3 Comparing Whole Genomes

As of today (mid 2003), Genbank contains “complete” genomes for more than 1,000 viruses, over 100 microbes, and about 100 eukariota. The abundance of complete genome sequences has given an enormous boost to comparative genomics. Association studies are emerging as a powerful tool for the functional identification of genes and molecular genetics has begun to reveal the biological basis of diversity. Comparing the genomes of related species gives us new insights into the complex structure of organisms at the *DNA*-level and protein-level.

The first step when comparing genomes is to produce an alignment, i.e., a collinear arrangement of sequence similarities. Alignment of nucleic or amino acid sequences has been one of the most important methods in sequence analysis, with much dedicated research and now many sophisticated algorithms available for aligning sequences with similar regions. These require assigning a score to all the possible alignments (typically, the sum of the similarity/identity values for each aligned symbol, minus a penalty for the introduction of gaps), along with a dynamic programming method to find optimal or near-optimal alignments according to this scoring scheme (see, e.g., [34]). These dynamic programming methods run in time proportional to the product of the length of the sequences to be aligned. Hence they are not suitable for aligning entire genomes. Recently several genome alignment programs have been developed, all using an anchor-based method to compute an alignment (for an overview see [13]). An *anchor* is an exact match of some minimum length occurring in all genomes to be aligned (see Figure 58.3). The anchor-based method can roughly be divided into the following three phases:

- (1) Computation of all potential anchors.
- (2) Computation of an optimal collinear sequence of non-overlapping potential anchors: these are the anchors that form the basis of the alignment (see Figure 58.4).
- (3) Closure of the gaps in between the anchors.

In the following, we will focus on phase (1) and explain two algorithms to compute potential anchors. The first algorithm allows one to align more than two genomes, while

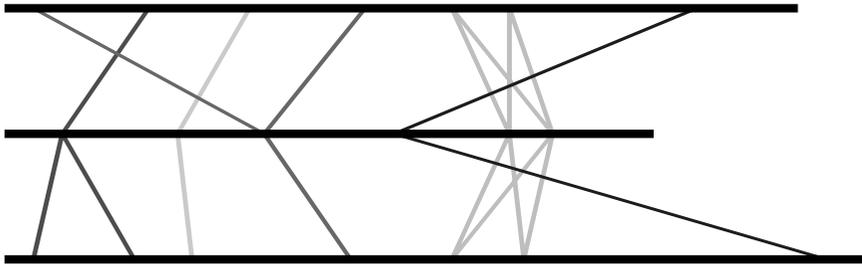


FIGURE 58.3: Three genomes represented as horizontal lines. Potential anchors of an alignment are connected by vertical lines. Each anchor is a sequence occurring at least once in all three genomes.

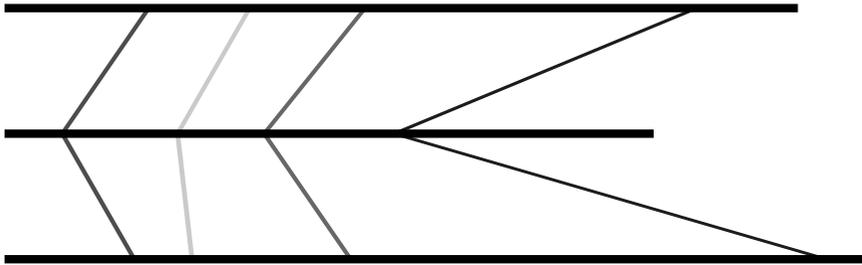


FIGURE 58.4: A collinear chain of non-overlapping anchors.

the second is limited to two genomes, but uses less space than the former. Both algorithms are based on suffix trees. For phase (2) of the anchor-based method, one uses methods from computational geometry. The interested reader is referred to the algorithms described in [1, 2, 33, 54]. In phase (3), one can apply any standard alignment methods, or even the same anchor-based method with relaxed parameters.

Basic Definitions

We recall and extend the definition introduced in Section 58.2. A *sequence*, or a *string*, S of length n is written as $S = S[0]S[1] \dots S[n-1] = S[0 \dots n-1]$. A *prefix* of S is a sequence $S[0 \dots i]$ for some $i \in [0, n-1]$. A *suffix* of S is a sequence $S[i \dots n-1]$ for some $i \in [0, n-1]$. Consider a set $\{G_0, \dots, G_{k-1}\}$ of $k \geq 2$ sequences (the genomes) over some alphabet Σ . Let $n_q = |G_q|$ for $q \in [0, k-1]$. To simplify the handling of boundary cases, assume that $G_0[-1] = \$_{-1}$ and $G_{k-1}[n_{k-1}] = \$_{k-1}$ are unique symbols not occurring in Σ . A *multiple exact match* is a $(k+1)$ -tuple $(l, p_0, p_1, \dots, p_{k-1})$ such that $l > 0$, $p_q \in [0, n_q - l]$, and $G_q[p_q \dots p_q + l - 1] = G_{q'}[p_{q'} \dots p_{q'} + l - 1]$ for all $q, q' \in [0, k-1]$. A multiple exact match is *left maximal* if for at least one pair $(q, q') \in [0, k-1] \times [0, k-1]$, we have $G_q[p_q - 1] \neq G_{q'}[p_{q'} - 1]$. A multiple exact match is *right maximal* if for at least one pair $(q, q') \in [0, k-1] \times [0, k-1]$, we have $G_q[p_q + l] \neq G_{q'}[p_{q'} + l]$. A multiple exact match is *maximal* if it is left maximal and right maximal. A maximal multiple exact match is also called *multiMEM*. Roughly speaking, a *multiMEM* is a sequence of length l that occurs in all sequences G_0, \dots, G_{k-1} (at positions p_0, \dots, p_{k-1}), and cannot simultaneously be extended to the left or to the right in every sequence. The ℓ -*multiMEM*-problem is to enumerate all *multiMEMs* of length at least ℓ for some given length threshold $\ell \geq 1$. For $k = 2$, we use the notion *MEM* and ℓ -*MEM*-problem.

Let $\$0, \dots, \$_{k-1}$ be pairwise different symbols not occurring in any G_q . These symbols are used to separate the sequences in the concatenation $S = G_0\$0G_1\$1 \dots G_{k-2}\$_{k-2}G_{k-1}\$_{k-1}$. $\$_{k-1}$ will be used as a sentinel attached to the end of G_{k-1} . All these special symbols are collected in a set $Seps = \{\$_{-1}, \$0, \dots, \$_{k-2}, \$_{k-1}\}$.

Let $n = |S| = k - 1 + \sum_{q=0}^{k-1} n_q$. For any $i \in [0, n]$, let $S_i = S[i \dots n - 1]\$_{k-1}$ denote the i th non-empty suffix of $S\$_{k-1}$. Hence $S_n = \$_{k-1}$. Define $t_0 = 0$ and $t_q = t_{q-1} + n_{q-1} + 1$ for any $q \in [1, k]$. t_q is the start position of G_q in S for $q \in [0, k - 1]$. Let $i \in [0, n - 1]$ such that $S[i] \notin \{\$0, \dots, \$_{k-2}\}$. Define two functions σ and ρ as follows:

- $\sigma(i) = q$ if and only if $i \in [t_q, t_{q+1} - 2]$
- $\rho(i) = i - t_{\sigma(i)}$

That is, position i in S is identified with the relative position $\rho(i)$ in sequence $G_{\sigma(i)}$.

We consider trees whose edges are labeled by non-empty sequences. For each symbol a , every node α in these trees has at most one a -edge $\alpha \xrightarrow{av} \beta$ for some sequence v and some node β . Suppose a tree T and let α be a node in T . A node α is denoted by \bar{w} if and only if w is the concatenation of the edge labels on the path from the root of T to α . A sequence w occurs in T if and only if T contains a node \bar{wv} , for some sequence v . The *suffix tree* for S , denoted by $ST(S)$, is the tree T with the following properties: (i) each node is either the root, a leaf or a branching node, and (ii) a sequence w occurs in T if and only if w is a substring of $S\$_{k-1}$. For each branching node \bar{au} in $ST(S)$, where a is a symbol and u is a string, \bar{u} is also a branching node, and there is a *suffix link* from \bar{au} to \bar{u} .

There is a one-to-one correspondence between the leaves of $ST(S)$ and the non-empty suffixes of $S\$_{k-1}$: Leaf \bar{S}_i corresponds to suffix S_i and vice versa.

For any node \bar{u} of $ST(S)$ (including the leaves), let $\mathcal{P}_{\bar{u}}$ be the set of positions i such that u is a prefix of S_i . In other words, $\mathcal{P}_{\bar{u}}$ is the set of positions in S where sequence u starts. $\mathcal{P}_{\bar{u}}$ is divided into disjoint and possibly empty position sets:

- For any $q \in [0, k - 1]$, define $\mathcal{P}_{\bar{u}}(q) = \{i \in \mathcal{P}_{\bar{u}} \mid \sigma(i) = q\}$, i.e. $\mathcal{P}_{\bar{u}}(q)$ is the set of positions i in S where u starts and i is a position in genome G_q .
- For any $a \in \Sigma \cup Seps$, define $\mathcal{P}_{\bar{u}}(a) = \{i \in \mathcal{P}_{\bar{u}} \mid S[i - 1] = a\}$, i.e. $\mathcal{P}_{\bar{u}}(a)$ is the set of positions i in S where u starts and the symbol to the left of this position is a .
- For any $q \in [0, k - 1]$ and any $a \in \Sigma \cup Seps$, define $\mathcal{P}_{\bar{u}}(q, a) = \mathcal{P}_{\bar{u}}(q) \cap \mathcal{P}_{\bar{u}}(a)$.

Computation of *multiMEMs*

We now describe an algorithm to compute all *multiMEMs*, using the suffix tree for S . The algorithm is part of the current version of the multiple genome alignment software *MGA*. It improves on the method which was described in [24] and used in an earlier version of *MGA*.

The algorithm computes position sets $\mathcal{P}_{\bar{u}}(q, a)$ by processing the edges of the suffix tree in a bottom-up traversal. That is, the edge leading to node \bar{u} is processed only after all edges in the subtree below \bar{u} have been processed.

If \bar{u} is a leaf corresponding to, say suffix S_i , then compute $\mathcal{P}_{\bar{u}}(q, a) = \{i\}$ if $\sigma(i) = q$ and $S[i - 1] = a$, and $\mathcal{P}_{\bar{u}}(q, a) = \emptyset$ otherwise. Now suppose that \bar{u} is a branching node with r outgoing edges. These are processed in any order. Consider an edge $\bar{u} \rightarrow \bar{w}$. Due to the bottom-up strategy, $\mathcal{P}_{\bar{w}}$ is already computed. However, only a subset of $\mathcal{P}_{\bar{w}}$ has been computed since only, say $j < r$, edges outgoing from \bar{u} have been processed. The corresponding subset of $\mathcal{P}_{\bar{w}}$ is denoted by $\mathcal{P}_{\bar{w}}^j$. The edge $\bar{u} \rightarrow \bar{w}$ is processed in the following way: At first, *multiMEMs* are output by combining the positions in $\mathcal{P}_{\bar{w}}^j$ and $\mathcal{P}_{\bar{u}}$. In particular, all $(k + 1)$ -tuples $(l, p_0, p_1, \dots, p_{k-1})$ satisfying the following conditions are enumerated:

- (1) $l = |u|$
- (2) $p_q \in \mathcal{P}_{\bar{u}}^j(q) \cup \mathcal{P}_{\bar{w}}(q)$ for any $q \in [0, k-1]$
- (3) $p_q \in \mathcal{P}_{\bar{u}}^j(q)$ for at least one $q \in [0, k-1]$
- (4) $p_q \in \mathcal{P}_{\bar{w}}(q)$ for at least one $q \in [0, k-1]$
- (5) $p_{q'} \in \mathcal{P}_{\bar{w}}(q', a)$ and $p_{q''} \in \mathcal{P}_{\bar{w}}(q'', b)$ for at least one pair $(q, q') \in [0, k-1] \times [0, k-1]$ and different symbols a and b .

By definition of $\mathcal{P}_{\bar{u}}$, u occurs at positions p_0, p_1, \dots, p_{k-1} in S . Moreover, for $q \in [0, k-1]$, $\rho(p_q)$ is a relative position of u in G_q . Hence $(l, \rho(p_0), \rho(p_1), \dots, \rho(p_{k-1}))$ is a multiple exact match. Conditions (3) and (4) guarantee that not all positions are exclusively taken from $\mathcal{P}_{\bar{u}}^j(q)$ or from $\mathcal{P}_{\bar{w}}(q)$. Hence at least two of the positions in $\{p_0, p_1, \dots, p_{k-1}\}$ are taken from different subtrees of \bar{u} . This implies right maximality. Condition (5) guarantees left maximality.

As soon as for the current edge $\bar{u} \rightarrow \bar{w}$ the *multiMEMs* are enumerated, the algorithm adds $\mathcal{P}_{\bar{w}}(q, a)$ to $\mathcal{P}_{\bar{u}}^j(q, a)$ to obtain position sets $\mathcal{P}_{\bar{u}}^{j+1}(q, a)$ for all $q \in [0, k-1]$ and all $a \in \Sigma \cup \text{Seps}$. That is, the position sets are inherited from node \bar{w} to the parent node \bar{u} . Finally, $\mathcal{P}_{\bar{u}}(q, a)$ is obtained as soon as all edges outgoing from \bar{u} are processed.

The algorithm performs two operations on position sets, as follows. Enumeration of multiple exact matches by combining position sets and accumulating position sets. A position set $\mathcal{P}_{\bar{u}}(q, a)$ is the union of position sets from the subtrees below \bar{u} . Recall that we considered processing an edge $\bar{u} \rightarrow \bar{w}$. If the edges to the children of \bar{w} have been processed, the position sets of the children are obsolete. Hence it is not required to copy position sets. At any time of the algorithm, each position is included in exactly one position set. Thus the position sets require $O(n)$ space. For each branching node one maintains a table of $k(|\Sigma|+1)$ references to possibly empty position sets. In particular, to achieve independence of the number of separator symbols, we store all positions from $\mathcal{P}_{\bar{u}}(q, a)$, $a \in \text{Seps}$, in a single set. Hence, the space requirement for the position sets is $O(|\Sigma|kn)$. The union operation for the position sets can be implemented in constant time using linked lists. For each node, there are $O(|\Sigma|k)$ union operations. Since there are $O(n)$ edges in the suffix tree, the union operations thus require $O(|\Sigma|kn)$ time.

Each combination of position sets requires to enumerate the following cartesian product:

$$\prod_{q=0}^{k-1} \left(\mathcal{P}_{\bar{u}}^j(q) \cup \mathcal{P}_{\bar{w}}(q) \right) \setminus \left(\left(\prod_{q=0}^{k-1} \mathcal{P}_{\bar{u}}^j(q) \right) \cup \left(\prod_{q=0}^{k-1} \mathcal{P}_{\bar{w}}(q) \right) \cup \left(\prod_{a \in \Sigma \cup \text{Seps}} (\mathcal{P}_{\bar{w}}(a) \cup \mathcal{P}_{\bar{u}}(a)) \right) \right) \quad (58.2)$$

The enumeration is done in three steps, as follows. In a first step one enumerates all possible k -tuples $(P_0, P_1, \dots, P_{k-1})$ of non-empty sets where each P_q is either $\mathcal{P}_{\bar{u}}^j(q)$ or $\mathcal{P}_{\bar{w}}(q)$. Such a k -tuple is called father/child choice, since it specifies to either choose a position from the father \bar{u} (a father choice) or from the child \bar{w} (a child choice). One rejects the two k -tuples specifying only father choices or only child choices and process the remaining father/child choices further. In the second step, for a fixed father/child choice $(P_0, P_1, \dots, P_{k-1})$ one enumerates all possible k -tuples (a_0, \dots, a_{k-1}) (called symbol choices) such that $P_q(a_q) \neq \emptyset$. At most $|\Sigma|$ symbol choices consisting of k identical symbols (this can be decided in constant time) are rejected. The remaining symbol choices are processed further. In the third step, for a fixed symbol choice (a_0, \dots, a_{k-1}) we enumerate all possible k -tuples (p_0, \dots, p_{k-1}) such that $p_q \in P_q(a_q)$ for $q \in [0, k-1]$. By construction, each of these k -tuples represents a *multiMEM* of length l . The cartesian product (58.2) thus can be enumerated in $O(k)$ space and in time proportional to its size.

The suffix tree construction and the bottom-up traversal (without handling of position sets) requires $O(n)$ time. Thus the algorithm described here runs in $O(|\Sigma|kn)$ space and

$O(|\Sigma|kn + z)$ time where z is the number of *multiMEMs*. It is still unclear if there is an algorithm which avoids the factors $|\Sigma|$ or k in the space or time requirement.

Space efficient computation of MEMs for two genomes

The algorithm computes all MEMs of length at least ℓ by constructing the suffix tree of G_0 and matches G_1 against it. The matching process delivers substrings of G_1 represented by locations in the suffix tree, where a location is defined as follows: Suppose a string u occurs in $\text{ST}(G_0)$. If there is a branching node \bar{u} , then \bar{u} is the *location of u* . If \bar{v} is the branching node of maximal depth, such that $u = vw$ for some non-empty string w , then (\bar{v}, w) is the *location of u* . The location of u is denoted by $\text{loc}(u)$.

$\text{ST}(G_0)$ represents all suffixes $T_i = G_0[i \dots n_0 - 1]_{\$0}$ of $G_0_{\$0}$. The algorithm processes G_1 suffix by suffix from longest to shortest. In the j th step, the algorithm processes suffix $R_j = G_1[j \dots n_1 - 1]_{\$2}$ and computes the locations of two prefixes p_{\min}^j and p_{\max}^j of R_j defined as follows:

- p_{\max}^j is the longest prefix of R_j that occurs in $\text{ST}(G_0)$.
- p_{\min}^j is the prefix of p_{\max}^j of length $\min\{\ell, |p_{\max}^j|\}$.

If $|p_{\min}^j| < \ell$, then one skips R_j . If $|p_{\min}^j| = \ell$, then at least one suffix represented in the subtree below $\text{loc}(p_{\min}^j)$ matches the first ℓ characters of R_j . To extract the MEMs, this subtree is traversed in a depth first order. The depth first traversal maintains for each visited branching node \bar{u} the length of the longest common prefix of u and R_j . Each time a leaf \bar{T}_i is visited, one first checks if $G_0[i - 1] \neq G_1[j - 1]$. If this is the case, then $(\perp, \rho(i), \rho(j))$ is a left maximal exact match and one determines the length l of the longest common prefix of T_i and R_j . By construction, $l \geq \ell$ and $G_0[i + l] \neq G_1[j + l]$. Hence $(l, \rho(i), \rho(j))$ is a MEM. Now consider the different steps of the algorithm in more detail:

Computation of $\text{loc}(p_{\min}^j)$: For $j = 0$, one computes $\text{loc}(p_{\min}^j)$ by greedily matching $G_1[0 \dots \ell - 1]$ against $\text{ST}(G_0)$. For $j \in [1, n_1 - 1]$, one follows the suffix link of $\text{loc}(p_{\min}^{j-1})$, if this is a branching node, or of \bar{v} if $\text{loc}(p_{\min}^{j-1}) = (\bar{v}, w)$. This shortcut via the suffix link leads to a branching node on the path from the root to $\text{loc}(p_{\min}^j)$, from which one matches the next characters. The method is similar to the matching-statistics computation of [14], and one can show that its overall running time for the computation of all $\text{loc}(p_{\min}^j)$, $j \in [0, n_1 - 1]$, is $O(n_1)$.

Computation of $\text{loc}(p_{\max}^j)$: Starting from $\text{loc}(p_{\min}^j)$ one computes $\text{loc}(p_{\max}^j)$ by greedily matching $G_1[|p_{\min}^j| \dots n_j - 1]$ against $\text{ST}(G_0)$. To facilitate the computation of longest common prefixes, one keeps track of the list of branching nodes on the path from $\text{loc}(p_{\min}^j)$ to $\text{loc}(p_{\max}^j)$. This list is called the *match path*. Since $|p_{\max}^{j-1}| \geq 1$ implies $|p_{\max}^j| \geq |p_{\max}^{j-1}| - 1$, we do not always have to match the edges of the suffix tree completely against the corresponding substring of G_1 . Instead, to reach $\text{loc}(p_{\max}^j)$, one rescans most of the edges by only looking at the first character of the edge label to determine the appropriate edge to follow. Thus the total time for this step is $O(n_1 + \alpha)$ where α is the total length of all match paths. α is upper bounded by the total size β of the subtrees below $\text{loc}(p_{\min}^j)$, $j \in [0, n_1 - 1]$. β is upper bounded by the number r of right maximal exact matches between G_0 and G_1 . Hence the running time for this step of the algorithm is $O(n_1 + r)$.

The Depth first traversal: This maintains an *lcp*-stack which stores for each visited branching node, say \bar{u} , a pair of values (*onmatchpath*, *lcpvalue*), where the boolean value *onmatchpath* is true, if and only if \bar{u} is on the match path, and *lcpvalue* stores the length of the longest common prefix of u and R_j . Given the match path, the *lcp*-stack can be maintained in constant time for each branching node visited. For each leaf \bar{T}_i visited during

the depth first traversal, the *lcp*-stack allows to determine in constant time the length of the longest common prefix of T_i and R_j . As a consequence, the depth first traversal requires time proportional to the size of the subtree. Thus the total time for all depth first traversals of the subtrees below $loc(p_{\min}^j)$, $j \in [0, n_1 - 1]$, is $O(r)$.

Altogether, the algorithm described here runs in $O(n_0 + n_1 + r)$ time and $O(n_0)$ space. It is implemented as part of the new version of the *MUMmer* genome alignment program, see <http://www.tigr.org/Software/mummer>.

Acknowledgment

This work was supported, in part, by Bourns College of Engineering, University of California, Riverside, and by the Center of Bioinformatics, University of Hamburg. We thank Jens Stoye and Klaus-Bernd Schürmann, University of Bielefeld, for helpful discussions and suggestions on the basic structure of the space efficient algorithm to compute *MEMs*.

References

- [1] ABOUELHODA, M. I., AND OHLEBUSCH, E. Multiple genome alignment: Chaining algorithms revisited. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, LNCS* (2003), vol. 2676, Springer-Verlag, pp. 1–16.
- [2] ABOUELHODA, M. I., AND OHLEBUSCH, E. A local chaining algorithm and its applications in comparative genomics. In *Proceedings of the 3rd Workshop on Algorithms in Bioinformatics, LNCS* (2003), vol. 2812, Springer-Verlag, pp. 1–16.
- [3] AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The design and analysis of computer algorithms*. Addison-Wesley, Reading, MA, 1974.
- [4] APOSTOLICO, A., BOCK, M. E., AND LONARDI, S. Monotony of surprise and large-scale quest for unusual words (extended abstract). In *Proc. of Research in Computational Molecular Biology (RECOMB)* (Washington, DC, April 2002), G. Myers, S. Hannenhalli, S. Istrail, P. Pevzner, and M. Waterman, Eds. Also in *J. Comput. Bio.*, 10:3-4, (July 2003), 283–311.
- [5] APOSTOLICO, A., BOCK, M. E., LONARDI, S., AND XU, X. Efficient detection of unusual words. *J. Comput. Bio.* 7, 1/2 (January 2000), 71–94.
- [6] APOSTOLICO, A., BOCK, M. E., AND XU, X. Annotated statistical indices for sequence analysis. In *Sequences* (Positano, Italy, 1998), B. Carpentieri, A. De Santis, U. Vaccaro, and J. Storer, Eds., IEEE Computer Society Press, pp. 215–229.
- [7] APOSTOLICO, A., GONG, F., AND LONARDI, S. Verbumculus and the discovery of unusual words. *Journal of Computer Science and Technology (special issue in bioinformatics)*, vol. 19, no. 1, pp. 22–41, 2004.
- [8] APOSTOLICO, A., AND LONARDI, S. A speed-up for the commute between subword trees and DAWGs. *Information Processing Letters* 83, 3 (2002), 159–161.
- [9] BLUMER, A., BLUMER, J., EHRENFUCHT, A., HAUSSLER, D., AND MCCONNEL, R. Complete inverted files for efficient text retrieval and analysis. *J. Assoc. Comput. Mach.* 34, 3 (1987), 578–595.
- [10] BRÄZMA, A., JONASSEN, I., UKKONEN, E., AND VILO, J. Predicting gene regulatory elements in silico on a genomic scale. *Genome Research* 8, 11 (1998), 1202–1215.
- [11] BURGE, C., CAMPBELL, A., AND KARLIN, S. Over- and under-representation of short oligonucleotides in DNA sequences. *Proc. Natl. Acad. Sci. U.S.A.* 89 (1992), 1358–1362.

- [12] CASTRIGNANO, T., COLOSIMO, A., MORANTE, S., PARISI, V., AND ROSSI, G. C. A study of oligonucleotide occurrence distributions in DNA coding segments. *J. Theor. Biol.* 184, 4 (1997), 451–69.
- [13] CHAIN, P., KURTZ, S., OHLEBUSCH, E., AND SLEZAK, T. An applications-focused review of comparative genomics tools: capabilities, limitations and future challenges. *Briefings in Bioinformatics* 4, 2 (2003), 105–123.
- [14] CHANG, W. I., AND LAWLER, E. L. Sublinear approximate string matching and biological applications. *Algorithmica* 12, 4/5 (October/November 1994), 327–344.
- [15] CHEN, T., AND SKIENA, S. S. Trie-based data structures for sequence assembly. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching* (Aarhus, Denmark, 1997), A. Apostolico and J. Hein, Eds., no. 1264 in Lecture Notes in Computer Science, Springer-Verlag, Berlin, pp. 206–223.
- [16] CLIFT, B., HAUSSLER, D., MCCONNELL, R., SCHNEIDER, T. D., AND STORMO, G. D. Sequence landscapes. *Nucleic Acids Res.* 14 (1986), 141–158.
- [17] DELCHER, A., KASIF, S., FLEISCHMANN, R., PETERSON, J., WHITE, O., AND SALZBERG, S. Alignment of whole genomes. *Nucleic Acids Research* 27, 11 (1999), 2369–2376.
- [18] DELCHER, A. L., PHILLIPPY, A., CARLTON, J., AND SALZBERG, S. L. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Research* 30, 11 (2002), 2478–2483.
- [19] FICKETT, J. W., TORNEY, D. C., AND WOLF, D. R. Base compositional structure of genomes. *Genomics* 13 (1992), 1056–1064.
- [20] GELFAND, M. S., AND KOONIN, E. Avoidance of palindromic words in bacterial and archaeal genomes: A close connection with restriction enzymes. *Nucleic Acids Res.* 25 (1997), 2430–2439.
- [21] GELFAND, M. S., KOONIN, E. V., AND MIRONOV, A. A. Prediction of transcription regulatory sites in archaea by a comparative genomic approach. *Nucleic Acids Res.* 28, 3 (2000), 695–705.
- [22] GENTLEMAN, J. The distribution of the frequency of subsequences in alphabetic sequences, as exemplified by deoxyribonucleic acid. *Appl. Statist.* 43 (1994), 404–414.
- [23] GUSFIELD, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [24] HÖHL, M., KURTZ, S., AND OHLEBUSCH, E. Efficient multiple genome alignment. *Bioinformatics* 18, Suppl. 1 (2002), S312–S320.
- [25] KARLIN, S., BURGE, C., AND CAMPBELL, A. M. Statistical analyses of counts and distributions of restriction sites in DNA sequences. *Nucleic Acids Res.* 20 (1992), 1363–1370.
- [26] KLEFFE, J., AND BORODOVSKY, M. First and second moment of counts of words in random texts generated by Markov chains. *Comput. Appl. Biosci.* 8 (1992), 433–441.
- [27] KURTZ, S., CHOUDHURI, J. V., OHLEBUSCH, E., SCHLEIERMACHER, C., STOYE, J., AND GIEGERICH, R. REPuter: The manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Res.* 29, 22 (2001), 4633–4642.
- [28] LEUNG, M. Y., MARSH, G. M., AND SPEED, T. P. Over and underrepresentation of short DNA words in herpes virus genomes. *J. Comput. Bio.* 3 (1996), 345–360.
- [29] LOTHAIRE, M., Ed. *Combinatorics on Words*, second ed. Cambridge University Press, 1997.
- [30] LOTHAIRE, M., Ed. *Algebraic Combinatorics on Words*. Cambridge University Press, 2002.
- [31] MARSAN, L., AND SAGOT, M.-F. Algorithms for extracting structured motifs using a suffix tree with application to promoter and regulatory site consensus identification.

- J. Comput. Bio.* 7, 3/4 (2000), 345–360.
- [32] MCCREIGHT, E. M. A space-economical suffix tree construction algorithm. *J. Assoc. Comput. Mach.* 23, 2 (April 1976), 262–272.
- [33] MYERS, E. W., AND MILLER, W. Chaining multiple-alignment fragments in sub-quadratic time. In *Proceedings of the 6th ACM-SIAM Annual Symposium on Discrete Algorithms* (San Francisco, CA, 1995), pp. 38–47.
- [34] NEEDLEMAN, S. B., AND WUNSCH, C. D. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.* 48 (1970), 443–453.
- [35] NUSSINOV, R. The universal dinucleotide asymmetry rules in DNA and the amino acid codon choice. *J. Mol. Evol.* 17 (1981), 237–244.
- [36] PAVESI, G., MAURI, G., AND PESOLE, G. An algorithm for finding signals of unknown length in DNA sequences. In *Proc. of the International Conference on Intelligent Systems for Molecular Biology* (2001), AAAI press, Menlo Park, CA, pp. S207–S214.
- [37] PAVESI, G., MAURI, G., AND PESOLE, G. Methods for pattern discovery in unaligned biological sequences. *Briefings in Bioinformatics* 2, 4 (December 2001), 417–430.
- [38] PEVZNER, P. A., BORODOVSKY, M. Y., AND MIRONOV, A. A. Linguistics of nucleotides sequences I: The significance of deviations from mean statistical characteristics and prediction of the frequencies of occurrence of words. *J. Biomol. Struct. Dynamics* 6 (1989), 1013–1026.
- [39] PEVZNER, P. A., BORODOVSKY, M. Y., AND MIRONOV, A. A. Linguistics of nucleotides sequences II: Stationary words in genetic texts and the zonal structure of DNA. *J. Biomol. Struct. Dynamics* 6 (1989), 1027–1038.
- [40] PHILLIPS, G. J., ARNOLD, J., AND IVARIE, R. The effect of codon usage on the oligonucleotide composition of the *e. coli* genome and identification of over- and underrepresented sequences by Markov chain analysis. *Nucleic Acids Res.* 15 (1987), 2627–2638.
- [41] RAHMANN, S. Rapid large-scale oligonucleotide selection for microarrays. In *Proceedings of the First IEEE Computer Society Bioinformatics Conference (CSB'02)* (2002), IEEE Press, pp. 54–63.
- [42] RASH, S., AND GUSFIELD, D. String barcoding: Uncovering optimal virus signatures. In *Proceedings of Sixth International Conference on Computational Molecular Biology (RECOMB 2002)* (2002), ACM Press, pp. 254–261.
- [43] RÉGNIER, M., AND SZPANKOWSKI, W. On pattern frequency occurrences in a Markovian sequence. *Algorithmica* 22 (1998), 631–649.
- [44] REINERT, G., SCHBATH, S., AND WATERMAN, M. S. Probabilistic and statistical properties of words: An overview. *J. Comput. Bio.* 7 (2000), 1–46.
- [45] STADEN, R. Methods for discovering novel motifs in nucleic acid sequences. *Comput. Appl. Biosci.* 5, 5 (1989), 293–298.
- [46] STOYE, J., AND GUSFIELD, D. Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree. *Theor. Comput. Sci.* 270, 1-2 (2002), 843–856.
- [47] STÜCKLE, E., EMMRICH, C., GROB, U., AND NIELSEN, P. Statistical analysis of nucleotide sequences. *Nucleic Acids Res.* 18, 22 (1990), 6641–6647.
- [48] TOMPA, M. An exact method for finding short motifs in sequences, with application to the ribosome binding site problem. In *Seventh International Conference on Intelligent Systems for Molecular Biology* (Heidelberg, Germany, August 1999), AAAI press, Menlo Park, CA, pp. 262–271.
- [49] VAN HELDEN, J., ANDRÉ, B., AND COLLADO-VIDES, J. Extracting regulatory

- sites from the upstream region of the yeast genes by computational analysis of oligonucleotides. *J. Mol. Biol.* 281 (1998), 827–842.
- [50] VAN HELDEN, J., RIOS, A. F., AND COLLADO-VIDES, J. Discovering regulatory elements in non-coding sequences by analysis of spaced dyads. *Nucleic Acids Res.* 28, 8 (2000), 1808–1818.
- [51] VOLINIA, S., GAMBARI, R., BERNARDI, F., AND BARRAI, I. Co-localization of rare oligonucleotides and regulatory elements in mammalian upstream gene regions. *J. Mol. Biol.* 5, 1 (1989), 33–40.
- [52] VOLINIA, S., GAMBARI, R., BERNARDI, F., AND BARRAI, I. The frequency of oligonucleotides in mammalian genir regions. *Comput. Appl. Biosci.* 5, 1 (1989), 33–40.
- [53] WEINER, P. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory* (Washington, DC, 1973), pp. 1–11.
- [54] ZHANG, Z., RAGHAVACHARI, B., HARDISON, R., AND MILLER, W. Chaining Multiple Alignment Blocks. *J. Comput. Bio.* 1, 3 (1994), 217–226.

59

Elimination Structures in Scientific Computing

59.1	The Elimination Tree	59-2
	The Elimination Game • The Elimination Tree Data Structure • An Algorithm • A Skeleton Graph • Supernodes	
59.2	Applications of Etrees	59-8
	Efficient Symbolic Factorization • Predicting Row and Column Nonzero Counts • Three Classes of Factorization Algorithms • Scheduling Parallel Factorizations • Scheduling Out-of-Core Factorizations	
59.3	The Clique Tree	59-12
	Chordal Graphs and Clique Trees • Design of Efficient Algorithms with Clique Trees • Compact Clique Trees	
59.4	Clique Covers and Quotient Graphs	59-17
	Clique Covers • Quotient Graphs • The Problem of Degree Updates • Covering the Column-Intersection Graph and Biclique Covers	
59.5	Column Elimination Trees and Elimination DAGS	59-20
	The Column Elimination Tree • Elimination DAGS • Elimination Structures for the Asymmetric Multifrontal Algorithm	

Alex Pothén
Old Dominion University

Sivan Toledo
Tel Aviv University

The most fundamental computation in numerical linear algebra is the factorization of a matrix as a product of two or more matrices with simpler structure. An important example is Gaussian elimination, in which a matrix is written as a product of a lower triangular matrix and an upper triangular matrix. The factorization is accomplished by elementary operations in which two or more rows (columns) are combined together to transform the matrix to the desired form. In Gaussian elimination, the desired form is an upper triangular matrix, in which nonzero elements below the diagonal have been transformed to be equal to zero. We say that the subdiagonal elements have been eliminated. (The transformations that accomplish the elimination yield a lower triangular matrix.)

The input matrix is usually sparse, i.e., only a few of the matrix elements are nonzero to begin with; in this situation, row operations constructed to eliminate nonzero elements in some locations might create new nonzero elements, called fill, in other locations, as a side-effect. Data structures that predict fill from graph models of the numerical algorithm, and algorithms that attempt to minimize fill, are key ingredients of efficient sparse matrix algorithms.

This chapter surveys these data structures, known as *elimination structures*, and the algorithms that construct and use them. We begin with the *elimination tree*, a data structure associated with symmetric Gaussian elimination, and we then describe its most important applications. Next we describe other data structures associated with symmetric Gaussian elimination, the *clique tree*, the *clique cover*, and the *quotient graph*. We then consider data structures that are associated with asymmetric Gaussian elimination, the *column elimination tree* and the *elimination directed acyclic graph*.

This survey has been written with two purposes in mind. First, we introduce the algorithms community to these data structures and algorithms from combinatorial scientific computing; the initial subsections should be accessible to the non-expert. Second, we wish to briefly survey the current state of the art, and the subsections dealing with the advanced topics move rapidly. A collection of articles describing developments in the field circa 1991 may be found in [24]; Duff provides a survey as of 1996 in [19].

59.1 The Elimination Tree

59.1.1 The Elimination Game

Gaussian elimination of a symmetric positive definite matrix A , which factors the matrix A into the product of a lower triangular matrix L and its transpose L^T , $A = LL^T$, is one of the fundamental algorithms in scientific computing. It is also known as Cholesky factorization. We begin by considering the graph model of this computation performed on a symmetric matrix A that is sparse, i.e., few of its matrix elements are nonzero. The number of nonzeros in L and the work needed to compute L depend strongly on the (symmetric) ordering of the rows and columns of A . The graph model of sparse Gaussian elimination was introduced by Parter [58], and has been called the *elimination game* by Tarjan [70]. The goal of the elimination game is to symmetrically order the rows and columns of A to minimize the number of nonzeros in the factor L .

We consider a sparse, symmetric positive definite matrix A with n rows and n columns, and its adjacency graph $G(A) = (V, E)$ on n vertices. Each vertex in $v \in V$ corresponds to the v -th row of A (and by symmetry, the v -th column); an edge $(v, w) \in E$ corresponds to the nonzero a_{vw} (and by symmetry, the nonzero a_{wv}). Since A is positive definite, its diagonal elements are positive; however, by convention, we do not explicitly represent a diagonal element a_{vv} by a loop (v, v) in the graph $G(A)$. (We use v, w, \dots to indicate unnumbered vertices, and i, j, k, \dots to indicate numbered vertices in a graph.)

We view the vertices of the graph $G(A)$ as being initially unnumbered, and number them from 1 to n , as a consequence of the elimination game. To number a vertex v with the next available number, add new *fill edges* to the current graph to make all currently unnumbered neighbors of v pairwise adjacent. (Note that the vertex v itself does not acquire any new neighbors in this step, and that v plays no further role in generating fill edges in future numbering steps.)

The graph that results at the end of the elimination game, which includes both the edges in the edge set E of the initial graph $G(A)$ and the set of fill edges, F , is called the filled graph. We denote it by $G^+(A) = (V, E \cup F)$. The numbering of the vertices is called an elimination ordering, and corresponds to the order in which the columns are factored. An example of a filled graph resulting from the elimination game on a graph is shown in Fig. 59.1. We will use this graph to illustrate various concepts throughout this paper.

The goal of the elimination game is to number the vertices to minimize the fill since it would reduce the storage needed to perform the factorization, and also controls the work in the factorization. Unfortunately, this is an NP-hard problem [74]. However, for

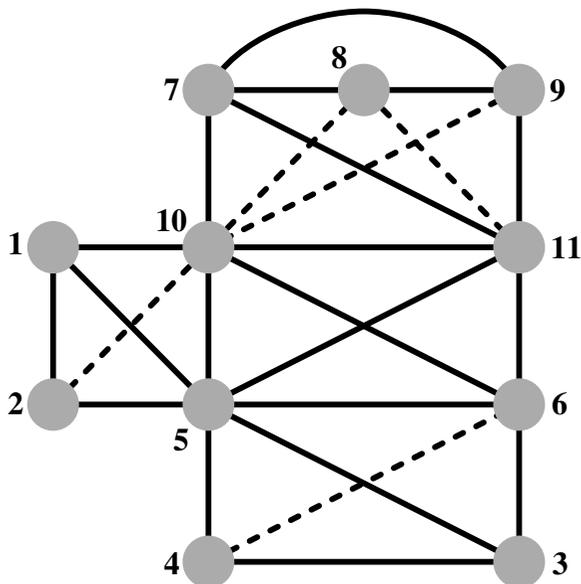


FIGURE 59.1: A filled graph $G^+(A)$ resulting from the elimination game on a graph $G(A)$. The solid edges belong to $G(A)$, and the broken edges are filled edges generated by the elimination game when vertices are eliminated in the order shown.

classes of graphs that have small separators, it is possible to establish upper bounds on the number of edges in the filled graph, when the graph is ordered by a nested dissection algorithm that recursively computes separators. Planar graphs, graphs of ‘well-shaped’ finite element meshes (aspect ratios bounded away from small values), and overlap graphs possess elimination orderings with bounded fill. Conversely, the fill is large for graphs that do not have good separators.

Approximation algorithms that incur fill within a polylog factor of the optimum fill have been designed by Agrawal, Klein and Ravi [1]; but since it involves finding approximate concurrent flows with uniform capacities, it is an impractical approach for large problems. A more recent approximation algorithm, due to Natanzon, Shamir and Sharan [57], limits fill to within the square of the optimal value; this approximation ratio is better than that of the former algorithm only for dense graphs.

The elimination game produces sets of cliques in the graph. Let $\text{hadj}^+(v)$ ($\text{ladj}^+(v)$) denote the higher-numbered (lower-numbered) neighbors of a vertex v in the graph $G^+(A)$; in the elimination game, $\text{hadj}^+(v)$ is the set of unnumbered neighbors of v immediately prior to the step in which v is numbered. When a vertex v is numbered, the set $\{v\} \cup \text{hadj}^+(v)$ becomes a clique by the rules of the elimination game. Future numbering steps and consequent fill edges added do not change the adjacency set (in the filled graph) of the vertex v . (We will use $\text{hadj}(v)$ and $\text{ladj}(v)$ to refer to higher and lower adjacency sets of a vertex v in the original graph $G(A)$.)

59.1.2 The Elimination Tree Data Structure

We define a forest from the filled graph by defining the parent of a vertex v to be the lowest numbered vertex in $\text{hadj}^+(v)$. It is clear that this definition of parent yields a forest since the parent of each vertex is numbered higher than itself. If the initial graph $G(A)$ is

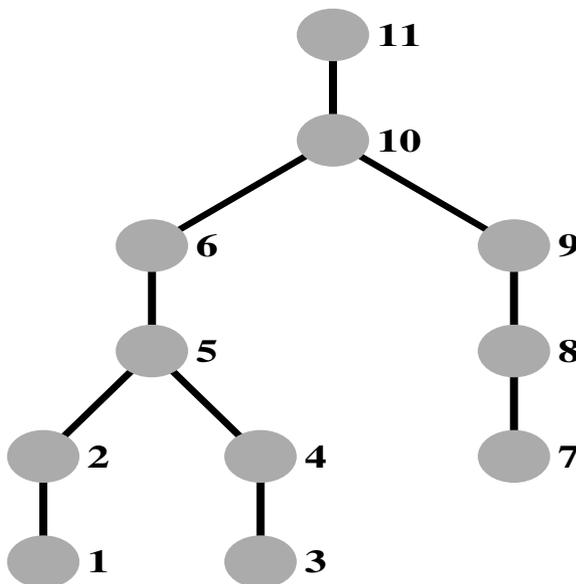


FIGURE 59.2: The elimination tree of the example graph.

connected, then indeed we have a tree, the *elimination tree*; if not we have an *elimination forest*.

In terms of the Cholesky factor L , the elimination tree is obtained by looking down each column below the diagonal element, and choosing the row index of the first subdiagonal nonzero to be the parent of a column. It will turn out that we can compute the elimination tree corresponding to a matrix and a given ordering without first computing the filled graph or the Cholesky factor.

The elimination tree of the graph in Fig. 59.1 with the elimination ordering given there is shown in Fig. 59.2.

A *fill path* joining vertices i and j is a path in the original graph $G(A)$ between vertices i and j , all of whose interior vertices are numbered lower than both i and j . The following theorem offers a static characterization of what causes fill in the elimination game.

THEOREM 59.1 [64] *The edge (i, j) is an edge in the filled graph if and only if a fill path joins the vertices i and j in the original graph $G(A)$.*

In the example graph in Fig. 59.1, vertices 9 and 10 are joined a fill path consisting of the interior vertices 7 and 8; thus $(9, 10)$ is a fill edge. The next theorem shows that an edge in the filled graph represents a dependence relation between its end points.

THEOREM 59.2 [69] *If (i, j) is an edge in the filled graph and $i < j$, then j is an ancestor of the vertex i in the elimination tree $T(A)$.*

This theorem suggests that the elimination tree represents the information flow in the elimination game (and hence sparse symmetric Gaussian elimination). Each vertex i influences only its higher numbered neighbors (the numerical values in the column i affect only those columns in $\text{had}j^+(i)$). The elimination tree represents the information flow in a minimal way in that we need consider only how the information flows from i to its parent in the elimination tree. If j is the parent of i and ℓ is another higher neighbor of i , then since the higher neighbors of i form a clique, we have an edge (j, ℓ) that joins j and ℓ ; since

by Theorem 59.2, ℓ is an ancestor of j , the information from i that affects ℓ can be viewed as being passed from i first to j , and then indirectly from j through its ancestors on the path in the elimination tree to ℓ .

An immediate consequence of the Theorem 59.2 is the following result.

COROLLARY 59.1 If vertices i and j belong to vertex-disjoint subtrees of the elimination tree, then no edge can join i and j in the filled graph.

Viewing the dependence relationships in sparse Cholesky factorization by means of the elimination tree, we see that any topological reordering of the elimination tree would be an elimination ordering with the same fill, since it would not violate the dependence relationships. Such reorderings would not change the fill or arithmetic operations needed in the factorization, but would change the schedule of operations in the factorization (i.e., when a specific operation is performed). This observation has been used in sparse matrix factorizations to schedule the computations for optimal performance on various computational platforms: multiprocessors, hierarchical memory machines, external memory algorithms, etc. A postordering of the elimination tree is typically used to improve the spatial and temporal data locality, and thereby the cache performance of sparse matrix factorizations.

There are two other perspectives from which we can view the elimination tree.

Consider directing each edge of the filled graph from its lower numbered endpoint to its higher numbered endpoint to obtain a directed acyclic graph (DAG). Now form the transitive reduction of the directed filled graph; i.e., delete an edge (i, k) whenever there is a directed path from i to k that does not use the edge (i, k) (this path necessarily consists of at least two edges since we do not admit multiple edges in the elimination game). The minimal graph that remains when all such edges have been deleted is unique, and is the elimination tree.

One could also obtain the elimination tree by performing a depth-first search (DFS) in the filled graph with the vertex numbered n as the initial vertex for the DFS, and choosing the highest numbered vertex in $\text{ladj}^+(i)$ as the next vertex to search from a vertex i .

59.1.3 An Algorithm

We begin with a consequence of the repeated application of the following fact: If a vertex i is adjacent to a higher numbered neighbor k in the filled graph, and k is not the parent of i , p_i , in the elimination tree, then i is adjacent to both k and p_i in the filled graph; when i is eliminated, by the rules of the elimination game, a fill edge joins p_i and k .

THEOREM 59.3 If (i, k) is an edge in the filled graph and $i < k$, then for every vertex j on an elimination tree path from i to k , (j, k) is also an edge in the filled graph.

This theorem leads to a characterization of $\text{ladj}^+(k)$, the set of lower numbered neighbors of a vertex k in the filled graph, which will be useful in designing an efficient algorithm for computing the elimination tree. The set $\text{ladj}^+(k)$ corresponds to the column indices of nonzeros in the k -th row of the Cholesky factor L , and $\text{ladj}(k)$ corresponds to the column indices of nonzeros in the lower triangle of the k -th row of the initial matrix A .

THEOREM 59.4 [51] Every vertex in the set $\text{ladj}^+(k)$ is a vertex reachable by paths in the elimination tree from a set of leaves to k ; each leaf l corresponds to a vertex in the set $\text{ladj}(k)$ such that no proper descendant d of l in the elimination tree belongs to the set $\text{ladj}(k)$.

Theorem 59.4 characterizes the k -th row of the Cholesky factor L as a row subtree $T_r(k)$

```

for  $k := 1$  to  $n \rightarrow$ 
     $p_k := 0;$ 
    for  $j \in \text{ladj}(k)$  (in increasing order)  $\rightarrow$ 
        find the root  $r$  of the tree containing  $j$ ;
        if ( $k \neq r$ ) then  $k := p_r$ ; fi
    rof
rof

```

FIGURE 59.3: An algorithm for computing an elimination tree. Initially each vertex is in a subtree with it as the root.

of the elimination subtree rooted at the vertex k , and pruned at each leaf l . The leaves of the pruned subtree are contained among $\text{ladj}(k)$, the column indices of the nonzeros in (the lower triangle of) the k -th row of A . In the elimination tree in Fig. 59.2, the pruned elimination subtree corresponding to row 11 has two leaves, vertices 5 and 7; it includes all vertices on the etree path from these leaves to the vertex 11.

The observation above leads to an algorithm, shown in Fig. 59.3, for computing the elimination tree from the row structures of A , due to Liu [51].

This algorithm can be implemented efficiently using the union-find data structure for disjoint sets. A height compressed version of the p array, ancestor, makes it possible to compute the root fast; and union by rank in merging subtrees helps to keep the merged tree shallow. The time complexity of the algorithm is $O(e\alpha(e, n) + n)$, where n is the number of vertices and e is the number of edges in $G(A)$, and $\alpha(e, n)$ is a functional inverse of Ackermann's function. Liu [54] shows experimentally that path compression alone is more efficient than path compression and union by rank, although the asymptotic complexity of the former is higher. Zmijewski and Gilbert [75] have designed a parallel algorithm for computing the elimination tree on distributed memory multiprocessors.

The concept of the elimination tree was implicit in many papers before it was formally identified. The term elimination tree was first used by Duff [17], although he studied a slightly different data structure; Schreiber [69] first formally defined the elimination tree, and its properties were established and used in several articles by Liu. Liu [54] also wrote an influential survey that delineated its importance in sparse matrix computations; we refer the reader to this survey for a more detailed discussion of the elimination tree current as of 1990.

59.1.4 A Skeleton Graph

The filled graph represents a supergraph of the initial graph $G(A)$, and a skeleton graph represents a subgraph of the latter. Many sparse matrix algorithms can be made more efficient by implicitly identifying the edges of a skeleton graph $G^-(A)$ from the graph $G(A)$ and an elimination ordering, and performing computations only on these edges. A skeleton graph includes only the edges that correspond to the leaves in each row subtree in Theorem 59.4. The other edges in the initial graph $G(A)$ can be discarded, since they will be generated as fill edges during the elimination game. Since each leaf of a row subtree corresponds to an edge in $G(A)$, the skeleton graph $G^-(A)$ is indeed a subgraph of the former. The skeleton graph of the example graph is shown in Fig. 59.4.

The leaves in a row subtree can be identified from the set $\text{ladj}(j)$ when the elimination tree is numbered in a postordering. The subtree $T(i)$ is the subtree of the elimination tree

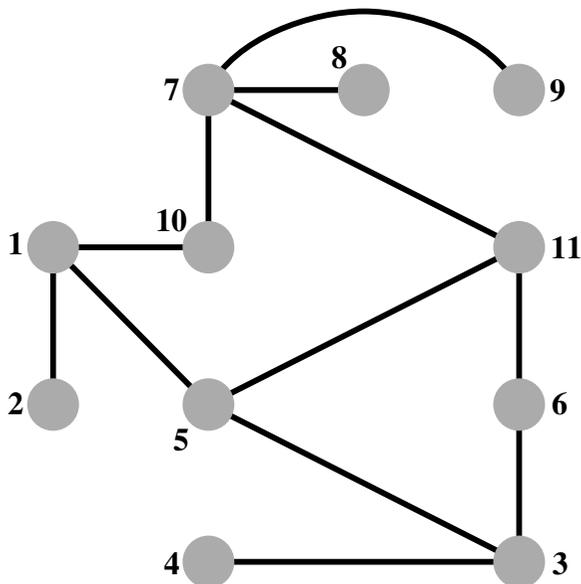


FIGURE 59.4: The skeleton graph $G^-(A)$ of the example graph.

rooted at a vertex i , and $|T(i)|$ is the number of vertices in that subtree. (It should not be confused with the row subtree $T_r(i)$, which is a pruned subtree of the elimination tree.)

THEOREM 59.5 [51] *Let $\text{ladj}(j) = \{i_1 < i_2 < \dots < i_s\}$, and let the vertices of a filled graph be numbered in a postordering of its elimination tree T . Then vertex i_q is a leaf of the row subtree $T_r(j)$ if and only if either $q = 1$, or for $q \geq 2$, $i_{q-1} < i_q - |T(i_q)| + 1$.*

59.1.5 Supernodes

A supernode is a subset of vertices S of the filled graph that form a clique and have the same higher neighbors outside S . Supernodes play an important role in numerical algorithms since loops corresponding to columns in a supernode can be blocked to obtain high performance on modern computer architectures. We now proceed to define a supernode formally.

A maximal clique in a graph is a set of vertices that induces a complete subgraph, but adding any other vertex to the set does not induce a complete subgraph. A supernode is a maximal clique $\{i_s, i_{s+1}, \dots, i_{s+t-1}\}$ in a filled graph $G^+(A)$ such that for each $1 \leq j \leq t-1$,

$$\text{hadj}^+(i_s) = \{i_{s+1}, \dots, i_{s+j}\} \cup \text{hadj}^+(i_{s+j}).$$

Let $\text{hd}^+(i_s) \equiv |\text{hadj}^+(i_s)|$; since $\text{hadj}^+(i_s) \subseteq \{i_{s+1}, \dots, i_{s+j}\} \cup \text{hadj}^+(i_{s+j})$, the relationship between the higher adjacency sets can be replaced by the equivalent test on higher degrees: $\text{hd}^+(i_s) = \text{hd}^+(i_{s+j}) + j$.

In practice, *fundamental supernodes*, rather than the maximal supernodes defined above, are used, since the former are easier to work with in the numerical factorization. A fundamental supernode is a clique but not necessarily a maximal clique, and satisfies two additional conditions: (1) i_{s+j-1} is the only child of the vertex i_{s+j} in the elimination tree, for each $1 \leq j \leq t-1$; (2) the vertices in a supernode are ordered consecutively, usually by post-ordering the elimination tree. Thus vertices in a fundamental supernode form a path in the elimination tree; each of the non-terminal vertices in this path has only one child,

and the child belongs to the supernode.

The fundamental supernodes corresponding to the example graph are: $\{1, 2\}$; $\{3, 4\}$; $\{5, 6\}$; $\{7, 8, 9\}$; and $\{10, 11\}$.

Just as we could compute the elimination tree directly from $G(A)$ without first computing $G^+(A)$, we can compute fundamental supernodes without computing the latter graph, using the theorem given below. Once the elimination tree is computed, this algorithm can be implemented in $O(n + e)$ time, where $e \equiv |E|$ is the number of edges in the original graph $G(A)$.

THEOREM 59.6 [56] *A vertex i is the first node of a fundamental supernode if and only if i has two or more children in the elimination tree T , or i is a leaf of some row subtree of T .*

59.2 Applications of Etrees

59.2.1 Efficient Symbolic Factorization

Symbolic factorization (or symbolic elimination) is a process that computes the nonzero structure of the factors of a matrix without computing the numerical values of the nonzeros.

The symbolic Cholesky factor of a matrix has several uses. It is used to allocate the data structure for the numeric factor and annotate it with all the row/column indices, which enables the removal of most of the non-numeric operations from the inner-most loop of the subsequent numeric factorization [20, 29]. It is also used to compute relaxed supernode (or amalgamated node) partitions, which group columns into supernodes even if they only have approximately the same structure [4, 21]. Symbolic factors can also be used in algorithms that construct approximate Cholesky factors by dropping nonzeros from a matrix A and factoring the resulting, sparser matrix B [6, 72]. In such algorithms, elements of A that are dropped from B but which appear in the symbolic factor of B can be added to the matrix B ; this improves the approximation without increasing the cost of factoring B . In all of these applications a supernodal symbolic factor (but not a relaxed one) is sufficient; there is no reason to explicitly represent columns that are known to be identical.

The following algorithm for symbolically factoring a symmetric matrix A is due to George and Liu [28] (and in a more graph-oriented form due to [64]; see also [29, Section 5.4.3] and [54, Section 8]).

The algorithm uses the elimination tree implicitly, but does not require it as input; the algorithm can actually compute the elimination tree on the fly. The algorithm uses the observation that

$$\text{hadj}^+(j) = \text{hadj}(j) \bigcup_{\cup_{i, p_i=j}} \text{hadj}^+(i) .$$

That is, the structure of a column of L is the union of the structure of its children in the elimination tree and the structure of the same column in the lower triangular part of A . Identifying the children can be done using a given elimination tree, or the elimination tree can be constructed on the fly by adding column i to the list of children of p_i when the structure of i is computed (p_i is the row index of the first subdiagonal nonzero in column i of L). The union of a set of column structures is computed using a boolean array P of size n (whose elements are all initialized to false), and an integer stack to hold the newly created structure. A row index k from a child column or from the column of A is added to the stack only if $P[k] = \text{false}$. When row index k is added to the stack, $P[k]$ is set to true to signal that k is already in the stack. When the computation of $\text{hadj}^+(j)$ is completed, the stack is used to clear P so that it is ready for the next union operation. The total work in

the algorithm is $\Theta(|L|)$, since each nonzero requires constant work to create and constant work to merge into the parent column, if there is a parent. (Here $|L|$ denotes the number of nonzeros in L , or equivalently the number of edges in the filled graph $G^+(A)$; similarly $|A|$ denotes the number of nonzeros in A , or the number of edges in the initial graph $G(A)$.)

The symbolic structure of the factor can usually be represented more compactly and computed more quickly by exploiting supernodes, since we essentially only need to represent the identity of each supernode (the constituent columns) and the structure of the first (lowest numbered) column in each supernode. The structure of any column can be computed from this information in time proportional to the size of the column. The George-Liu column-merge algorithm presented above can compute a supernodal symbolic factorization if it is given as input a supernodal elimination tree; such a tree can be computed in $O(|A|)$ time by the Liu-Ng-Peyton algorithm [56]. In practice, this approach saves a significant amount of work and storage.

Clearly, column-oriented symbolic factorization algorithms can also generate the structure of rows in the same asymptotic work and storage. But a direct symbolic factorization by rows is less obvious. Whitten [73], in an unpublished manuscript cited by Tarjan and Yannakakis [71], proposed a row-oriented symbolic factorization algorithm (see also [51] and [54, Sections 3.2 and 8.2]). The algorithm uses the characterization of the structure of row i in L as the row subtree $T_r(i)$. Given the elimination tree and the structure of A by rows, it is trivial to traverse the i th row subtree in time proportional to the number of nonzeros in row i of L . Hence, the elimination tree along with a row-oriented representation of A is an effective implicit symbolic row-oriented representation of L ; an explicit representation is usually not needed, but it can be generated in work and space $O(|L|)$ from this implicit representation.

59.2.2 Predicting Row and Column Nonzero Counts

In some applications the explicit structure of columns of L is not required, only the number of nonzeros in each column or each row. Gilbert, Ng, and Peyton [38] describe an almost-linear-time algorithm for determining the number of nonzeros in each row and column of L . Applications for computing these counts fast include comparisons of fill in alternative matrix orderings, preallocation of storage for a symbolic factorization, finding relaxed supernode partitions quickly, determining the load balance in parallel factorizations, and determining synchronization events in parallel factorizations.

The algorithm to compute row counts is based on Whitten's characterization [73]. We are trying to compute $|L_{i*}| = |T_r(i)|$. The column indices $j < i$ in row i of A define a subset of the vertices in the subtree of the elimination tree rooted at the vertex i , $T[i]$. The difficulty, of course, is counting the vertices in $T_r(i)$ without enumerating them. The Gilbert-Ng-Peyton algorithm counts these vertices using three relatively simple mechanisms: (1) processing the column indices $j < i$ in row i of A in postorder of the etree, (2) computing the distance of each vertex in the etree from the root, and (3) setting up a data structure to compute the least-common ancestor (LCA) of pairs of etree vertices. It is not hard to show that the once these preprocessing steps are completed, $|T_r(i)|$ can be computed using $|A_{i*}|$ LCA computations. The total cost of the preprocessing and the LCA computations is almost linear in $|A|$.

Gilbert, Ng, and Peyton show how to further reduce the number of LCA computations. They exploit the fact that the leaves of $T_r(i)$ are exactly the indices j that cause the creation of new supernodes in the Liu-Ng-Peyton supernode-finding algorithm [56]. This observation limits the LCA computations to leaves of row subtrees, i.e., edges in the skeleton graph $G^-(A)$. This significantly reduces the running time in practice.

Efficiently computing the column counts in L is more difficult. The Gilbert-Ng-Peyton algorithm assigns a weight $w(j)$ to each etree vertex j , such that $|L_{*j}| = \sum_{k \in T[j]} w(k)$. Therefore, the column-count of a vertex is the sum of the column counts of its children, plus its own weight. Hence, w_j must compensate for (1) the diagonal elements of the children, which are not included in the column count for j , (2) for rows that are nonzero in column j but not in its children, and (3) for duplicate counting stemming from rows that appear in more than one child. The main difficulty lies in accounting for duplicates, which is done using least-common-ancestor computations, as in the row-counts algorithm. This algorithm, too, benefits from handling only skeleton-graph edges.

Gilbert, Ng, and Peyton [38] also show in their paper how to optimize these algorithms, so that a single pass over the nonzero structure of A suffices to compute the row counts, the column counts, and the fundamental supernodes.

59.2.3 Three Classes of Factorization Algorithms

There are three classes of algorithms used to implement sparse direct solvers: left-looking, right-looking, and multifrontal; all of them use the elimination tree to guide the computation of the factors. The major difference between the first two of these algorithms is in how they schedule the computations they perform; the multifrontal algorithm organizes computations differently from the other two, and we explain this after introducing some concepts.

The computations on the sparse matrix are decomposed into subtasks involving computations among dense submatrices (supernodes), and the precedence relations among them are captured by the supernodal elimination tree. The computation at each node of the elimination tree (subtask) involves the partial factorization of the dense submatrix associated with it.

The right-looking algorithm is an eager updating scheme: Updates generated by the submatrix of the current subtask are applied immediately to future subtasks that it is linked to by edges in the filled graph of the sparse matrix. The left-looking algorithm is a lazy updating scheme: Updates generated by previous subtasks linked to the current subtask by edges in the filled adjacency graph of the sparse matrix are applied just prior to the factorization of the current submatrix. In both cases, updates always join a subtask to some ancestor subtask in the elimination tree. In the multifrontal scheme, updates always go from a child task to its parent in the elimination tree; an update that needs to be applied to some ancestor subtask is passed incrementally through a succession of vertices on the elimination tree path from the subtask to the ancestor.

Thus the major difference among these three algorithms is how the data accesses and the computations are organized and scheduled, while satisfying the precedence relations captured by the elimination tree. An illustration of these is shown in [Fig. 59.5](#).

59.2.4 Scheduling Parallel Factorizations

In a parallel factorization algorithm, dependences between nonzeros in L determine the set of admissible schedules. A diagonal nonzero can only be factored after all updates to it from previous columns have been applied, a subdiagonal nonzero can be scaled only after updates to it have been applied (and after the diagonal element has been factored), and two subdiagonal nonzeros can update elements in the reduced system only after they have been scaled.

The elimination tree represents very compactly and conveniently a superset of these dependences. More specifically, the etree represents dependences between columns of L . A

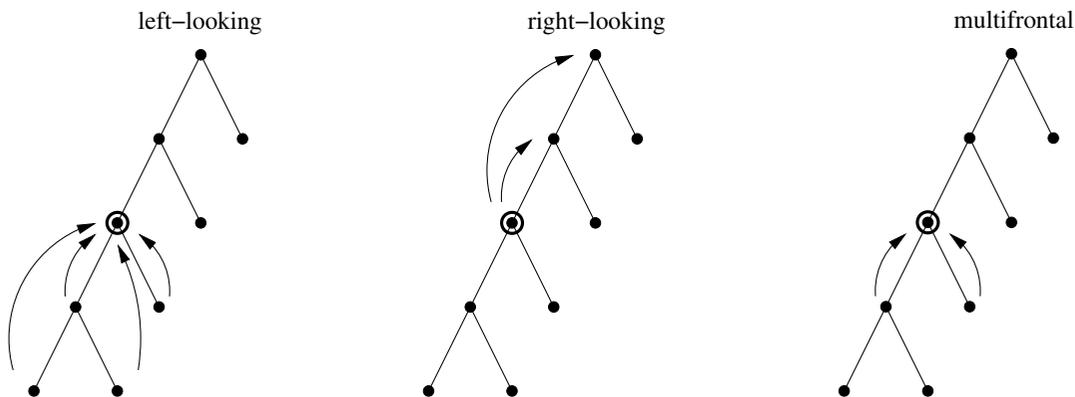


FIGURE 59.5: Patterns of data access in the left-looking, right-looking, and multifrontal algorithms. A subtree of the elimination tree is shown, and the circled node corresponds to the current submatrix being factored.

column can be completely factored only after all its descendants have been factored, and two columns that are not in an ancestor-descendant relationship can be factored in any order. Note that this is a superset of the element dependences, since a partially factored column can already perform some of the updates to its ancestors. But most sparse elimination algorithms treat column operations (or row operations) as atomic operations that are always performed by a single processor sequentially and with no interruption. For such algorithms, the etree represents exactly the relevant dependences.

In essence, parallel column-oriented factorizations can factor the columns associated with different children of an etree vertex simultaneously, but columns in an ancestor-descendant relationship must be processed in postorder. Different algorithms differ mainly in how updates are represented and scheduled.

By computing the number of nonzeros in each column, a parallel factorization algorithm can determine the amount of computation and storage associated with each subtree in the elimination tree. This information can be used to assign tasks to processors in a load-balanced way.

Duff was the first to observe that the column dependences represented by the elimination tree can guide a parallel factorization [18]. In that paper Duff proposed a parallel multifrontal factorization. The paper also proposed a way to deal with indefinite and asymmetric systems, similar to Duff and Reid's sequential multifrontal approach [21]. For further references up to about 1997, see Heath's survey [43]. Several implementations described in papers published after 1997 include PaStiX [45], PARADISO [68], WSSMP [42], and MUMPS [3], which also includes indefinite and unsymmetric factorizations. All four are message-passing codes.

59.2.5 Scheduling Out-of-Core Factorizations

In an out-of-core factorization at least some of the data structures are stored on external-memory devices (today almost exclusively magnetic disks). This allows such factorization algorithms to factor matrices that are too large to factor in main memory. The factor, which is usually the largest data structure in the factorization, is the most obvious candidate for storing on disks, but other data structures, for example the stack of update matrices in a multifrontal factorization, may also be stored on disks.

Planning and optimizing out-of-core factorization schedules require information about

data dependences in the factorization and about the number of nonzeros in each column of the factor. The etree describes the required dependence information, and as explained above, it is also used to compute nonzero counts.

Following Rothberg and Schreiber [66], we classify out-of-core algorithms into *robust* algorithms and *non-robust algorithms*. Robust algorithms adapt the factorization to complete with the core memory available by performing the data movement and computations at a smaller granularity when necessary. They partition the submatrices corresponding to the supernodes and stacks used in the factorization into smaller units called panels to ensure that the factorization completes with the available memory. Non-robust algorithms assume that the stack or a submatrix corresponding to a supernode fits within the core memory provided. In general, non-robust algorithms read elements of the input matrix only once, read from disk nothing else, and they only write the factor elements to disk; Dobrian and Pothen refer to such algorithms as read-once-write-once, and to robust ones as read-many-write-many [15].

Liu proposed [53] a non-robust method that works as long as for all $j = 1, \dots, n$, all the nonzeros in the submatrix $L_{j:n,1:j}$ of the factor fit simultaneously in main memory. Liu also shows in that paper how to reduce the amount of main memory required to factor a given matrix using this technique by reordering the children of vertices in the etree.

Rothberg and Schreiber [65, 66] proposed a number of robust out-of-core factorization algorithms. They proposed multifrontal, left-looking, and hybrid multifrontal/left-looking methods. Rotkin and Toledo [67] proposed two additional robust methods, a more efficient left-looking method, and a hybrid right/left-looking method. All of these methods use the etree together with column-nonzero counts to organize the out-of-core factorization process.

Dobrian and Pothen [15] analyzed the amount of main memory required for read-once-write-once factorizations of matrices with several regular etree structures, and the amount of I/O that read-many-write-many factorizations perform on these matrices. They also provided simulations on problems with irregular elimination tree structures. These studies led them to conclude that an external memory sparse solver library needs to provide at least two of the factorization methods, since each method can out-perform the others on problems with different characteristics. They have provided implementations of out-of-core algorithms for all three of the multifrontal, left-looking, and right-looking factorization methods; these algorithms are included in the direct solver library OBLIO [16].

In addition to out-of-core techniques, there exist techniques that reduce the amount of main memory required to factor a matrix without using disks. Liu [52] showed how to minimize the size of the stack of update matrices in the multifrontal method by reordering the children of vertices in the etree; this method is closely related to [53]. Another approach, first proposed by Eisenstat, Schultz and Sherman [23] uses a block factorization of the coefficient matrix, but drops some of the off-diagonal blocks. Dropping these blocks reduces the amount of main memory required for storing the partial factor, but requires recomputation of these blocks when linear systems are solved using the partial factor. George and Liu [29, Chapter 6] proposed a general algorithm to partition matrices into blocks for this technique. Their algorithm uses *quotient graphs*, data structures that we describe later in this chapter.

59.3 The Clique Tree

59.3.1 Chordal Graphs and Clique Trees

The filled graph $G^+(A)$ that results from the elimination game on the matrix A (the adjacency graph of the Cholesky factor L) is a *chordal* graph, i.e., a graph in which every cycle on four or more vertices has an edge joining two non-consecutive vertices on the cycle [63].

(The latter edge is called a chord, whence the name chordal graph. This class of graphs has also been called triangulated or rigid circuit graphs.)

A vertex v in a graph G is *simplicial* if its neighbors $\text{adj}(v)$ form a clique. Every chordal graph is either a clique, or it has two non-adjacent simplicial vertices. (The simplicial vertices in the filled graph in Fig. 59.1 are 1, 2, 3, 4, 7, 8, and 9.) We can eliminate a simplicial vertex v without causing any fill by the rules of the elimination game, since $\text{adj}(v)$ is already a clique, and no fill edge needs to be added. A chordal graph from which a simplicial vertex is eliminated continues to be a chordal graph. A *perfect elimination ordering* of a chordal graph is an ordering in which simplicial vertices are eliminated successively without causing any fill during the elimination game. A graph is chordal if and only if it has a perfect elimination ordering.

Suppose that the vertices of the adjacency graph $G(A)$ of a sparse, symmetric matrix A have been re-numbered in an elimination ordering, and that $G^+(A)$ corresponds to the filled graph obtained by the elimination game on $G(A)$ with that ordering. This elimination ordering is a perfect elimination ordering of the filled graph $G^+(A)$. Many other perfect elimination orderings are possible for $G^+(A)$, since there are at least two simplicial vertices that can be chosen for elimination at each step, until the graph has one uneliminated vertex.

It is possible to design efficient algorithms on chordal graphs whose time complexity is much less than $O(|E \cup F|)$, where $E \cup F$ denotes the set of edges in the chordal filled graph. This is accomplished by representing chordal graphs by tree data structures defined on the maximal cliques of the graph. (Recall that a clique K is maximal if $K \cup \{v\}$ is not a clique for any vertex $v \notin K$.)

THEOREM 59.7 *Every maximal clique of a chordal filled graph $G^+(A)$ is of the form $K(v) = \{v\} \cup \text{adj}^+(v)$, with the vertices ordered in a perfect elimination ordering.*

The vertex v is the lowest-numbered vertex in the maximal clique $K(v)$, and is called the *representative vertex* of the clique. Since there can be at most $n \equiv |V|$ representative vertices, a chordal graph can have at most n maximal cliques. The maximal cliques of the filled graph in Fig. 59.1 are: $K_1 = \{1, 2, 5, 10\}$; $K_2 = \{3, 4, 5, 6\}$; $K_3 = \{5, 6, 10, 11\}$; and $K_4 = \{7, 8, 9, 10, 11\}$. The lowest-numbered vertex in each maximal clique is its representative; note that in our notation $K_2 = K(3)$, $K_1 = K(1)$, $K_3 = K(5)$, and $K_4 = K(7)$.

Let $\mathcal{K}_G = \{K_1, K_2, \dots, K_m\}$ denote the set of maximal cliques of a chordal graph G . Define a clique intersection graph with the maximal cliques as its vertices, with two maximal cliques K_i and K_j joined by an edge (K_i, K_j) of weight $|K_i \cap K_j|$. A *clique tree* corresponds to a maximum weight spanning tree (MST) of the clique intersection graph. Since the MST of a weighted graph need not be unique, a clique tree of a chordal graph is not necessarily unique either.

In practice, a rooted clique tree is used in sparse matrix computations. Lewis, Peyton, and Pothén [48] and Pothén and Sun [62] have designed algorithms for computing rooted clique trees. The former algorithm uses the adjacency lists of the filled graph as input, while the latter uses the elimination tree. Both algorithms identify representative vertices by a simple degree test. We will discuss the latter algorithm.

First, to define the concepts needed for the algorithm, consider that the maximal cliques are ordered according to their representative vertices. This ordering partitions each maximal clique $K(v)$ with representative vertex v into two subsets: $\text{new}(K(v))$ consists of vertices in the clique $K(v)$ whose higher adjacency sets are contained in it but not in any earlier ordered maximal clique. The residual vertices in $K(v) \setminus \text{new}(K(v))$ form the ancestor set $\text{anc}(K(v))$. If a vertex $w \in \text{anc}(K(v))$, by definition of the ancestor set, w has a higher neighbor that is not adjacent to v ; then by the rules of the elimination game, any higher-

```

for  $v := 1$  to  $n \rightarrow$ 
  if  $v$  has a child  $u$  in etree with  $\text{hd}^+(v) + 1 = \text{hd}^+(u)$  then
    let  $K_u$  be the clique in which  $u$  is a new vertex;
    add  $v$  to the set  $\text{new}(K_u)$ ;
  else
    make  $v$  the representative vertex of a maximal clique  $K(v)$ ;
    add  $v$  to the set  $\text{new}(K(v))$ ;
  fi
  for each child  $s$  of  $v$  in etree such that  $v$  and  $s$  are new vertices in different cliques  $\rightarrow$ 
    let  $K_s$  be the clique in which  $s$  is a new vertex;
    make  $K_s$  a child of the clique  $K_v$  in which  $v$  is a new vertex;
  rof
rof

```

FIGURE 59.6: An algorithm for computing a clique tree from an elimination tree, whose vertices are numbered in postorder. The variable $\text{hd}^+(v)$ is the higher degree of a vertex v in the filled graph.

numbered vertex $x \in K(v)$ also belongs to $\text{anc}(K(v))$. Thus the partition of a maximal clique into new and ancestor sets is an ordered partition: vertices in $\text{new}(K(v))$ are ordered before vertices in $\text{anc}(K(v))$. We denote the lowest numbered vertex f in $\text{anc}(K(v))$ the first ancestor of the clique $K(v)$. A rooted clique tree may be defined as follows: the parent of a clique $K(v)$ is the clique P in which the first ancestor vertex f of K appears as a vertex in $\text{new}(P)$.

The reason for calling these subsets ‘new’ and ‘ancestor’ sets can be explained with respect to a rooted clique tree. We can build the chordal graph beginning with the root clique of the clique tree, successively adding one maximal clique at a time, proceeding down the clique tree in in-order. When a maximal clique $K(v)$ is added, vertices in $\text{anc}(K(v))$ also belong to some ancestor clique(s) of $K(v)$, while vertices in $\text{new}(K(v))$ appear for the first time. A rooted clique tree, with vertices in $\text{new}(K)$ and $\text{anc}(K)$ identified for each clique K , is shown in Fig. 59.7.

This clique tree algorithm can be implemented in $O(n)$ time, once the elimination tree and the higher degrees have been computed. The rooted clique tree shown in Fig. 59.7, is computed from the example elimination tree and higher degrees of the vertices in the example filled graph, using the clique tree algorithm described above. The clique tree obtained from this algorithm is not unique. A second clique tree that could be obtained has the clique $K(5)$ as the root clique, and the other cliques as leaves.

A comprehensive review of clique trees and chordal graphs in sparse matrix computations, current as of 1991, is provided by Blair and Peyton [7].

59.3.2 Design of Efficient Algorithms with Clique Trees

Shortest Elimination Trees. Jess and Kees [46] introduced the problem of modifying a fill-reducing elimination ordering to enhance concurrency in a parallel factorization algorithm. Their approach was to generate a chordal filled graph from the elimination ordering, and then to eliminate a maximum independent set of simplicial vertices at each step, until all the vertices are eliminated. (This is a greedy algorithm in which the largest number of pairwise independent columns that do not cause fill are eliminated in one step.) Liu

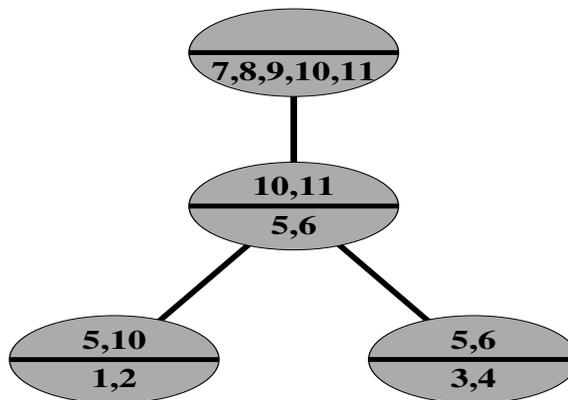


FIGURE 59.7: A clique tree of the example filled graph, computed from its elimination tree. Within each clique K in the clique tree, the vertices in $\text{new}(K)$ are listed below the bar, and the vertices in $\text{anc}(K)$ are listed above the bar.

and Mirzaian [55] showed that this approach computed a shortest elimination tree over all perfect elimination orderings for a chordal graph, and provided an implementation linear in the number of edges of the filled graph. Lewis, Peyton, and Pothen [55] used the clique tree to provide a faster algorithm; their algorithm runs in time proportional to the size of the clique tree: the sum of the sizes of the maximal cliques of the chordal graph.

A vertex is simplicial if and only if it belongs to exactly one maximal clique in the chordal graph; a maximum independent set of simplicial vertices is obtained by choosing one such vertex from each maximal clique that contains simplicial vertices, and thus the clique tree is a natural data structure for this problem. The challenging aspect of the algorithm is to update the rooted clique tree when simplicial vertices are eliminated and cliques that become non-maximal are *absorbed* by other maximal cliques.

Parallel Triangular Solution. In solving systems of linear equations by factorization methods, usually the work involved in the factorization step dominates the work involved in the triangular solution step (although the communication costs and synchronization overheads of both steps are comparable). However, in some situations, many linear systems with the same coefficient matrix but with different right-hand-side vectors need to be solved. In such situations, it is tempting to replace the triangular solution step involving the factor matrix L by explicitly computing an inverse L^{-1} of the factor. Unfortunately L^{-1} can be much less sparse than the factor, and so a more space efficient ‘product-form inverse’ needs to be employed. In this latter form, the inverse is represented as a product of triangular matrices such that all the matrices in the product together require exactly as much space as the original factor.

The computation of the product form inverse leads to some interesting chordal graph partitioning problems that can be solved efficiently by using a clique tree data structure.

We begin by directing each edge in the chordal filled graph $G^+(A)$ from its lower to its higher numbered end point to obtain a directed acyclic graph (DAG). We will denote this

DAG by $G(L)$. Given an edge (i, j) directed from i to j , we will call i the predecessor of j , and j the successor of i . The elimination ordering must eliminate vertices in a topological ordering of the DAG such that all predecessors of a vertex must be eliminated before it can be eliminated. The requirement that each matrix in the product form of the inverse must have the same nonzero structure as the corresponding columns in the factor is expressed by the fact that the subgraph corresponding to the matrix should be transitively closed. (A directed graph is *transitively closed* if whenever there is a directed path from a vertex i to a vertex j , there is an edge directed from i to j in the graph.) Given a set of vertices P_i , the *column subgraph* of P_i includes all the vertices in P_i and vertices reached by directed edges leaving vertices in P_i ; the edges in this subgraph include all edges with one or both endpoints in P_i .

The simpler of the graph partitioning problems is the following:

Find an ordered partition $P_1 \prec P_2 \prec \dots P_m$ of the vertices of a directed acyclic filled graph $G(L)$ such that

1. every $v \in P_i$ has all of its predecessors included in P_1, \dots, P_i ;
2. the column subgraph of P_i is transitively closed; and
3. the number of subgraphs m is minimum over all topological orderings of $G(L)$.

Pothen and Alvarado [61] designed a greedy algorithm that runs in $O(n)$ time to solve this partitioning problem by using the elimination tree.

A more challenging variant of the problem minimizes the number of transitively closed subgraphs in $G(L)$ over all perfect elimination orderings of the undirected chordal filled graph $G^+(A)$. This variant could change the edges in the DAG $G(L)$, (but not the edges in $G^+(A)$) since the initial ordering of the vertices is changed by the perfect elimination ordering, and after the reordering, edges are directed from the lower numbered end point to its higher numbered end point.

This is quite a difficult problem, but two surprisingly simple greedy algorithms solve it. Peyton, Pothen, and Yuan provide two different algorithms for this problem; the first algorithm uses the elimination tree and runs in time linear in the number of edges in the filled graph [59]. The second makes use of the clique tree, and computes the partition in time linear in the size of the clique tree [60]. Proving the correctness of these algorithms requires a careful study of the properties of the minimal vertex separators (these are vertices in the intersections of the maximal cliques) in the chordal filled graph.

59.3.3 Compact Clique Trees

In analogy with skeleton graphs, we can define a space-efficient version of a clique tree representation of a chordal graph, called the compact clique tree. If K is the parent clique of a clique C in a clique tree, then it can be shown that $\text{anc}(C) \subset K$. Thus trading space for computation, we can delete the vertices in K that belong to the ancestor sets of its children, since we can recompute them when necessary by unioning the ancestor sets of the children. The partition into new and ancestor sets can be obtained by storing the lowest numbered ancestor vertex for each clique. A compact clique K^c corresponding to a clique K is:

$$K^c = K \setminus \cup_{C \in \text{child}(K)} \text{anc}(C).$$

Note that the compact clique depends on the specific clique tree from which it is computed.

A compact clique tree is obtained from a clique tree by replacing cliques by compact cliques for vertices. In the example clique tree, the compact cliques of the leaves are unchanged from the corresponding cliques; and the compact cliques of the interior cliques are $K^c(5) = \{11\}$, and $K^c(7) = \{7, 8, 9\}$.

The compact clique tree is potentially sparser (asymptotically $O(n)$ instead of $O(n^2)$ even) than the skeleton graph on pathological examples, but on “practical” examples, the size difference between them is small. Compact clique trees were introduced by Pothen and Sun [62].

59.4 Clique Covers and Quotient Graphs

Clique covers and quotient graphs are data structures that were developed for the efficient implementation of minimum-degree reordering heuristics for sparse matrices. In Gaussian elimination, an elimination step that uses a_{ij} as a pivot (the elimination of the j th unknown using the i th equation) modifies every coefficient a_{kl} for which $a_{kj} \neq 0$ and $a_{il} \neq 0$. Minimum-degree heuristics attempt to select pivots for which the number of modified coefficients is small.

59.4.1 Clique Covers

Recall the graph model of symmetric Gaussian elimination discussed in subsection 59.1.1. The adjacency graph of the matrix to be factored is an undirected graph $G = (V, E)$, $V = \{1, 2, \dots, n\}$, $E = \{(i, j) : a_{ij} \neq 0\}$. The elimination of a row/column j corresponds to eliminating vertex j and adding edges to the remaining graph so that the neighbors of j become a clique. If we represent the edge set E using a clique cover, a set of cliques $\mathcal{K} = \{K : K \subseteq V\}$ such that $E = \cup_{K \in \mathcal{K}} \{(i, j) : i, j \in K\}$, the vertex elimination process becomes a process of merging cliques [63]: The elimination of vertex j corresponds to merging all the cliques that j belongs to into one clique and removing j from all the cliques. Clearly, all the old cliques that j used to belong to are now covered by the new clique, so they can be removed from the cover. The clique-cover can be initialized by representing every nonzero of A by a clique of size 2. This process corresponds exactly to symbolic elimination, which we have discussed in Section 59.2, and which costs $\Theta(|L|)$ work. The cliques correspond exactly to frontal matrices in the multifrontal factorization method.

In the sparse-matrix literature, this model of Gaussian elimination has been sometimes called the *generalized-element* model or the *super-element* model, due to its relationship to finite-element models and matrices.

The significance of clique covers is due to the fact that in minimum-degree ordering codes, there is no need to store the structure of the partially computed factor, so when one clique is merged into another, it can indeed be removed from the cover. This implies that the total size $\sum_{K \in \mathcal{K}} |K|$ of the representation of the clique cover, which starts at exactly $|A| - n$, shrinks in every elimination step, so it is always bounded by $|A| - n$. Since exactly one clique is formed in every elimination step, the total number of cliques is also bounded, by $n + (|A| - n) = |A|$. In contrast, the storage required to explicitly represent the symbolic factor, or even to just explicitly represent the edges in the reduced matrix, can grow in every elimination step and is not bounded by $O(|A|)$.

Some minimum-degree codes represent cliques fully explicitly [8, 36]. This representation uses an array of cliques and an array of vertices; each clique is represented by a linked list of vertex indices, and each vertex is represented by a linked list of clique indices to which it belongs. The size of this data structure never grows—linked-list elements are moved from one list to another or are deleted during elimination steps, but new elements never need to be allocated once the data structure is initialized.

Most codes, however, use a different representation for clique covers, which we describe next.

59.4.2 Quotient Graphs

Most minimum-degree codes represent the graphs of reduced matrices during the elimination process using *quotient graphs* [25]. Given a graph $G = (V, E)$ and a partition \mathcal{S} of V into disjoint sets $S_j \in \mathcal{S}$, the quotient graph G/\mathcal{S} is the undirected graph $(\mathcal{S}, \mathcal{E})$ where $\mathcal{E} = \{(\mathcal{S}_i, \mathcal{S}_j) : \text{adj}(\mathcal{S}_i) \cap \mathcal{S}_j \neq \emptyset\}$.

The representation of a graph G after the elimination of vertices $1, 2, \dots, j-1$, but before the elimination of vertex j , uses a quotient graph G/\mathcal{S} , where \mathcal{S} consists of sets S_k of eliminated vertices that form maximal connected components in G , and sets $S_i = \{i\}$ of uneliminated vertices $i \geq j$. We denote a set S_k of eliminated vertices by the index k of the highest-numbered vertex in it.

This quotient graph representation of an elimination graph corresponds to a clique cover representation as follows. Each edge in the quotient graph between uneliminated vertices $S_{\{i_1\}}$ and $S_{\{i_2\}}$ corresponds to a clique of size 2; all the neighbors of an eliminated set S_k correspond to a clique, the clique that was created when vertex k was eliminated. Note that all the neighbors of an uneliminated set S_k are uneliminated vertices, since uneliminated sets are maximal with respect to connectivity in G .

The elimination of vertex j in the quotient-graph representation corresponds to marking S_j as eliminated and merging it with its eliminated neighbors, to maintain the maximal connectivity invariant.

Clearly, a representation of the initial graph G using adjacency lists is also a representation of the corresponding quotient graph. George and Liu [26] show how to maintain the quotient graph efficiently using this representation without allocating more storage through a series of elimination steps.

Most of the codes that implement minimum-degree ordering heuristics, such as GEN-MMD [50], AMD [2], and Spindle [14, 47], use quotient graphs to represent elimination graphs.

It appears that the only advantage of a quotient graph over an explicit clique cover in the context of minimum-degree algorithms is a reduction by a small constant factor in the storage requirement, and possibly in the amount of work required. Quotient graphs, however, can also represent symmetric partitions of symmetric matrices in applications that are not directly related to elimination graphs. For example, George and Liu use quotient graphs to represent partitions of symmetric matrices into block matrices that can be factored without fill in blocks that only contain zeros [29, Chapter 6].

In [27], George and Liu showed how to implement the minimum degree algorithm without modifying the representation of the input graph at all. In essence, this approach represents the quotient graph implicitly using the input graph and the indices of the eliminated vertices. The obvious drawback of this approach is that vertex elimination (as well as other required operations) are expensive.

59.4.3 The Problem of Degree Updates

The minimum-degree algorithm works by repeatedly eliminating the vertex with the minimum degree and turning its neighbors into a clique. If the reduced graph is represented by a clique cover or a quotient graph, then the representation does not reveal the degree of vertices. Therefore, when a vertex is eliminated from a graph represented by a clique cover or a quotient graph, the degrees of its neighbors must be recomputed. These degree updates can consume much of the running time of minimum-degree algorithms.

Practical minimum-degree codes use several techniques to address this issue. Some techniques reduce the running time while preserving the invariant that the vertex that is elim-

inated always has the minimum degree. For example, mass elimination, the elimination of all the vertices of a supernode consecutively without recomputing degrees, can reduce the running time significantly without violating this invariant. Other techniques, such as multiple elimination and the use of approximate degrees, do not preserve the minimum-degree invariant. This does not imply that the elimination orderings that such technique produce are inferior to true minimum-degree orderings. They are often superior to them. This is not a contradiction since the minimum-degree rule is just a heuristic which is rarely optimal. For further details, we refer the reader to George and Liu's survey [30], to Amestoy, Davis, and Duff's paper on approximate minimum-degree rules [2], and to Kumfert and Pothen's work on minimum-degree variants [14, 47]. Heggernes, Eisenstat, Kumfert and Pothen prove upper bounds on the running time of space-efficient minimum-degree variants [44].

59.4.4 Covering the Column-Intersection Graph and Biclique Covers

Column orderings for minimizing fill in Gaussian elimination with partial pivoting and in the orthogonal-triangular (QR , where Q is an orthogonal matrix, and R is an upper triangular matrix) factorization are often based on symmetric fill minimization in the symmetric factor of $A^T A$, whose graph is known as the the column intersection graph $G_{\cap}(A)$ (we ignore the possibility of numerical cancellation in $A^T A$). To run a minimum-degree algorithm on the column intersection graph, a clique cover or quotient graph of it must be constructed. One obvious solution is to explicitly compute the edge-set of $G_{\cap}(A)$, but this is inefficient, since $G_{\cap}(A)$ can be much denser than $G(A)$.

A better solution is to initialize the clique cover using a clique for every row of A ; the vertices of the clique are the indices of the nonzeros in that row [30]. It is easy to see that each row in A indeed corresponds to a clique in $G_{\cap}(A)$. This approach is used in the COLMMD routine in MATLAB [36] and in COLAMD [11].

A space-efficient quotient-graph representation for $G_{\cap}(A)$ can be constructed by creating an adjacency-list representation of the symmetric 2-by-2 block matrix

$$\begin{pmatrix} I & A \\ A^T & 0 \end{pmatrix}$$

and eliminating vertices 1 through n . The graph of the Schur complement matrix

$$G(0 - A^T I^{(-1)} A) = G(A^T A) = G_{\cap}(A).$$

If we maintain a quotient-graph representation of the reduced graph through the first n elimination steps, we obtain a space-efficient quotient graph representation of the column-intersection graph. This is likely to be more expensive, however, than constructing the clique-cover representation from the rows of A . We learned of this idea from John Gilbert; we are not aware of any practical code that uses it.

The nonzero structure of the Cholesky factor of $A^T A$ is only an upper bound on the structure of the LU factors in Gaussian elimination with partial pivoting. If the identities of the pivots are known, the nonzero structure of the reduced matrices can be represented using biclique covers. The nonzero structure of A is represented by a bipartite graph $(\{1, 2, \dots, n\} \cup \{1', 2', \dots, n'\}, \{(i, j') : a_{ij} \neq 0\})$. A biclique is a complete bipartite graph on a subset of the vertices. Each elimination step corresponds to a removal of two connected vertices from the bipartite graph, and an addition of a new biclique. The vertices of the new biclique are the neighbors of the two eliminated vertices, but they are not the union of a set of bicliques. Hence, the storage requirement of this representation may exceed the storage required for the initial representation. Still, the storage requirement is always smaller

than the storage required to represent each edge of the reduced matrix explicitly. This representation poses the same degree update problem that symmetric clique covers pose, and the same techniques can be used to address it. Version 4 of UMFPACK, an asymmetric multifrontal LU factorization code, uses this idea together with a degree approximation technique to select pivots corresponding to relatively sparse rows in the reduced matrix [9].

59.5 Column Elimination Trees and Elimination DAGS

Elimination structures for asymmetric Gaussian elimination are somewhat more complex than the equivalent structures for symmetric elimination. The additional complexity arises because of two issues. First, the factorization of a sparse asymmetric matrix A , where A is factored into a lower triangular factor L and an upper triangular factor U , $A = LU$, is less structured than the sparse symmetric factorization process. In particular, the relationship between the nonzero structure of A and the nonzero structure of the factors is much more complex. Consequently, data structures for predicting fill and representing data-flow and control-flow dependences in elimination algorithms are more complex and more diverse.

Second, factoring an asymmetric matrix often requires *pivoting*, row and/or column exchanges, to ensure existence of the factors and numerical stability. For example, the 2-by-2 matrix $A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ does not have an LU factorization, because there is no way to eliminate the first variable from the first equation: that variable does not appear in the equation at all. But the permuted matrix PA does have a factorization, if P is a permutation matrix that exchanges the two rows of A . In finite precision arithmetic, row and/or column exchanges are necessary even when a nonzero but small diagonal element is encountered. Some sparse LU algorithms perform either row or column exchanges, but not both. The two cases are essentially equivalent (we can view one as a factorization of A^T), so we focus on row exchanges (partial pivoting). Other algorithms, primarily multifrontal algorithms, perform both row and column exchanges; these are discussed toward the end of this section.

For completeness, we note that pivoting is also required in the factorization of sparse symmetric indefinite matrices. Such matrices are usually factored into a product LDL^T , where L is lower triangular and D is a block diagonal matrix with 1-by-1 and 2-by-2 blocks. There has not been much research about specialized elimination structures for these factorization algorithms; such codes invariably use the symmetric elimination tree of A to represent dependences for structure prediction and for scheduling the factorization.

The complexity and diversity of asymmetric elimination arises not only due to pivoting, but also because asymmetric factorizations are less structured than symmetric ones, so a rooted tree can no longer represent the factors. Instead, directed acyclic graphs (dags) are used to represent the factors and dependences in the elimination process. We discuss *elimination dags* (*edags*) in Section 59.5.2.

Surprisingly, dealing with partial pivoting turns out to be simpler than dealing with the asymmetry, so we focus next on the *column elimination tree*, an elimination structure for LU factorization with partial pivoting.

59.5.1 The Column Elimination Tree

The *column elimination tree* (*col-etree*) is the elimination tree of $A^T A$, under the assumption that no numerical cancellation occurs in the formation of $A^T A$. The significance of this tree to LU with partial pivoting stems from a series of results that relate the structure of the LU factors of PA , where P is some permutation matrix, to the structure of the Cholesky factor of $A^T A$.

block-triangular form), there exists a permutation matrix P such that every edge of the col-etree corresponds to a nonzero in the upper-triangular factor of PA . This implies that the a-priori symbolic column-dependence structure predicted by the col-etree is as tight as possible.

Like the etree of a symmetric matrix, the col-etree can be computed in time almost linear in the number of nonzeros in A [34]. This is done by an adaptation of the symmetric etree algorithm, an adaptation that does not compute explicitly the structure of $A^T A$. Instead of constructing $G(A^T A)$, the algorithm constructs a much sparser graph G' with the same elimination tree. The main idea is that each row of A contributes a clique to $G(A^T A)$; this means that each nonzero index in the row must be an ancestor of the preceding nonzero index. A graph in which this row-clique is replaced by a path has the same elimination tree, and it has only as many edges as there are nonzeros in A . The same paper shows not only how to compute the col-etree in almost linear time, but also how to bound the number of nonzeros in each row and column of the factors L and U , using again an extension of the symmetric algorithm to compute the number of nonzeros in the Cholesky factor of $A^T A$. The decomposition of this Cholesky factor into fundamental supernodes, which the algorithm also computes, can be used to bound the extent of fundamental supernodes that will arise in L .

59.5.2 Elimination DAGS

The col-etree represents all possible column dependences for any sequence of pivot rows. For a specific sequence of pivots, the col-etree includes dependences that do not occur during the factorization with these pivots. There are two typical situations in which the pivoting sequence is known. The first is when the matrix is known to have a stable LU factorization without pivoting. The most common case is when A^T is strictly diagonally dominant. Even if A is not diagonally dominant, its rows can be pre-permuted to bring large elements to the diagonal. The permuted matrix, even if its transpose is not diagonally dominant, is fairly likely to have a relatively stable LU factorization that can be used to accurately solve linear systems of equations. This strategy is known as *static pivoting* [49]. The other situation in which the pivoting sequence is known is when the matrix, or part of it, has already been factored. Since virtually all sparse factorization algorithms need to collect information from the already-factored portion of the matrix before they factor the next row and column, a compact representation of the structure of this portion is useful.

Elimination dags (edags) are directed acyclic graphs that capture a minimal or near minimal set of dependences in the factors. Several edags have been proposed in the literature. There are two reasons for this diversity. First, edags are not always as sparse and easy to compute as elimination trees, so researchers have tried to find edags that are easy to compute, even if they represent a superset of the actual dependences. Second, edags often contain information only about a specific structure in the factors or a specific dependence in a specific elimination algorithm (e.g., data dependence in a multifrontal algorithm), so different edags are used for different applications. In other words, edags are not as universal as etrees in their applications.

The simplest edag is the graph $G(L^T)$ of the transpose of the lower triangular factor, if we view every edge in this graph as directed from the lower-numbered vertex to a higher-numbered vertex. This corresponds to orienting edges from a row index to a column index in L . For example, if $L_{6,3} \neq 0$, we view the edge $(6, 3)$ as a directed edge $3 \rightarrow 6$ in $G(L^T)$. Let us denote by $G((L^{(j-1)})^T)$ the partial lower triangular factor after $j - 1$ columns have been factored. Gilbert and Peierls showed that the nonzeros in the j th rows of L and U are exactly the vertices reachable, in $G((L^{(j-1)})^T)$, from the nonzero indices in the j th column

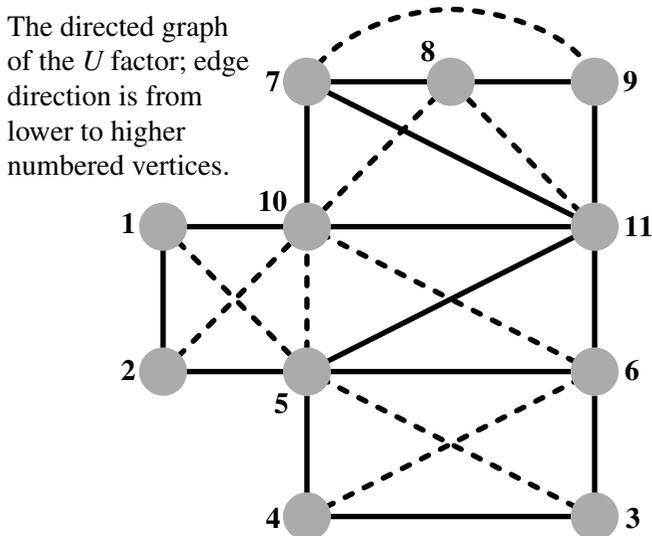


FIGURE 59.9: The directed graph of the U factor of the matrix whose graph is shown in Figure 59.8. In this particular case, the graph of L^T is exactly the same graph, only with the direction of the edges reversed. Fill is indicated by dashed lines. Note that the fill is indeed bounded by the fill in the column-intersection graph, which is shown in Figure 59.1. However, that upper bound is not realized in this case: the edge $(9, 10)$ fills in the column-intersection graph, but not in the LU factors.

of A [39]. This observation allowed them to use a depth-first search (DFS) to quickly find the columns in the already-factored part of the matrix that update the j th column before it can be factored. This resulted in the first algorithm that factored a general sparse matrix in time linear in the number of arithmetic operations (earlier algorithms sometimes performed much more work to manipulate the sparse data structure than the amount of work in the actual numerical computations).

Eisenstat and Liu showed that a simple modification of the graph $G((L^{(j-1)})^T)$ can often eliminate many of its edges without reducing its ability to predict fill [22]. They showed that if both L_{ik} and U_{ki} are nonzeros, then all the edges $i \rightarrow \ell$ for $\ell > i$ can be safely pruned from the graph. In other words, the nonzeros in column k of L below row i can be pruned. This is true since if $U_{ki} \neq 0$, then column k of L updates column i , so all the pruned nonzeros appear in column i , and since the edge $k \rightarrow i$ is in the graph, they are all reachable when k is reachable. This technique is called *symmetric pruning*. This edge is used in the SuperLU codes [12, 13] to find the columns that update the next supernode (set of consecutive columns with the same nonzero structure in L). Note that the same codes use the col-*etree* to predict structure before the factorization begins, and an edge to compactly represent the structure of the already-factored block of A .

Gilbert and Liu went a step further and studied the *minimal* edges that preserve the reachability property that is required to predict fill [35]. These graphs, which they called the *elimination dags* are the transitive reductions of the directed graphs $G(L^T)$ and $G(U)$. (The graph of U can be used to predict the row structures in the factors, just as $G(L^T)$ can predict the column structures.) Since these graphs are acyclic, each graph has a unique transitive reduction; If A is symmetric, the transitive reduction is the symmetric elimination tree. Gilbert and Liu also proposed an algorithm to compute these transitive reductions row by row. Their algorithm computes the next row i of the transitive reduction of L by

The minimal dag of the U factor; edge direction is from lower to higher numbered vertices.

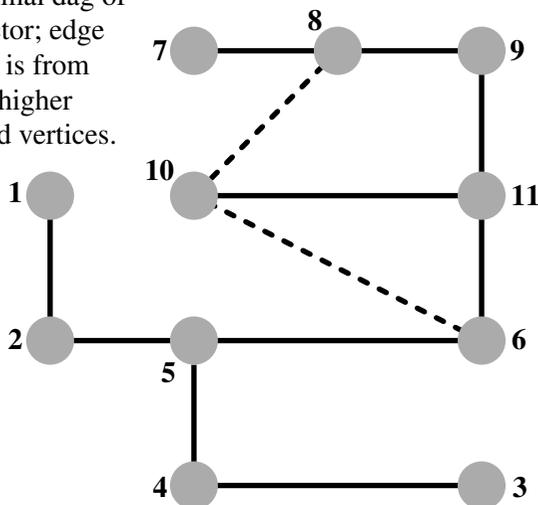


FIGURE 59.10: The minimal edag of U ; this graph is the transitive reduction of the graph shown in Figure 59.9.

traversing the reduction of U to compute the structure of row i of L , and then reducing it. Then the algorithm computes the structure of row i of U by combining the structures of earlier rows whose indices are the nonzeros in row i of L . In general, these minimal edags are often more expensive to compute than the symmetrically-pruned edags, due to the cost of transitively reducing each row. Gupta recently proposed a different algorithm for computing the minimal edags [41]. His algorithm computes the minimal structure of U by rows and of L by columns. His algorithm essentially applies to both L and U the rule that Gilbert and Liu apply to U . By computing the structure of U by rows and of L by columns, Gupta's algorithm can cheaply detect supernodes that are suitable for asymmetric multifrontal algorithms, where a supernode consists of a set of consecutive indices for which both the rows of U all have the same structure and the columns of L have the same structure (but the rows and columns may have different structures).

59.5.3 Elimination Structures for the Asymmetric Multifrontal Algorithm

Asymmetric multifrontal LU factorization algorithms usually use both row and column exchanges. UMFPACK, the first such algorithm, due to Davis and Duff [10], used a pivoting strategy that factored an arbitrary row and column permutation of A . The algorithm tried to balance numerical and degree considerations in selecting the next row and column to be factored, but in principle, all row and column permutations were possible. Under such conditions, not much structure prediction is possible. The algorithm still used a clever elimination structure that we described earlier, a biclique cover, to represent the structure of the Schur complement (the remaining uneliminated equations and variables), but it did not use etrees or edags.

Recent unsymmetric multifrontal algorithms still use pivoting strategies that allow both row and column exchanges, but the pivoting strategies are restricted enough that structure prediction is possible. These pivoting strategies are based on *delayed pivoting*, which was originally invented for symmetric indefinite factorizations. One such code, Davis's

UMFPACK 4, uses the column elimination tree to represent control-flow dependences, and a biclique cover to represent data dependences [9]. Another code, Gupta's WSMP, uses conventional minimal edags to represent control-flow dependences, and specialized dags to represent data dependences [41]. More specifically, Gupta shows how to modify the minimal edags so they exactly represent data dependences in the unsymmetric multifrontal algorithm with no pivoting, and how to modify the edags to represent dependences in an unsymmetric multifrontal algorithm that employs delayed pivoting.

Acknowledgment

Alex Pothen was supported, in part, by the National Science Foundation under grant CCR-0306334, and by the Department of Energy under subcontract DE-FC02-01ER25476. Sivan Toledo was supported in part by an IBM Faculty Partnership Award, by grant 572/00 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities), and by Grant 2002261 from the US-Israeli Binational Science Foundation.

References

- [1] Ajit Agrawal, Philip Klein, and R. Ravi. Cutting down on fill using nested dissection: Provably good elimination orderings. In [24], pages 31–55, 1993.
- [2] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.
- [3] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.
- [4] Cleve Ashcraft and Roger Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Transactions on Mathematical Software*, 15(4):291–309, 1989.
- [5] Haim Avron and Sivan Toledo. A new parallel unsymmetric multifrontal sparse LU factorization. In preparation; presented at the 11th SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, February 2004.
- [6] Marshall Bern, John R. Gilbert, Bruce Hendrickson, Nhat Nguyen, and Sivan Toledo. Support-graph preconditioners. Submitted to the *SIAM Journal on Matrix Analysis and Applications*, 29 pages, January 2001.
- [7] Jean R. S. Blair and Barry W. Peyton. An introduction to chordal graphs and clique trees. In [24], pages 1–30, 1993.
- [8] Tzu-Yi Chen, John R. Gilbert, and Sivan Toledo. Toward an efficient column minimum degree code for symmetric multiprocessors. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, 1999. 11 pp. on CDROM.
- [9] Timothy A. Davis. A column pre-ordering strategy for the unsymmetric-pattern multifrontal method. Technical Report TR-03-006, Department of Computer and Information Science and Engineering, University of Florida, May 2003.
- [10] Timothy A. Davis and Iain S. Duff. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM Journal on Matrix Analysis and Applications*, 18:140–158, 1997.
- [11] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond G. Ng. A column approximate minimum degree ordering algorithm. Technical Report TR-00-005, De-

- partment of Computer and Information Science and Engineering, University of Florida, 2000.
- [12] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 20:720–755, 1999.
 - [13] James W. Demmel, John R. Gilbert, and Xiaoye S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20:915–952, 1999.
 - [14] Florin Dobrián, Gary Kumfert, and Alex Pothén. The design of a sparse direct solver library using object-oriented techniques. In A. M. Bruaset, H. P. Langtangen, and E. Quak, editors, *Modern Software Tools in Scientific Computing*, pages 89–131. Springer-Verlag, 2000.
 - [15] Florin Dobrián and Alex Pothén. The design of I/O-efficient sparse direct solvers. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*. ACM Press, 2001. 21 pp. on CDROM.
 - [16] Florin Dobrián and Alex Pothén. A comparison of three external memory algorithms for factoring sparse matrices. In *Proceedings of the SIAM Conference on Applied Linear Algebra*, 11 pages, SIAM, 2003.
 - [17] Iain S. Duff. Full matrix techniques in sparse matrix factorizations. In G. A. Watson, editor, *Lecture Notes in Mathematics*, pages 71–84. Springer-Verlag, 1982.
 - [18] Iain S. Duff. Parallel implementation of multifrontal schemes. *Parallel Computing*, 3:193–204, 1986.
 - [19] Iain S. Duff. Sparse numerical linear algebra: direct methods and preconditioning. Technical Report RAL-TR-96-047, Department of Computation and Information, Rutherford Appleton Laboratory, Oxon OX11 0QX England, 1996.
 - [20] Iain S. Duff, Albert M. Erisman, and John K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.
 - [21] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.
 - [22] Stanley C. Eisenstat and Joseph W. H. Liu. Exploiting structural symmetry in unsymmetric sparse symbolic factorization. *SIAM Journal on Matrix Analysis and Applications*, 13:202–211, January 1992.
 - [23] Stanley C. Eisenstat, Martin H. Schultz, and Andrew H. Sherman. Software for sparse Gaussian elimination with limited core memory. In Ian S. Duff and C. W. Stewart, editors, *Sparse Matrix Proceedings*, pages 135–153. SIAM, Philadelphia, 1978.
 - [24] Alan George, John R. Gilbert, and Joseph W. H. Liu, editors. *Graph Theory and Sparse Matrix Computation*. Springer-Verlag, 1993. IMA Volumes in Applied Mathematics, Volume 56.
 - [25] Alan George and Joseph W. H. Liu. A quotient graph model for symmetric factorization. In Iain S. Duff and G. W. Stewart, editors, *Sparse Matrix Proceedings*, pages 154–175. SIAM, Philadelphia, 1978.
 - [26] Alan George and Joseph W. H. Liu. A fast implementation of the minimum degree algorithm using quotient graphs. *ACM Transactions on Mathematical Software*, 6(3):337–358, 1980.
 - [27] Alan George and Joseph W. H. Liu. A minimal storage implementation of the minimum degree algorithm. *SIAM Journal on Numerical Analysis*, 17:283–299, 1980.
 - [28] Alan George and Joseph W. H. Liu. An optimal algorithm for symbolic factorization of symmetric matrices. *SIAM Journal on Computing*, 9:583–593, 1980.
 - [29] Alan George and Joseph W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, 1981.

- [30] Alan George and Joseph W. H. Liu. The evolution of the minimum-degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.
- [31] Alan George and Esmond G. Ng. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM Journal on Scientific and Statistical Computing*, 6:390–409, 1985.
- [32] John R. Gilbert. An efficient parallel sparse partial pivoting algorithm. Technical Report 88/45052-1, Christian Michelsen Institute, Bergen, Norway, 1988.
- [33] John R. Gilbert and Laura Grigori. A note on the column elimination tree. *SIAM Journal on Matrix Analysis and Applications*, 25:143–151, 2003.
- [34] John R. Gilbert, Xiaoye S. Li, Esmond G. Ng, and Barry W. Peyton. Computing row and column counts for sparse QR and LU factorization. *BIT*, 41:693–710, 2001.
- [35] John R. Gilbert and Joseph W. H. Liu. Elimination structures for unsymmetric sparse LU factors. *SIAM Journal on Matrix Analysis and Applications*, 14:334–352, 1993.
- [36] John R. Gilbert, Cleve Moler, and Robert S. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.
- [37] John R. Gilbert and Esmond G. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In [24], pages 107–139, 1993.
- [38] John R. Gilbert, Esmond G. Ng, and Barry W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 15(4):1075–1091, 1994.
- [39] John R. Gilbert and Tim Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM Journal on Scientific and Statistical Computing*, 9:862–874, 1988.
- [40] John R. Gilbert and Sivan Toledo. High-performance out-of-core sparse LU factorization. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, 1999. 10 pp. on CDROM.
- [41] Anshul Gupta. Improved symbolic and numerical factorization algorithms for unsymmetric sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 24:529–552, 2002.
- [42] Anshul Gupta, Mahesh V. Joshi, and Vipin Kumar. WSSMP: A high-performance serial and parallel symmetric sparse linear solver. In Bo Kågström, Jack Dongarra, Erik Elmroth, and Jerzy Wasniewski, editors, *Proceedings of the 4th International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems (PARA '98)*, volume 1541 of *Lecture Notes in Computer Science*, pages 182–194, Umeå, Sweden, June 1998. Springer.
- [43] Michael T. Heath. Parallel direct methods for sparse linear systems. In David E. Keyes, Ahmed Sameh, and V. Venkatakrisnan, editors, *Parallel Numerical Algorithms*, pages 55–90. Kluwer, 1997.
- [44] Pinar Heggeres, Stanley C. Eisenstat, Gary Kumfert, and Alex Pothen. The computational complexity of the minimum degree algorithm. In *Proceedings of the 14th Norwegian Computer Science Conference (NIK 2001)*, Tromsø, Norway, November 2001, 12 pages. Also available as ICASE Report 2001-42, NASA/CR-2001-211421, NASA Langley Research Center.
- [45] Pascal Hénon, Pierre Ramet, and Jean Roman. PaStiX: A high-performance parallel direct solver for sparse symmetric definite systems. *Parallel Computing*, 28(2):301–321, 2002.
- [46] J. Jess and H. Kees. A data structure for parallel LU decomposition. *IEEE Transactions on Computers*, C-31:231–239, 1982.
- [47] Gary Karl Kumfert. *An object-oriented algorithmic laboratory for ordering sparse*

- matrices*. PhD thesis, Old Dominion University, December 2000.
- [48] John G. Lewis, Barry W. Peyton, and Alex Pothen. A fast algorithm for reordering sparse matrices for parallel factorization. *SIAM Journal on Scientific Computing*, 6:1147–1173, 1989.
 - [49] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A scalable distributed memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29:110–140, 2003.
 - [50] Joseph W. H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2):141–153, 1985.
 - [51] Joseph W. H. Liu. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Transactions on Mathematical Software*, 12(2):127–148, 1986.
 - [52] Joseph W. H. Liu. On the storage requirement in the out-of-core multifrontal method for sparse factorization. *ACM Transactions on Mathematical Software*, 12(3):249–264, 1986.
 - [53] Joseph W. H. Liu. An adaptive general sparse out-of-core Cholesky factorization scheme. *SIAM Journal on Scientific and Statistical Computing*, 8(4):585–599, 1987.
 - [54] Joseph W. H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11:134–172, 1990.
 - [55] Joseph W. H. Liu and Andrew Mirzaian. A linear reordering algorithm for parallel pivoting of chordal graphs. *SIAM Journal on Discrete Mathematics*, 2:100–107, 1989.
 - [56] Joseph W. H. Liu, Esmond G. Ng, and Barry W. Peyton. On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 14:242–252, 1993.
 - [57] Assaf Natanzon, Ron Shamir, and Roded Sharan. A polynomial approximation algorithm for the minimum fill-in problem. In *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing (STOC 98)*, pages 41–47, 1998.
 - [58] Seymour V. Parter. The use of linear graphs in Gaussian elimination. *SIAM Review*, 3:119–130, 1961.
 - [59] Barry W. Peyton, Alex Pothen, and Xiaoqing Yuan. Partitioning a chordal graph into transitive subgraphs for parallel sparse triangular solution. *Linear Algebra and its Applications*, 192:329–354, 1993.
 - [60] Barry W. Peyton, Alex Pothen, and Xiaoqing Yuan. A clique tree algorithm for partitioning a chordal graph into transitive subgraphs. *Linear Algebra and its Applications*, 223:553–588, 1995.
 - [61] Alex Pothen and Fernando Alvarado. A fast reordering algorithm for parallel sparse triangular solution. *SIAM Journal on Scientific and Statistical Computing*, 13:645–653, 1992.
 - [62] Alex Pothen and Chunguang Sun. Compact clique tree data structures for sparse matrix factorizations. In Thomas F. Coleman and Yuying Li, editors, *Large Scale Numerical Optimization*, pages 180–204. SIAM, Philadelphia, 1990.
 - [63] Donald J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In R. C. Read, editor, *Graph Theory and Computing*, pages 183–217. Academic Press, New York, 1972.
 - [64] Donald J. Rose, Robert E. Tarjan, and George S. Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on Computing*, 5:266–283, 1976.
 - [65] Edward Rothberg and Robert S. Schreiber. An alternative approach to sparse out-of-core factorization. presented at the 2nd SIAM Conference on Sparse Matrices, Coeur d’Alene, Idaho, October 1996.
 - [66] Edward Rothberg and Robert S. Schreiber. Efficient methods for out-of-core sparse

- Cholesky factorization. *SIAM Journal on Scientific Computing*, 21:129–144, 1999.
- [67] Vladimir Rotkin and Sivan Toledo. The design and implementation of a new out-of-core sparse Cholesky factorization method. *ACM Transactions on Mathematical Software*, 30(1), March 2004. In press.
- [68] Olaf Schenk and Klaus Gärtner. Sparse factorization with two-level scheduling in PARADISO. In *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*, Portsmouth, Virginia, March 2001. 10 pp. on CDROM.
- [69] Robert S. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Transactions on Mathematical Software*, 8:256–276, 1982.
- [70] Robert E. Tarjan. Unpublished lecture notes. 1975.
- [71] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs. *SIAM Journal of Computing*, 13:566–579, 1984. Addendum in 14:254–255, 1985.
- [72] Pravin M. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. Unpublished manuscript. A talk based on the manuscript was presented at the IMA Workshop on Graph Theory and Sparse Matrix Computation, October 1991, Minneapolis.
- [73] G. F. Whitten. Computation of fill-in for sparse symmetric positive definite matrices. Technical report, Computer Science Department, Vanderbilt University, Nashville, TN, 1978.
- [74] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 2:77–79, 1981.
- [75] Earl Zmijewski and John R. Gilbert. A parallel algorithm for sparse symbolic Cholesky factorization on a multiprocessor. *Parallel Computing*, 7:199–210, 1988.

60

Data Structures for Databases

	60.1	Overview of the Functionality of a Database Management System.....	60-1
	60.2	Data Structures for Query Processing..... Index Structures • Sorting Large Data Sets • The Parse Tree • Expression Trees • Histograms	60-3
Joachim Hammer <i>University of Florida</i>	60.3	Data Structures for Buffer Management	60-11
Markus Schneider <i>University of Florida</i>	60.4	Data Structures for Disk Space Management .. Record Organizations • Page Organizations • File Organization	60-15
	60.5	Conclusion	60-21

60.1 Overview of the Functionality of a Database Management System

Many of the previous chapters have shown that efficient strategies for complex data-structuring problems are essential in the design of fast algorithms for a variety of applications, including combinatorial optimization, information retrieval and Web search, databases and data mining, and geometric applications. The goal of this chapter is to provide the reader with an overview of the important data structures that are used in the implementation of a modern, general-purpose database management system (DBMS). In earlier chapters of the book the reader has already been exposed to many of the data structures employed in a DBMS context (e.g., B-trees, buffer trees, quad trees, R-trees, interval trees, hashing). Hence, we will focus mainly on their application but also introduce other important data structures to solve some of the fundamental data management problems such as *query processing and optimization*, *efficient representation of data on disk*, as well as the *transfer of data from main memory to disk*. Due to space constraints, we cannot cover applications of data structures to manage non-standard data such as multi-dimensional data, spatial and temporal data, multimedia data, or XML.

Before we begin our treatment of how data structures are used in a DBMS, we briefly review the basic architecture, its components, and their functionality. Unless otherwise noted, our discussion applies to a class of DBMSs that are based on the relational data model. These so-called relational database management systems make up the majority of systems in use today and are offered by all major vendors including IBM, Microsoft, Oracle, and Sybase. Most of the components described here can also be found in DBMSs based on other models such as the object-based model or XML.

Figure 60.1 depicts a conceptual overview of the main components that make up a DBMS. Rectangles represent system components, double-sided arrows represent input and output, and the solid connectors indicate data as well as process flow between two components.

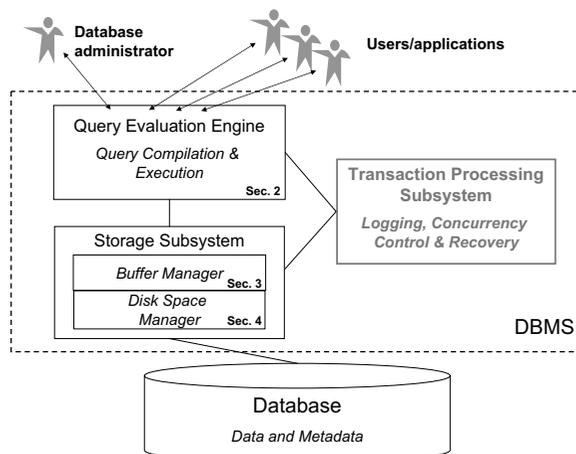


FIGURE 60.1: A simplified architecture of a database management system (DBMS).

Please note that the inner workings of a DBMS are quite complex and we are not attempting to provide a detailed discussion of its implementation. For an in-depth treatment the reader should refer to one of the many excellent database textbooks books, e.g., [3–8].

Starting from the top, users interact with the DBMS via commands generated from a variety of user interfaces or application programs. These commands can either retrieve or update the data that is managed by the DBMS or create or update the underlying metadata that describes the schema of the data. The former are called queries, the latter are called data definition statements. Both types of commands are processed by the *Query Evaluation Engine* which contains components for parsing the input, producing an execution plan, and executing the plan against the underlying database. In the case of queries, the parsed command is presented to a query optimizer component, which uses information about how the data is stored to produce an efficient execution plan from the possibly many alternatives. We discuss data structures that represent parsed queries, execution plans, and statistics about a database, including the data structures that are used by an external sorting algorithm in Section 60.2 when we focus on the query evaluation engine.

Since databases are normally too large to fit into the main memory of a computer, the data of a database resides in secondary memory, generally on one or more magnetic disks. However, to execute queries or modifications on data, that data must first be transferred to main memory for processing and then back to disk for persistent storage. It is the job of the *Storage Subsystem* to accomplish a sophisticated placement of data on disk, to assure an efficient localization of these persistent data, to enable their bidirectional transfer between disk and main memory, and to allow direct access to these data from other DBMS subsystems. The storage subsystem consists of two components: The *Disk Space Manager* is responsible for storing physical data items on disk, managing free regions of the disk space, hiding device properties from higher architecture levels, mapping physical blocks to tracks and sectors of a disc, and controlling the transfer of data items between external and main memory. The *Buffer Manager* organizes an assigned, limited main memory area called *buffer* and may comprise several smaller buffers (buffer pool). Other subsystems may have direct access to data items in these buffers.

In Sections 60.3 and 60.4, we discuss data structures that are used to represent both data

in memory as well as on disk such as fixed and variable-length records, large binary objects (LOBs), heap, sorted, and clustered files, as well as different types of index structures. Given the fact that a database management system must manage data that is both resident in main memory as well as on disk, one has to deal with the reality that the most appropriate data structure for data stored on disk is different from the data structures used for algorithms that run in main memory. Thus when implementing the storage manager, one has to pay careful attention to select not only the appropriate data structures but also to map the data between them in an efficient manner.

In addition to the above two subsystems, today's modern DBMSs include a *Transaction Management Subsystem* to support concurrent execution of queries against the database and recovery from failure. Although transaction processing is an important and complex topic, it is less interesting for our investigation of data structures and is mentioned here only for completeness.

The rest of this chapter is organized as follows. Section 60.2 describes important data structures used during query evaluation. Data structures used for buffer management are described in Section 60.3, and data structures used by the disk space manager are described in Section 60.4. Section 60.5 concludes the chapter.

60.2 Data Structures for Query Processing

Query evaluation is performed in main memory in several steps as outlined in [Figure 60.2](#). Starting with the high-level input query expressed in a declarative language called SQL (see, for example, [2]) the *Parser* scans, parses, and validates the query. The goal is to check whether the query is formulated according to the syntax rules of the language supported in the DBMS. The parser also validates that all attribute and relation names are part of the database schema that is being queried.

The parser produces a *parse tree* which serves as input to the *Query Translation and Rewrite* module shown underneath the parser. Here the query is translated into an internal representation, which is based on the relational algebra notation [1]. Besides its compact form, a major advantage of using relational algebra is that there exist transformations (re-write rules) between equivalent expressions to explore alternate, more efficient forms of the same query. Different algebraic expressions for a query are called *logical query plans* and are represented as *expression trees* or *operator trees*. Using the re-write rules, the initial logical query plan is transformed into an equivalent plan that is expected to execute faster. Query re-writing is guided by heuristics which help reduce the amount of intermediary work that must be performed by the query in order to arrive at the same result.

A particularly challenging problem is the selection of the best join ordering for queries involving the join of three or more relations. The reason is that the *order* in which the input relations are presented to a join operator (or any other binary operator for that matter) tends to have an important impact on the cost of the operation. Unfortunately, the number of candidate plans grows rapidly when the number of input relations grows.

The outcome of the query translation and rewrite module is a set of “improved” logical query plans representing different execution orders or combinations of operators of the original query. The *Physical Plan Generator* converts the logical query plans into *physical query plans* which contain information about the algorithms to be used in computing the relational operators represented in the plan. In addition, physical query plans also contain information about the access methods available for each relation. Access methods are ways of retrieving tuples from a table and consist of either a file scan (i.e., a complete retrieval of all tuples) or an index plus a matching selection condition. Given the many different

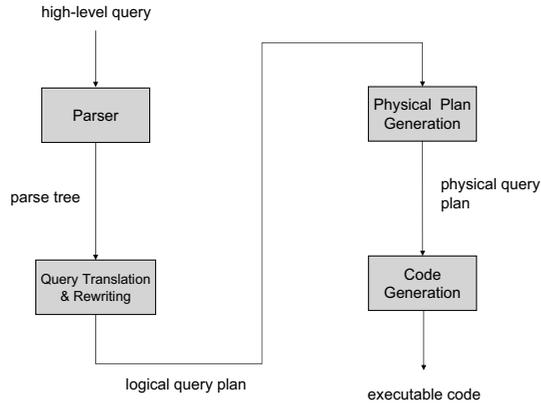


FIGURE 60.2: Outline of query evaluation.

options for implementing relational operators and for accessing the data, each logical plan may lead to a large number of possible physical plans. Among the many possible plans, the physical plan generator evaluates the cost for each and chooses the one with the lowest overall cost.

Finally, the best physical plan is submitted to the *Code Generator* which produces the executable code that is either executed directly or is stored and executed later whenever needed. Query re-writing and physical plan generation are referred to as *query optimization*. However, the term is misleading since in most cases the chosen plan represents a reasonably efficient strategy for executing a query.

In the following paragraphs, we focus on several important data structures that are used during query evaluation, some of which have been mentioned above: The *parse tree* for storing the parsed and validated input query (Section 60.2.3), the *expression tree* for representing logical and physical query plans (Section 60.2.4), and the *histogram* which is used to approximate the distribution of attribute values in the input relations (Section 60.2.5). We start with a summary of the well-known *index structures* and how they are used to *speed up database operations*. Since sorting plays an important role in query processing, we include a separate description of the data structures used to *sort large data sets using external memory* (Section 60.2.2).

60.2.1 Index Structures

An important part of the work of the physical plan generator is to choose an efficient implementation for each of the operators in the query. For each relational operator (e.g., **selection**, **projection**, **join**) there are several alternative algorithms available for implementation. The best choice usually depends on factors such as size of the relation, available memory in the buffer pool, sort order of the input data, and availability of index structures. In the following, we briefly highlight some of the important index structures that are used by a modern DBMS.

One-dimensional Indexes

One-dimensional indexes contain a single search key, which may be composed of multiple attributes. The most frequently used data structures for one-dimensional database indexes are dynamic tree-structured indexes such as B/B^+ -Trees (from now on collectively referred to as B-Trees, see also [Chapter 15](#)) and *hash-based indexes* using extendible and linear hashing (see [Chapter 9](#)). In general, hash-based indexes are especially good for equality searches. For example, in the case of an equality selection operation, one can use a one-dimensional hash-based index structure to examine just the tuples that satisfy the given condition. Consider the selection of student records having a certain grade point average (GPA). Assuming students are randomly distributed throughout the relation, an index on the GPA value could lead us to only those records satisfying the selection condition and resulting in a lot fewer data transfers than a sequential scan of the relation (if we assume the tuples satisfying the condition make up only a fraction of the entire relation).

Given their superior performance for equality searches hash-based indexes prove to be particularly useful in implementing relational operations such as joins. For example, the index-nested-loop join algorithm generates many equality selection queries, making the difference in cost between a hash-based and the slightly more expensive tree-based implementation significant.

B-Trees provide efficient support for range searches (all data items that fall within a range of values) and are almost as good as hash-based indexes for equality searches. Besides their excellent performance, B-Trees are “self-tuning”, meaning they maintain as many levels of the index as is appropriate for the size of the relation being indexed. Unlike hash-based indexes, B-Trees manage the space on the disk blocks they use and do not require any overflow blocks. As we have mentioned in [Sec. 60.1](#), database index structures are an example of data structures that have been designed as secondary memory structures.

Multi-dimensional Indexes

In addition to these one-dimensional index structures, many applications (e.g., geographic database, inventory and sales database for decision-support) also require data structures capable of indexing data existing in two or higher-dimensional spaces. In these domains, important database operations are selections involving partial matches (all points that match specified values in one or more dimensions), range queries (all points that fall within a range of values in one or more dimensions), nearest-neighbor queries (closest point to a given point), and so-called “where-am-I” queries (all the regions in which a given point is located).

The following are some of the most important data structures that support these types of operations.

Grid file. A multi-dimensional extension of one-dimensional hash tables. Grid files support range queries, partial-match queries, and nearest-neighbor queries well, as long as data is uniformly distributed.

Multiple-key index. The index on one attribute leads to indexes on another attribute for each value of the first. Multiple-key indexes are useful for range and nearest-neighbor queries.

R-tree. A B-Tree generalization suitable for collections of regions. R-Trees are used to represent a collection of regions by grouping them into a hierarchy of larger regions. They are well suited to support “where-am-I” queries as well as the other types of queries mentioned above if the atomic regions are individual points. (See also [Chapter 21](#).)

Quad tree. Recursively divide a multi-dimensional data set into quadrants until each quadrant contains a minimal number of points (e.g., amount of data that can fit on a disk block). Quad trees support partial-match, range, and nearest-neighbor queries well. (See also [Chapter 19](#)/)

Bitmap index. A collection of bit vectors which encode the location of records with a given value in a given field. Bitmap indexes support range, nearest-neighbor, and partial-match queries and are often employed in data warehouses and decision-support systems. Since bitmap indexes tend to get large when the underlying attributes have many values, they are often compressed using a run-length encoding.

Given the importance of database support for non-standard applications, many relational database management systems support one or more of these multi-dimensional indexes, either directly (e.g., bitmap indexes), or as part of a special-purpose extension to the core database engine (e.g., R-trees in a spatial extender).

In general, indexes are also used to answer certain types of queries without having to access the data file. For example, if we need only a few attribute values from each tuple and there is an index whose search key contains all these fields, we can choose an index scan instead of examining all data tuples. This is faster since index records are smaller (and hence fit into fewer buffer pages). Note that an index scan does not make use of the search structure of the index: for example, in a B-Tree index one would examine all leaf pages in sequence. All commercial relational database management systems support B-Trees and at least one type of hash-based index structure.

60.2.2 Sorting Large Data Sets

The need to sort large data sets arises frequently in data management. Besides outputting the result of a query in sorted order, sorting is useful for eliminating duplicate data items during the processing of queries. In addition, an efficient algorithm for performing a join operation (sort-merge join) requires the input relations to be sorted. Since the size of databases routinely exceeds the amount of available main memory, most DBMSs use an external sorting technique called *merge sort*, which is based on the main-memory version with the same name. The idea behind merge sort is that a file which does not fit into main memory can be sorted by breaking it into smaller pieces (runs), sorting the smaller runs individually, and then merging them to produce a single run that contains the original data items in sorted order. External merge sort is another example where main memory versions of algorithms and data structures need to be changed to accommodate a computing environment where all data resides on secondary and perhaps even tertiary storage. We will point out more such examples in Section 60.4 when we describe the disk space manager.

During the first phase, also called the run-generation phase, merge-sort fills the available buffer pages in main memory with blocks containing the data records from a file stored on disk. We will have more to say about the management of buffer pages when we discuss data structures for buffer management in Section 60.3. Sorting is done using any of the main-memory algorithms (e.g., Heapsort, Quicksort). The sorted records are written back to new blocks on disk, forming a sorted run containing as many blocks as there are available buffer pages in main memory. This process is repeated until all records in the data file are in one of the sorted runs. The arrangement of buffer pages and disk blocks during run generation is depicted in [Figure 60.3](#).

In the second phase, also called the merging phase, all but one of the main memory buffers are used to hold input data from one of the sorted runs. In most instances, the number

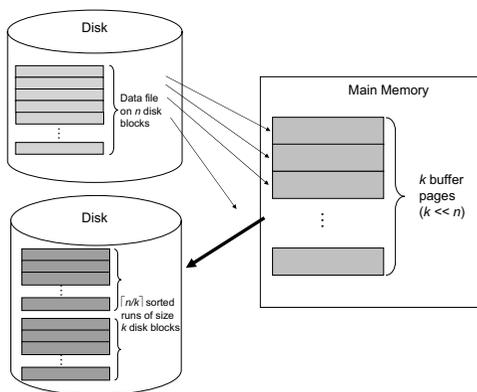


FIGURE 60.3: Arrangement of buffer pages and disk blocks during the run generation phase.

of sorted runs is less than the number of buffer pages and the merging can be done in one pass. Note, this so-called *multi-way merging* is different from the main-memory version of merge sort which merges pairs of runs (two-way merge). The arrangement of buffers and disk blocks to complete this one-pass multi-way merging step is shown in Figure 60.4. Note that the two-way merge strategy results in reading data in and out of memory $2 * \log_2(n)$ times for n runs (versus reading all n runs only once for the n -way strategy).

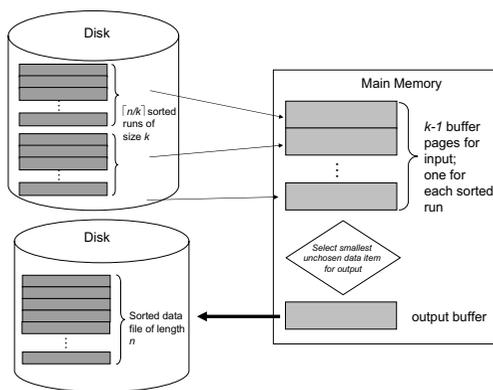


FIGURE 60.4: Arrangement of buffer pages and disk blocks during the one pass, multi-way merging phase.

In situations when the number of sorted runs exceeds the available buffer pages in main memory, the merging step must be performed in *several passes* as follows: assuming k buffer pages in main memory, each pass involves the repeated merging of $k - 1$ runs until all runs have been merged. At this point the number of runs has been reduced by a factor of $k - 1$.

If the reduced number of sublists is still greater than k , the merging is repeated until the number of sublists is less than k . A final merge generates the sorted output. In this scenario, the number of merge passes required is $\lceil \log_{k-1}(n/k) \rceil$.

60.2.3 The Parse Tree

A *parse tree* is an m -ary tree data structure that represents the structure of a query. Each interior node of the tree is labeled with a non-terminal symbol from the grammar of the query language. The root node is labeled with the goal symbol. The query being parsed appears at the bottom with each token of the query being a leaf in the tree. In the case of SQL, leaf nodes are lexical elements such as keywords of the language (e.g., *SELECT*), names of attributes or relations, operators, and other schema elements.

The parse tree for the SQL query selecting all the enrolled students with a GPA higher than 3.5.

```
SELECT Name
FROM Enrollment, Student
WHERE ID = SID AND GPA > 3.5;
```

is shown in Figure 60.5. For this example, we are tacitly assuming the existence of two relations called **Enrollment** and **Student** which store information about enrollment records for students in a school or university.

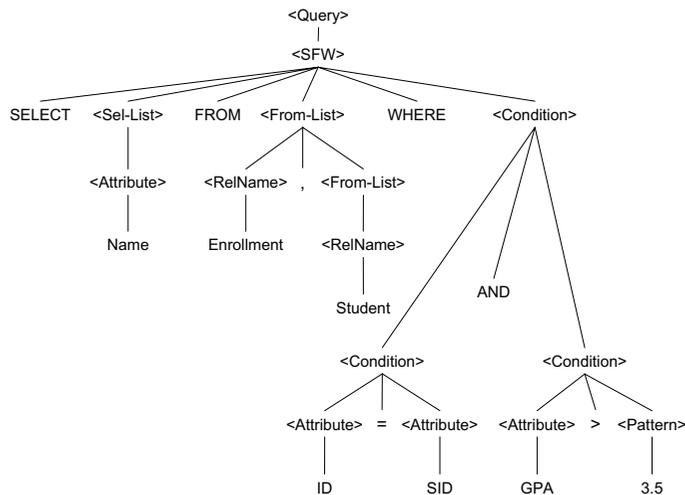


FIGURE 60.5: Sample parse tree for an SQL query showing query goal, interior and leaf nodes. Adapted from Garcia-Molina et al. [4].

The parse tree shown in Figure 60.5 is based on the grammar for SQL as defined in [4] (which is a subset of the full grammar for SQL). Non-terminal symbols are enclosed in angled brackets. At the root is the category $\langle Query \rangle$ which forms the goal for the parsed query. Descending down the tree, we see that this query is of the form *SFW* (select-from-where). In case one of the relations in the *FROM* clause is a view, it must be replaced by

its own parse tree since a view is essentially a query. A parse tree is said to be valid if it conforms to the syntactic rules of the grammar as well as the semantic rules on the use of the schema names.

60.2.4 Expression Trees

An *expression tree* is a binary tree data structure that represents a logical query plan for a query after it has been translated into a relational algebra expression. The expression tree represents the input relations of the query as *leaf nodes* of the tree, and the relational algebra operations together with estimates for result sizes as *internal nodes*.

Figure 60.6 shows an example of three expression trees representing different logical query plans for the following SQL query, which selects all the students enrolled in the course 'COP 4720' during the term 'Sp04' who have a grade point average of 3.5 (result sizes are omitted in the figure):

```
SELECT Name FROM Enrollment, Student
WHERE Enrollment.ID = Student.SID AND Enrollment.Course = 'COP 4720'
AND Enrollment.TermCode = 'Sp04' AND Student.GPA = 3.5;
```

An execution of a tree consists of executing an internal node operation whenever its operands are available and then replacing that internal node by the relation that results from executing the operation. The execution terminates when the root node is executed and produces the result relation for the query.

Note that expression trees representing *physical* query plans differ in the information that is stored in the nodes. For example, internal nodes contain information such as the operation being performed, any parameters if necessary, general strategy about the algorithm that is used, whether materialization of intermediate results or pipelining is used, and the anticipated number of buffers the operation will require (rather than result size as in logical query plans). At the leaf nodes table names are replaced by scan operators such as *TableScan*, *SortScan*, *IndexScan*, etc.

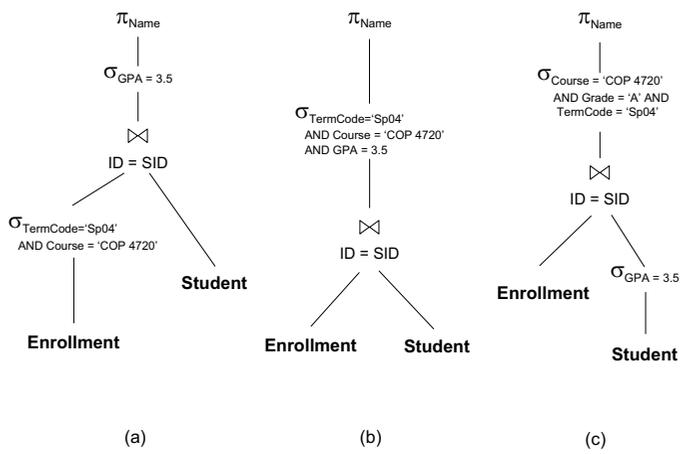


FIGURE 60.6: Three expression trees representing different logical query plans for the same query.

There is an interesting twist to the types of expression trees that are actually considered by the query optimizer. As we have previously pointed out, the number of different query plans (both logical and physical) for a given query can be very large. This is even more so the case, when the query involves the join of two or more relations since we need to take the join order into account when choosing the best possible plan. Today's query optimizers prune a large portion of the candidate expression trees and concentrate only on the class of *left-deep trees*. A left-deep tree is an expression tree in which the right child of each join is a leaf (i.e., a base table). For example, in [Figure 60.6](#), the tree labeled (a) is an example of a left-deep tree. The tree labeled (b) is an example of a *nonlinear* or bushy tree (a join node may have no leaf nodes), tree (c) is an example of a *right-deep tree* (the left child of each join node is a base table).

Besides the fact that the number of left-deep trees is smaller than the number of all trees, there is another advantage for considering only left-deep expression trees: Left-deep trees allow the query optimizer to generate more efficient plans by avoiding the intermediate storage (materialization) of join results. Note that in most join algorithms, the inner table must be materialized because we must examine the entire inner table for each tuple of the outer table. However, in a left-deep tree, all inner tables exist as base tables and are therefore already materialized. IBM DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase ASE all search for left-deep trees using a dynamic programming approach [7].

60.2.5 Histograms

Whether choosing a logical query plan or constructing a physical query plan from a logical query plan, the query evaluation engine needs to have information about the expected cost of evaluating the expressions in a query. Cost estimation is based on statistics about the database which include number of tuples in a relation, number of disk blocks used, distribution of values for each attribute, etc. Frequent computation of statistics, especially in light of many changes to the database, lead to more accurate cost estimates. However, the drawback is increased overhead since counting tuples and values is expensive.

An important data structure for cost estimation is the *histogram*, which is used by the DBMS to *approximate* the distribution of values for a given attribute. Note that in all but the smallest databases, counting the exact occurrence of values is usually not an option. Having access to accurate distributions is essential in determining how many tuples satisfy a certain selection predicate, for example, how many students there are with a GPA value of 3.5. This is especially important in the case of joins, which are among the most expensive operations. For example, if a value of the join attribute appears in the histograms for both relations, we can determine exactly how many tuples of the result will have this value.

Using a histogram, the data distribution is approximated by dividing the range of values, for example, GPA values, into subranges or *buckets*. Each bucket contains the number of tuples in the relation with GPA values within that bucket. Histograms are more accurate than assuming a uniform distribution across all values.

Depending on how one divides the range of values into the buckets, we distinguish between *equiwidth* and *equidepth* histograms [7]. In equiwidth histograms, the value range is divided into buckets of equal size. In equidepth histograms, the value range is divided so that the number of tuples in each bucket is the same (usually within a small delta). In both cases, each bucket contains the average frequency. When the number of buckets gets large, histograms can be compressed, for example, by combining buckets with similar distributions.

Consider the **Students-Enrollments** scenario from above. [Figure 60.7](#) depicts two sample histograms for attribute GPA in relation **Student**. Values along the horizontal axis denote GPA, the vertical bars indicate the number of students that fall in each range. For this

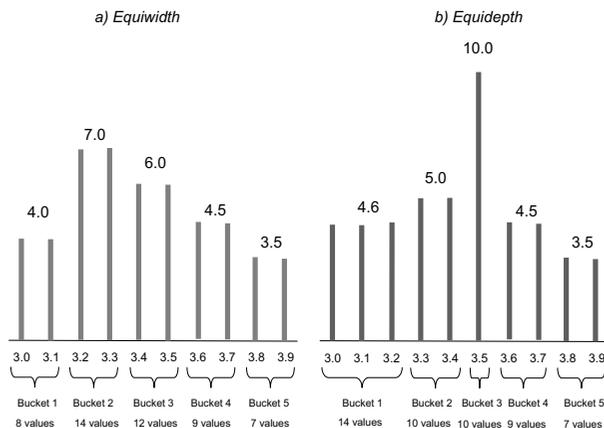


FIGURE 60.7: Two sample histograms approximating the distribution of GPA values in relation *Student*. Adapted from Ramakrishnan and Gehrke [7].

example we are assuming that GPA values are rounded to one decimal and that there are 50 students total. Histogram a) is an equiwidth histogram with bucket size = 2. Histogram b) is an equidepth histogram containing between 7 and 10 students per bucket.

Consider the selection $GPA = 3.5$. Using the equidepth histogram, we are led to bucket 3, which contains only the GPA value 3.5 and we arrive at the correct answer, 10 (vs. $1/2$ of $12 = 6$ in the equiwidth histogram). In general, equidepth histograms provide better estimates than equiwidth histograms. This is due to the fact that buckets with very frequently occurring values contain fewer values. Thus the uniform distribution assumption is applied to a smaller range of values, leading to a more accurate estimate. The converse is true for buckets containing infrequent values, which are better approximated by equiwidth histograms. However, in query optimization, good estimation for frequent values are more important.

Histograms are used by the query optimizers of all of the major DBMS vendors. For example, Sybase ASE, IBM DB2, Informix, and Oracle all use one-dimensional, equidepth histograms. Microsoft's SQL Server uses one-dimensional *equiarea histograms* (a combination of equiwidth and equidepth) [7].

60.3 Data Structures for Buffer Management

A *buffer* is partitioned into an array of *frames* each of which can keep a *page*. Usually a page of a buffer is mapped to a *block** of a file so that reading and writing of a page only require one disk access each. Application programs and queries make requests on the buffer manager when they need a block from disk, that contains a data item of interest. If the

*A block is a contiguous sequence of bytes and represents the unit used for both storage allocation and data transfer. It is usually a multiple of 512 Bytes and has a typical size of 1KB to 8KB. It may contain several data items. Usually, a data item does not span two or more blocks.

block is already in the buffer, the buffer manager conveys the address of the block in main memory to the requester. If the block is not in main memory, the buffer manager first allocates space in the buffer for the block, throwing out some other block if necessary, to make space for the new block. The displaced block is written back to disk if it was modified since the most recent time that it was written to the disk. Then, the buffer manager reads in the requested block from disk into the free frame of the buffer and passes the page address in main memory to the requester. A major goal of buffer management is to minimize the number of block transfers between the disk and the buffer.

Besides pages, so-called *segments* are provided as a counterpart of files in main memory. This allows one to define different segment types with additional attributes, which support varying requirements concerning data processing. A segment is organized as a contiguous subarea of the buffer in a virtual, linear address space with visible page borders. Thus, it consists of an ordered sequence of pages. Data items are managed so that page borders are respected. If a data item is required, the address of the page in the buffer containing the item is returned.

An important question now is how segments are mapped to files. An appropriate mapping enables the storage system to preserve the merits of the file concept. The distribution of a segment over several files turns out to be unfavorable in the same way as the representation of a data item over several pages. Hence, a segment S_k is assigned to exactly one file F_j , and m segments can be stored in a file. Since block size and page size are the same, each page $P_{k_i} \in S_k$ is assigned to a block $B_{j_l} \in F_j$. We distinguish four methods of realizing this mapping.

The *direct page addressing* assumes an implicitly given mapping between the pages of a segment S_k and the blocks of a file F_j . The page P_{k_i} ($1 \leq i \leq s_k$) is stored in the block B_{j_l} ($1 \leq l \leq d_j$) so that $l = K_j - 1 + i$ and $d_j \geq K_j - 1 + s_k$ holds. K_j denotes the number of the first block reserved for S_k (Figure 60.8). Frequently, we have a restriction to a 1:1-mapping, i.e., $K_j = 1$ and $s_k = d_j$ hold. Only in this case, a dynamic extension of segments is possible. A drawback is that at the time of the segment creation the assigned file area has to be allocated so that a block is occupied for each empty page. For segments whose data stock grows slowly, the fixed block allocation leads to a low storage utilization.

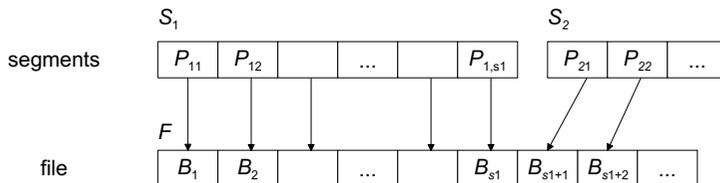


FIGURE 60.8: Direct page addressing.

The *indirect page addressing* offers a much larger flexibility for the allocation of pages to blocks and, in addition, dynamic update and extension functionality (Figure 60.9). It requires two auxiliary data structures.

- Each segment S_k is associated with a *page table* T_k which for each page of the segment contains an entry indicating the block currently assigned to the page. Empty pages obtain a special null value in the page table.
- Each file F_j is associated with a *bit table* M_j which serves for free disk space

management and quotes for each block whether currently a page is mapped to it or not. $M_j(l) = 1$ means that block B_{j_l} is occupied; $M_j(l) = 0$ says that block B_{j_l} is free. Hence, the bit table enables a dynamic assignment between pages and blocks.

Although this concept leads to an improved storage utilization, for large segments and files, the page tables and bit tables have to be split because of their size, transferred into main memory and managed in a special buffer. The provision of a page P_{k_i} that is not in the buffer can require two physical block accesses (and two enforced page removals), because, if necessary, the page table T_k has to be loaded first in order to find the current block address $j = T_k(i)$.

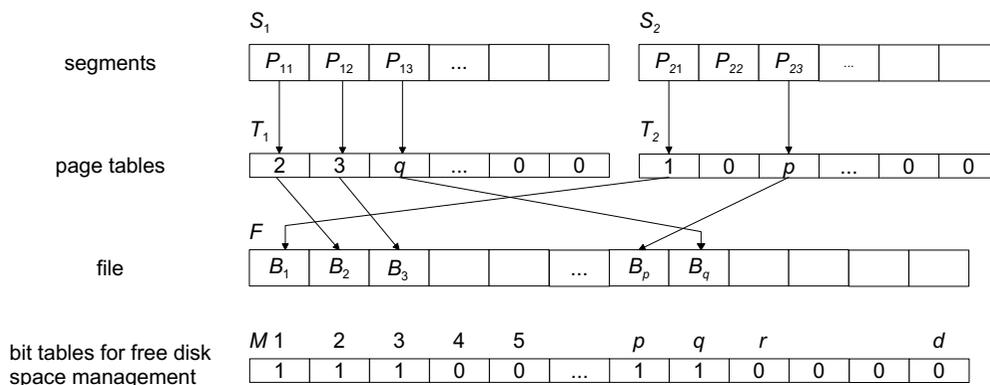


FIGURE 60.9: Indirect page addressing.

The two methods described so far assume that a modified page is written back to the block that has once been assigned to it (*update in place*). If an error occurs within a transaction, as a result of the direct placement of updates, the recovery manager must provide enough log information (*undo* information) to be able to restore the old state of a page. Since the writing of large volumes of log data leads to notable effort, it is often beneficial to perform updates in a page in a manner so that the old state of the page is available until the end of the transaction. The following two methods are based on an indirect update of changes and provide extensive support for recovery.

The *twin slot method* can be regarded as a modification of the direct page addressing. It causes very low costs for recovery but partially compensates this advantage through double disk space utilization. For a page P_{k_i} of a segment S_k , two physically consecutive blocks $B_{j_{i-1}}$ and B_{j_i} of a file F_j with $l = K_j - 1 + 2 \cdot i$ are allocated. Alternately, at the beginning of a transaction, one of both block keeps the current state of the page whereas changes are written to the other block. In case of a page request, both blocks are read, and the block with the more recent state is provided as the current page in the buffer. The block with the older state then stores the changed page. By means of page locks, a transaction-oriented recovery concept can be realized without explicitly managing log data.

The *shadow paging concept* (Figure 60.10) represents an extension of the indirect page addressing method and also supports indirect updates of changes. Before the beginning of a

new *save-interval* given by two *save-points*[†] the contents of all current pages of a segment are duplicated as so-called *shadow pages* and can thus be kept unchanged and consistent. This means, when a new save-point is created, all data structures belonging to the representation of a segment S_k (i.e., all occupied pages, the page table T_k , the bit table M) are stored as a consistent snapshot of the segment on disk. All modifications during a save-interval are performed on copies T'_k and M' of T_k and M . Changed pages are not written back to their original but to free blocks. At the creation of a new save-point, which must be an atomic operation, the tables T'_k and M' as well as all pages that belong to this state and have been changed are written back to disk. Further, all those blocks are released whose pages were subject to changes during the last save-interval. This just concerns those shadow pages for which a more recent version exists. At the beginning of the next save-interval the current contents of T'_k and M' has to be copied again to T_k and M . In case of an error within a save-interval, the DBMS can roll back to the previous consistent state represented by T_k and M .

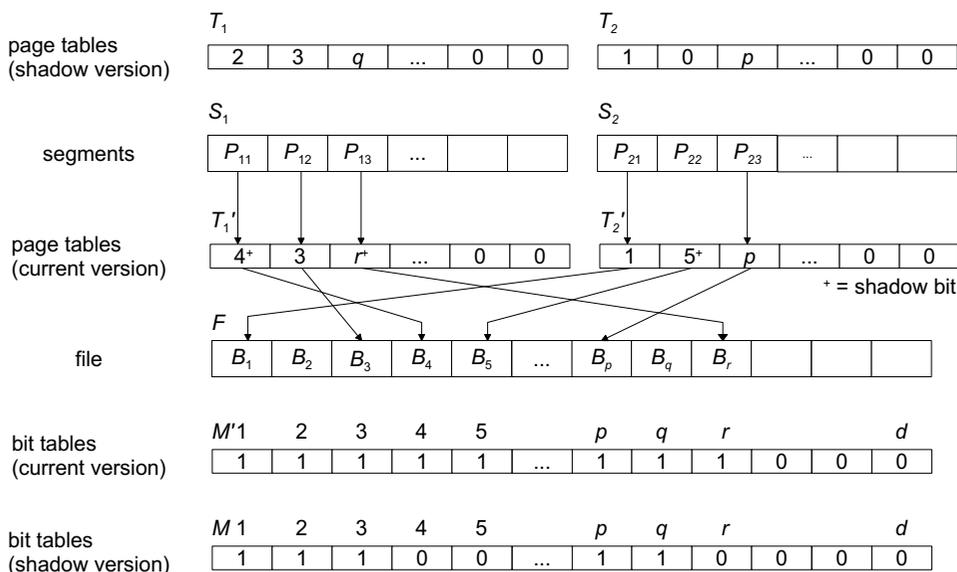


FIGURE 60.10: The shadow paging concept (segments S_1 and S_2 currently in process).

As an example, Figure 60.10 shows several changes of pages in two segments S_1 and S_2 . These changes are marked by so-called *shadow bits* in the page tables. Shadow bits are employed for the release of shadow pages at the creation time of new save-points. If a segment consists of s pages, the pertaining file must allocate s further blocks, because each changed page occupies two blocks within a save-interval.

The save-points orientate themselves to segments and not to transaction borders. Hence,

[†]Transactions are usually considered as being atomic. But a limited concept of “subtransactions” allows one to establish intermediate *save-points* while the transaction is executing, and subsequently to roll back to a previously established save-point, if required, instead of having to roll back all the way to the beginning. Note that updates made at save-points are invisible to other transactions.

in an error case, a *segment-oriented recovery* is executed. For a *transaction-oriented recovery* additional log data have to be collected.

60.4 Data Structures for Disk Space Management

Placing data items on disc is usually performed at different logical granularities. The most basic items found in relational or object-oriented database systems are the values of attributes. They consist of one or several bytes and are represented by *fields*. Fields, in turn, are put together in collections called *records*, which correspond to tuples or objects. Records need to be stored in physical *blocks* (see Section 60.3). A collection of records that forms a relation or the extent of a class is stored in some useful way as a collection of blocks, called a *file*.

60.4.1 Record Organizations

A collection of field names and their corresponding data types constitute a *record format* or *record type*. The data type of a field is usually one of the standard data types (e.g., *integer, float, bool, date, time*). If all records in a file have the same size in bytes, we call them *fixed-length records*. The fields of a record all have a fixed length and are stored consecutively. If the *base address*, i.e., the start position, of a record is given, the address of a specific field can be calculated as the sum of the lengths of the preceding fields. The sum assigned to each field is called the *offset* of this field. Record and field information are stored in the data dictionary. Figure 60.11 illustrates this record organization.

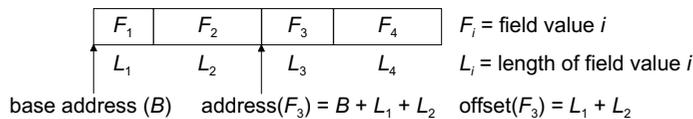


FIGURE 60.11: Organization of records with fields of fixed length.

Fixed-length records are easy to manage and allow the use of efficient search methods. But this implies that all fields have a size so that all data items that potentially are to be stored may find space. This can lead to a waste of disk space and to more unfavorable access times.

If we assume that each record of a file has the same, fixed number of fields, a *variable-length record* can only be formed if some fields have a variable length. For example, a string representing the name of an employee can have a varying length in different records. Different data structures exist for implementing variable-length records. A first possible organization amounts to a consecutive sequence of fields which are interrupted by separators (such as ? or % or \$). *Separators* are special symbols that do not occur in data items. A special *terminator* symbol indicates the end of the record. But this organization requires a pass (scan) of the record to be able to find a field of interest (Figure 60.12). Instead of separators, each field of variable length can also start with a counter that specifies the needed number of bytes of a field value.

Another alternative is that a header precedes the record. A *header* represents the “administrative” part of the record and can include information about integer offsets of the

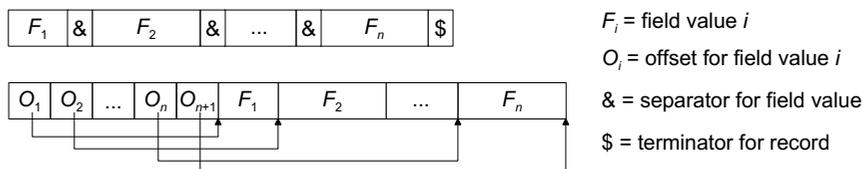


FIGURE 60.12: Alternative organizations of records with fields of variable length.

beginnings of the field values (Figure 60.12). The i th integer number is then the start address of the i th field value relatively to the beginning of the record. Also for the end of the record we must store an offset in order to know the end of the last field value. This alternative is usually the better one. Costs arise due to the header in terms of storage; the benefit is direct field access. Problems arise with changes. An update can let a field value grow which necessitates a “shift” of all consecutive fields. Besides, it can happen that a modified record does not fit any more on the page assigned to it and has to be moved to another page. If record identifiers contain a page number, on this page the new page number has to be left behind pointing to the new location of the record.

A further problem of variable-length records arises if such a record grows to such an extent that it does not fit on a page any more. For example, field values storing image data in various formats (e.g., GIF or JPEG), movies in formats such as MPEG, or spatial objects such as polygons can extend from the order of many kilobytes to the order of many megabytes or even gigabytes. Such truly large values for records or field values of records are called *large objects (lobs)* with the distinction of *binary large objects (blobs)* for large byte sequences and *character large objects!character (clobs)* for large strings.

Since, in general, lobes exceed page borders, only the non-lob fields are stored on the original page. Different data structures are conceivable for representing lobes. They all have in common that a lobe is subdivided into a collection of linked pages. This organization is also called *spanned*, because records can span more than one page, in contrast to the *unspanned* organization where records are not allowed to cross page borders. The first alternative is to keep a pointer instead of the lobe on the original page as attribute value. This pointer (also called *page reference*) points to the start of a linked page or block list keeping the lobe (Figure 60.13(a)). Insertions, deletions, and modifications are simple but direct access to pages is impossible. The second alternative is to store a *lob directory* as attribute value (Figure 60.13(b)). Instead of a pointer, a directory is stored which includes the lobe size, further administrative data, and a *page reference list* pointing to the single pages or blocks on a disk. The main benefit of this structure is the direct and sequential access to pages. The main drawback is the fixed and limited size of the lobe directory and thus the lobe. A lobe directory can grow so much that it needs itself a lobe for its storage.

The third alternative is the usage of *positional B⁺-trees* (Figure 60.14). Such a B-tree variant stores relative byte positions in its inner nodes as separators. Its leaf nodes keep the actual data pages of the lobe. The original page only stores as the field value a pointer to the root of the tree.

60.4.2 Page Organizations

Records are positioned on pages (or blocks). In order to reference a record, often a *pointer* to it suffices. Due to different requirements for storing records, the structure of pointers can vary. The most obvious pointer type is the *physical address* of a record on disk or in a virtual storage and can easily be used to compute the page to be read. The main advantage

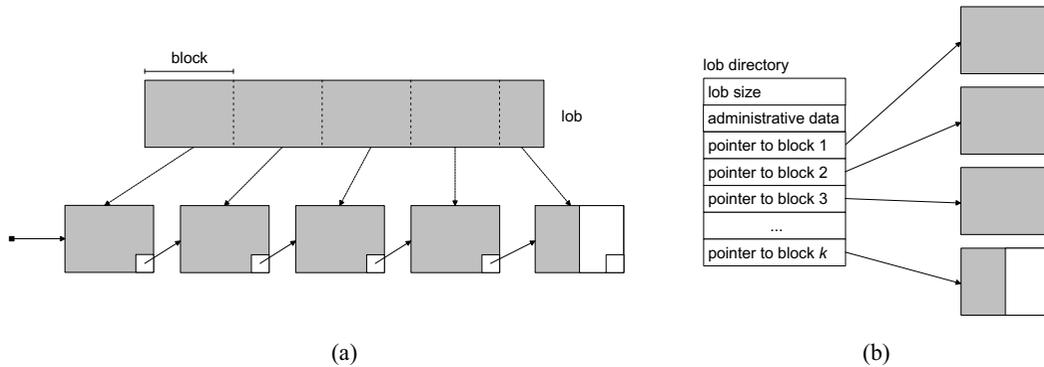


FIGURE 60.13: A lob as a linked list of pages (a), and the use of a lob directory (b).

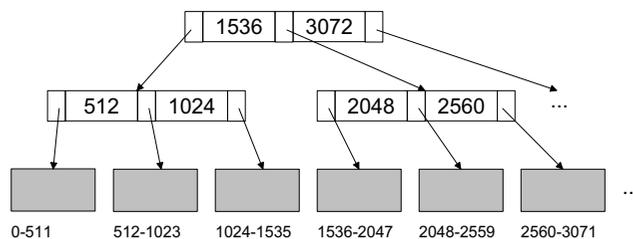


FIGURE 60.14: A lob managed by a positional B⁺-tree.

is a direct access to the searched record. But it is impossible to move a record within a page, because this requires the locating and changing of all pointers to this record. We call these pointers *physical pointers*. Due to this drawback, a pointer is often described as a pair (p, n) where p is the number of the page where the record can be found and where n is a number indicating the location of the record on the page. The parameter n can be interpreted differently, e.g., as a relative byte address on a page, as a number of a slot, or as an index of a directory in the *page header*. The entry at this index position yields the relative position of the record on the page. All pointers (s, p) remain unchanged and are named *page-related pointers*. Pointers that are completely stable against movements in main memory can be achieved if a record is associated with a *logical address* that reveals nothing about its storage. The record can be moved freely in a file without changing any pointers. This can be realized by *indirect addressing*. If a record is moved, only the respective entry in a *translation table* has to be changed. All pointers remain unchanged, and we call them *logical pointers*. The main drawback is that each access to a record needs an additional access to the translation table. Further, the table can become so large that it does not fit completely in main memory.

A page can be considered as a collection of *slots*. Each slot can capture exactly one record. If all records have the same length, all slots have the same size and can be allocated consecutively on the page. Hence, a page contains so many records as slots fit on a page plus page information like directories and pointers to other pages. A first alternative for arranging a set of N fixed-length records is to place them in the first N slots (see Figure 60.15). If a record is deleted in slot $i < N$, the last record on the page in slot N is moved to the

free slot i . However, this causes problems if the record to be moved is pinned[‡] and the slot number has to be changed. Hence, this “packed” organization is problematic, although it allows one to easily compute the location of the i th record. A second alternative is to manage deletions of records on each page and thus information about free slots by means of a directory represented as a bitmap. The retrieval of the i th record as well as finding the next free slot on a page require a traversal of the directory. The search for the next free slot can be sped up if an additional, special field stores a pointer on the first slot whose deletion flag is set. The slot itself then contains a pointer to the next free slot so that a chaining of free slots is achieved.

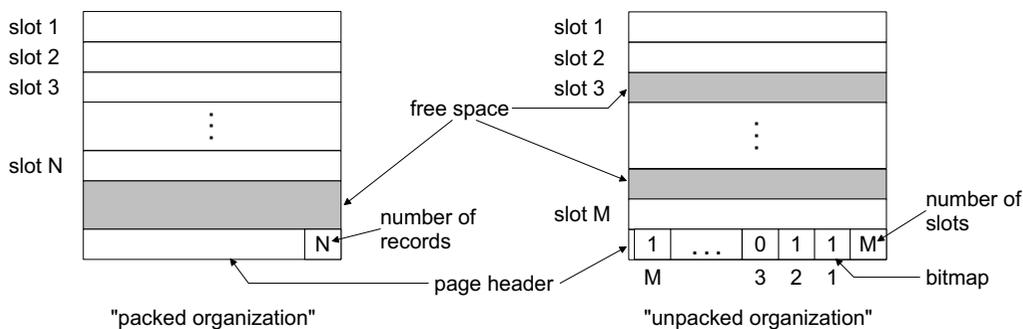


FIGURE 60.15: Alternative page organizations for fixed-length records.

Also variable-length records can be positioned consecutively on a page. But deletions of records must be handled differently now because slots cannot be reused in a simple manner any more. If a new record is to be inserted, first a free slot of “the right size” has to be found. If the slot is too large, storage is wasted. If it is too small, it cannot be used. In any case, unused storage areas (*fragmentation*) at the end of slots can only be avoided if records on a page are moved and condensed. This leads to a connected, free storage area. If the records of a page are unpinned, the “packed” representation for fixed-length records can be adapted. Either a special terminator symbol marks the end of the record, or a field at the beginning of the record keeps its length. In the general case, indirect addressing is needed which permits record movements without negative effects and without further access costs. The most flexible organization of variable-length records is provided by the *tuple identifier (TID) concept* (Figure 60.16). Each record is assigned a unique, stable pointer consisting of a page number and an index into a page-internal directory. The entry at index i contains the relative position, i.e., the offset, of slot i and hence a pointer to record i on the page. The length information of a record is stored either in the directory entry or at the beginning of the slot (L_i in Figure 60.16). Records which grow or shrink can be moved on the page without being forced to modify their TIDs. If a record is deleted, this is registered in the corresponding directory entry by means of a deletion flag.

Since a page cannot be subdivided into predefined slots, some kind of free disk space management is needed on each page. A pointer to the beginning of the free storage space on the page can be kept in the page header. If a record does not fit into the currently

[‡]If pointers of unknown origin reference a record, we call the record *pinned*, otherwise *unpinned*.

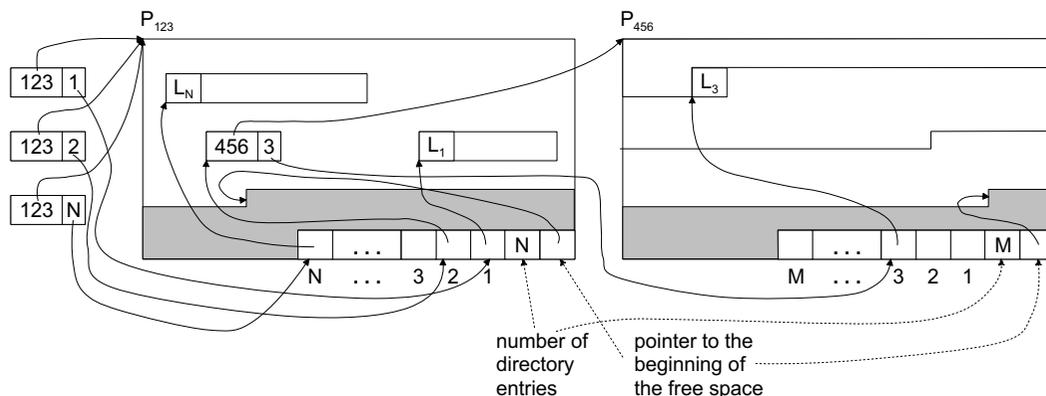


FIGURE 60.16: Page organization for variable-length records.

available free disk space, the page is compressed (i.e., defragmented) and all records are placed consecutively without gaps. The effect is that the maximally available free space is obtained and is located after the record representations.

If, despite defragmentation, a record does still not fit into the available free space, the record must be moved from its “home page” to an “overflow page”. The respective TID can be kept stable by storing a “proxy TID” instead of the record on the home page. This proxy TID points to the record having been moved to the overflow page. An overflow record is not allowed to be moved to another, second overflow page. If an overflow record has to leave its overflow page, its placement on the home page is attempted. If this fails due to a lack of space, a new overflow page is determined and the overflow pointer is placed on the home page. This procedure assures that each record can be retrieved with a maximum of two page accesses.

If a record is deleted, we can only replace the corresponding entry in the directory by a deletion flag. But we cannot compress the directory since the indexes of the directory are used to identify records. If we deleted an entry and compress, the indexes of the subsequent slots in the directory would be decremented so that TIDs would point to wrong slots and thus wrong records. If a new record is inserted, the first entry of the directory containing a deletion flag is selected for determining the new TID and pointing to the new record.

If a record represents a large object, i.e., it does not fit on a single page but requires a collection of linked pages, the different data structures for blobs can be employed.

60.4.3 File Organization

A *file (segment)* can be viewed as a sequence of *blocks (pages)*. Four fundamental file organizations can be distinguished, namely files of unordered records (heap files), files of ordered records (sorted files), files with dispersed records (hash files), and tree-based files (index structures).

Heap files are the simplest file organization. Records are inserted and stored in their unordered, chronological sequence. For each heap file we have to manage their assigned pages (blocks) to support scans as well as the pages containing free space to perform insertions efficiently. Doubly-linked lists of pages or directories of pages using both page numbers for page addressing are possible alternatives. For the first alternative, the DBMS uses a *header page* which is the first page of a heap file, contains the address of the first data page, and information about available free space on the pages. For the second alternative, the DBMS

must keep the first page of the heap file in mind. The directory itself represents a collection of pages and can be organized as a linked list. Each directory entry points to a page of the heap file. The free space on each page is recorded by a counter associated with each directory entry. If a record is to be inserted, its length can be compared to the number of free bytes on a page.

Sorted files physically order their records based on the values of one (or several) of their fields, called the *ordering field(s)*. If the ordering field is also a *key field* of the file, i.e., a field guaranteed to have a unique value in each record, then the field is called the *ordering key* for the file. If all records have the same fixed length, binary search on the ordering key can be employed resulting in faster access to records.

Hash files are a file organization based on hashing and representing an important indexing technique. They provide very fast access to records on certain search conditions. Internal hashing techniques have been discussed in different chapters of this book; here we are dealing with their external variants and will only explain their essential features. The fundamental idea of hash files is the distribution of the records of a file into so-called *buckets*, which are organized as heaps. The distribution is performed depending on the value of the *search key*. The direct assignment of a record to a bucket is computed by a *hash function*. Each bucket consists of one or several pages of records. A *bucket directory* is used for the management of the buckets, which is an array of pointers. The entry for index i points to the first page of bucket i . All pages for bucket i are organized as a linked list. If a record has to be inserted into a bucket, this is usually done on its last page since only there space can be found. Hence, a pointer to the last page of a bucket is used to accelerate the access to this page and to avoid traversing all the pages of the bucket. If there is no space left on the last page, overflow pages are provided. This is called a *static hash file*. Unfortunately, this strategy can cause long chains of overflow pages. *Dynamic hash files* deal with this problem by allowing a variable number of buckets. *Extensible hash files* employ a directory structure in order to support insertion and deletion efficiently without the employment of overflow pages. *Linear hash files* apply an intelligent strategy to create new buckets. Insertion and deletion are efficiently realized without using a directory structure.

Index structures are a fundamental and predominantly tree-based file organization based on the search key property of values and aiming at speeding up the access to records. They have a paramount importance in query processing. Many examples of index structures are already described in detail in this handbook, e.g., B-trees and variants, quad-trees and oct-trees, R-trees and variants, and other multidimensional data structures. We will not discuss them further here. Instead, we mention some basic and general organization forms for index structures that can also be combined. An index structure is called a *primary organization* if it contains search key information together with an embedding of the respective records; it is named a *secondary organization* if it includes besides search key information only TIDs or TID lists to records in separate file structures (e.g., heap files or sorted files). An index is called a *dense index* if it contains (at least) one index entry for each search key value which is part of a record of the indexed file; it is named a *sparse index* (Figure 60.17) if it only contains an entry for each page of records of the indexed file. An index is called a *clustered index* (Figure 60.17) if the logical order of records is equal or almost equal to their physical order, i.e., records belonging logically together are physically stored on neighbored pages. Otherwise, the index is named *non-clustered*. An index is called a *one-dimensional index* if a linear order is defined on the set of search key values used for organizing the index entries. Such an order cannot be imposed on a *multi-dimensional index* where the organization of index entries is based on spatial relationships. An index is called a *single-level index* if the index only consists of a single file; otherwise, if the index is composed of several files, it is named a *multi-level index*.

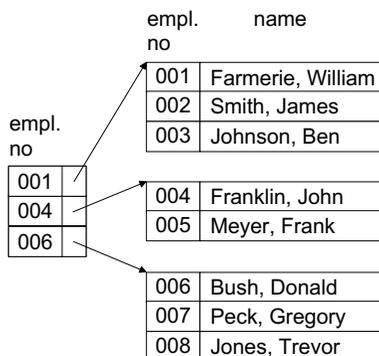


FIGURE 60.17: Example of a clustered, sparse index as a secondary organization on a sorted file.

60.5 Conclusion

A modern database management system is a complex software system that leverages many sophisticated algorithms, for example, to evaluate relational operations, to provide efficient access to data, to manage the buffer pool, and to move data between disk and main memory. In this chapter, we have shown how many of the data structures that were introduced in earlier parts of this book (e.g., B-trees, buffer trees, quad trees, R-trees, interval trees, hashing) including a few new ones such as histograms, LOBs, and disk pages, are being used in a real-world application. However, as we have noted in the introduction, our coverage of the data structures that are part of a DBMS is not meant to be exhaustive since a complete treatment would have easily exceeded the scope of this chapter. Furthermore, as the functionality of a DBMS must continuously grow in order to support new applications (e.g., GIS, federated databases, data mining), so does the set of data structures that must be designed to efficiently manage the underlying data (e.g., spatio-temporal data, XML, bio-medical data). Many of these new data structure challenges are being actively studied in the database research communities today and are likely to form a basis for tomorrow's systems.

References

- [1] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
- [2] Chris J. Date and Hugh Darwen. *A Guide to The SQL Standard*. Addison-Wesley Publishing Company, Inc., third edition, 1997.
- [3] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, fourth edition, 2003.
- [4] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems - The Complete Book*. Prentice Hall, Upper Saddle River, New Jersey, first edition, 2002.
- [5] Philip M. Lewis, Arthur Bernstein, and Michael Kifer. *Databases and Transaction Processing*. Addison Wesley, first edition, 2002.
- [6] Patrick O'Neil and Elizabeth O'Neil. *Database: Principles, Programming, and Performance*. Morgan Kaufmann, second edition, 2000.
- [7] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, third edition, 2003.

- [8] Abraham Silberschatz, Henry F. Korth, and S. Sudharshan. *Database System Concepts*. McGraw-Hill, fourth edition, 2002.

Vipin Kumar
University of Minnesota

Pang-Ning Tan
Michigan State University

Michael Steinbach
University of Minnesota

61.1	Introduction.....	61-1
	Data Mining Tasks and Techniques • Challenges of Data Mining • Data Mining and the Role of Data Structures and Algorithms	
61.2	Classification.....	61-6
	Nearest-Neighbor Classifiers • Proximity Graphs for Enhancing Nearest Neighbor Classifiers	
61.3	Association Analysis.....	61-8
	Hash Tree Structure • FP-Tree Structure	
61.4	Clustering.....	61-15
	Hierarchical and Partitional Clustering • Nearest Neighbor Search and Multi-Dimensional Access Methods	
61.5	Conclusion.....	61-19

61.1 Introduction

Recent years have witnessed an explosive growth in the amounts of data collected, stored, and disseminated by various organizations. Examples include (1) the large volumes of point-of-sale data amassed at the checkout counters of grocery stores, (2) the continuous streams of satellite images produced by Earth-observing satellites, and (3) the avalanche of data logged by network monitoring software. To illustrate how much the quantity of data has grown over the years, [Figure 61.1](#) shows an example of the number of Web pages indexed by a popular Internet search engine since 1998.

In each of the domains described above, data is collected to satisfy the information needs of the various organizations: Commercial enterprises analyze point-of-sale data to learn the purchase behavior of their customers; Earth scientists use satellite image data to advance their understanding of how the Earth system is changing in response to natural and human-related factors; and system administrators employ network traffic data to detect potential network problems, including those resulting from cyber-attacks.

One immediate difficulty encountered in these domains is how to extract useful information from massive data sets. Indeed, getting information out of the data is like *drinking from a fire hose*. The sheer size of the data simply overwhelms our ability to manually sift through the data, hoping to find useful information. Fueled by the need to rapidly analyze and summarize the data, researchers have turned to *data mining* techniques [22, 27, 29, 30, 50]. In a nutshell, data mining is the task of *discovering interesting knowledge automatically from large data repositories*.

Interesting knowledge has different meanings to different people. From a business perspective, knowledge is interesting if it can be used by analysts or managers to make profitable

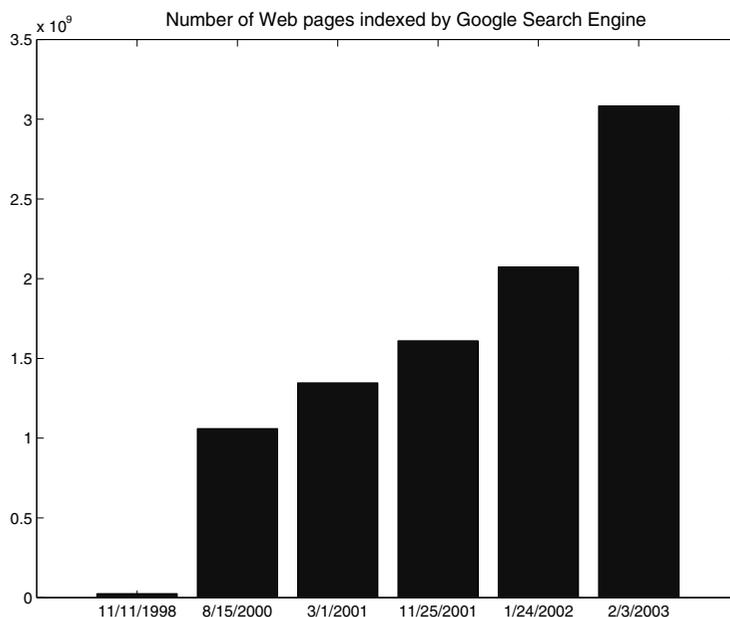


FIGURE 61.1: Number of Web pages indexed by the Google[©] search engine (Source: Internet Archive, <http://www.archive.org>).

business decisions. For Earth Scientists, knowledge is interesting if it reveals previously unknown information about the characteristics of the Earth system. For system administrators, knowledge is interesting if it indicates unauthorized or illegitimate use of system resources.

Data mining is often considered to be an integral part of another process, called *Knowledge Discovery in Databases* (or KDD). KDD refers to the overall process of turning raw data into interesting knowledge and consists of a series of transformation steps, including data preprocessing, data mining, and postprocessing. The objective of data preprocessing is to convert data into the right format for subsequent analysis by selecting the appropriate data segments and extracting attributes that are relevant to the data mining task (feature selection and construction). For many practical applications, more than half of the knowledge discovery efforts are devoted to data preprocessing. Postprocessing includes all additional operations performed to make the data mining results more accessible and easier to interpret. For example, the results can be sorted or filtered according to various *measures* to remove uninteresting patterns. In addition, *visualization* techniques can be applied to help analysts explore data mining results.

61.1.1 Data Mining Tasks and Techniques

Data mining tasks are often divided into two major categories:

Predictive The goal of predictive tasks is to use the values of some variables to predict the values of other variables. For example, in Web mining, e-tailers are interested in predicting which online users will make a purchase at their Web site. Other examples include biologists, who would like to predict the functions of proteins, and stock market analysts, who would like to forecast the future prices of various stocks.

Descriptive The goal of descriptive tasks is to find human-interpretable patterns that describe the underlying relationships in the data. For example, Earth Scientists are interested in discovering the primary forcings influencing observed climate patterns. In network intrusion detection, analysts want to know the kinds of cyber-attacks being launched against their networks. In document analysis, it is useful to find groups of documents, where the documents in each group share a common topic.

Data mining tasks can be accomplished using a variety of data mining techniques, as shown in Figure 61.2.

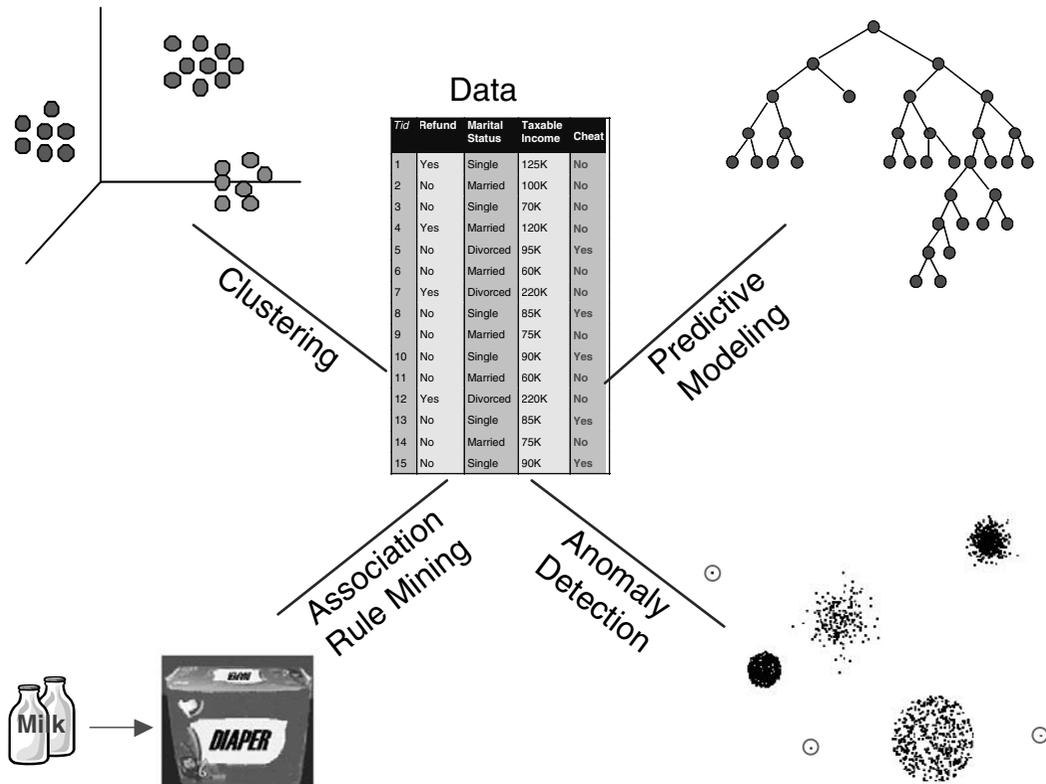


FIGURE 61.2: Data mining techniques.

- Predictive modeling** is used primarily for predictive data mining tasks. The input data for predictive modeling consists of two distinct types of variables: (1) explanatory variables, which define the essential properties of the data, and (2) one or more target variables, whose values are to be predicted. For the Web mining example given in the previous section, the input variables correspond to the demographic features of online users, such as age, gender, and salary, along with their browsing activities, e.g., what pages are accessed and for how long. There is one binary target variable, Buy, which has values, Yes or No,

indicating, respectively, whether the user will buy anything from the Web site or not. Predictive modeling techniques can be further divided into two categories: *classification* and *regression*. Classification techniques are used to predict the values of discrete target variables, such as the **Buy** variable for online users at a Web site. For example, they can be used to predict whether a customer will most likely be lost to a competitor, i.e., customer churn or attrition, and to determine the category of a star or galaxy for sky survey cataloging. Regression techniques are used to predict the values of continuous target variables, e.g., they can be applied to forecast the future price of a stock.

- **Association rule mining** seeks to produce a set of dependence rules that predict the occurrence of a variable given the occurrences of other variables. For example, association analysis can be used to identify products that are often purchased together by sufficiently many customers, a task that is also known as *market basket analysis*. Furthermore, given a database that records a sequence of events, e.g., a sequence of successive purchases by customers, an important task is that of finding dependence rules that capture the temporal connections of events. This task is known as *sequential pattern analysis*.
- **Cluster analysis** finds groupings of data points so that data points that belong to one cluster are more similar to each other than to data points belonging to a different cluster, e.g., clustering can be used to perform market segmentation of customers, document categorization, or land segmentation according to vegetation cover. While cluster analysis is often used to better understand or describe the data, it is also useful for summarizing a large data set. In this case, the objects belonging to a single cluster are replaced by a single *representative* object, and further data analysis is then performed using this reduced set of representative objects.
- **Anomaly detection** identifies data points that are significantly different than the rest of the points in the data set. Thus, anomaly detection techniques have been used to detect network intrusions and to predict fraudulent credit card transactions. Some approaches to anomaly detection are statistically based, while other are based on distance or graph-theoretic notions.

61.1.2 Challenges of Data Mining

There are several important challenges in applying data mining techniques to large data sets:

Scalability Scalable techniques are needed to handle the massive size of some of the datasets that are now being created. As an example, such datasets typically require the use of efficient methods for storing, indexing, and retrieving data from secondary or even tertiary storage systems. Furthermore, parallel or distributed computing approaches are often necessary if the desired data mining task is to be performed in a timely manner. While such techniques can dramatically increase the size of the datasets that can be handled, they often require the design of new algorithms and data structures.

Dimensionality In some application domains, the number of dimensions (or attributes of a record) can be very large, which makes the data difficult to analyze because of the ‘curse of dimensionality’ [9]. For example, in bioinformatics, the development of advanced microarray technologies allows us to analyze gene

expression data with thousands of attributes. The dimensionality of a data mining problem may also increase substantially due to the temporal, spatial, and sequential nature of the data.

Complex Data Traditional statistical methods often deal with simple data types such as continuous and categorical attributes. However, in recent years, more complicated types of structured and semi-structured data have become more important. One example of such data is graph-based data representing the linkages of web pages, social networks, or chemical structures. Another example is the free-form text that is found on most web pages. Traditional data analysis techniques often need to be modified to handle the complex nature of such data.

Data Quality Many data sets have one or more problems with data quality, e.g., some values may be erroneous or inexact, or there may be missing values. As a result, even if a 'perfect' data mining algorithm is used to analyze the data, the information discovered may still be incorrect. Hence, there is a need for data mining techniques that can perform well when the data quality is less than perfect.

Data Ownership and Distribution For a variety of reasons, e.g., privacy and ownership, some collections of data are distributed across a number of sites. In many such cases, the data cannot be centralized, and thus, the choice is either distributed data mining or no data mining. Challenges involved in developing distributed data mining solutions include the need for efficient algorithms to cope with distributed and heterogeneous data sets, the need to minimize the cost of communication, and the need to accommodate data security and data ownership policies.

61.1.3 Data Mining and the Role of Data Structures and Algorithms

Research in data mining is motivated by a number of factors. In some cases, the goal is to develop an approach with greater efficiency. For example, a current technique may work well as long as all of the data can be held in main memory, but the size of data sets has grown to the point where this is no longer possible. In other cases, the goal may be to develop an approach that is more flexible. For instance, the nature of the data may be continually changing, and it may be necessary to develop a model of the data that can also change. As an example, network traffic varies in volume and kind, often over relatively short time periods. In yet other cases, the task is to obtain a more accurate model of the data, i.e., one that takes into account additional factors that are common in many real world situations.

The development and success of new data mining techniques is heavily dependent on the creation of the proper algorithms and data structures to address the needs such as those just described: efficiency, flexibility, and more accurate models. (This is not to say that system or applications issues are unimportant.) Sometimes, currently existing data structures and algorithms can be directly applied, e.g., data access methods can be used to efficiently organize and retrieve data. However, since currently existing data structures and algorithms were typically not designed with data mining tasks in mind, it is frequently the case that some modifications, enhancements, or completely new approaches are needed, i.e., new work in data structures and algorithms is needed. We would emphasize, though, that sometimes it is the concepts and viewpoints associated with currently existing algorithms and data structures that are the most useful. Thus, the realization that a problem can be formulated as a particular type of a graph or tree may quickly lead to a solution.

In the following sections, we provide some examples of how data structures play an important role, both conceptually and practically, for classification, association analysis, and clustering.

61.2 Classification

Classification [21, 43] is the task of assigning objects to their respective categories. For example, stock analysts are interested in classifying the stocks of publicly-owned companies as `buy`, `hold`, or `sell`, based on the financial outlook of these companies. Stocks classified as `buy` are expected to have stronger future revenue growth than those classified as `sell`. In addition to the practical uses of classification, it also helps us to understand the similarities and differences between objects that belong to different categories.

The data set in a classification problem typically consists of a collection of *records* or data objects. Each record, also known as an *instance* or *example*, is characterized by a tuple (\mathbf{x}, y) , where \mathbf{x} is the set of explanatory variables associated with the object and y is the object's class label. A record is said to be *labeled* if the value of y is known; otherwise, the record is *unlabeled*. Each attribute $x_k \in \mathbf{x}$ can be discrete or continuous. On the other hand, the class label y must be a discrete variable whose value is chosen from a finite set $\{y_1, y_2, \dots, y_c\}$. If y is a continuous variable, then this problem is known as *regression*.

The classification problem can be stated formally as follows:

Classification is the task of learning a function, $f : \mathbf{x} \rightarrow y$, that maps the explanatory variables \mathbf{x} of an object to one of the class labels for y .

f is known as the *target function* or *classification model*.

61.2.1 Nearest-Neighbor Classifiers

Typically, the classification framework presented involves a two-step process: (1) an inductive step for constructing classification models from data, and (2) a deductive step for applying the derived model to previously unseen instances. For decision tree induction and rule-based learning systems, the models are constructed immediately after the training set is provided. Such techniques are known as *eager learners* because they intend to learn the model as soon as possible, once the training data is available.

An opposite strategy would be to delay the process of generalizing the training data until it is needed to classify the unseen instances. One way to do this is to find all training examples that are relatively similar to the attributes of the test instance. Such examples are known as the nearest neighbors of the test instance. The test instance can then be classified according to the class labels of its neighbors. This is the central idea behind the *nearest-neighbor classification* scheme [4, 17, 18, 21], which is useful for classifying data sets with continuous attributes. A nearest neighbor classifier represents each instance as a data point embedded in a d -dimensional space, where d is the number of continuous attributes. Given a test instance, we can compute its distance to the rest of the data objects (data points) in the training set by using an appropriate distance or similarity measure, e.g., the standard Euclidean distance measure.

The k -nearest neighbors of an instance z are defined as the data points having the k smallest distances to z . [Figure 61.3](#) illustrates an example of the 1-, 2- and 3-nearest neighbors of an unknown instance, \times , located at the center of the circle. The instance can be assigned to the class label of its nearest neighbors. If the nearest neighbors contain more

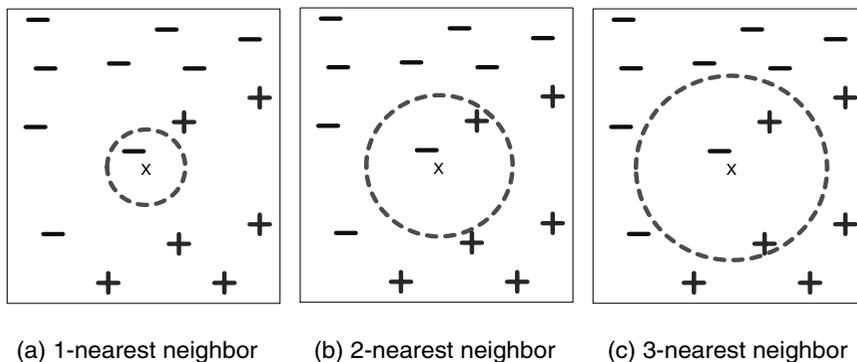


FIGURE 61.3: The 1-, 2- and 3-nearest neighbors of an instance.

than one class label, then one takes a majority vote among the class labels of the nearest neighbors.

The nearest data point to the unknown instance shown in Figure 61.3(a) has a negative class label. Thus, in a 1-nearest neighbor classification scheme, the unknown instance would be assigned to a negative class. If we consider a larger number of nearest neighbors, such as three, the list of nearest neighbors would contain training examples from 2 positive classes and 1 negative class. Using the majority voting scheme, the instance would be classified as a positive class. If the number of instances from both classes are the same, as in the case of the 2-nearest neighbor classification scheme shown in Figure 61.3(b), we could choose either one of the classes (or the default class) as the class label.

A summary of the k -nearest neighbor classification algorithm is given in Figure 61.4. Given an unlabeled instance, we need to determine its distance or similarity to all the training instances. This operation can be quite expensive and may require efficient indexing techniques to reduce the amount of computation.

(k : number of nearest neighbor, E : training instances, z : unlabeled instance)

- 1: Compute the distance or similarity of z to all the training instances
- 2: Let $E' \subset E$ be the set of k closest training instances to z
- 3: Return the predicted class label for z : $class \leftarrow Voting(E')$.

FIGURE 61.4: k -nearest neighbor classification algorithm.

While one can take a majority vote of the nearest neighbors to select the most likely class label, this approach may not be desirable because it assumes that the influence of each nearest neighbor is the same. An alternative approach is to weight the influence of each nearest neighbor according to its distance, so that the influence is weaker if the distance is too large.

61.2.2 Proximity Graphs for Enhancing Nearest Neighbor Classifiers

The nearest neighbor classification scheme, while simple, has a serious problem as currently presented: It is necessary to store all of the data points, and to compute the distance between an object to be classified and all of these stored objects. If the set of original data

points is large, then this can be a significant computational burden. Hence, a considerable amount of research has been conducted into strategies to alleviate this problem.

There are two general strategies for addressing the problem just discussed:

Condensing The idea is that we can often eliminate many of the data points without affecting classification performance, or without affecting it very much. For instance, if a data object is in the ‘middle’ of a group of other objects with the same class, then its elimination will likely have no effect on the nearest neighbor classifier.

Editing Often, the classification performance of a nearest neighbor classifier can be enhanced by deleting certain data points. More specifically, if a given object is compared to its nearest neighbors and most of them are of another class (i.e., if the points that would be used to classify the given point are of another class), then deleting the given object will often improve classifier performance.

While various approaches to condensing and editing points to improve the performance of nearest neighbor classifiers have been proposed, there has been a considerable amount of work that approaches this problem from the viewpoint of computational geometry, especially proximity graphs [49]. Proximity graphs include nearest neighbor graphs, minimum spanning trees, relative neighborhood graphs, Gabriel graphs, and the Delaunay triangulation [35]. We can only indicate briefly the usefulness of this approach, and refer the reader to [49] for an in depth discussion.

First, we consider how Voronoi diagrams can be used to eliminate points that add nothing to the classification. (The Voronoi diagram for a set of data points is the set of polygons formed by partitioning all of points in the space into a set of convex regions such that every point in a region is closer to the data point in the same region than to any other data point. [Figure 61.5](#) shows a Voronoi diagram for a number of two-dimensional points.) Specifically, if all the Voronoi neighbors of a point, i.e., those points belonging to Voronoi regions that touch the Voronoi region of the given point, have the same class as the given data point, then discarding that point cannot affect the classification performance. The reason for this is that the Voronoi regions of the neighboring points will expand to ‘occupy’ the space once occupied by the the Voronoi region of the given point, and thus, classification behavior is unchanged. More sophisticated approaches based on proximity graphs are possible [49].

For editing, i.e., discarding points to approve classification performance, proximity graphs can also be useful. In particular, instead of eliminating data points whose k nearest neighbors are of a different class, we build a proximity graph and eliminate those points where a majority of the neighbors in the proximity graph are of a different class. Of course, the results will depend on the type of proximity graph. The Gabriel graph has been found to be the best, but for further discussion, we once again refer the reader to [49], and the extensive list of references that it contains.

In summary, our goal in this section was to illustrate that—for one particular classification scheme, nearest neighbor classification—the rich set of data structures and algorithms of computational geometry, i.e., proximity graphs, have made a significant contribution, both practically and theoretically.

61.3 Association Analysis

An important problem in data mining is the discovery of association patterns [1] present in large databases. This problem was originally formulated in the context of *market basket* data, where the goal is to determine whether the occurrence of certain items in a transaction

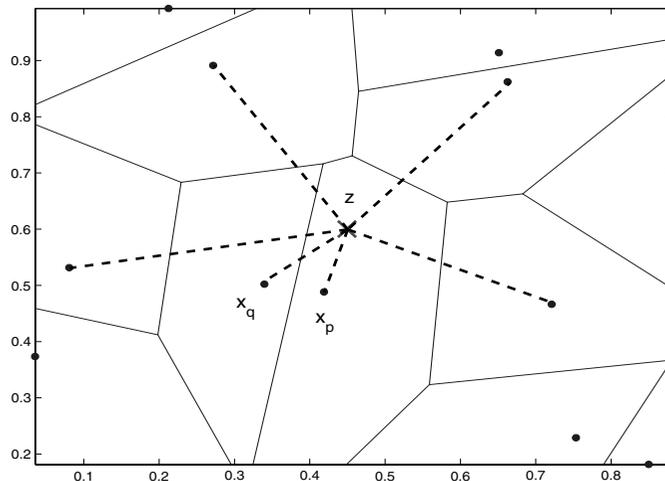


FIGURE 61.5: Voronoi diagram.

TID	Items
1	Bread, Coke, Milk
2	Beer, Bread
3	Beer, Coke, Diaper, Milk
4	Beer, Bread, Diaper, Milk
5	Coke, Diaper, Milk

FIGURE 61.6: Market-basket transactions

can be used to infer the occurrence of other items. If such interesting relationships are found, then they can be put to various profitable uses such as marketing promotions, shelf management, inventory management, etc.

To formalize the problem, let $T = \{t_1, t_2, \dots, t_N\}$ be the set of all transactions and $I = \{i_1, i_2, \dots, i_d\}$ be the set of all items. Any subset of I is known as an *itemset*. The *support count* of an itemset C is defined as the number of transactions in T that contain C , i.e.,

$$\sigma(C) = |\{t | t \in T, C \subseteq t\}|.$$

An *association rule* is an implication of the form $X \rightarrow Y$, where X and Y are itemsets and $X \cap Y = \emptyset$. The strength of an association rule is given by its *support* (s) and *confidence* (c) measures. The support of the rule is defined as the fraction of transactions in T that contain itemset $X \cup Y$.

$$s(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{|T|}.$$

Confidence, on the other hand, provides an estimate of the conditional probability of finding items of Y in transactions that contain X .

$$c(X \rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)}$$

For example, consider the market basket transactions shown in Figure 61.6. The support for the rule $\{\text{Diaper, Milk}\} \rightarrow \{\text{Beer}\}$ is $\sigma(\text{Diaper, Milk, Beer}) / 5 = 2/5 = 40\%$, whereas its confidence is $\sigma(\text{Diaper, Milk, Beer}) / \sigma(\text{Diaper, Milk}) = 2/3 = 66\%$.

Support is useful because it reflects the significance of a rule. Rules that have very low support are rarely observed, and thus, are more likely to occur by chance. Confidence

is useful because it reflects the reliability of the inference made by each rule. Given an association rule $X \longrightarrow Y$, the higher the confidence, the more likely it is to find Y in transactions that contain X . Thus, the goal of association analysis is to automatically discover association rules having relatively high support and high confidence. More specifically, an association rule is considered to be interesting only if its support is greater than or equal to a minimum support threshold, *minsup*, and its confidence is greater than or equal to a minimum confidence threshold, *minconf*.

The association analysis problem is far from trivial because of the exponential number of ways in which items can be grouped together to form a rule. In addition, the rules are constrained by two completely different conditions, stated in terms of the *minsup* and *minconf* thresholds. A standard way for generating association rules is to divide the process into two steps. The first step is to find all itemsets that satisfy the minimum support threshold. Such itemsets are known as *frequent itemsets* in the data mining literature. The second step is to generate high-confidence rules only from those itemsets found to be frequent. The completeness of this two-step approach is guaranteed by the fact that any association rule $X \longrightarrow Y$ that satisfies the *minsup* requirement can always be generated from a frequent itemset $X \cup Y$.

Frequent itemset generation is the computationally most expensive step because there are 2^d possible ways to enumerate all itemsets from I . Much research has therefore been devoted to developing efficient algorithms for this task. A key feature of these algorithms lies in their strategy for controlling the exponential complexity of enumerating candidate itemsets. Briefly, the algorithms make use of the anti-monotone property of itemset support, which states that all subsets of a frequent itemset must be frequent. Put another way, if a candidate itemset is found to be infrequent, we can immediately prune the search space spanned by supersets of this itemset. The *Apriori* algorithm, developed by Agrawal et al. [2], pioneered the use of this property to systematically enumerate the candidate itemsets. During each iteration k , it generates only those *candidate* itemsets of length k whose $(k - 1)$ -subsets are found to be frequent in the previous iteration. The support counts of these candidates are then determined by scanning the transaction database. After counting their supports, candidate k -itemsets that pass the *minsup* threshold are declared to be frequent.

Well-designed data structures are central to the efficient mining of association rules. The *Apriori* algorithm, for example, employs a hash-tree structure to facilitate the support counting of candidate itemsets. On the other hand, algorithms such as FP-growth [31] and H-Miner [47] employ efficient data structures to provide a compact representation of the transaction database. A brief description of the hash tree and FP-tree data structures is presented next.

61.3.1 Hash Tree Structure

Apriori is a level-wise algorithm that generates frequent itemsets one level at a time, from itemsets of size-1 up to the longest frequent itemsets. At each level, candidate itemsets are generated by extending the frequent itemsets found at the previous level. Once the candidate itemsets have been enumerated, the transaction database is scanned once to determine their actual support counts. This generate-and-count procedure is repeated until no new frequent itemsets are found.

Support counting of candidate itemsets is widely recognized as the key bottleneck of frequent itemset generation. This is because one has to determine the candidate itemsets contained in each transaction of the database. A naive way for doing this is to simply match each transaction against every candidate itemset. If the candidate is a subset of the transaction, its support count is incremented. This approach can be prohibitively expensive

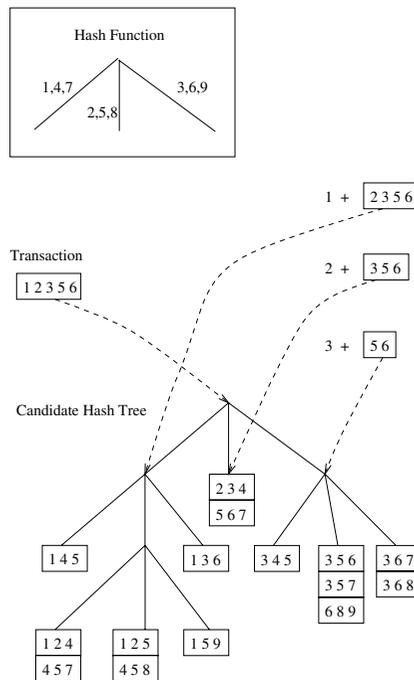


FIGURE 61.7: Hashing a transaction at the root node of a hash tree.

if the number of candidates and number of transactions are large.

In the *Apriori* algorithm, candidate itemsets are hashed into different buckets and stored in a hash tree structure. During support counting, each transaction is also hashed into its appropriate buckets. This way, instead of matching a transaction against all candidate itemsets, the transaction is matched only to those candidates that belong to the same bucket.

Figure 61.7 illustrates an example of a hash tree for storing candidate itemsets of size 3. Each internal node of the hash tree contains a hash function that determines which branch of the current node is to be followed next. The hash function used by the tree is also shown in this figure. Specifically, items 1, 4 and 7 are hashed to the left child of the node; items 2, 5, 8 are hashed to the middle child; and items 3, 6, 9 are hashed to the right child. Candidate itemsets are stored at the leaf nodes of the tree. The hash tree shown in Figure 61.7 contains 15 candidate itemsets, distributed across 9 leaf nodes.

We now illustrate how to enumerate candidate itemsets contained in a transaction. Consider a transaction t that contains five items, $\{1, 2, 3, 5, 6\}$. There are ${}^5C_3 = 10$ distinct itemsets of size 3 contained in this transaction. Some of these itemsets may correspond to the candidate 3-itemsets under investigation, in which case, their support counts are incremented. Other subsets of t that do not correspond to any candidates can be ignored.

Figure 61.8 shows a systematic way for enumerating size-3 itemsets contained in the transaction t by specifying the items one-by-one. It is assumed that items in every 3-itemset are stored in increasing lexicographic order. Because of the ordering constraint, all itemsets of size-3 derived from t must begin with item 1, 2, or 3. No 3-itemset may begin with item 5 or 6 because there are only two items in this transaction that are greater than or equal to 5. This is illustrated by the level 1 structures depicted in Figure 61.8. For example, the structure $1 \boxed{2\ 3\ 5\ 6}$ represents an itemset that begins with 1, followed by two

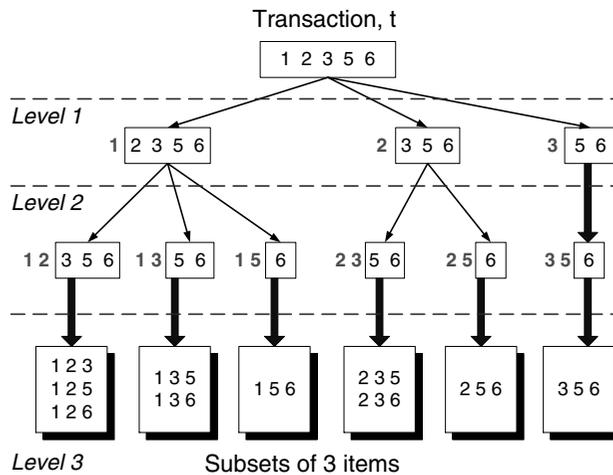


FIGURE 61.8: Enumerating subsets of three items from a transaction t .

more items chosen from the set $\{2, 3, 5, 6\}$.

After identifying the first item, the structures at level 2 denote the various ways to select the second item. For example, the structure $1\ 2\ \boxed{3\ 5\ 6}$ corresponds to itemsets with prefix $\{1\ 2\}$, followed by either item 3, 5, or 6. Once the first two items have been chosen, the structures at level 3 represent the complete set of 3-itemsets derived from transaction t . For example, the three itemsets beginning with the prefix $\{1\ 2\}$ are shown in the leftmost box at level 3 of this figure.

The tree-like structure shown in Figure 61.8 is simply meant to demonstrate how subsets of a transaction can be enumerated, i.e., by specifying the items in the 3-itemsets one-by-one, from its left-most item to its right-most item. For support counting, we still have to match each subset to its corresponding candidate. If there is a match, then the support count for the corresponding candidate is incremented.

We now describe how a hash tree can be used to determine candidate itemsets contained in the transaction $t = \{1, 2, 3, 5, 6\}$. To do this, the hash tree must be traversed in such a way that all leaf nodes containing candidate itemsets that belong to t are visited. As previously noted, all size-3 candidate itemsets contained in t must begin with item 1, 2, or 3. Therefore, at the root node of the hash tree, we must hash on items 1, 2, and 3 separately. Item 1 is hashed to the left child of the root node; item 2 is hashed to the middle child of the root node; and item 3 is hashed to the right child of the root node. Once we reach a child of the root node, we need to hash on the second item of the level 2 structures given in Figure 61.8. For example, after hashing on item 1 at the root node, we need to hash on items 2, 3, and 5 at level 2. Hashing on items 2 or 5 will lead us to the middle child node while hashing on item 3 will lead us to the right child node, as depicted in Figure 61.9. This process of hashing on items that belong to the transaction continues until we reach the leaf nodes of the hash tree. Once a leaf node is reached, all candidate itemsets stored at the leaf are compared against the transaction. If a candidate belongs to the transaction, its support count is incremented. In this example, 6 out of the 9 leaf nodes are visited and 11 out of the 15 itemsets are matched against the transaction.

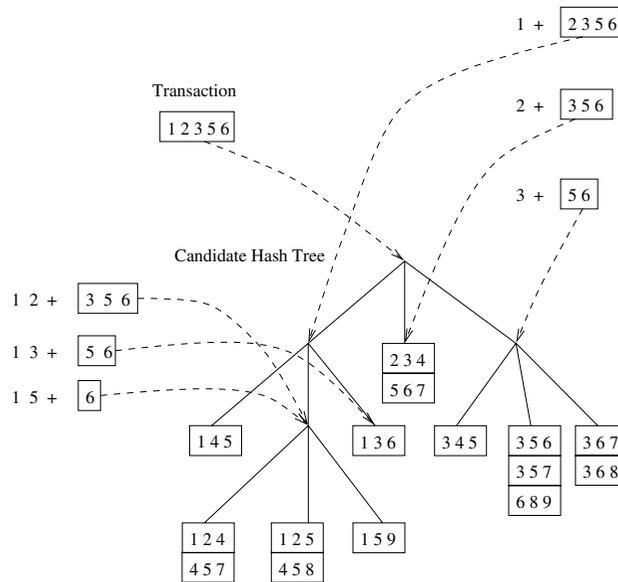


FIGURE 61.9: Subset operation on the left most subtree of the root of a candidate hash tree.

61.3.2 FP-Tree Structure

Recently, an interesting algorithm called FP-growth was proposed that takes a radically different approach to discovering frequent itemsets. The algorithm does not subscribe to the generate-and-count paradigm of Apriori. It encodes the database using a compact data structure called an FP-tree and infers frequent itemsets directly from this structure.

First, the algorithm scans the database once to find the frequent singleton items. An order is then imposed on the items based on decreasing support counts. Figure 61.10 illustrates an example of how to construct an FP-tree from a transaction database that contains five items, A, B, C, D, and E. Initially, the FP-tree contains only the root node, which is represented by a null symbol. Next, each transaction is used to create a path from the root node to some node in the FP-tree.

After reading the first transaction, $\{A, B\}$, a path is formed from the root node to its child node, labeled as A, and subsequently, to another node labeled as B. Each node in the tree contains the symbol of the item along with a count of the transactions that reach the particular node. In this case, both nodes A and B would have a count equal to one. After reading the second transaction $\{B, C, D\}$ a new path extending from $\text{null} \rightarrow B \rightarrow C \rightarrow D$ is created. Again, the nodes along this path have support counts equal to one. When the third transaction is read, the algorithm will discover that this transaction shares a common prefix A with the first transaction. As a result, the path $\text{null} \rightarrow A \rightarrow C \rightarrow D$ is merged to the existing path $\text{null} \rightarrow A \rightarrow B$. The support count for node A is incremented to two, while the newly-created nodes, C and D, each have a support count equal to one. This process is repeated until all the transactions have been mapped into one of the paths in the FP-tree. For example, the state of the FP-tree after reading the first ten transactions is shown at the bottom of Figure 61.10.

By looking at the way the tree is constructed, we can see why an FP-tree provides a compact representation of the database. If the database contains many transactions that share common items, then the size of an FP-tree will be considerably smaller than the size

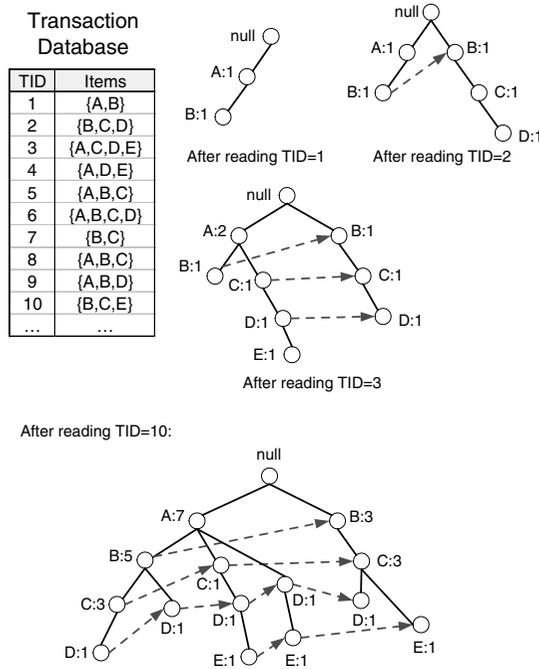


FIGURE 61.10: Construction of an FP-tree.

of the database. The best-case scenario would be that the database contains the same set of items for all transactions. The resulting FP-tree would contain only a single branch of nodes. The worst-case scenario would be that each transaction contains a unique set of items. In this case, there is no sharing of transactions among the nodes and the size of the FP-tree is the same as the size of the database.

During tree construction, the FP-tree structure also creates a linked-list access mechanism for reaching every individual occurrence of each frequent item used to construct the tree. In the above example, there are five such linked lists, one for each item, A, B, C, D, and E.

The algorithm used for generating frequent itemsets from an FP-tree is known as *FP-growth*. Given the FP-tree shown in Figure 61.10, the algorithm divides the problem into several subproblems, where each subproblem involves finding frequent itemsets having a particular suffix. In this example, the algorithm initially looks for frequent itemsets that end in E by following the linked list connecting the nodes for E. After all frequent itemsets ending in E are found, the algorithm looks for frequent itemsets that end in D by following the linked list for D, and so on.

How does FP-growth find all the frequent itemsets ending in E? Recall that an FP-tree stores the support counts of every item along each path, and that these counts reflect the number of transactions that are collapsed onto that particular path. In our example, there are only three occurrences of the node E. By collecting the *prefix paths* of E, we can solve the subproblem of finding frequent itemsets ending in E. The prefix paths of E consist of all paths starting from the root node up to the parent nodes of E. These prefix paths can form a new FP-tree to which the FP-growth algorithm can be recursively applied.

Before creating a new FP-tree from the prefix paths, the support counts of items along each prefix path must be updated. This is because the initial prefix path may include several transactions that do not contain the item E. For this reason, the support count of each item along the prefix path must be adjusted to have the same count as node E for

that particular path. After updating the counts along the prefix paths of E , some items may no longer be frequent, and thus, must be removed from further consideration (as far as our new subproblem is concerned). An FP-tree of the prefix paths is then constructed by removing the infrequent items. This recursive process of breaking the problem into smaller subproblems will continue until the subproblem involves only a single item. If the support count of this item is greater than the minimum support threshold, then the label of this item will be returned by the FP-growth algorithm. The returned label is appended as a prefix to the frequent itemset ending in E .

61.4 Clustering

Cluster analysis [6, 7, 33, 39] groups data objects based on information found in the data that describes the objects and their relationships. The goal is that the objects in a group be similar (or related) to one another and different from (or unrelated to) the objects in other groups. The greater the similarity (or homogeneity) within a group, and the greater the difference between groups, the ‘better’ or more distinct the clustering.

61.4.1 Hierarchical and Partitional Clustering

The most commonly made distinction between clustering techniques is whether the resulting clusters are nested or unnested or, in more traditional terminology, whether a set of clusters is *hierarchical* or *partitional*. A partitional or unnested set of clusters is simply a division of the set of data objects into non-overlapping subsets (clusters) such that each data object is in exactly one subset, i.e., a partitioning of the data objects. The most common partitional clustering algorithm is K-means, whose operation is described by the psuedo-code in Figure 61.11. (K is a user specified parameter, i.e., the number of clusters desired, and a centroid is typically the mean or median of the points in a cluster.)

- 1: Initialization: Select K points as the initial centroids.
- 2: **repeat**
- 3: Form K clusters by assigning all points to the closest centroid.
- 4: Recompute the centroid of each cluster.
- 5: **until** The centroids do not change

FIGURE 61.11: Basic K-means algorithm.

A hierarchical or nested clustering is a set of nested clusters organized as a hierarchical tree, where the leaves of the tree are singleton clusters of individual data objects, and where the cluster associated with each interior node of the tree is the union of the clusters associated with its child nodes. Typically, hierarchical clustering proceeds in an agglomerative manner, i.e., starting with each point as a cluster, we repeatedly merge the closest clusters, until only one cluster remains. A wide variety of methods can be used to define the distance between two clusters, but this distance is typically defined in terms of the distances between pairs of points in different clusters. For instance, the distance between clusters may be the minimum distance between any pair of points, the maximum distance, or the average distance. The algorithm for agglomerative clustering is described by the psuedo-code in [Figure 61.12](#).

- 1: Compute the pairwise distance matrix.
- 2: **repeat**
- 3: Merge the closest two clusters.
- 4: Update the distance matrix to reflect the distance between the new cluster and the original clusters.
- 5: **until** Only one cluster remains

FIGURE 61.12: Basic agglomerative hierarchical clustering algorithm

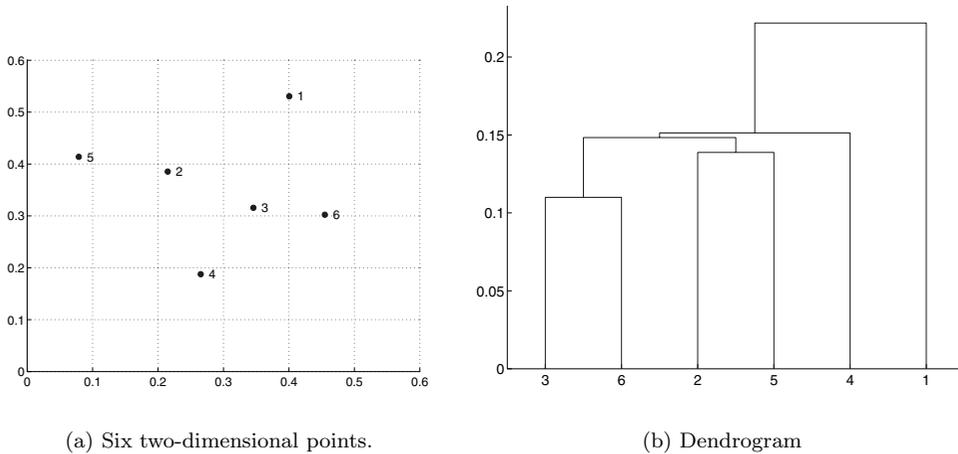


FIGURE 61.13: A hierarchical clustering of six points.

The tree that represents a hierarchical clustering is called a *dendrogram*, a term that comes from biological taxonomy. Figure 61.13 shows six points and the hierarchical clustering that is produced by the MIN clustering technique. This approach creates a hierarchical clustering by starting with the individual points as clusters, and then successively merges pairs of clusters with the minimum distance, i.e., that have the closest pair of points.

61.4.2 Nearest Neighbor Search and Multi-Dimensional Access Methods

In both the K-means and agglomerative hierarchical clustering algorithms, the time required is heavily dependent on the amount of time that it takes to find the the distance between sets of points—step 3 in both algorithms. This is true of most other clustering schemes as well, and thus, efficient implementations of clustering techniques often require considerations of nearest-neighbor search and the related area of multi-dimensional access methods. In the remainder of this section, we discuss these areas and their relevance to cluster analysis.

We begin by considering a general situation. Given a set of objects or data points, including possibly complicated data such as images, text strings, DNA sequences, polygons, etc., two issues are key to efficiently utilizing this data:

1. **How can items be located efficiently?** While a ‘representative’ feature vector is commonly used to ‘index’ these objects, these data points will normally be very sparse in the space, and thus, it is not feasible to use an array to store the data. Also, many sets of data do not have any features that constitute a ‘key’ that would

allow the data to be accessed using standard and efficient database techniques.

2. **How can similarity queries be efficiently conducted?** Many applications, including clustering, require the nearest neighbor (or the k nearest neighbors) of a point. For instance, the clustering techniques DBSCAN [24] and Chameleon [37] will have a time complexity of $O(n^2)$ unless they can utilize data structures and algorithms that allow the nearest neighbors of a point to be located efficiently. As a non-clustering example of an application of similarity queries, a user may want to find all the pictures similar to a particular photograph in a database of photographs.

Techniques for nearest-neighbor search are often discussed in papers describing multi-dimensional access methods or spatial access methods, although strictly speaking the topic of multi-dimensional access methods is broader than nearest-neighbor search since it addresses all of the many different types of queries and operations that a user might want to perform on multi-dimensional data. A large amount of work has been done in the area of nearest neighbor search and multi-dimensional access methods. Examples of such work include the kdb tree [14, 48], the R [28] tree, the R* tree [8], the SS-tree [34], the SR-tree [38], the X-tree [11], the GNAT tree [13], the M-tree [16], the TV tree [41], the hB tree [42], the “pyramid technique” [10], and the ‘hybrid’ tree [15]. A good survey of nearest-neighbor search, albeit from the slightly more general perspective of multi-dimensional access methods is given by [26].

As indicated by the prevalence of the word ‘tree’ in the preceding references, a common approach for nearest neighbor search is to create tree-based structures, such that the ‘closeness’ of the data increases as the tree is traversed from top to bottom. Thus, the nodes towards the bottom of the tree and their children can often be regarded as representing ‘clusters’ of data that are relatively cohesive. In the reverse directions, we also view clustering as being potentially useful for finding nearest neighbors. Indeed, one of the simplest techniques for generating a nearest neighbor tree is to cluster the data into a set of clusters and then, recursively break each cluster into subclusters until the subclusters consist of individual points. The resulting cluster tree consists of the clusters generated along the way. Regardless of how a nearest neighbor search tree is obtained, the general approach for performing a k -nearest-neighbor query is given by the algorithm in [Figure 61.14](#).

This seems fairly straightforward and, thus it seems as though nearest neighbor trees should be useful for clustering data, or conversely, that clustering would be a practical way to find nearest neighbors based on the results of clustering. However, there are some problems.

Goal Mismatch One of the goals of many nearest-neighbor tree techniques is to serve as efficient secondary storage based access methods for non-traditional databases, e.g., spatial databases, multimedia databases, document databases, etc. Because of requirements related to page size and efficient page utilization, ‘natural’ clusters may be split across pages or nodes. Nonetheless, data is normally highly ‘clustered’ and this can be used for actual clustering as shown in [25], which uses an R* tree to improve the efficiency of a clustering algorithm introduced in [46].

Problems with High-dimensional Data Because of the nature of nearest-neighbor trees, the tree search involved is a branch-and-bound technique and needs to search large parts of the tree, i.e., at any particular level, many children and their descendants may need to be examined. To see this, consider a point and all points that are within a given distance of it. This hyper-sphere (or hyper-rectangle in the case of multi-dimensional range queries) may very well cut across a number of nodes (clusters)—particularly if the point is on the edge of a cluster

```

1: Add the children of the root node to a search queue
2: while the search queue is not empty do
3:   Take a node off the search queue
4:   if that node and its descendants are not ‘close enough’ to be considered for the search
   then
5:     Discard the subtree represented by this node
6:   else
7:     if the node is not a leaf node, i.e., a data point then
8:       Add the children of this node to a search queue
9:     else {if the node is a leaf node}
10:      add the node to a list of possible solutions
11:    end if
12:  end if
13: end while
14: Sort the list of possible solutions by distance from the query point and return the
    k-nearest neighbors

```

FIGURE 61.14: Basic algorithm for a nearest neighbor query

and/or the query distance being considered is greater than the distance between clusters. More specifically, it is difficult for the algorithms that construct nearest neighbor trees to avoid a significant amount of overlap in the volumes represented by different nodes in the tree. In [11], it has been shown that the degree of overlap in a frequently used nearest neighbor tree, the R^* tree, approaches 100% as the dimensionality of the vectors exceeds 5. Even in two dimensions the overlap was about 40%. Other nearest neighbor search techniques suffer from similar problems.

Furthermore, in [12] it was demonstrated that the concept of “nearest neighbor” is not meaningful in many situations, since the minimum and maximum distances of a point to its neighbors tend to be very similar in high dimensional space. Thus, unless there is significant clustering in the data and the query ranges stay within individual clusters, the points returned by nearest neighbor queries are not much closer to the query point than are the points that are not returned. In this latter case, the nearest neighbor query is ‘unstable, to use the terminology of [12]. Recently, e.g., in [10], there has been some work on developing techniques that avoid this problem. Nonetheless, in some cases, a linear scan can be more efficient at finding nearest neighbors than more sophisticated techniques.

Outliers Typically, outliers are not discarded, i.e., all data points are stored. However, if some of the data points do not fit into clusters particularly well, then the presence of outliers can have deleterious effects on the lookups of other data points.

To summarize, there is significant potential for developments in the areas of nearest neighbor search and multidimensional access methods to make a contribution to cluster analysis. The contribution to efficiency is obvious, but the notions of distance (or similarity) are central to both areas, and thus, there is also the possibility of conceptual contributions as well. However, currently, most clustering methods that utilize nearest neighbor search or multidimensional access methods are interested only in the efficiency aspects [5, 25, 45].

61.5 Conclusion

In this chapter we have provided some examples to indicate the role that data structures play in data mining. For classification, we indicated how proximity graphs can play an important role in understanding and improving the performance of nearest neighbor classifiers. For association analysis, we showed how data structures are currently used to address the exponential complexity of the problem. For clustering, we explored its connection to nearest neighbor search and multi-dimensional access methods—a connection that has only been modestly exploited.

Data mining is a rapidly evolving field, with new problems continually arising, and old problems being looked at in the light of new developments. These developments pose new challenges in the areas of data structures and algorithms. Some of the most promising areas in current data mining research include multi-relational data mining [20, 23, 32], mining streams of data [19], privacy preserving data mining [3], and mining data with complicated structures or behaviors, e.g., graphs [32, 40] and link analysis [36, 44].

Acknowledgment

This work was partially supported by NASA grant # NCC 2 1231 and by the Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAD19-01-2-0014. The content of this work does not necessarily reflect the position or policy of the government and no official endorsement should be inferred. Access to computing facilities was provided by the AHPCRC and the Minnesota Supercomputing Institute. Figures 61.1-61.11, 61.13 and some parts of the text were taken from *Introduction to Data Mining* by Pang-Ning Tan, Michael Steinbach, and Vipin Kumar, published by Addison-Wesley, and are reprinted with the permission of Addison-Wesley.

References

- [1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD Intl. Conf. Management of Data*, pages 207–216, Washington D.C., USA, 1993.
- [2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th VLDB Conference*, pages 487–499, Santiago, Chile, 1994.
- [3] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 439–450. ACM Press, May 2000.
- [4] D. Aha. *A study of instance-based algorithms for supervised learning tasks: mathematical, empirical, and psychological evaluations*. PhD thesis, University of California, Irvine, 1990.
- [5] Alsabti, Ranka, and Singh. An efficient parallel algorithm for high dimensional similarity join. In *IPPS: 11th International Parallel Processing Symposium*. IEEE Computer Society Press, 1998.
- [6] M. R. Anderberg. *Cluster Analysis for Applications*. Academic Press, New York, December 1973.
- [7] P. Arabie, L. Hubert, and G. De Soete. An overview of combinatorial data analysis. In P. Arabie, L. Hubert, and G. De Soete, editors, *Clustering and Classification*, pages 188–217. World Scientific, Singapore, January 1996.

- [8] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r^* -tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331. ACM Press, 1990.
- [9] R. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, 1961.
- [10] S. Berchtold, C. Böhm, and H.-P. Kriegel. The pyramid-technique: towards breaking the curse of dimensionality. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 142–153. ACM Press, 1998.
- [11] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Databases*, pages 28–39, San Francisco, U.S.A., 1996. Morgan Kaufmann Publishers.
- [12] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is “nearest neighbor” meaningful? In *Proceedings 7th International Conference on Database Theory (ICDT’99)*, pages 217–235, 1999.
- [13] S. Brin. Near neighbor search in large metric spaces. In *The VLDB Journal*, pages 574–584, 1995.
- [14] T. B. B. Yu, R. Orlandic, and J. Somavaram. Kdb $_{KD}$ -tree: A compact kdb-tree structure for indexing multidimensional data. In *Proceedings of the 2003 IEEE International Symposium on Information Technology (ITCC 2003)*. IEEE, April, 28-30, 2003.
- [15] K. Chakrabarti and S. Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering, 23-26 March 1999, Sydney, Australia*, pages 440–447. IEEE Computer Society, 1999.
- [16] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *The VLDB Journal*, pages 426–435, 1997.
- [17] S. Cost and S. Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–78, 1993.
- [18] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *Knowledge Based Systems*, 8(6):373–389, 1995.
- [19] P. Domingos and G. Hulten. A general framework for mining massive data streams. *Journal of Computational and Graphical Statistics*, 12, 2003.
- [20] P. Domingos. Prospects and challenges for multirelational data mining. *SIGKDD Explorations*, 5(1), July 2003.
- [21] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley & Sons, Inc., New York, second edition, 2001.
- [22] M. H. Dunham. *Data Mining: Introductory and Advanced Topics*. Prentice Hall, 2002.
- [23] S. Dzeroski and L. D. Raedt. Multi-relational data mining: The current frontiers. *SIGKDD Explorations*, 5(1), July 2003.
- [24] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD96*, pages 226–231, 1996.
- [25] M. Ester, H.-P. Kriegel, and X. Xu. Knowledge discovery in large spatial databases: focusing techniques for efficient class identification. In M. Egenhofer and J. Herring, editors, *Advances in Spatial Databases, 4th International Symposium, SSD’95*, volume 951, pages 67–82, Portland, ME, 1995. Springer.
- [26] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing*

- Surveys (CSUR)*, 30(2):170–231, 1998.
- [27] R. L. Grossman, C. Kamath, V. K. Philip Kegelmeyer, and R. R. Namburu, editors. *Data Mining for Scientific and Engineering Applications*. Kluwer Academic Publishers, October 2001.
- [28] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57. ACM Press, 1984.
- [29] D. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.
- [30] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, 2001.
- [31] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM-SIGMOD Int'l Conf on Management of Data (SIGMOD'00)*, Dallas, TX, May 2000.
- [32] L. B. Holder and D. J. Cook. Graph-based relational learning: Current and future directions. *SIGKDD Explorations*, 5(1), July 2003.
- [33] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall Advanced Reference Series. Prentice Hall, Englewood Cliffs, New Jersey, March 1988.
- [34] R. Jain and D. A. White. Similarity indexing with the ss-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523, 1996.
- [35] J. W. Jaromczyk and G. T. Toussaint. Relative neighborhood graphs and their relatives. *Proceedings of the IEEE*, 80(9):1502–1517, September 1992.
- [36] D. Jensen and J. Neville. Data mining in social networks. In *National Academy of Sciences Symposium on Dynamic Social Network Analysis*, 2002.
- [37] G. Karypis, E.-H. Han, , and V. Kumar. Chameleon: Hierarchical clustering using dynamic modeling. *Computer*, 32(8):68–75, 1999.
- [38] N. Katayama and S. Satoh. The sr-tree: an index structure for high-dimensional nearest neighbor queries. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 369–380. ACM Press, 1997.
- [39] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley Series in Probability and Statistics. John Wiley and Sons, New York, November 1990.
- [40] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *The 2001 IEEE International Conference on Data Mining*, pages 313–320, 2001.
- [41] K.-I. Lin, H. V. Jagadish, and C. Faloutsos. The tv-tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, 1994.
- [42] D. B. Lomet and B. Salzberg. The hb-tree: a multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems (TODS)*, 15(4):625–658, 1990.
- [43] T. M. Mitchell. *Machine Learning*. McGraw-Hill, March 1997.
- [44] D. Mladenic, M. Grobelnik, N. Milic-Frayling, S. Donoho, and T. D. (editors). Kdd 2003: Workshop on link analysis for detecting complex behavior, August 2003.
- [45] F. Murtagh. Clustering in massive data sets. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 501–543. Kluwer Academic Publishers, Dordrecht, Netherlands, May 2002.
- [46] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 144–155, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.
- [47] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyperstructure mining of frequent patterns in large databases. In *Proc. 2001 Int'l Conf on Data Mining*

- (*ICDM'01*), San Jose, CA, Nov 2001.
- [48] J. T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18. ACM Press, 1981.
 - [49] G. T. Toussaint. Proximity graphs for nearest neighbor decision rules: recent progress. In *Interface-2002, 34th Symposium on Computing and Statistics, Montreal, Canada*, April 17-20 2002.
 - [50] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

Computational Geometry: Fundamental Structures

	62.1	Introduction.....	62-1
	62.2	Arrangements.....	62-2
		Substructures and Complexity • Decomposition • Duality	
	62.3	Convex Hulls.....	62-7
		Complexity • Construction • Dynamic Convex Hulls	
	62.4	Voronoi Diagrams.....	62-11
		Complexity • Construction • Variations	
	62.5	Triangulations.....	62-14
		Delaunay Triangulation • Polygons • Polyhedra • Pseudo-Triangulations	

Mark de Berg
Technical University, Eindhoven

Bettina Speckmann
Technical University, Eindhoven

62.1 Introduction

Computational geometry deals with the design and analysis of algorithms and data structures for problems involving spatial data. The questions that are studied range from basic problems such as line-segment intersection (“Compute all intersection points in a given set of line segments in the plane.”) to quite involved problems such as motion-planning (“Compute a collision-free path for a robot in workspace from a given start position to a given goal position.”) Because spatial data plays an important role in many areas within and outside of computer science—CAD/CAM, computer graphics and virtual reality, and geography are just a few examples—computational geometry has a broad range of applications. Computational geometry emerged from the general algorithms area in the late 1970s. It experienced a rapid growth in the 1980s and by now is a recognized discipline with its own conferences and journals and many researchers working in the area. It is a beautiful field with connections to other areas of algorithms research, to application areas like the ones mentioned earlier, and to areas of mathematics such as combinatorial geometry.

To design an efficient geometric algorithm or data structure, one usually needs two ingredients: a toolbox of algorithmic techniques and geometric data structures and a thorough understanding of the geometric properties of the problem at hand. As an example, consider the classic *post-office problem*, where we want to preprocess a set S of n points in the plane—the points in S are usually called *sites*—for the following queries: report the site in S that is closest to a query point q . A possible approach is to subdivide the plane into n regions, one for each site, such that the region of a site $s \in S$ consists of exactly those points $q \in \mathbb{R}^2$ for which s is the closest site. This subdivision is called the Voronoi

diagram of S . A query with a point q can now be answered by locating the region in which q lies, and reporting the site defining that region. To make this idea work, one needs an efficient data structure for point location. But one also needs to understand the geometry: What does the Voronoi diagram look like? What is its complexity? How can we construct it efficiently?

In [Part IV](#), many data structures for spatial data were already discussed. Hence, in this chapter we will focus on the second ingredient: we will discuss a number of basic geometric concepts. In particular, we will discuss arrangements in [Section 62.2](#), convex hulls in [Section 62.3](#), Voronoi diagrams in [Section 62.4](#), and triangulations in [Section 62.5](#).

More information on computational geometry can be found in various sources: there are several general textbooks on computational geometry [7, 8, 45, 49], as well as more specialized books e.g. on arrangements [20, 53] and Voronoi diagrams [44]. Finally, there are two handbooks that are devoted solely to (discrete and) computational geometry [24, 50].

62.2 Arrangements

The *arrangement* $\mathcal{A}(S)$ defined by a finite collection S of curves in the plane is the subdivision of the plane into open cells of dimensions 2 (the *faces*), 1 (the *edges*), and 0 (the *vertices*), induced by S —see [Fig. 62.1](#) for an example. This definition generalizes readily to

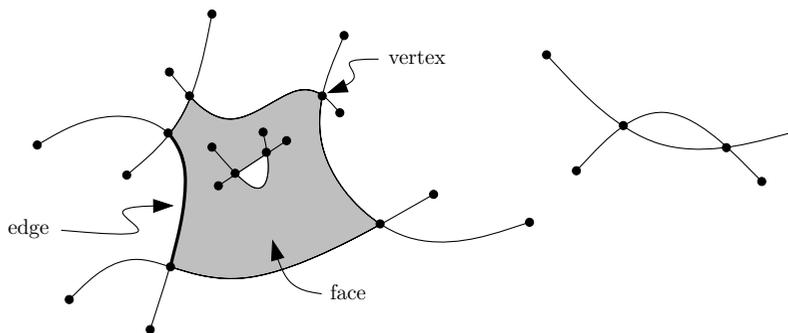


FIGURE 62.1: An arrangement of curves in the plane.

higher dimensions: the arrangement defined by a set S of geometric objects in \mathbb{R}^d such as hyperplanes or surfaces, is the decomposition of \mathbb{R}^d into open cells of dimensions $0, \dots, d$ induced by S . The cells of dimension k are usually called *k-cells*. The 0-cells are called *vertices*, the 1-cells are called *edges*, the 2-cells are called *faces*, the $(d - 1)$ -cells are called *facets*, and the d -cells are sometimes just called *cells*.

Arrangements have turned out to form a fundamental concept underlying many geometric problems and the efficiency of geometric algorithms is often closely related to the combinatorial complexity of (certain parts of) some arrangement. Moreover, to solve a certain problem geometric algorithms often rely on some decomposition of the arrangement underlying the problem. Hence, the following subsections give some more information on the complexity and the decomposition of arrangements.

		single cell	upper envelope	reference
$d = 2$	lines	$\Theta(n)$	$\Theta(n)$	trivial
	segments	$\Theta(n\alpha(n))$	$\Theta(n\alpha(n))$	[48]
	circles	$\Theta(n)$	$\Theta(n)$	linearization
	Jordan arcs	$\Theta(\lambda_{s+2}(n))$	$\Theta(\lambda_{s+2}(n))$	[28]
$d = 3$	planes	$\Theta(n)$	$\Theta(n)$	Euler's formula
	triangles	$\Omega(n^2\alpha(n)), O(n^2 \log n)$	$\Theta(n^2\alpha(n))$	[56], [21]
	spheres	$\Theta(n^2)$	$\Theta(n^2)$	linearization
	surfaces	$\Omega(n\lambda_q(n)), O(n^{2+\epsilon})$	$\Omega(n\lambda_q(n)), O(n^{2+\epsilon})$	[30]
$d > 3$	hyperplanes	$\Theta(n^{\lfloor d/2 \rfloor})$	$\Theta(n^{\lfloor d/2 \rfloor})$	Upper Bound Thm [37]
	$(d-1)$ -simplices	$\Omega(n^{d-1}\alpha(n)), O(n^{d-1} \log n)$	$\Theta(n^{d-1}\alpha(n))$	[56], [21]
	$(d-1)$ -spheres	$\Theta(n^{\lceil d/2 \rceil})$	$\Theta(n^{\lceil d/2 \rceil})$	linearization
	surfaces	$\Omega(n^{d-2}\lambda_q(n)), O(n^{d-1+\epsilon})$	$O(n^{d-1+\epsilon})$	[6], [54]

TABLE 62.1 Maximum complexity of single cells and envelopes in arrangements. The parameter s is the maximum number of points in which any two curves meet; the parameter q is a similar parameter for higher dimensional surfaces. The function $\lambda_t(n)$ is the maximum length of a Davenport-Schinzel sequence [53] of order t on n symbols, and is only slightly super-linear for any constant t . Bounds of the form $O(n^{d-1+\epsilon})$ hold for any constant $\epsilon > 0$.

62.2.1 Substructures and Complexity

Let H be a collection of n hyperplanes in \mathbb{R}^d . As stated earlier, the arrangement $\mathcal{A}(H)$ is the decomposition of \mathbb{R}^d into open cells of dimensions $0, \dots, d$ induced by H . The *combinatorial complexity* of $\mathcal{A}(H)$ is defined to be the total number of cells of the various dimensions. This definition immediately carries over to arrangements induced by other objects, such as segments in the plane, or surfaces in \mathbb{R}^d . For example, the complexity of the arrangement in Fig. 62.1 is 58, since it consists of 27 vertices, 27 edges, and 4 faces (one of which is the unbounded face).

It is easy to see that the maximum complexity of an arrangement of n lines in the plane is $\Theta(n^2)$: there can be at most $n(n-1)/2$ vertices, at most n^2 edges, and at most $n^2/2 + n/2 + 1$ faces. Also for an arrangement of curves the maximum complexity is $\Theta(n^2)$, provided that any pair of curves intersects at most s times, for a constant s . More generally, the maximum complexity of an arrangement of n hyperplanes in \mathbb{R}^d is $\Theta(n^d)$. The same bound holds for well-behaved surfaces (such as algebraic surfaces of constant maximum degree) or well-behaved surface patches in \mathbb{R}^d .

Single cells. It becomes more challenging to bound the complexity when we consider only a part of an arrangement. For example, what is the maximum complexity of a *single cell*, that is, the maximum number of i -cells, for $i < d$, on the boundary of any given d -cell? For lines in the plane this is still rather easy—the maximum complexity is $\Theta(n)$, since any line can contribute at most one edge to a given face—but the question is already quite hard for arrangements of line segments in the plane. Here it turns out that the maximum complexity can be $\Theta(n\alpha(n))$, where $\alpha(n)$ is the extremely slowly growing functional inverse of Ackermann's function. More generally, for Jordan curves where each pair intersects in at most s points, the maximum complexity is $\Theta(\lambda_{s+2}(n))$, where $\lambda_{s+2}(n)$ is the maximum length of a Davenport-Schinzel sequence of order $s+2$ on n symbols. The function $\lambda_{s+2}(n)$ is only slightly super-linear for any constant s . In higher dimensions, tight bounds are known for hyperplanes: the famous *Upper Bound Theorem* states that the maximum complexity of a single cell is $\Theta(n^{\lfloor d/2 \rfloor})$. For most other objects, the known upper and lower bounds are close but not tight—see Table 62.1.

Lower envelopes. Another important substructure is the *lower envelope*. Intuitively, the

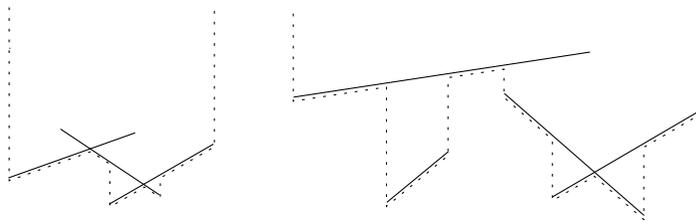


FIGURE 62.2: The lower envelope of a set of segments in the plane.

lower envelope of a set of segments in the plane is what one would see when looking at the segments from below—see Fig. 62.2. More formally, if we view the segments as graphs of partially defined (linear) functions, then the lower envelope is the point-wise minimum of these functions. Similarly, the *upper envelope* is defined as the point-wise maximum of the functions. The definition readily extends to x -monotone curves in the plane, to planes, triangles, or xy -monotone surface patches in \mathbb{R}^3 , etc.

Envelopes are closely related to single cells. The lower envelope of a set of lines in the plane, for instance, is the boundary of the single cell in the arrangement that is below all the lines. The vertices of the lower envelope of a set of segments in the plane are also vertices of the unbounded cell defined by those segments, but here the reverse is not true: vertices of the unbounded cell that are above other segments are not on the lower envelope. Nevertheless, the worst-case complexities of lower envelopes and single cells are usually very similar—see Table 62.1.

Other substructures. More types of substructures have been studied than single cells and envelopes: zones, levels, multiple cells, etc. The interested reader may consult the Chapter 21 of the *CRC Handbook of Discrete and Computational Geometry* [24], or the books by Edelsbrunner [20] or Sharir and Agarwal [53].

62.2.2 Decomposition

Full arrangements, or substructures in arrangements, are by themselves not convenient to work with, because their cells can be quite complex. Thus it is useful to further decompose the cells of interest into constant-complexity subcells: triangles or trapezoids in 2D, and simplices or trapezoid-like cells in higher dimensions. There are several ways of doing this.

Bottom-vertex triangulations. For arrangements of hyperplanes, the so-called *bottom-vertex triangulation* is often used. This decomposition is obtained as follows. Consider a bounded face f in a planar arrangement of lines. We can decompose f into triangles by drawing a line segment from the bottommost vertex v of f to all other vertices of f , except the vertices that are already adjacent to v —see Fig. 62.3(a). Note that this easy method for triangulating f is applicable since f is always convex. To decompose the whole arrangement of lines (or some substructure in it) we simply decompose each face in this manner.*

To decompose a d -cell C in a higher-dimensional arrangement of hyperplanes, we proceed inductively as follows. We first decompose each $(d - 1)$ -cell on the boundary of C , and then extend each $(d - 1)$ -simplex in this boundary decomposition into a d -simplex by connecting

*Unbounded faces require a bit of care, but they can be handled in a similar way.

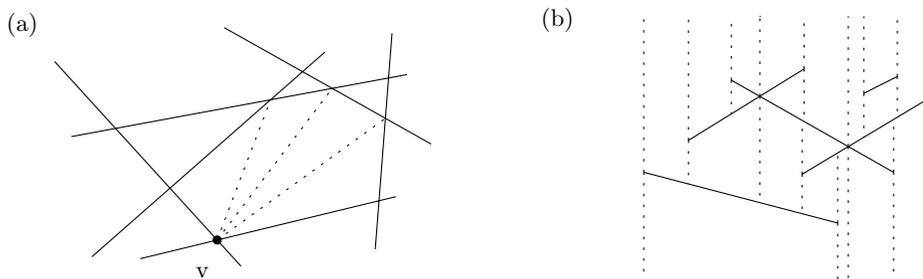


FIGURE 62.3: (a) The bottom-vertex triangulation of a face in a planar arrangement. (b) The vertical decomposition of a planar arrangement of line segments.

each of its vertices to the bottommost vertex of C . Again, this can readily be used to decompose any subset of cells in an arrangement of hyperplanes.

The total number of simplices in a bottom-vertex decomposition is linear in the total complexity of the cells being decomposed.

Vertical decompositions. The bottom-vertex triangulation requires the cells to be convex, so it does not work for arrangements of segments or for arrangements of surfaces. For such arrangements one often uses the *vertical decomposition* (or: *trapezoidal decomposition*). For an arrangement of line segments or curves in the plane, this decomposition is defined as follows. Each vertex of the arrangement—this can be a segment endpoint or an intersection point—has a vertical connection to the segment immediately above it, and to the segment immediately below it. If there is no segment below or above a vertex, then the connection extends to infinity. This decomposes each cell into trapezoids: subcells that are bounded by at most two vertical connections and by at most two segments—see Fig. 62.3(b). This definition can be generalized to higher dimensions as follows. Suppose we wish to decompose the arrangement $\mathcal{A}(S)$ induced by a collection S of surfaces in \mathbb{R}^d , where each surface is vertically monotone (that is, any line parallel to the x_d -axis intersects the surface in at most one point). Each point of any $(d-2)$ -dimensional cell of $\mathcal{A}(S)$ is connected by a vertical segment to the surface immediately above it and to the surface immediately below it. In other words, from each $(d-2)$ -cell we extend a vertical wall upward and downward. These walls together decompose the cells into subcells bounded by vertical walls and by at most two surfaces from S —one from above and one from below. These subcells are vertically monotone, but do not yet have constant complexity. Hence, we recursively decompose the bottom of the cell, and then extend this decomposition vertically upward to obtain a decomposition of the entire cell.

The vertical decomposition can be used for most arrangements (or substructures in them). In the plane, the maximum complexity of the vertical decomposition is linear in the total complexity of the decomposed cells. However, in higher dimensions this is no longer true. In \mathbb{R}^3 , for instance, the vertical decomposition of an arrangement of n disjoint triangles can consist of $\Theta(n^2)$ subcells, even though in this case the total complexity of the arrangement is obviously linear. Unfortunately, this is unavoidable, as there are collections of disjoint triangles in \mathbb{R}^3 for which any decomposition into convex subcells must have $\Omega(n^2)$ subcells. For n intersecting triangles, the vertical decomposition has complexity $O(n^2 \alpha(n) \log n + K)$, where K is the complexity of the arrangement of triangles [56]. More information about the complexity of vertical decompositions in various settings can be found in Halperin's survey on arrangements [24, Chapter 21]. In many cases, vertical decompositions can be constructed in time proportional to their complexity, with perhaps a small (logarithmic or

$O(n^\epsilon)$ multiplicative factor.

62.2.3 Duality

Consider the transformation in the plane that maps the point $p = (p_x, p_y)$ to the line $p^* : y = p_x x - p_y$, and the line $\ell : y = ax + b$ to the point $\ell^* = (a, -b)$. Such a transformation that maps points to lines and vice versa is called a *duality transform*. Often the term *primal plane* is used for the plane in which the original objects live, and the term *dual plane* is used for the plane in which their images live. The duality transform defined above has a few easy-to-verify properties:

- (i) It is *incidence preserving*: if a point p lies on a line ℓ , then the point ℓ^* dual to ℓ lies on the line p^* dual to p .
- (ii) It is *order preserving*: if a point p lies above a line ℓ , then ℓ^* lies above p^* .

These properties imply several others. For example, three points on a line become three lines through a point under the duality transform—see Fig. 62.4. Another property is that for any point p we have $(p^*)^* = p$. Notice that the duality transform above is not defined for vertical lines. This technicality is usually not a problem, as vertical lines can often be handled separately.

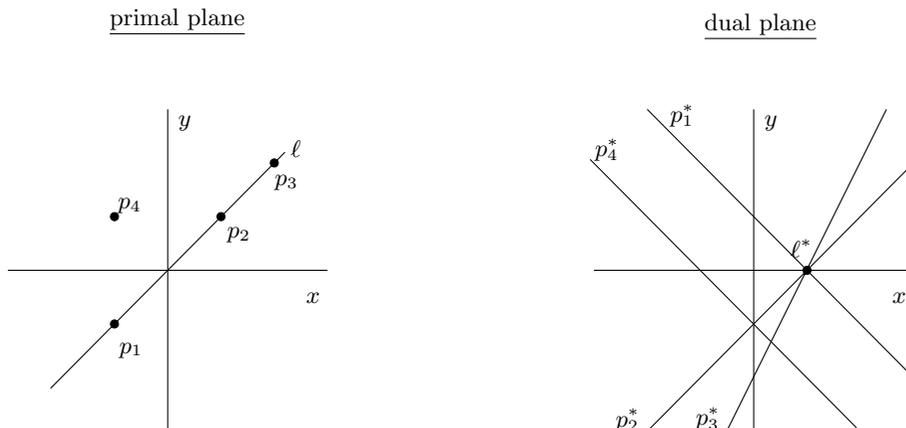


FIGURE 62.4: Illustration of the duality transform.

This duality transform is so simple that it does not seem very interesting at first sight. However, it turns out to be extremely useful. As an example, consider the following problem. We are given a set P of n points in the plane, which we wish to preprocess for *strip-emptiness* queries: given a query strip—a strip is the region between two parallel lines—decide whether that strip is empty or if it contains one or more points from P . If we know about duality, and we know about data structures for point location, then this problem is easy to solve: we take the duals of the points in P to obtain a set P^* of lines in the plane, and we preprocess the arrangement $\mathcal{A}(P^*)$ induced by P^* for logarithmic-time point location. To decide whether the strip bounded by lines ℓ_1 and ℓ_2 is empty, we perform point locations with ℓ_1^* and ℓ_2^* in $\mathcal{A}(P^*)$; the strip is empty if and only if ℓ_1^* and ℓ_2^* lie in the same face

of the arrangement. (This does not work if ℓ_1 and ℓ_2 are vertical, since then their duals are undefined. But for this case we can simply build one extra data structure, which is a balanced binary search tree on the x -coordinates of the points.)

In principle, of course, we could also have arrived at this solution without using duality. After all, duality does not add any new information: it is just a different way of looking at things. Hence, every algorithm or data structure that works in the dual plane can also be interpreted as working in the primal plane. But some things are simply more easy to see in the dual plane. For example, a face in the arrangement $\mathcal{A}(P^*)$ in the dual plane is much more visible than the collection of all lines in the primal plane dividing P into two subsets in a certain way. So without duality, we would not have realized that we could solve the strip-emptiness problem with a known data structure, and we would probably not have been able to develop that structure ourselves either.

This is just one example of the use of duality. There are many more problems where duality is quite useful. In fact, we will see another example in the next section, when we study convex hulls.

62.3 Convex Hulls

A set $A \subset \mathbb{R}^d$ is *convex* if for any two points $p, q \in A$ the segment \overline{pq} is completely contained in A . The *convex hull* of a set S of objects is the smallest convex set that contains all objects in S , that is, the most tightly fitting convex bounding volume for S . For example, if S is a set of objects in the plane, we can obtain the convex hull by taking a large rubber band around the objects and then releasing the band; the band will snap around the objects and the resulting shape is the convex hull. More formally, we can define $\mathcal{CH}(S)$ as the intersection of all convex sets containing all objects in S :

$$\mathcal{CH}(S) := \bigcap \{A : A \text{ is convex, and } o \subset A \text{ for all } o \in S \}.$$

We denote the convex hull of the objects in S by $\mathcal{CH}(S)$.

It is easy to see that the convex hull of a set of line segments in the plane is the same as the convex hull of the endpoints of the segments. More generally, the convex hull of a set of bounded polygonal objects in \mathbb{R}^d is the same as the convex hull of the vertices of the objects. Therefore we will restrict our discussion to convex hulls of sets of points. Table 62.2 gives an overview of the results on the complexity and construction of convex hulls discussed below.

	complexity	construction	reference
$d = 2$, worst case	$\Theta(n)$	$O(n \log n)$	[7, 26]
$d = 3$, worst case	$\Theta(n)$	$O(n \log n)$	[17, 39, 43, 51]
$d > 3$, worst case	$\Theta(n^{\lfloor d/2 \rfloor})$	$O(n^{\lfloor d/2 \rfloor})$	[14, 17, 39, 51]
$d \geq 2$, uniform distr.	$\Theta(\log^{d-1} n)$	$O(n)$	[18]

TABLE 62.2 Maximum complexity of the convex hull of a set of n points, and the time needed to construct the convex hull. The bounds on uniform distribution refer to points drawn uniformly at random from a hypercube or some other convex polytope.

62.3.1 Complexity

Let P be a set of n points in \mathbb{R}^d . The *convex hull* of P , denoted by $\mathcal{CH}(P)$, is a convex polytope whose vertices are a subset of the points in P . The complexity of a polytope is defined as the total number of k -facets[†] (that is, k -dimensional features) on the boundary of the polytope, for $k = 0, 1, \dots, d-1$: the complexity of a planar polygon is the total number of vertices and edges, the complexity of a 3-dimensional polytope is the total number of vertices, edges, and faces, and so on.

Because the vertices of $\mathcal{CH}(P)$ are a subset of the points in P , the number of vertices of $\mathcal{CH}(P)$ is at most n . In the plane this means that the total complexity of the convex hull is $O(n)$, because the number of edges of a planar polygon is equal to the number of vertices. In higher dimensions this is no longer true: the number of k -facets ($k > 0$) of a polytope can be larger than the number of vertices. How large can this number be in the worst case? In \mathbb{R}^3 , the total complexity is still $O(n)$. This follows from Euler's formula, which states that for a convex polytope in \mathbb{R}^3 with V vertices, E edges, and F faces it holds that $V - E + F = 2$. In higher dimensions, the complexity can be significantly higher: the worst-case complexity of a convex polytope with n vertices in \mathbb{R}^d is $\Theta(n^{\lfloor d/2 \rfloor})$.

In fact, the bound on the complexity of the convex hull immediately follows from the results of the previous section if we apply duality. To see this, consider a set P of n points in the plane. For simplicity, let us suppose that $\mathcal{CH}(P)$ does not have any vertical edges and that no three points in P are collinear. Define the *upper hull* of P , denoted by $\mathcal{UH}(P)$, as the set of edges of $\mathcal{CH}(P)$ that bound $\mathcal{CH}(P)$ from above. Let P^* be the set of lines that are the duals of the points in P . A pair $p, q \in P$ defines an edge of $\mathcal{UH}(P)$ if and only if all other points $r \in P$ lie below the line through p and q . In the dual this means that all lines $r^* \in P^*$ lie above the intersection point $p^* \cap q^*$. In other words, $p^* \cap q^*$ is a vertex on the lower envelope $\mathcal{LE}(P^*)$ of the lines in P^* . Furthermore, a point $p \in P$ is a vertex of $\mathcal{UH}(P)$ if and only if its dual p^* defines an edge of $\mathcal{LE}(P^*)$. Thus there is a one-to-one correspondence between the vertices (or, edges) of $\mathcal{UH}(P)$, and the edges (or, vertices) of $\mathcal{LE}(P^*)$. In higher dimensions a similar statement is true: there is a one-to-one correspondence between the k -facets of the upper hull of P and the $(d-k-1)$ -facets of the lower envelope of P^* . The bound on the complexity of the convex hull of a set of n points in \mathbb{R}^d therefore follows from the $\Theta(n^{\lfloor d/2 \rfloor})$ bound on the complexity of the lower envelope of a set of n hyperplanes \mathbb{R}^d .

The $\Theta(n^{\lfloor d/2 \rfloor})$ bound implies that the complexity of the convex hull can be quite high when the dimension gets large. Fortunately this is not the case if the points in P are distributed uniformly: in that case only few of the points in P are expected to show up as a vertex on the convex hull. More precisely, the expected complexity of the convex hull of a set P that is uniformly distributed in the unit hypercube is $O(\log^{d-1} n)$ [18, 19].

62.3.2 Construction

We now turn our attention to algorithms for computing the convex hull of a set P of n points in \mathbb{R}^d . In the previous subsection we have shown that there is a correspondence between the upper hull of P and the lower envelope of P^* , where P^* is the set of hyperplanes dual to the points in P . It follows that any algorithm that can compute the convex hull of a set

[†]In the previous section we used the term k -cells for the k -dimensional features in an arrangement, but since we are dealing here with a single polytope we prefer the term k -facet.

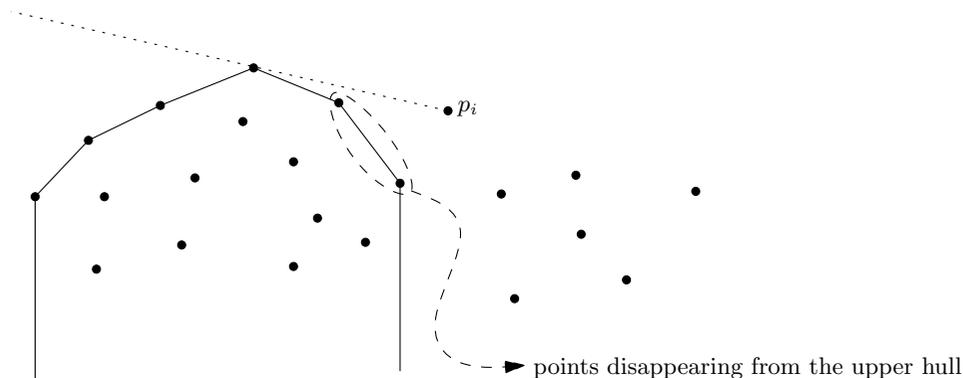


FIGURE 62.5: Constructing the upper hull.

of points in \mathbb{R}^d can also be used to compute the intersection of a set of half-spaces in \mathbb{R}^d , and vice versa.

First consider the planar case. By a reduction from sorting, one can show that $\Omega(n \log n)$ is a lower bound on the worst-case running time of any convex-hull algorithm. There are many different algorithms that achieve $O(n \log n)$ running time and are thus optimal in the worst case. One of the best known algorithms is called *Graham's scan* [7, 26]. It treats the points from left to right, and maintains the upper hull of all the points encountered so far. To handle a point p_i , it is first added to the end of the current upper hull. The next step is to delete the points that should no longer be part of the hull. They always form a consecutive portion at the right end of the old hull, and can be identified easily—see Fig. 62.5.

After these points have been deleted, the next point is handled. Graham's scan runs in linear time, after the points have been sorted from left to right. This is optimal in the worst case.

The $\Omega(n \log n)$ lower bound does not hold if only few points show up on the convex hull. Indeed, in this case it is possible to do better: Kirkpatrick and Seidel [34], and later Chan [10], gave output-sensitive algorithms that compute the convex hull in $O(n \log k)$ time, where k is the number of vertices of the hull.

In \mathbb{R}^3 , the worst-case complexity of the convex hull is still linear, and it can be computed in $O(n \log n)$ time, either by a deterministic divide-and-conquer algorithm [43] or by a simpler randomized algorithm [17, 39, 51]. In dimensions $d > 3$, the convex hull can be computed in $\Theta(n^{\lfloor d/2 \rfloor})$ time [14]. This is the same as the worst-case complexity of the hull, and therefore optimal. Again, the simplest algorithms that achieve this bound are randomized [17, 39, 51]. There is also an output-sensitive algorithm by Chan [10], which computes the convex hull in $O(n \log k + (nk)^{1-1/(\lfloor d/2 \rfloor + 1)} \log^{O(1)} n)$ time, where k is its complexity.

As remarked earlier, the expected complexity of the convex hull of a set of n uniformly distributed points is much smaller than the worst-case complexity. This means that if we use an output-sensitive algorithm to compute the convex hull, its expected running time will be better than its worst-case running time. One can do even better, however, by using specialized algorithms. For example, Dwyer [18] has shown that the convex hull of points distributed uniformly in e.g. the unit hypercube can be computed in linear expected time.

62.3.3 Dynamic Convex Hulls

In some applications the set P changes over time: new points are inserted into P , and some existing points are deleted. The convex hull can change drastically after an update: the insertion of a single point can cause $\Theta(n)$ points to disappear from the convex hull, and the deletion of a single point can cause $\Theta(n)$ points to appear on the convex hull. Surprisingly, it is nevertheless possible to store the convex hull of a planar point set in such a way that any update can be processed in $O(\log^2 n)$ in the worst case, as shown by Overmars and van Leeuwen [47]. The key to their result is to not only store the convex hull of the whole set, but also information about the convex hull of certain subsets. The structure of Overmars and van Leeuwen roughly works as follows. Suppose we wish to maintain $\mathcal{UH}(P)$, the upper hull of P ; maintenance of the lower hull can be done similarly. The structure to maintain $\mathcal{UH}(P)$ is a balanced binary tree \mathcal{T} , whose leaves store the points from P sorted by x -coordinate. The idea is that each internal node ν stores the upper hull of all the points in the subtree rooted at ν . Instead of storing the complete upper hull at each node ν , however, we only store those parts that are not already on the upper hull of nodes higher up in the tree. In other words, the point corresponding to a leaf μ is stored at the highest ancestor of μ where it is on the upper hull—see Fig. 62.6 for an illustration.

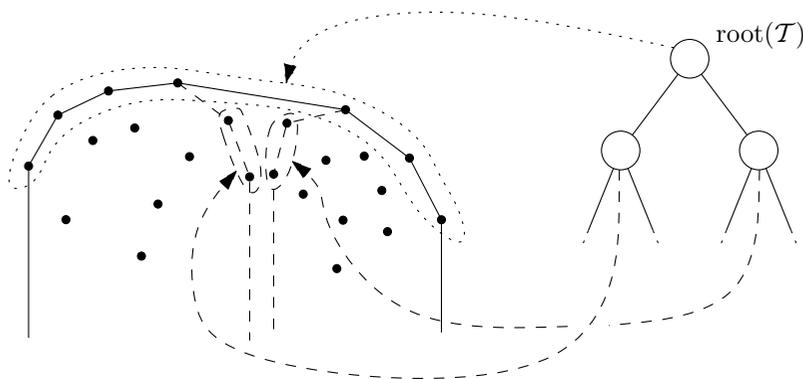


FIGURE 62.6: The Overmars-van Leeuwen structure to maintain the convex hull.

Note that the root still stores the upper hull of the entire set P . Because a point is stored in only one upper hull, the structure uses $O(n)$ storage. Overmars and van Leeuwen show how to update the structure in $O(\log^2 n)$ time in the worst case.

Although the result by Overmars and van Leeuwen is more than 20 years old by now, it still has not been improved in its full generality. Nevertheless, there have been several advances in special cases. For example, for the semi-dynamic case (only insertions, or only deletions), there are structures with $O(\log n)$ update time [30, 42]. Furthermore, $O(\log n)$ update time can be achieved in the off-line setting, where the sequence of insertions and deletions is known in advance [32]. Improvements are also possible if one does not need to maintain the convex hull explicitly. For example, in some applications the reason for maintaining the convex hull is to be able to find the point that is extreme in a query direction, or the intersection of the convex hull with a query line. Such queries can be answered in logarithmic time if the convex hull vertices are stored in order in a balanced binary tree, but it is not necessary to know all the convex hull vertices explicitly to answer such queries.

This observation was used by Chan [11], who described a fully dynamic structure that can answer such queries in $O(\log n)$ time and that can be updated in $O(\log^{1+\varepsilon} n)$ time. This result was recently improved by Brodal and Jacob [9], who announced a structure that uses $O(n)$ storage, has $O(\log n)$ update time, and can answer queries in $O(\log n)$ time. Neither Chan's structure nor the structure of Brodal and Jakob, however, can report the convex hull in time linear in its complexity, and the update times are amortized.

62.4 Voronoi Diagrams

Recall the *post-office problem* mentioned in the introduction. Here we want to preprocess a set S of n points, referred to as *sites*, in the plane such that we can answer the query: which site in S is closest to a given query point q ? In order to solve this problem we can divide the plane into regions according to the nearest-neighbor rule: each site s gets assigned the region which is closest to s . This subdivision, which compactly captures the distance information inherent in a given configuration, is called the *Voronoi diagram* of S —see Fig. 62.7(a).

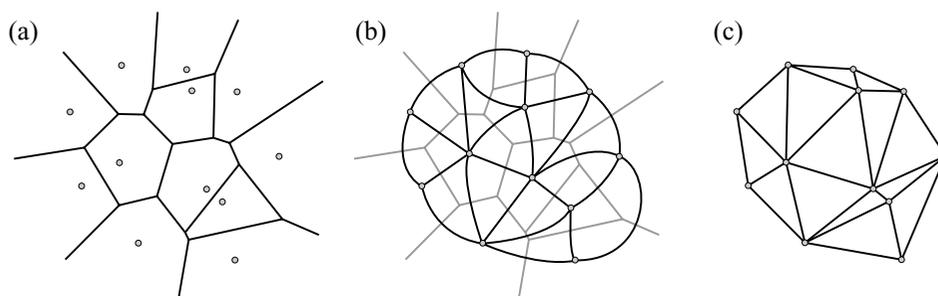


FIGURE 62.7: (a) The Voronoi diagram of a set of points. (b) The dual graph of the Voronoi diagram. (c) The Delaunay triangulation of the points.

More formally, the Voronoi diagram of a set of sites $S = \{s_1, \dots, s_n\}$ in \mathbb{R}^d , which we refer to as $\text{Vor}(S)$, partitions space into n regions—one for each site—such that the region for a site s_i consists of all points that are closer to s_i than to any other site $s_j \in S$. The set of points that are closest to a particular site s_i forms the so-called *Voronoi cell* of s_i , and is denoted by $V(s_i)$. Thus, when S is a set of sites in the plane we have

$$V(s_i) = \{p \in \mathbb{R}^2 : \text{dist}(p, s_i) < \text{dist}(p, s_j) \text{ for all } j \neq i\},$$

where $\text{dist}(\cdot, \cdot)$ denotes the Euclidean distance.

Now consider the dual graph of the Voronoi diagram, that is, the graph that has a node for every Voronoi cell and an arc between any two Voronoi cells that share a common edge—see Fig. 62.7(b). (Observe that the concept of dual graph used here has nothing to do with the duality transform discussed in Section 62.2.3.) Suppose we embed this graph in the plane, by using the site s_i to represent the node corresponding to the cell $V(s_i)$ and by drawing the edges as straight line segments, as in Fig. 62.7(c). Somewhat surprisingly perhaps, this graph is always planar. Moreover, it is actually a triangulation of the point set S , assuming that no four points in S are co-circular. More details on this special triangulation, which is called the *Delaunay triangulation*, will be given in Section 62.5.1.

There exists a fascinating connection between Voronoi diagrams in \mathbb{R}^d and half-space intersections in \mathbb{R}^{d+1} . Assume for simplicity that $d = 2$, and consider the transformation that maps a site $s = (s_x, s_y)$ in \mathbb{R}^2 to the non-vertical plane $h(s) : z = 2s_x x + 2s_y y - (s_x^2 + s_y^2)$ in \mathbb{R}^3 . Geometrically, $h(s)$ is the plane tangent to the unit paraboloid $z = x^2 + y^2$ at the point vertically above $(s_x, s_y, 0)$. Let $H(S)$ be the set of planes that are the image of a set of point sites S in the plane. Let \mathcal{S} denote the convex polyhedron that is formed by the intersection of the positive half-spaces defined by the planes in $H(S)$, that is, $\mathcal{S} = \bigcap_{h \in H(S)} h^+$, where h^+ denotes the half-space above h . Surprisingly, the projection of the edges and vertices of \mathcal{S} vertically downward on the xy -plane is exactly the Voronoi diagram of S .

The Voronoi diagram can be defined for various sets of sites, for example points, line segments, circles, or circular arcs, and for various metrics. Sections 62.4.1 and 62.4.2 discuss the complexity and construction algorithms of Voronoi diagrams for the usual case of point sites in \mathbb{R}^d , while Section 62.4.3 describes some of the possible variations. Additional details and proofs of the material presented in this section can be found in [4, 5, 22].

62.4.1 Complexity

Let S be a set of n points in \mathbb{R}^d . The Voronoi diagram of S is a cell complex in \mathbb{R}^d . If $d = 2$ then the Voronoi cell of a site is the interior of a convex, possibly infinite polygon. Its boundary consists of *Voronoi edges*, which are equidistant from two sites, and *Voronoi vertices*, which are equidistant from at least three sites. The Voronoi diagram of $n \geq 3$ sites has at most $2n - 5$ vertices and at most $3n - 6$ edges, which implies that the Delaunay triangulation has at most $2n - 5$ triangles and $3n - 6$ edges. What is the complexity of the Voronoi diagram in $d \geq 3$? Here the connection between Voronoi diagrams in \mathbb{R}^d and intersections of half-spaces in \mathbb{R}^{d+1} comes in handy: we know from the Upper Bound Theorem that the intersection of n half-spaces in \mathbb{R}^{d+1} has complexity $O(n^{\lfloor (d+1)/2 \rfloor}) = O(n^{\lceil d/2 \rceil})$. This bound is tight, so the maximum complexity of the Voronoi diagram—and of the Delaunay triangulation, for that matter—in \mathbb{R}^d is $\Theta(n^{\lceil d/2 \rceil})$.

62.4.2 Construction

In this section we present several algorithms to compute the Voronoi diagram or its dual, the Delaunay triangulation, for point sites in \mathbb{R}^d . Several of these algorithms can be generalized to work with metrics other than the Euclidean, and to sites other than points. Since the Voronoi diagram in the plane has only linear complexity one might be tempted to search for a linear time construction algorithm. However the problem of sorting n real numbers is reducible to the problem of computing Voronoi diagrams and, hence, any algorithm for computing the Voronoi diagram must take $\Omega(n \log n)$ time in the worst case.

Two data structures that are particularly well suited to working with planar subdivisions like the Voronoi diagram are the *doubly-connected edge list* (DCEL) by Muller and Preparata [38] and the *quad-edge structure* by Guibas and Stolfi [29]. Both structures require $O(n)$ space, where n is the complexity of the subdivision, and allow to efficiently traverse the edges adjacent to a vertex and the edges bounding a face. In both structures, one can easily obtain in $O(n)$ time a structure for the Voronoi diagram from a structure for the Delaunay triangulation, and vice versa. In fact, the quad-edge structure, as well as a variant of the DCEL [7], are representations of a planar subdivision and its dual at the same time, so there is nothing to convert (except, perhaps, that one may wish to explicitly compute the coordinates of the vertices in the Voronoi diagram, if the quad-edge structure or DCEL describes the Delaunay triangulation).

Divide-and-conquer. The first deterministic worst-case optimal algorithm to compute the Voronoi diagram was given by Shamos and Hoey [52]. Their divide-and-conquer algorithm splits the set S of point sites by a dividing line into subsets L and R of approximately the same size. The Voronoi diagrams $\text{Vor}(L)$ and $\text{Vor}(R)$ are computed recursively and then merged into $\text{Vor}(S)$ in linear time. This algorithm constructs the Voronoi diagram of a set of n points in the plane in $O(n \log n)$ time and linear space.

Plane Sweep. The strategy of a sweep line algorithm is to move a line, called the *sweep line*, from top to bottom over the plane. While the sweep is performed, information is maintained regarding the structure one wants to compute. It is tempting to apply the same approach to Voronoi diagrams, by keeping track of the Voronoi edges that are currently intersected by the sweep line. It is problematic, however, to discover a new Voronoi region in time: when the sweep line reaches a new site, then the line has actually been intersecting the Voronoi edges of its region for a while. Fortune [23] was the first to find a way around this problem. *Fortune's algorithm* applies the plane sweep paradigm in a slightly different fashion: instead of maintaining the intersection of the sweep line with the Voronoi diagram, it maintains information of the part of the Voronoi diagram of the sites above the line that can not be changed by sites below it. This algorithm provides an alternative way of computing the Voronoi diagram of n points in the plane in $O(n \log n)$ time and linear space.

Randomized incremental construction. A natural idea is to construct the Voronoi diagram by *incremental insertion*, that is, to obtain $\text{Vor}(S)$ from $\text{Vor}(S \setminus \{s\})$ by inserting the site s . Insertion of a site means integrating its Voronoi region into the diagram constructed so far. Unfortunately the region of s can have up to $n - 1$ edges, for $|S| = n$, which may lead to a running time of $O(n^2)$. The insertion process is probably better described and implemented in the dual environment, for the Delaunay triangulation DT : construct $DT_i = DT(\{s_1, \dots, s_i\})$ by inserting s_i into DT_{i-1} . The advantage of this approach over a direct construction of $\text{Vor}(S)$ is that Voronoi vertices that appear only in intermediate diagrams but not in the final one need not be computed or stored. DT_i is constructed by exchanging edges, using edge flips [36], until all edges invalidated by s_i have been removed. Still, the worst-case running time of this algorithm can be quadratic. However, if we insert the sites in *random order*, and the algorithm is implemented carefully, then one can prove that the expected running time is $O(n \log n)$, and that the expected amount of storage is $O(n)$.

Other approaches. Finally, recall the connection between Delaunay triangulations and convex hulls. Since there exist $O(n \log n)$ algorithms to compute the convex hull of points in \mathbb{R}^3 (see [Section 62.3](#)) we therefore have yet another optimal algorithm for the computation of Voronoi diagrams.

62.4.3 Variations

In this section we present some of the common variations on Voronoi diagrams. The first is the *order- k Voronoi diagram* of a set S of n sites, which partitions \mathbb{R}^d on the basis of the first k closest point sites. In other words, each cell in the order- k Voronoi diagram of a set S of sites in the plane corresponds to a k -tuple of sites from S and consists of those points in the plane for which that k -tuple are the k closest sites. One might fear that the order- k Voronoi diagram has $\Theta(n^k)$ cells, but this is not the case. In two dimensions, for example, its complexity is $O(k(n-k))$, and it can be computed in $O(k(n-k) \log n + n \log^3 n)$ expected time [2].

The *furthest-site Voronoi diagram* partitions \mathbb{R}^d according to the furthest site, or equivalently, according to the closest $n - 1$ of n sites. The furthest-site Voronoi diagram can be computed in $O(n \log n)$ time in two dimensions, and in $O(n^{\lceil d/2 \rceil})$ in dimension $d \geq 3$.

One can also consider different distance functions than the Euclidean distance. For example, one can alter the distance function by the addition of *additive* or *multiplicative weights*. In this case every point site s_i is associated with a weight w_i and the distance function $d(s_i, x)$ between a point x and a site s_i becomes $d(s_i, x) = w_i + \text{dist}(s_i, x)$ (additive weights) or $d(s_i, x) = w_i \cdot \text{dist}(s_i, x)$ where $\text{dist}(s_i, x)$ denotes the Euclidean distance between s_i and x . The Voronoi diagram for point sites in 2 dimensions with additive weights can be computed in $O(n \log n)$ time, for multiplicative weights the time increases to $O(n^2)$ time.

Finally the *power diagram*, or *Laguerre diagram*, is another Voronoi diagram for point sites s_i that are associated with weights w_i . Here the distance function is the *power distance* introduced by Aurenhammer [3], where the distance from a point x to a site s_i is measured along a line tangent to the sphere of radius $\sqrt{w_i}$ centered at s_i , i.e., $d(s_i, x) = \sqrt{\text{dist}(s_i, x)^2 - w_i}$. The power diagram can be computed in $O(n \log n)$ time in two dimensions.

62.5 Triangulations

In geometric data processing, structures that partition the geometric input, as well as connectivity structures for geometric objects, play an important role. Versatile tools in this context are *triangular meshes*, often called *triangulations*. A triangulation of a geometric domain such as a polygon in \mathbb{R}^2 or a polyhedron in \mathbb{R}^3 is a partition into simplices that meet only at shared faces. A triangulation of a point set S is a triangulation of the convex hull of S , such that the vertices in the triangulation are exactly the points in S .

In the following sections, we first discuss the most famous of all triangulations, the Delaunay triangulation. We then address triangulations of polygons and polyhedra in \mathbb{R}^3 . Finally we describe a recent generalization of triangulations: the pseudo-triangulation.

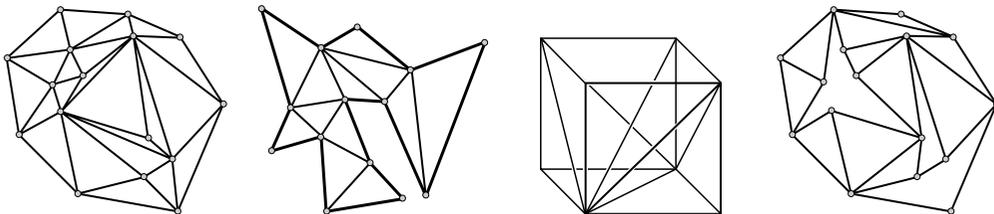


FIGURE 62.8: Triangulations of a point set, a simple polygon, and a polyhedron; a pseudo-triangulation of a point set.

62.5.1 Delaunay Triangulation

In this section we provide additional detail on the Delaunay triangulation of a set $S = \{s_1, \dots, s_n\}$ of points in \mathbb{R}^d , which was introduced in Section 62.4. There we defined the Delaunay triangulation as the dual of the Voronoi diagram. In the plane, for instance, the Delaunay triangulation has an edge between sites s_i and s_j if and only if the Voronoi cells of s_i and s_j share a boundary edge. In higher dimensions, there is an edge between s_i and s_j

if their Voronoi cells share a $(d - 1)$ -facet. The Delaunay triangulation can also be defined directly. If we restrict ourselves for the moment to a set S of points in \mathbb{R}^2 then the Delaunay triangulation of S , $DT(S)$, is defined by the *empty-circle condition*: a triangle Δ defined by three points s_i , s_j , and s_k is part of $DT(S)$ if and only if Δ 's circumcircle neither encloses nor passes through any other points of S . More generally, $d + 1$ points in \mathbb{R}^d define a simplex in the Delaunay triangulation if and only if its circumscribed sphere neither encloses nor passes through any other points of S . If no $d + 1$ points of S are co-spherical then $DT(S)$ is indeed a triangulation. If $d + 2$ or more points are co-spherical, then $DT(S)$ can contain cells with more than $d + 1$ sides. Fortunately, such cells can easily be further decomposed in simplices—with a bottom-vertex triangulation, for example—so such degeneracies are not a real problem. To simplify the description, we from now on assume that these degeneracies do not occur.

In the previous section we have seen a close connection between Voronoi diagrams in \mathbb{R}^d and intersections of half-spaces in \mathbb{R}^{d+1} . Similarly, there is a close connection between the Delaunay triangulation in \mathbb{R}^d and convex hulls in \mathbb{R}^{d+1} . Let's again restrict ourselves to the case $d = 2$. Consider the transformation that maps a site $s = (s_x, s_y)$ in \mathbb{R}^2 onto the point $\lambda(s_x, s_y) = (s_x, s_y, s_x^2 + s_y^2)$ in \mathbb{R}^3 . In other words, s is “lifted” vertically onto the unit paraboloid to obtain $\lambda(s)$. Let $\lambda(S)$ be the set of lifted sites. Then if we project the lower convex hull of $\lambda(S)$ —the part of the convex hull consisting of the facets facing downward—back onto the xy -plane, we get the Delaunay triangulation of S .

The Delaunay triangulation is the “best” triangulation with respect to many optimality criteria. The Delaunay triangulation:

- minimizes the maximum radius of a circumcircle;
- maximizes the minimum angle;
- maximizes the sum of inscribed circle radii;
- minimizes the “potential energy” of a piecewise-linear interpolating surface.

Also, the distance between any two vertices of the Delaunay triangulation along the triangulation edges is at most 2.42 times their Euclidean distance, that is, the *dilation* of the Delaunay triangulation is 2.42. Finally, the Delaunay triangulation contains as a subgraph many other interesting graphs:

$$EMST \subseteq RNG \subseteq GG \subseteq DT$$

where $EMST$ is the Euclidean minimum spanning tree, RNG is the relative neighborhood graph, and GG is the Gabriel graph (see [49] for details on these graphs).

Since the Delaunay triangulation is the dual of the Voronoi diagram, any algorithm presented in Section 62.4.2 can be used to efficiently compute the Delaunay triangulation. We therefore refrain from presenting any additional algorithms at this point.

62.5.2 Polygons

Triangulating a simple polygon P is not only an interesting problem in its own right, but it is also an important preprocessing step for many algorithms. For example, many shortest-path and visibility problems on a polygon P can be solved in linear time if a triangulation of P is given [27]. It is fairly easy to show that any polygon can indeed be decomposed into triangles by adding a number of diagonals and, moreover, that the number of triangles in any triangulation of a simple polygon with n vertices is $n - 2$.

There are many algorithms to compute a triangulation of a simple polygon. Most of them run in $O(n \log n)$ time. Whether it is possible to triangulate a polygon in linear time

was a prominent open problem for several years until Chazelle [13], after a series of interim results, devised a linear-time algorithm. Unfortunately, his algorithm is more of theoretical than of practical interest, so it is probably advisable to use a deterministic algorithm with $O(n \log n)$ running time, such as the one described below, or one of the slightly faster randomized approaches with a time complexity of $O(n \log^* n)$ [39].

In the remainder of this section we sketch a deterministic algorithm that triangulates a simple polygon P with n vertices in $O(n \log n)$ time. We say that a polygon P is *monotone* with respect to a line ℓ if the intersection of any line ℓ' perpendicular to ℓ with P is connected. A polygon that is monotone with respect to the x -axis is called *x -monotone*. Now the basic idea is to decompose P into monotone polygons and then to triangulate each of these monotone polygons in linear time.

There are several methods to decompose a simple polygon into x -monotone polygons in $O(n \log n)$ time. One approach is to sweep over P twice, from left to right and then from right to left, and to add appropriate edges to vertices that did not previously have at least one edge extending to the left and at least one edge extending to the right. A more detailed description of this or related approaches can be found in [7, 24].

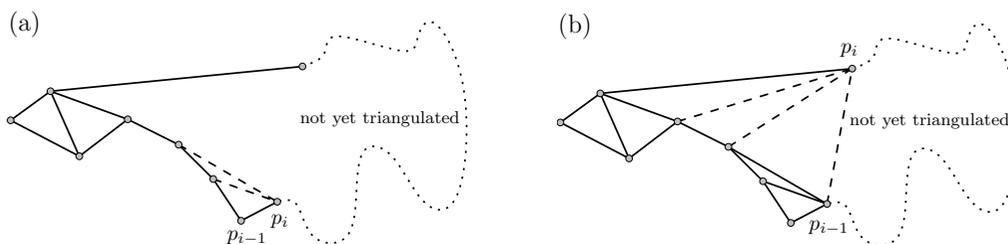


FIGURE 62.9: Triangulating an x -monotone polygon.

Triangulating a monotone polygon. Now suppose we are given an x -monotone polygon P —see Fig. 62.9. We consider its vertices p_1, \dots, p_n from left to right and use a stack to store the vertices of the not-yet-triangulated part of the polygon (which necessarily form a reflex chain) to the left of our current vertex p_i . If p_i is adjacent to p_{i-1} , as in see Fig. 62.9(a), then we pop vertices from the stack and connect them to p_i until the stack (including p_i) forms a reflex chain again. In particular that might mean that we simply add p_i to the stack. If p_i is adjacent to the leftmost vertex on the stack, which could be p_{i-1} , as in see Fig. 62.9(b), then we connect p_i to each vertex of the stack and clear the stack of all vertices except p_i and p_{i-1} . This algorithm triangulates P in linear time.

62.5.3 Polyhedra

In this section we briefly discuss triangulations (or *tetrahedralizations*) of three-dimensional polyhedra. A polyhedron P is a connected solid with a piecewise linear boundary (that is, its boundary is composed of polygons). We assume P to be non-degenerate: A sufficiently small ball around any point of the boundary of P contains a connected component of the interior as well as the exterior of P . The number of vertices, edges, and faces of a non-degenerate tetrahedron are linearly related.

Three dimensions unfortunately do not behave as nicely as two. Two triangulations of the same input data may contain quite different numbers of tetrahedra. For example,

a triangulation of a convex polyhedron with n vertices may contain between $n - 3$ and $\binom{n-2}{2}$ tetrahedra. Furthermore, some non-convex polyhedra can not be triangulated at all without the use of additional (so-called *Steiner*) points. The most famous example is *Schönhardt's* polyhedron. Finally, it is NP-complete to determine whether a polyhedron can be triangulated without Steiner points and to test whether k Steiner points are sufficient [46].

Chazelle [12] constructed a polyhedron with n vertices that requires $\Omega(n^2)$ Steiner points. On the other hand, any polyhedron can be triangulated with $O(n^2)$ tetrahedra, matching his lower bound. If one pays special attention to the reflex vertices of the input polyhedron P , then it is even possible to triangulate P using only $O(n + r^2)$ tetrahedra, where r is the number of reflex edges of P [15].

62.5.4 Pseudo-Triangulations

In recent years a relaxation of triangulations, called *pseudo-triangulations*, has received considerable attention. Here, faces are bounded by three concave chains, rather than by three line segments. More formally, a pseudo-triangle is a planar polygon that has exactly three convex vertices with internal angles less than π , all other vertices are concave. Note that every triangle is a pseudo-triangle as well. The three convex vertices of a pseudo-triangle are called its *corners*. Three concave chains, called *sides*, join the three corners. A pseudo-triangulation for a set S of points in the plane is a partition of the convex hull of S into pseudo-triangles whose vertex set is exactly S —see Fig. 62.8. Pseudo-triangulations, also called *geodesic triangulations*, were originally studied for convex sets and for simple polygons in the plane because of their applications to visibility [40, 41] and ray shooting [16, 25]. But in the last few years they also found application in robot motion planning [55] and kinetic collision detection [1, 35].

Of particular interest are the so-called *minimum pseudo-triangulations*, which have the minimum number of pseudo-triangular faces among all possible pseudo-triangulations of a given domain. They were introduced by Streinu [55], who established that every minimum pseudo-triangulation of a set S of n points consists of exactly $n - 2$ pseudo-triangles (here we do not count the outer face). Note that such a statement cannot be made for ordinary triangulations: there the number of triangles depends on the number of points that show up on the convex hull of S . Minimum pseudo-triangulations are also referred to as *pointed pseudo-triangulations*. The name stems from the fact that every vertex v of a minimum pseudo-triangulation has an incident region whose angle at v is greater than π . The converse is also true (and can be easily established via Euler's relation): If every vertex of a pseudo-triangulation is *pointed*—it has an incident angle greater than π —then this pseudo-triangulation has exactly $n - 2$ pseudo-triangles and is therefore minimum. A pseudo-triangulation is called *minimal* (as opposed to minimum) if the union of any two faces is not a pseudo-triangle. In general, all minimum pseudo-triangulations are also minimal, but the opposite is not necessarily true—see Figure 62.10(a) for an example of a minimal but not minimum pseudo-triangulation.

The great variety of applications in which pseudo-triangulations are successfully employed prompted a growing interest in their geometric and combinatoric properties, which often deviate substantially from those of ordinary triangulations. An example of a nice property of pseudo-triangulations is that every point set in the plane admits a pseudo-triangulation of maximum vertex degree 5 [33]. (This bound is tight.) Again, this is not the case for ordinary triangulations: any ordinary triangulation of the point set depicted in Figure 62.10(b), contains a vertex of degree $n - 1$.

Up to now there are unfortunately no extensions of pseudo-triangulations to dimensions higher than two which retain a significant subset of their useful planar properties.

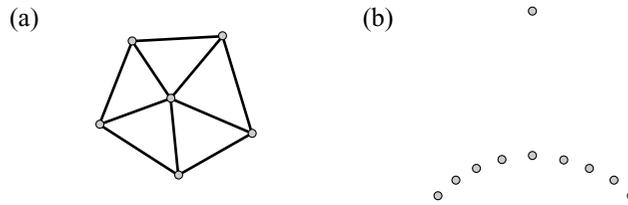


FIGURE 62.10: (a) A minimal but not minimum pseudo-triangulation. (b) A point set for which every triangulation has a vertex of high degree.

References

- [1] P. K. Agarwal, J. Basch, L. J. Guibas, J. Hershberger, and L. Zhang. Deformable free space tilings for kinetic collision detection. *Int. J. Robotics Research* 21(3):179–197, 2002.
- [2] P. K. Agarwal, M. de Berg, J. Matousek, and O. Schwartzkopf. Constructing levels in arrangements and higher order Voronoi diagrams. In *Proc. 11th Symp. Comput. Geom.*, pages 71–78, 1995.
- [3] F. Aurenhammer. Power diagrams: properties, algorithms, and applications. *Siam J. Comput.*, 16:78–96, 1987.
- [4] F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. In *ACM Comput. Surv.*, 23:345–405, 1991.
- [5] F. Aurenhammer and R. Klein. Voronoi diagrams. In J. Sack and G. Urrutia, editors, *Handbook of Computational Geometry*, Chapter V, pages 201–290. Elsevier Science Publishing, 2000.
- [6] S. Basu. On the combinatorial and topological complexity of a single cell. In *Proc. 39th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 606–616, 1998.
- [7] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwartzkopf. *Computational Geometry: Algorithms and Applications (2nd Ed.)*. Springer-Verlag, 2000.
- [8] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, 1998.
- [9] G. S. Brodal and R. Jacob. Dynamic Planar Convex Hull. In *Proc. 43rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 617–626, 2002.
- [10] T. Chan. Output-sensitive results on convex hulls, extreme points, and related problems. *Discr. Comput. Geom.* 16:369–387 (1996).
- [11] T. Chan. Dynamic planar convex hull operations in near-logarithmic amortized time. *J. ACM.* 48:1–12 (2001).
- [12] B. Chazelle. Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. *SIAM J. Comput.*, 13:488–507, 1984.
- [13] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete Comput. Geom.*, 6:485–524, 1991.
- [14] B. Chazelle. An optimal convex hull algorithm in any fixed dimension. *Discr. Comput. Geom.* 10:377–409 (1993).
- [15] B. Chazelle and L. Palios. Triangulating a nonconvex polytope. *Discrete Comput. Geom.*, 5:505–526, 1990.
- [16] B. Chazelle, H. Edelsbrunner, M. Grigni, L. J. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12:54–68, 1994.
- [17] K. L. Clarkson and P.W. Shor. Applications of random sampling in computational geometry, II. *Discr. Comput. Geom.* 4:387–421 (1989).

- [18] R. A. Dwyer. On the convex hull of random points in a polytope. *J. Appl. Probab.* 25:688–699 (1988).
- [19] R. A. Dwyer. Kinder, gentler average-case analysis for convex hulls and maximal vectors. *SIGACT News* 21:64–71 (1990).
- [20] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, 1987.
- [21] H. Edelsbrunner. The upper envelope of piecewise linear functions: Tight complexity bounds in higher dimensions. *Discr. Comput. Geom.* 4:337–343 (1989).
- [22] S. Fortune. Voronoi Diagrams and Delaunay Triangulations. In: F.K. Hwang and D.-Z. Du (eds.), *Computing in Euclidean Geometry, 2nd Edition*. World Scientific, pages 225–265, 1995.
- [23] S. Fortune. A sweepline algorithms for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.
- [24] J. E. Goodman and J. O’Rourke (Eds.). *Handbook of Discrete and Computational Geometry*. CRC Press LLC, 1997.
- [25] M. Goodrich and R. Tamassia. Dynamic ray shooting and shortest paths in planar subdivision via balanced geodesic triangulations. *J. Algorithms* 23:51–73, 1997.
- [26] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inf. Proc. Lett.* 1:132–133 (1972).
- [27] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.
- [28] L. J. Guibas, M. Sharir, and S. Sifrony. On the general motion planning problem with two degrees of freedom. *Discr. Comput. Geom.* 4:491–521 (1989).
- [29] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph.*, 4:74–123, 1985.
- [30] D. Halperin and M. Sharir. New bounds for lower envelopes in three dimensions, with applications to visibility in terrains. *Discr. Comput. Geom.* 12:313–326 (1994).
- [31] J. Hershberger and S. Suri. Applications of semi-dynamic convex hull algorithms. *BIT* 32:249–267 (1992).
- [32] J. Hershberger and S. Suri. Off-line maintenance of planar configurations. *J. Algorithms* 21:453–475 (1996).
- [33] L. Kettner, D. Kirkpatrick, A. Mantler, J. Snoeyink, B. Speckmann, and F. Takeuchi. Tight degree bounds for pseudo-triangulations of points. *Comp. Geom. Theory Appl.*, 25:1–12, 2003.
- [34] D. Kirkpatrick and R. Seidel. The ultimate planar convex hull algorithm? *SIAM J. Comput.* 15:287–299 (1986).
- [35] D. Kirkpatrick, J. Snoeyink, and B. Speckmann. Kinetic collision detection for simple polygons. *Int. J. Comput. Geom. Appl.* 12(1&2):3–27, 2002.
- [36] C. L. Lawson. Software for C^1 surface interpolation. In J.R. Rie, editor, *Math. Software III*, pages 161–194, New York, NY, 1977. Academic Press.
- [37] P. McMullen. The maximal number of faces of a convex polytope. *Mathematica* 17:179–184 (1970).
- [38] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoret. Comput. Sci.*, 7:217–236, 1978.
- [39] K. Mulmuley. *Computational Geometry: an Introduction through Randomized Algorithms*. Prentice Hall, 1994.
- [40] M. Pocchiola and G. Vegter. Minimal tangent visibility graphs. *Comput. Geom. Theory Appl.*, 6:303–314, 1996.
- [41] M. Pocchiola and G. Vegter. Topologically sweeping visibility complexes via pseudo-triangulations *Discrete Comp. Geom.*, 16:419–453, 1996.

- [42] F. P. Preparata. An optimal real-time algorithm for planar convex hulls. *Comm. ACM* 22:405–408 (1979).
- [43] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Comm. ACM* 20:87–93 (1977).
- [44] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams (2nd Ed.)*. Wiley, 2000.
- [45] J. O’Rourke. *Computational Geometry in C (2nd Ed.)*. Cambridge University Press, 1998.
- [46] J. Rupert and R. Seidel. On the difficulty of tetrahedralizing 3-dimensional non-convex polyhedra. *Discrete Comput. Geom.*, 7:227–253, 1992.
- [47] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comput. Sys. Sci.* 23:166–204 (1981)
- [48] R. Pollack, M. Sharir, and S. Sifrony. Separating two simple polygons by a sequence of translations. *Discr. Comput. Geom.* 3:123–136 (1988).
- [49] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.
- [50] J.-R. Sack and J. Urrutia (Eds.). *Handbook of Computational Geometry*. Elsevier Science, 2000.
- [51] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discr. Comput. Geom.* 6:423–434 (1991).
- [52] M. I. Shamos and D. Hoey. Closest-point problems. In *Proc. 16th IEEE Symp. Found. Comput. Science*, pages 151–162, 1975.
- [53] M. Sharir and P. K. Agarwal. *Davenport-Schinzel Sequences and Their Geometric Applications*. Cambridge University Press, 1995.
- [54] M. Sharir. Almost tight upper bounds for lower envelopes in higher dimensions. *Discr. Comput. Geom.* 12:327–345 (1994).
- [55] I. Streinu. A combinatorial approach to planar non-colliding robot arm motion planning. In *Proc. 41st FOCS*, 2000, pp. 443–453.
- [56] B. Tagansky. A new technique for analyzing substructures in arrangements of piecewise linear surfaces. *Discr. Comput. Geom.* 16:455–479 (1996).

63

Computational Geometry: Proximity and Location

63.1	Introduction.....	63-1
63.2	Point Location.....	63-2
	Kirkpatrick's Algorithm • Slab-Based Methods and Persistent Trees • Separating Chains and Fractional Cascading • Trapezoidal Maps and the History Graph • Worst- and Expected-Case Optimal Point Location	
63.3	Proximity Structures.....	63-9
	Voronoi Diagrams • Delaunay Triangulations • Other Geometric Proximity Structures	
63.4	Nearest Neighbor Searching	63-12
	Nearest Neighbor Searching through Point Location • K-d trees • Other Approaches to Nearest Neighbor Searching • Approximate Nearest Neighbor Searching • Approximate Voronoi Diagrams	
63.5	Sources and Related Material	63-18

Sunil Arya

Hong Kong University of Science and Technology

David M. Mount

University of Maryland

63.1 Introduction

Proximity and location are fundamental concepts in geometric computation. The term *proximity* refers informally to the quality of being close to some point or object. Typical problems in this area involve computing geometric structures based on proximity, such as the Voronoi diagram, Delaunay triangulation and related graph structures such as the relative neighborhood graph. Another class of problems are retrieval problems based on proximity. These include nearest neighbor searching and the related concept of range searching. (See [Chapter 18](#) for a discussion of data structures for range searching.) Instances of proximity structures and proximity searching arise in many fields of applications and in many dimensions. These applications include object classification in pattern recognition, document analysis, data compression, and data mining.

The term *location* refers to the position of a point relative to a geometric subdivision or a given set of disjoint geometric objects. The best known example is the point location problem, in which a subdivision of space into disjoint regions is given, and the problem is to identify which region contains a given query point. This problem is widely used in areas such as computer graphics, geographic information systems, and robotics. Point location is also used as a method for proximity searching, when applied in conjunction with Voronoi diagrams.

In this chapter we will present a number of geometric data structures that arise in the context of proximity and location. The area is so vast that our presentation will be limited to a relatively few relevant results. We will discuss data structures for answering point location queries first. After this we will introduce proximity structures, including Voronoi diagrams and Delaunay triangulations. Our presentation of these topics will be primarily restricted to the plane. Finally, we will present results on multidimensional nearest neighbor searching.

63.2 Point Location

The planar *point location* problem is one of the most fundamental query problems in computational geometry. Consider a *planar straight line graph* S . (See [Chapter 17](#) for details.) This is an undirected graph, drawn in the plane, whose edges are straight line segments that have pairwise disjoint interiors. The edges of S subdivide the plane into (possibly unbounded) polygonal regions, called *faces*. Henceforth, such a structure will be referred to as a *polygonal subdivision*. Throughout, we let n denote the *combinatorial complexity* of S , that is, the total number of vertices, edges and faces. (We shall occasionally abuse notation and use n to refer to the specific number of vertices, edges, or faces of S .) A planar subdivision is a special case of the more general topological concept of a *cell complex* [35], in which vertices, edges, and generally faces of various dimensions are joined together so that the intersection of any two faces is either empty or is a face of lower dimension.

The *point location problem* is to preprocess a polygonal subdivision S in the plane into a data structure so that, given any query point q , the polygonal face of the subdivision containing q can be reported quickly. (In [Figure 63.1\(a\)](#), face A would be reported.) This problem is a natural generalization of the binary search problem in 1-dimensional space, where the faces of the subdivision correspond to the intervals between the 1-dimensional key values. By analogy with the 1-dimensional case, the goal is to preprocess a subdivision into a data structure of size $O(n)$ so that point location queries can be answered in $O(\log n)$ time.

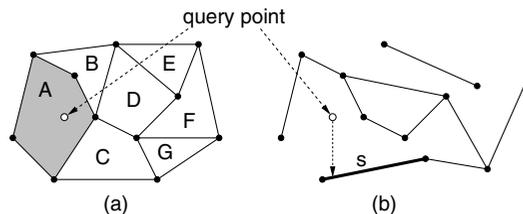


FIGURE 63.1: Illustration of (a) point location and (b) vertical ray shooting queries.

A slightly more general formulation of the problem, which is applicable even when the input is not a subdivision is called *vertical ray shooting*. A set S of line segments is given with pairwise disjoint interiors. Given a query point q , the problem is to determine the line segment of S that lies vertically below q . (In [Figure 63.1\(b\)](#), the segment s would be reported.) If the ray hits no segment, a special value is returned. When S is a polygonal subdivision, point location can be reduced to vertical ray shooting by associating each edge of S with the face that lies immediately above it.

63.2.1 Kirkpatrick's Algorithm

Kirkpatrick was the first to present a simple point location data structure that is asymptotically optimal [52]. It answers queries in $O(\log n)$ time using $O(n)$ space. Although this is not the most practical approach to point location, it is quite easy to understand.

Kirkpatrick starts with the assumption that the planar subdivision has been refined (through the addition of $O(n)$ new edges and vertices) so that it is a triangulation whose external face is a triangle. Let T_0 denote this initial triangulation subdivision. Kirkpatrick's method generates a finite sequence of increasingly coarser triangulations, $\langle T_0, T_1, T_2, \dots, T_m \rangle$, where T_m consists of the single triangle forming the outer face of the original triangulation. This sequence satisfies the following constraints: (a) each triangle of T_{i+1} intersects a constant number of triangles of T_i , and (b) the number of vertices of T_{i+1} is smaller than the number of vertices of T_i by a constant fraction. (See Figure 63.2.)

The data structure itself is a rooted DAG (directed acyclic graph), where the root of the structure corresponds to the single triangle of T_m , and the leaves correspond to the triangles of T_0 . The interior nodes of the DAG correspond to the triangles of each of the triangulations. A directed edge connects each triangle in T_{i+1} with each triangle in T_i that it overlaps.

Given a query point q , the point location query proceeds level-by-level through the DAG, visiting the nodes corresponding to the triangles that contain q . By property (a), each triangle in T_{i+1} overlaps a constant number of triangles of T_i , which implies that it is possible to descend one level in the data structure in $O(1)$ time. It follows that the running time is proportional to the number of levels in the tree. By property (b), the number of vertices decreases at each level by a fixed constant fraction, and hence, the number of levels is $O(\log n)$. Thus the overall query time is $O(\log n)$.

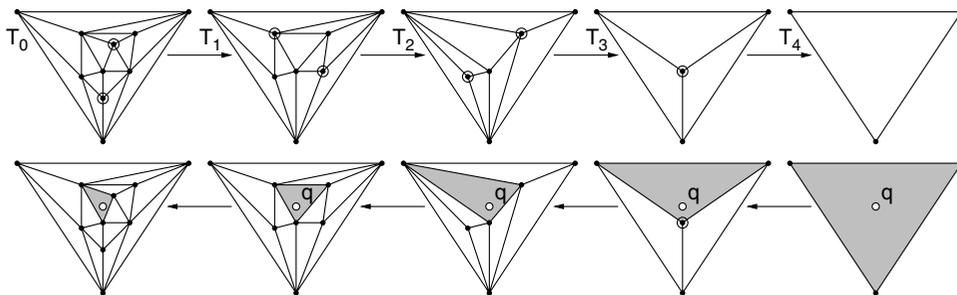


FIGURE 63.2: The sequence of triangulations generated in the construction of Kirkpatrick's structure (above) and the triangles visited in answering a point location query (below).

Kirkpatrick showed how to build the data structure by constructing a sequence of triangulations satisfying the above properties. Kirkpatrick's approach is to compute an *independent set* of vertices (that is, a set of mutually nonadjacent vertices) in T_i where each vertex of the independent set has constant degree. (An example is shown at the top of Figure 63.2. The vertices of the independent set are highlighted.) The three vertices of the outer face are not included. Kirkpatrick showed that there exists such a set whose size is a constant fraction of the total number of vertices, and it can be computed in linear time. These vertices are removed along with any incident edges, and the resulting "holes" are then retriangulated. Kirkpatrick showed that the two properties hold for the resulting sequence of triangulations.

63.2.2 Slab-Based Methods and Persistent Trees

Many point location methods operate by refining the given subdivision to form one that is better structured, and hence, easier to search. One approach for generating such a refinement is to draw a vertical line through each vertex of the subdivision. These lines partition the plane into a collection of $O(n)$ *vertical slabs*, such that there is no vertex within each slab. As a result, the intersection of the subdivision with each slab consists of a set of line segments, which cut clear through the slab. These segments thus partition the slab into a collection of disjoint trapezoids with vertical sides. (See Figure 63.3.)

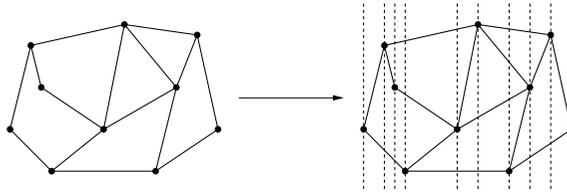


FIGURE 63.3: Slab refinement of a subdivision.

Point location queries can be answered in $O(\log n)$ time by applying two binary searches. The first search accesses the query point's x coordinate to determine the slab containing the query point. The second binary search tests whether the query point lies above or below individual lines of the slab, in order to determine which trapezoid contains the query point. Since each slab can be intersected by at most n lines, this second search can be done in $O(\log n)$ time as well.

A straightforward implementation of this method is not space efficient, since there are $\Omega(n)$ slabs,* each having up to $\Omega(n)$ intersecting segments, for a total of $\Omega(n^2)$ space. However, adjacent slabs are very similar, since the only segments that change are those that are incident to the vertices lying on the slab boundary. Sarnak and Tarjan [67] exploited this idea to produce an optimal point location data structure. To understand their algorithm, imagine sweeping a line segment continuously from left to right. Consider the sorted order of subdivision line segments intersecting this sweep line. Whenever the sweep line encounters a vertex of the subdivision, the edges incident to this vertex lying to the left of the vertex are removed from the sweep-line order and incident edges to the right of the vertex are inserted. Since every edge is inserted once and deleted once in this process, the total number of changes over the entire sweep process is $O(n)$.

Sarnak and Tarjan proposed maintaining a persistent variant of the search tree. A *persistent search tree* is a dynamic search tree (supporting insertion and deletion) which can answer queries not only to the current tree, but to any of the previous versions in the history of the tree's lifetime as well. (See Chapter 31.) In this context, the history of changes to the search tree is maintained in a left to right sweep of the plane. The persistent search tree supports queries to any of these trees, that is, in any of the slabs, in $O(\log n)$ time. The clever aspect of Sarnak and Tarjan's tree is that it can be stored in $O(n)$ total space

*For readers unfamiliar with this notation, $\Omega(f(n))$ is analogous to the notation $O(f(n))$, but it provides an asymptotic lower bound rather than an upper bound. The notation $\Theta(f(n))$ means that both upper and lower bounds apply [30].

(as opposed to $O(n^2)$ space, which would result by generating $O(n)$ copies of the tree). This is done by a method called *limited node copying*. Thus, this provides an asymptotically optimal point location algorithm. A similar approach was discovered independently by Cole [29].

63.2.3 Separating Chains and Fractional Cascading

Slab methods use vertical lines to help organize the search. An alternative approach, first suggested by Lee and Preparata [55], is to use a divide-and-conquer approach based on a hierarchy of monotone polygon chains, called *separating chains*. A simple polygon is said to be *x-monotone* if the intersection of the interior of the polygon with a vertical line is connected. An *x-monotone subdivision* is one in which all the faces are *x-monotone*. The separating chain method requires that the input be an *x-monotone subdivision*. Fortunately, it is possible to convert any polygonal subdivision in the plane into an *x-monotone subdivision* in $O(n \log n)$ time, through the addition of $O(n)$ new edges. (See, for example, [31, 55, 64].) For example, Figure 63.4(a) shows a subdivision that is not *x-monotone*, but the addition of two edges suffice to produce an *x-monotone subdivision* shown in Figure 63.4(b).

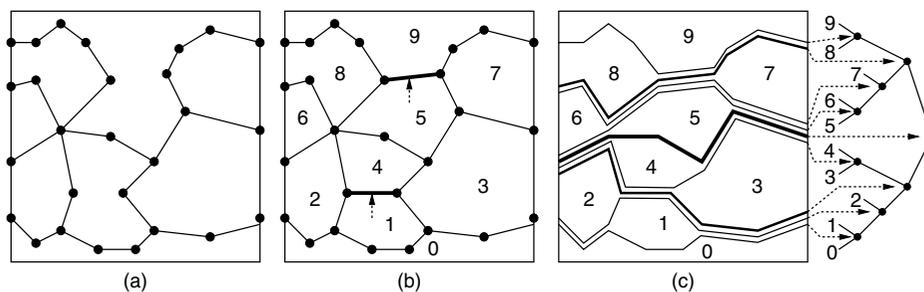


FIGURE 63.4: Point location by separating chains: (a) the original subdivision, (b) the addition of one or more edges to make the subdivision *x-monotone*, (c) decomposition of the subdivision into a hierarchy of separating chains.

Consider an *x-monotone* subdivision with n faces. It is possible to order the faces f_0, f_1, \dots, f_{n-1} such that if $i < j$, then every vertical line that intersects both of these faces intersects f_i below f_j . (See Figure 63.4(b).) For each i , $0 < i < n$, define the i th separating chain to be the *x-monotone* polygonal chain separating faces whose indices are less than i from those that are greater than or equal to i .

Observe that, given a chain with m edges, it is possible to determine whether a given query point lies above or below the chain in $O(\log m)$ time, by first performing a binary search on the x -coordinates of the chain, in order to find which chain edge overlaps the query point, and then determining whether the query point lies above or below this edge in $O(1)$ time. The separating chain method works intuitively by performing a binary search on these chains. The binary search can be visualized as a binary tree imposed on the chains, as shown in Figure 63.4(c).

Although many chains traverse the same edge, it suffices to store each edge only once in the structure, namely with the chain associated with the highest node in the binary tree. This is because once a discrimination of the query point is made with respect to such an

edge, its relation is implicitly known for all other chains that share the same edge. It follows that the total space is $O(n)$.

As mentioned above, at each chain the search takes logarithmic time to determine whether the query point is above or below the chain. Since there are $\Omega(n)$ chains, this would lead to an $\Omega(\log^2 n)$ algorithm [55]. There is a clever way to reduce the search time to $O(\log n)$, through the use of a simple and powerful method called *fractional cascading* [24, 36]. Intuitively, fractional cascading seeks to replace a sequence of independent binary searches with a more efficient sequence of coordinated searches. After searching through a parent's chain, it is known which edge of this chain the query point overlaps. Thus, it is not necessary to search the entire range of x -coordinates for the child's chain, just the sublist of x -coordinates that overlap this interval.

However, in general, the number of edges of the child's chain that overlaps this interval may be as large as $\Omega(n)$, and so this observation would seem to be of no help. In fractional cascading, this situation is remedied by augmenting each list. Starting with the leaf level, the x -coordinate of every fourth vertex is passed up from each child's sorted list of x -coordinates and inserted into its parent's list. This is repeated from the parent to the grandparent, and so on. After doing this, once the edge of the parent's chain that overlaps the query point has been determined, there can be at most four edges of the child's chain that overlap this interval. (For example, in Figure 63.5 the edge \overline{pq} is overlapped by eight edges at the next lower level. After cascading, it is broken into three subedges, each of which overlaps at most four edges at the next level.) Thus, the overlapping edge in the child's chain can be found in $O(1)$ time. The root requires $O(\log n)$ time, and each of the subsequent $O(\log n)$ searches can be performed in $O(1)$ additional time. It can be shown that this augmentation of the lists increases the total size of all the lists by at most a constant factor, and hence the total space is still $O(n)$.

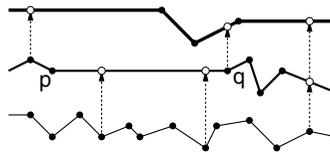


FIGURE 63.5: Example of fractional cascading. Every fourth vertex is sampled from each chain and inserted in its parent's chain.

63.2.4 Trapezoidal Maps and the History Graph

Next we describe a randomized approach for point location. It is quite simple and practical. Let us assume that the planar subdivision is presented simply as a set of n line segments $S = \{s_1, s_2, \dots, s_n\}$ with pairwise disjoint interiors. The algorithm answers vertical ray-shooting queries as described earlier. This approach was developed by Mulmuley [60]. Also see Seidel [68].

The algorithm is based on a structure called a *trapezoidal map* (or *trapezoidal decomposition*). First, assume that the entire domain of interest is enclosed in a large rectangle. Imagine shooting a bullet vertically upwards and downwards from each vertex in the polygonal subdivision until it hits another segment of S . To simplify the presentation, we shall assume that the x -coordinates of no two vertices are identical. The segments of S together

with the resulting bullet paths subdivide the plane into $O(n)$ trapezoidal cells with vertical sides, which may degenerate to triangles. (See Figure 63.6(a).)

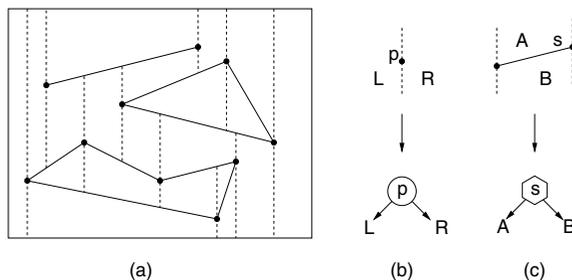


FIGURE 63.6: A trapezoidal map of a set of segments (a), and the two types of internal nodes: x -node (b) and y -node (c).

For the purposes of point location, the trapezoidal map is created by a process called a *randomized incremental construction*. The process starts with the initial bounding rectangle (that is, one trapezoid) and then the segments of S are inserted one by one in random order. As each segment is added, the trapezoidal map is updated by “walking” the segment through the subdivision, and updating the map by shooting new bullet paths through the segment endpoints and trimming existing paths that hit the new segment. See [31, 60, 68] for further details. The number of changes in the diagram with each insertion is proportional to the number of vertical segments crossed by the newly added segment, which in the worst case may be as high as $\Omega(n)$. It can be shown, however, that on average each insertion of a new segment results in $O(1)$ changes. This is true irrespective of the distribution of the segments, and the expectation is taken over all possible insertion orders.

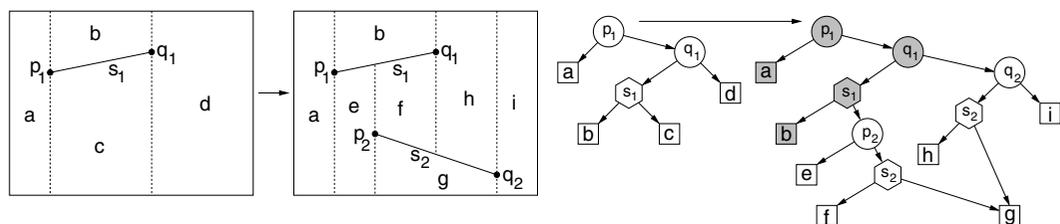


FIGURE 63.7: Example of incremental construction of a trapezoidal map and the associated history DAG. The insertion of segment s_2 replaces the leaves associated with destroyed trapezoids c and d with an appropriate search structure for the new trapezoids $e-i$.

The point location data structure is based on a rooted directed acyclic graph, or DAG, called the *history DAG*. Each node has either two outgoing edges (internal nodes) or none (leaves). Leaves correspond one-to-one with the cells of the trapezoidal map. There are two types of internal nodes, x -nodes and y -nodes. Each x -node contains the x -coordinate x_0 of an endpoint of one of the segments, and its two children correspond to the points lying to the left and to the right of the vertical line $x = x_0$. Each y -node contains a pointer to

a line segment of the subdivision. The left and right children correspond to whether the query point is above or below the line containing this segment, respectively. (In Figure 63.7, x -nodes are shown as circles, y -nodes as hexagons, and leaves as squares.)

As with Kirkpatrick's algorithm, the construction of the point location data structure encodes the history of the randomized incremental construction. Let $\langle T_0, T_1, \dots, T_n \rangle$ denote the sequence of trapezoidal maps that result through the randomized incremental process. The point location structure after insertion of the i th segment has one leaf for each trapezoid in T_i . Whenever a segment is inserted, the leaf nodes corresponding to trapezoids that were destroyed are replaced with internal x - and y -nodes that direct the search to the location of the query point in the newly created trapezoids, after the insertion. (This is illustrated in Figure 63.7.) Through the use of node sharing, the resulting data structure can be shown to have expected size $O(n)$, and its expected depth is $O(\log n)$, where the expectation is over all insertion orders. Details can be found in [31, 60, 68].

63.2.5 Worst- and Expected-Case Optimal Point Location

Goodrich, Orletsky and Ramaiyer [43] posed the question of bounding the minimum number of comparisons required, in the worst case, to answer point location queries in a subdivision of n segments. Adamy and Seidel [1] provided a definitive answer by showing that point location queries can be answered with $\log_2 n + 2\sqrt{\log_2 n} + o(\sqrt{\log n})$ primitive comparisons. They also gave a similar lower bound.

Another natural question involves the expected-case complexity of point location. Given a polygonal subdivision S , assume that each cell $z \in S$ is associated with the probability p_z that a query point lies in z . The problem is to produce a point location data structure whose expected search time is as low as possible. The appropriate target bound on the number of comparisons is given by the *entropy* of the subdivision, which is denoted by H and defined:

$$\text{entropy}(S) = H = \sum_{z \in S} p_z \log_2(1/p_z).$$

In the 1-dimensional case, a classical result due to Shannon implies that the expected number of comparisons needed to answer such queries is at least as large as the entropy of the probability distribution [53, 71]. Mehlhorn [58] showed that in the 1-dimensional case it is possible to build a binary search tree whose expected search time is at most $H + 2$.

Arya, Malamatos, and Mount [5, 6] presented a number of results on this problem in the planar case, and among them they showed that for a polygonal subdivision of size n in which each cell has constant combinatorial complexity, it is possible to answer point location queries with $H + o(H)$ comparisons in the expected case using space that is nearly linear in n . Their results also applied to subdivisions with convex cells, assuming the query distribution is uniform within each cell. Their approach was loosely based on computing a binary space partition (BSP) tree (see Chapter 20) satisfying two properties:

- (a) The entropy of the subdivision defined by the leaves of the BSP should be close to the entropy of the original subdivision.
- (b) The depth of a leaf should be close to $\log_2(1/p)$, where p is the probability that a query point lies within the leaf.

Arya, Malamatos, and Mount [7] also presented a simple weighted variant of the randomized incremental algorithm and showed that it can answer queries in $O(H)$ expected time and $O(n)$ space. Iacono [48] presented a deterministic weighted variant based on Kirkpatrick's algorithm.

63.3 Proximity Structures

Proximity structures arise from numerous applications in science and engineering. It is a fundamental fact that nearby objects tend to exert a greater influence and have greater relevance than more distant objects. Proximity structures are discrete geometric and graph structures that encode proximity information. We discuss a number of such structures, including Voronoi diagrams, Delaunay triangulations, and various geometric graph structures, such as the relative neighborhood graph.

63.3.1 Voronoi Diagrams

The *Voronoi diagram* of a set of sites S is a partition of space into regions, one per site, where the region for site s is the set of points that are closer to s than to any other site of S . This structure has been rediscovered and applied in many different branches of science and goes by various names, including Thiessen diagrams and Dirichlet tessellations.

Henceforth, we consider the most common case in which the sites S consist of a set of n points in real d -dimensional space, \mathbb{R}^d , and distances are measured using the Euclidean metric. The set of points of \mathbb{R}^d that are closer to some site $s \in S$ than any other site is called the *Voronoi cell* of s , or $V(s)$. (See Figure 63.8.) The union of the boundaries of the Voronoi cells is the *Voronoi diagram* of S , denoted $Vor(S)$. Observe that the set of points of \mathbb{R}^d that are closer to s than some other site t consists of the points that lie in the open halfspace defined by a plane that bisects the pair (s, t) . It follows that each Voronoi cell is the intersection of $n - 1$ halfspaces, and hence, it is a (possibly unbounded) convex polyhedron. A Voronoi diagram in dimension d is a cell complex whose faces of all dimensions are convex polyhedra. In the plane a Voronoi diagram is a planar straight line graph with possibly unbounded edges. It can be represented using standard methods for representing polygonal subdivisions and cell complexes (see [Chapter 17](#)).

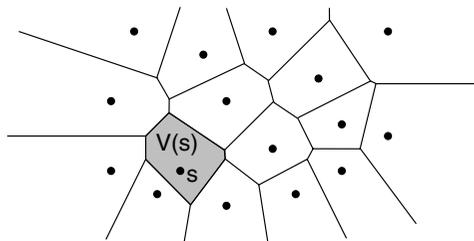
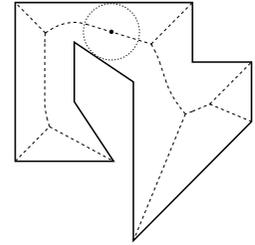


FIGURE 63.8: The Voronoi diagram and a Voronoi cell $V(s)$.

The Voronoi diagram possesses a number of useful geometric properties. For example, for a set of points in the plane, each edge of the Voronoi diagram lies on the perpendicular bisector between two sites. The vertices of the Voronoi diagram lie at the center of an empty circle passing through the incident sites. If the points are in general position (and in particular if no four points are cocircular) then every vertex of the diagram is incident to exactly three edges. In fact, it is not hard to show that the largest empty circle whose center lies within the convex hull of a given point set will coincide with a Voronoi vertex. In higher dimensions, each face of dimension k of the Voronoi diagram consists of the points of \mathbb{R}^d that are equidistant from a subset of $d - k + 1$ sites, and all other sites are strictly

farther away. In the plane the combinatorial complexity of the Voronoi diagram is $O(n)$, and in dimension d its complexity is $\Theta(n^{\lceil d/2 \rceil})$.

Further information on algorithms for constructing Voronoi diagrams as well as variants of the Voronoi diagram can be found in [Chapter 62](#). Although we defined Voronoi diagrams for point sites, it is possible to define them for any type of geometric object. One such variant involves replacing point sites with line segments or generally the boundary of any region of the plane. Given a region P (e.g., a simple polygon), the *medial axis* is defined to be the set of centers of maximal balls contained in P , that is, balls contained in P that are not contained in another ball in P [32]. The medial axis is frequently used in pattern recognition and shape matching. It consists of a combination of straight-line segments and hyperbolic arcs. It can be computed in $O(n \log n)$ time by a modification of Fortune's swepline algorithm [39]. Finally, it is possible to generalize Voronoi diagrams to other metrics, such as the L_1 and L_∞ metrics (see [Section 63.4](#)).



63.3.2 Delaunay Triangulations

The Delaunay triangulation is a structure that is closely related to the Voronoi diagram. The Delaunay triangulation is defined as follows for a set S of n point sites in the plane. Consider any subset $T \subseteq S$ of sites, such that there exists a circle that passes through all the points of T , and contains no point of S in its interior. Such a subset is said to satisfy the *empty circumcircle property*. For example, in [Figure 63.9\(a\)](#), the pair $\{p, q\}$ and triple $\{r, s, t\}$ both satisfy the empty circumcircle property. The *Delaunay triangulation* is defined to be the union of the convex hulls of all such subsets. It can be shown that the result is a cell complex. Furthermore, if the points are in general position, and in particular, no four points are cocircular, then the resulting structure is a triangulation of S . (If S is not in general position, then some faces may have more than three edges, and it is common to *complete* the triangulation by triangulating each such face.) A straightforward consequence of the above definition is that the Delaunay triangulation is dual to the Voronoi diagram. For example, [Figure 63.9\(b\)](#) shows the overlay of these two structures in the plane.

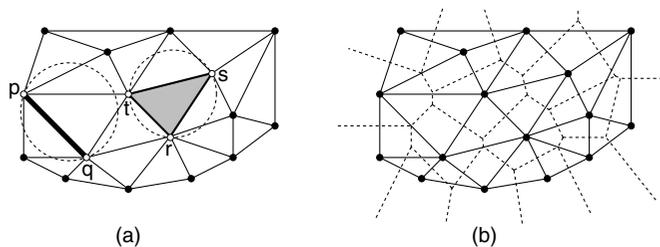


FIGURE 63.9: (a) The Delaunay triangulation of a set of points and (b) its overlay with the Voronoi diagram.

Delaunay triangulations are widely used in practice, and they possess a number of useful properties. For example, among all triangulations of a planar point set the Delaunay triangulation maximizes the minimum angle. Also, in all dimensions, the Euclidean minimum

spanning tree (defined below) is a subgraph of the Delaunay triangulation. Proofs of these facts can be found in [31].

In the plane the Delaunay triangulation of a set of points has $O(n)$ edges and $O(n)$ faces. The above definition can be generalized to arbitrary dimensions. In dimension d , the Delaunay triangulation can have as many as $\Theta(n^{\lceil d/2 \rceil})$ faces. However, it can be much smaller. In particular, Dwyer [34] has shown that in any fixed dimension, if n points are drawn from a uniform distribution from within a unit ball, then the expected number of simplices is $O(n)$.

There is an interesting connection between Delaunay triangulations in dimension d and convex hulls in dimension $d + 1$. Consider the *lifting map* $f : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ defined $f(x, y) = (x, y, x^2 + y^2)$. This projects points in the plane onto the paraboloid $z = x^2 + y^2$. Given a planar point set S , let S' denote the set of points of \mathbb{R}^3 that results by applying this map to each point of S . Define the lower hull of S' to be the set of faces whose outward pointing normal has a negative z coordinate. It can be shown that, when projected back to the plane, the edges of the lower convex hull of S' are exactly the edges of the Delaunay triangulation of S . (See Figure 63.10.)

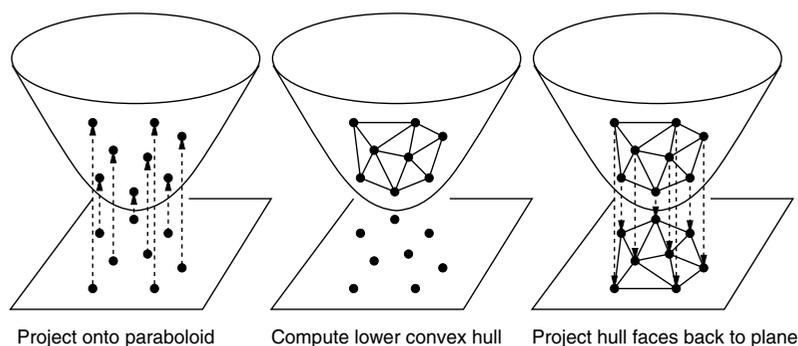


FIGURE 63.10: The Delaunay triangulation can be computed by lifting the points to the paraboloid, computing the lower convex hull, and projecting back to the plane.

Although there exist algorithms specially designed for computing Delaunay triangulations, the above fact makes it possible to compute Delaunay triangulations in any dimension by computing convex hulls in the next higher dimension. There exist $O(n \log n)$ time algorithms for computing planar Delaunay triangulations, for example, based on divide-and-conquer [70] and plane sweep [39]. Perhaps the most popular method is based on randomized incremental point insertion [45]. In dimension $d \geq 3$, Delaunay triangulations can be computed in $O(n^{\lceil d/2 \rceil})$ time through randomized incremental point insertion [27].

63.3.3 Other Geometric Proximity Structures

The Delaunay triangulation is perhaps the best known example of a proximity structure. There are a number of related graph structures that are widely used in pattern recognition, learning, and other applications. Given a finite set S of points in d -dimensional Euclidean space, we can define a graph on these points by joining pairs of points that satisfy certain neighborhood properties. In this section we will consider a number of such neighborhood graphs.

Let us first introduce some definitions. For $p, q \in \mathbb{R}^d$ let $\text{dist}(p, q)$ denote the Euclidean distance from p to q . Given positive $r \in \mathbb{R}$, let $B(p, r)$ be the open ball consisting of points whose distance from point p is strictly less than r . Define the *lune*, denoted $L(p, q)$, to be the intersection of two balls both of radius $\text{dist}(p, q)$ centered at these points, that is,

$$L(p, q) = B(p, \text{dist}(p, q)) \cap B(q, \text{dist}(p, q)).$$

The following geometric graphs are defined for a set S consisting of n points in \mathbb{R}^d . (See Figure 63.11.)

Nearest Neighbor Graph (NNG): The directed graph containing an edge (p, q) if q is the nearest neighbor of p , that is, $B(p, \text{dist}(p, q)) \cap S = \emptyset$.

Euclidean Minimum Spanning Tree (EMST): This is an undirected spanning tree on S that minimizes the sum of the Euclidean edge lengths.

Relative Neighborhood Graph (RNG): The undirected graph containing an edge (p, q) if there is no point $r \in S$ that is simultaneously closer to p and q than $\text{dist}(p, q)$ [74]. Equivalently, (p, q) is an edge if $L(p, q) \cap S = \emptyset$.

Gabriel Graph (GG): The undirected graph containing an edge (p, q) if the ball whose diameter is \overline{pq} does not contain any other points of S [42], that is, if

$$B\left(\frac{p+q}{2}, \frac{\text{dist}(p, q)}{2}\right) \cap S = \emptyset.$$

Delaunay Graph (DT): The 1-skeleton (edges) of the Delaunay triangulation.

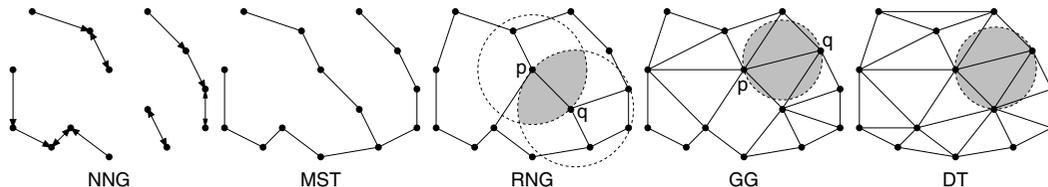


FIGURE 63.11: Common geometric graphs on a point set.

These graphs form an interesting hierarchical relationship. If we think of each edge of an undirected graph as consisting of two directed edges, then we have the following hierarchical relationship, which was first established in [74]. Also see [50].

$$\text{NNG} \subseteq \text{MST} \subseteq \text{RNG} \subseteq \text{GG} \subseteq \text{DT}.$$

This holds in all finite dimensions and generalizes to Minkowski (L_m) metrics, as well.

63.4 Nearest Neighbor Searching

Nearest neighbor searching is an important problem in a variety of applications, including knowledge discovery and data mining, pattern recognition and classification, machine learning, data compression, multimedia databases, document retrieval, and statistics. We

are given a set S of objects in some space to be preprocessed, so that given a query object q , the closest object (or objects) of S can be reported quickly.

There are many ways in which to define the notion of similarity. Because the focus of this chapter is on geometric approaches, we shall assume that proximity is defined in terms of the well known Euclidean distance. Most of the results to be presented below can be generalized to any *Minkowski* (or L_m) *metric*, in which the distance between two points \mathbf{p} and \mathbf{q} is defined to be

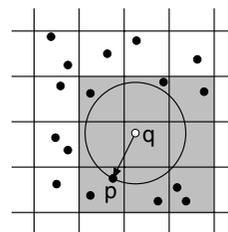
$$\text{dist}_m(\mathbf{p}, \mathbf{q}) = \left(\sum_{i=1}^d |p_i - q_i|^m \right)^{1/m}$$

where $m \geq 1$ is a constant. The case $m = 2$ is the Euclidean distance, the case $m = 1$ is the Manhattan distance, and the limiting case $m = \infty$ is the max distance. In typical geometric applications the dimension d is assumed to be a fixed constant. There has also been work on high dimensional proximity searching in spaces of arbitrarily high dimensions [49] and in arbitrary (nongeometric) metric spaces [23], which we shall not cover here.

There are a number of natural extensions to the nearest neighbor problem as described above. One is to report the k nearest neighbors to the query point, for some given integer k . Another is to compute all the points lying within some given distance, that is, a range query in which the range is defined by the distance function.

Obviously, without any preprocessing whatsoever, the nearest neighbor search problem can be solved in $O(n)$ time through simple brute-force search. A number of very simple methods have been proposed which assume minimal preprocessing. For example, points can be sorted according to their projection along a line, and the projected distances can be used as a method to prune points from consideration [40, 44, 54]. These methods are only marginally effective, and provide significant improvements over brute-force search only in very low dimensions.

For uniformly distributed point sets, good expected case performance can be achieved by simple decompositions of space into a regular grid of hypercubes. Rivest [65] and later Cleary [28] provided analyses of these methods. Bentley, Weide, and Yao [17] also analyzed a grid-based method for distributions satisfying certain bounded-density assumptions. Intuitively, the points are bucketed into grid cells, where the size of the grid cell is based on the expected distance to the nearest neighbor. To answer a query, the grid cell containing the query point is located, and a spiral-like search working outwards from this cell is performed to identify nearby points. Suppose for example that q is the query point and p is its closest neighbor. Then all the grid cells overlapping a ball centered at q of radius $\text{dist}(p, q)$ would be visited.



Grids are easy to implement, since each bucket can be stored as a simple list of points, and the complete set of buckets can be arranged in a multi-dimensional array. Note that this may not be space efficient, since it requires storage for empty cells. A more space-efficient method is to assign a hash code to each grid cell based on its location, and then store only the nonempty grid buckets in a hash table. In general, grid methods do not work well for nearest neighbor search unless the point distribution is roughly uniform. As will be discussed below, more sophisticated methods are needed to achieve good efficiency for nonuniformly distributed data.

63.4.1 Nearest Neighbor Searching through Point Location

One of the original motivations for the Voronoi diagram is nearest neighbor searching. By definition, the Voronoi diagram subdivides space into cells according to which site is the closest. So, in order to determine the closest site, it suffices to compute the Voronoi diagram and generate a point location data structure for the Voronoi diagram. In this way, nearest neighbor queries are reduced to point location queries. This provides an optimal $O(n)$ space and $O(\log n)$ query time method for answering point location queries in the plane. Unfortunately, this solution does not generalize well to higher dimensions. The worst-case combinatorial complexity of the Voronoi diagram in dimension d grows as $\Theta(n^{\lceil d/2 \rceil})$, and optimal point location data structures are not known to exist in higher dimensions.

63.4.2 K-d trees

Perhaps the most popular class of approaches to nearest neighbor searching involves some sort of hierarchical spatial subdivision. Let S denote the set of n points in \mathbb{R}^d for which queries are to be answered. In such an approach, the entire space is subdivided into successively smaller regions, and the resulting hierarchy is represented by a rooted tree. Each node of the tree represents a region of space, called a *cell*. Implicitly, each node represents the subset of points of S that lie within its cell. The root of the tree is associated with the entire space and the entire point set S . For some arbitrary node u of the tree, if the number of points of S associated with u is less than some constant, then this node is declared to be a leaf of the tree. Otherwise, the cell associated with u is recursively subdivided into smaller (possibly overlapping) subcells according to some *splitting rule*. Then the associated points of S are distributed among these children according to which subcell they lie in. These subcells are then associated with the children of u in the tree.

There are many ways in which to define such a subdivision. Perhaps the earliest and best known example is that of the k-d tree data structure. Bentley [16] introduced the *k-d tree* data structure (or *kd-tree*) as a practical general-purpose data structure for many types of geometric retrieval problems. Although it is not the asymptotically most efficient solution for these problems, its flexibility makes it a popular choice for implementation. The cells of a k-d tree are axis-aligned hyperrectangles. Each internal node is associated with an axis-orthogonal splitting hyperplane. This hyperplane splits the rectangular cell into two rectangular subcells, each of which is associated with one of the two children. An example is shown in Figure 63.12.

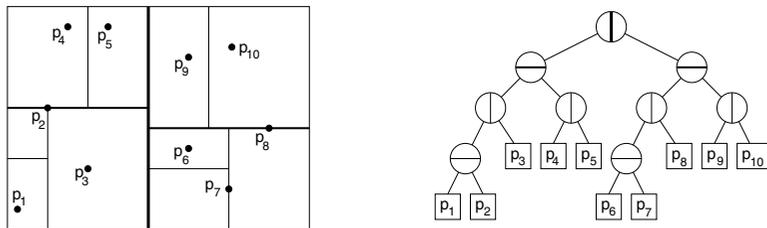


FIGURE 63.12: An example of a k-d tree of a set of points in the plane, showing both the associated spatial subdivision (left) and the binary tree structure (right).

The choice of the splitting hyperplane is an important issue in the implementation of the

k-d tree. For the purpose of nearest neighbor searching, a good split is one that divides the points into subsets of similar cardinalities and which produces cells that are not too skinny, that is, the ratio between the longest and shortest sides is bounded. However, it is not always possible to achieve these goals. A simple and commonly used method is to cycle through the various coordinate axes (that is, splitting along x , then y , then z , then back to x , and so on). Each time the split is made through the median coordinate along the splitting dimension [31, 66]. Friedman, Bentley and Finkel [41] suggested the following method, which is more sensitive to the data distribution. First, compute the minimum axis-aligned bounding box for the set of points associated with the current cell. Next choose the splitting axis to be the one that is parallel to the longest side of this box. Finally, split the points by a hyperplane that is orthogonal to this axis, and which splits the points into two sets of equal size. A number of other splitting rules have been proposed for k-d trees, including the sliding midpoint rule by Arya and Fu [3] and Maneewongvatana and Mount [57], variance minimization by White and Jain [76], and methods by Silva Filho [37] and Sproull [73]. We will discuss other subdivision methods in the next section as well.

It is possible to construct the k-d tree of an n -element point set in $O(n \log n)$ time by a simple top-down recursive procedure. The process involves determining the splitting axis and the splitting coordinate along this axis, and then partitioning the point set about this coordinate. If the splitting rule partitions the point set about its median coordinate then it suffices to compute the median by any linear-time algorithm for computing medians [30]. Some splitting methods may not evenly partition the point set. In the worst case this can lead to quadratic construction time. Vaidya showed that it is possible to achieve $O(n \log n)$ construction time, even when unbalanced splitting takes place [75]. The total space requirements are $O(n)$ for the tree itself.

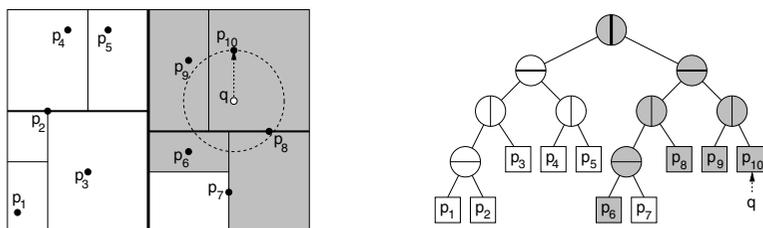


FIGURE 63.13: Nearest neighbor search in a k-d tree. The point p_{10} is the initial closest, and only the shaded cells and nodes are visited. The final answer is p_8 .

Given a query point q , a nearest neighbor search is performed by the following recursive algorithm [41]. Throughout, the algorithm maintains the closest point to q encountered so far in the search, and the distance to this closest point. As the nodes of the tree are traversed, the algorithm maintains the d -dimensional hyperrectangular cell associated with each node. (This is updated incrementally as the tree is traversed.) When the search arrives at a leaf node, it computes the distance from q to the associated point(s) of this node, and updates the closest point if necessary. (See Figure 63.13.) Otherwise, when it arrives at an internal node, it first computes the distance from the query point to the associated cell. If this distance is greater than the distance to the closest point so far, the search returns immediately, since the subtree rooted at this node cannot provide a closer point. Otherwise, it is determined which side of the splitting hyperplane contains the query point. First, the

closer child is visited and then the farther child. A somewhat more intelligent variant of this method, called *priority search*, involves storing the unvisited nodes in a priority queue, sorted according to the distance from the query point to the associated cell, and then processes the nodes in increasing order of distance from the query point [9].

63.4.3 Other Approaches to Nearest Neighbor Searching

The k-d tree is but one example of a general class of nearest neighbor search structures that are based on hierarchical space decomposition. A good survey of methods from database literature was given by Böhm, Berchtold, and Keim [20]. These include the R-tree [46] and its variants, the R*-tree [15], the R⁺-tree [69], and the X-tree [18], which are all based on recursively decomposing space into (possibly overlapping) hyperrectangles. (See Chapter 21 for further information.) For the cases studied, the X-tree is reported to have the best performance for nearest neighbor searching in high dimensional spaces [20]. The SS-tree [76] is based on subdividing space using (possibly overlapping) hyperspheres rather than rectangles. The SR-tree [51] uses the intersection of an enclosing rectangle and enclosing sphere to represent a cell. The TV-tree [56] applies a novel approach of considering projections of the data set onto higher dimensional subspaces at successively deeper levels in the search tree.

A number of algorithms for nearest neighbor searching have been proposed in the algorithms and computational geometry literature. Higher dimensional solutions with sublinear worst-case performance were considered by Yao and Yao [77]. Clarkson [25] showed that queries could be answered in $O(\log n)$ time with $O(n^{\lceil d/2 \rceil + \delta})$ space, for any $\delta > 0$. The O -notation hides constant factors that are exponential in d . Agarwal and Matoušek [2] generalized this by providing a tradeoff between space and query time. Meiser [59] showed that queries could be answered in $O(d^5 \log n)$ time and $O(n^{d+\delta})$ space, for any $\delta > 0$, thus showing that exponential factors in query time could be eliminated by using sufficient space.

63.4.4 Approximate Nearest Neighbor Searching

In any fixed dimensions greater than two, no method for exact nearest neighbor searching is known that achieves the simultaneous goals of roughly linear space and logarithmic query time. For methods achieving roughly linear space, the constant factors hidden in the asymptotic running time grow at least as fast as 2^d (depending on the metric). Arya *et al.* [11] showed that if n is not significantly larger than 2^d , then boundary effects decrease this exponential dimensional dependence. Nonetheless, the so called “curse of dimensionality” is a significant impediment to computing nearest neighbors efficiently in high dimensional spaces.

This suggests the idea of computing nearest neighbors approximately. Consider a set of points S and a query point q . For any $\epsilon > 0$, we say that a point $p \in S$ is an ϵ -*approximate nearest neighbor* of q if

$$\text{dist}(p, q) \leq (1 + \epsilon)\text{dist}(p^*, q),$$

where p^* is the true nearest neighbor of q in S . The approximate nearest neighbor problem was first considered by Bern [19]. He proposed a data structure that achieved a fixed approximation factor depending on dimension. Arya and Mount [10] proposed a randomized data structure that achieves polylogarithmic query time in the expected case, and nearly linear space. Their approach was based on a combination of the notion of neighborhood graphs, as described in Section 63.3.3, and skip lists. In their algorithm the approximation error factor ϵ is an arbitrary positive constant, which is given at preprocessing time.

Arya *et al.* [12] proposed a hierarchical spatial subdivision data structure, called the *BBD-tree*. This structure has the nice features of having $O(n)$ size, $O(\log n)$ depth, and each cell has bounded aspect ratio, that is, the ratio between its longest and shortest side is bounded. They achieved this by augmenting the axis-aligned splitting operation of the *k-d tree* (see Figure 63.14(a)) with an additional subdivision operation called *shrinking* (see Figure 63.14(b)). A shrinking node is associated with an axis-aligned rectangle, and the two children correspond to the portions of space lying inside and outside of this rectangle, respectively. The resulting cells are either axis-aligned hyperrectangles, or the set-theoretic difference of two axis-aligned hyperrectangles. They showed that, in all fixed dimensions d and for $\epsilon > 0$, it is possible to answer ϵ -nearest neighbor queries in $O(\log n)$ time using the *BBD-tree*. The hidden asymptotic constants in query time grow as $(1/\epsilon)^d$.

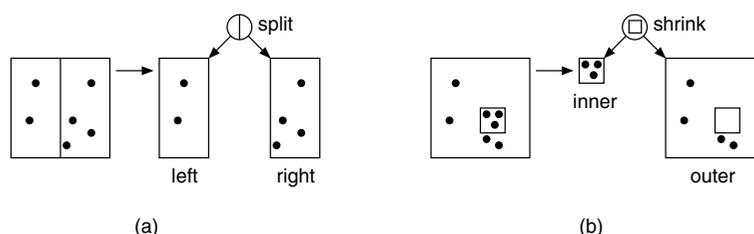


FIGURE 63.14: Splitting nodes (a) and shrinking nodes (b) in a BBD-tree.

Duncan *et al.* [33] proposed an alternative structure, called the *BAR tree*, which achieves all of these combinatorial properties and has convex cells. The *BAR tree* achieves this by using cutting planes that are not necessarily axis-aligned. Clarkson [26] and Chan [22] presented data structures that achieved better ϵ dependency in the query time. In particular, they showed that queries could be answered in $O((1/\epsilon)^{d/2} \log n)$ time.

63.4.5 Approximate Voronoi Diagrams

As mentioned in Section 63.4.1 it is possible to answer nearest neighbor queries by applying a point location query to the Voronoi diagram. However, this approach does not generalize well to higher dimensions, because of the rapid growth rate of the Voronoi diagram and the lack of good point location structures in dimension higher than two.

Har-Peled [47] proposed a method to overcome these problems. Given an error bound $\epsilon > 0$, an *approximate Voronoi diagram* (AVD) of a point set S is defined to be a partition of space into cells, where each cell c is associated with a *representative* $r_c \in S$, such that r_c is an ϵ -nearest neighbor for all the points in c [47]. Arya and Malamatos [4] generalized this by allowing up to some given number $t \geq 1$ representatives to be stored with each cell, subject to the requirement that for any point in the cell, one of these t representatives is an ϵ -nearest neighbor. Such a decomposition is called a (t, ϵ) -AVD. (See Figure 63.15.)

Of particular interest are AVDs that are constructed from hierarchical spatial decompositions, such as quadtrees and their variants, since such structures support fast point location in all dimensions. This yields a very simple method for performing approximate nearest neighbor searching. In particular, a tree descent determines the leaf cell containing the query point and then the closest of the t representatives is reported.

Har-Peled [47] showed that it is possible to construct a $(1, \epsilon)$ AVD in which the number of

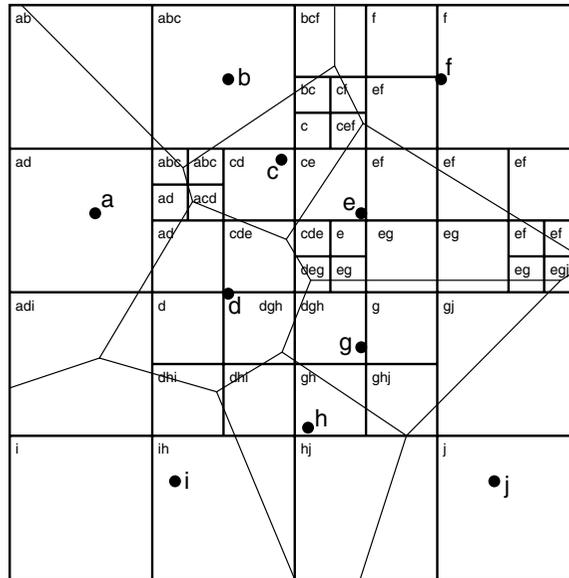


FIGURE 63.15: A $(3,0)$ -AVD implemented as a quadtree subdivision for the set $\{a, b, \dots, j\}$. Each cell is labeled with its representatives. The Voronoi diagram is shown for reference.

leaf cells is $O((n/\epsilon^d)(\log n) \log(n/\epsilon))$. Arya and Malamatos [4] and later Arya, Malamatos, and Mount [8] improved these results by showing how to construct more space-efficient AVDs. In all constant dimensions d , their results yield a data structure of $O(n)$ space (including the space for representatives) that can answer ϵ -nearest neighbor queries in $O(\log n + (1/\epsilon)^{(d-1)/2})$ time. This is the best asymptotic result known for approximate nearest neighbor searching in fixed dimensional spaces.

63.5 Sources and Related Material

General information regarding the topics presented in the chapter can be found in standard texts on computational geometry, including those by Preparata and Shamos [64], Edelsbrunner [35], Mulmuley [61], de Berg *et al.* [31], and Boissonnat and Yvinec [21] as well as Samet's book on spatial data structures [66]. Further information on point location can be found in a survey paper written by Snoeyink [72]. For information on Voronoi diagrams see the book by Okabe, Boots and Sugihara [62] or surveys by Aurenhammer [13], Aurenhammer and Klein [14], and Fortune [38]. For further information on geometric graphs see the survey by O'Rourke and Toussaint [63]. Further information on nearest neighbor searching can be found in surveys by Böhm *et al.* [20], Indyk [49], and Chavez *et al.* [23].

Acknowledgments

The work of the first author was supported in part by a grant from the Hong Kong Research Grants Council (HKUST6229/00E and HKUST6080/01E). The work of the second author was supported in part by National Science Foundation grant number CCR-0098151.

References

- [1] U. Adamy and R. Seidel. Planar point location close to the information-theoretic lower bound. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, 1998.
- [2] P. K. Agarwal and J. Matoušek. Ray shooting and parametric search. *SIAM J. Comput.*, 22(4):794–806, 1993.
- [3] S. Arya and H. Y. Fu. Expected-case complexity of approximate nearest neighbor searching. *SIAM J. of Comput.*, 32:793–815, 2003.
- [4] S. Arya and T. Malamatos. Linear-size approximate Voronoi diagrams. In *Proc. 13th ACM-SIAM Sympos. Discrete Algorithms*, pages 147–155, 2002.
- [5] S. Arya, T. Malamatos, and D. M. Mount. Nearly optimal expected-case planar point location. In *Proc. 41 Annu. IEEE Sympos. Found. Comput. Sci.*, pages 208–218, 2000.
- [6] S. Arya, T. Malamatos, and D. M. Mount. Entropy-preserving cuttings and space-efficient planar point location. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, 2001. 256–261.
- [7] S. Arya, T. Malamatos, and D. M. Mount. A simple entropy-based algorithm for planar point location. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, pages 262–268, 2001.
- [8] S. Arya, T. Malamatos, and D. M. Mount. Space-efficient approximate Voronoi diagrams. In *Proc. 34th Annual ACM Sympos. Theory Comput.*, pages 721–730, 2002.
- [9] S. Arya and D. M. Mount. Algorithms for fast vector quantization. In *Data Compression Conference*, pages 381–390. IEEE Press, 1993.
- [10] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.
- [11] S. Arya, D. M. Mount, and O. Narayan. Accounting for boundary effects in nearest-neighbor searching. *Discrete Comput. Geom.*, 16:155–176, 1996.
- [12] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45:891–923, 1998.
- [13] F. Aurenhammer. Voronoi diagrams: A survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, September 1991.
- [14] F. Aurenhammer and R. Klein. Voronoi diagrams. In J.-R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, pages 201–290. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 2000.
- [15] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 322–331, 1990.
- [16] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.
- [17] J. L. Bentley, B. W. Weide, and A. C. Yao. Optimal expected-time algorithms for closest point problems. *ACM Trans. on Math. Softw.*, 6(4):563–580, 1980.
- [18] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. 22nd VLDB Conference*, pages 28–39, 1996.
- [19] M. Bern. Approximate closest-point queries in high dimensions. *Inform. Process. Lett.*, 45:95–99, 1993.
- [20] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surveys*, 33:322–373, 2001.
- [21] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press,

- UK, 1998. Translated by H. Brönnimann.
- [22] T. Chan. Approximate nearest neighbor queries revisited. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 352–358, 1997.
 - [23] E. Chavez, G. Navarro, R. A. Baeza-Yates, and J. L. Marroquin. Searching in metric spaces. *ACM Comput. Surveys*, 33:273–321, 2001.
 - [24] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(3):133–162, 1986.
 - [25] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Comput.*, 17(4):830–847, 1988.
 - [26] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 160–164, 1994.
 - [27] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry II. *Discrete Comput. Geom.*, 4:387–421, 1989.
 - [28] J. G. Cleary. Analysis of an algorithm for finding nearest neighbors in Euclidean space. *ACM Trans. on Math. Softw.*, 5(2):183–192, 1979.
 - [29] R. Cole. Searching and storing similar lists. *J. Algorithms*, 7:202–220, 1986.
 - [30] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.
 - [31] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.
 - [32] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. Wiley-Interscience, New York, 1973.
 - [33] C. A. Duncan, M. T. Goodrich, and S. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. *J. Algorithms*, 38:303–333, 2001.
 - [34] R. A. Dwyer. Higher-dimensional Voronoi diagrams in linear expected time. *Discrete Comput. Geom.*, 6:343–367, 1991.
 - [35] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, West Germany, 1987.
 - [36] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15(2):317–340, 1986.
 - [37] Y. V. Silva Filho. Optimal choice of discriminators in a balanced K-D binary search tree. *Inform. Proc. Lett.*, 13:67–70, 1981.
 - [38] S. Fortune. Voronoi diagrams and Delaunay triangulations. In Jacob E. Goodman and Joseph O’Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 20, pages 377–388. CRC Press LLC, Boca Raton, FL, 1997.
 - [39] S. J. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2:153–174, 1987.
 - [40] J. H. Friedman, F. Baskett, and L. J. Shustek. An algorithm for finding nearest neighbors. *IEEE Trans. Comput.*, C-24(10):1000–1006, 1975.
 - [41] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, 1977.
 - [42] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18:259–278, 1969.
 - [43] M. T. Goodrich, M. Orletsky, and K. Ramaiyer. Methods for achieving fast query times in point location data structures. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 757–766, 1997.
 - [44] L. Guan and M. Kamel. Equal-average hyperplane partitioning method for vector quantization of image data. *Pattern Recogn. Lett.*, 13:693–699, 1992.

- [45] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992.
- [46] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 47–57, 1984.
- [47] S. Har-Peled. A replacement for Voronoi diagrams of near linear size. In *Proc. 42 Annu. IEEE Sympos. Found. Comput. Sci.*, pages 94–103, 2001.
- [48] J. Iacono. Optimal planar point location. In *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, pages 340–341, 2001.
- [49] P. Indyk. Nearest neighbors in high-dimensional spaces. In Jacob E. Goodman and Joseph O’Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press LLC, Boca Raton, FL, 2004. (To appear).
- [50] J. W. Jaromczyk and G. T. Toussaint. Relative neighborhood graphs and their relatives. *Proc. IEEE*, 80(9):1502–1517, September 1992.
- [51] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 369–380, 1997.
- [52] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12(1):28–35, 1983.
- [53] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, second edition, 1998.
- [54] C.-H. Lee and L.-H. Chen. Fast closest codeword search algorithm for vector quantisation. *IEE Proc.-Vis. Image Signal Process.*, 141:143–148, 1994.
- [55] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6(3):594–606, 1977.
- [56] K. I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, 1994.
- [57] S. Maneewongvatana and D. M. Mount. Analysis of approximate nearest neighbor searching with clustered point sets. In M. H. Goldwasser, C. C. McGeoch, and D. S. Johnson, editors, *Data Structures, Near Neighbor Searches, and Methodology*, volume 59 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 105–123. American Mathematics Society, 2002.
- [58] K. Mehlhorn. Best possible bounds on the weighted path length of optimum binary search trees. *SIAM J. Comput.*, 6:235–239, 1977.
- [59] S. Meiser. Point location in arrangements of hyperplanes. *Information and Computation*, 106(2):286–303, 1993.
- [60] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbolic Comput.*, 10(3–4):253–280, 1990.
- [61] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, Englewood Cliffs, NJ, 1993.
- [62] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Chichester, UK, 1992.
- [63] J. O’Rourke and G. T. Toussaint. Pattern recognition. In Jacob E. Goodman and Joseph O’Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 43, pages 797–814. CRC Press LLC, Boca Raton, FL, 1997.
- [64] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, 3rd edition, October 1990.
- [65] R. L. Rivest. On the optimality of Elias’s algorithm for performing best-match searches. In *Information Processing*, pages 678–681. North Holland Publishing Company, 1974.
- [66] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

- [67] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 29(7):669–679, July 1986.
- [68] R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Comput. Geom. Theory Appl.*, 1(1):51–64, 1991.
- [69] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+ -tree: A dynamic index for multi-dimensional objects. In *Proc. 13th VLDB Conference*, pages 507–517, 1987.
- [70] M. I. Shamos and D. Hoey. Closest-point problems. In *Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 151–162, 1975.
- [71] C. E. Shannon. A mathematical theory of communication. *Bell Sys. Tech. Journal*, 27:379–423, 623–656, 1948.
- [72] J. Snoeyink. Point location. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 30, pages 559–574. CRC Press LLC, Boca Raton, FL, 1997.
- [73] R. F. Sproull. Refinements to nearest-neighbor searching in k -dimensional trees. *Algorithmica*, 6(4):579–589, 1991.
- [74] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recogn.*, 12:261–268, 1980.
- [75] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbors problem. *Discrete Comput. Geom.*, 4:101–115, 1989.
- [76] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. 12th IEEE Internat. Conf. Data Engineering*, pages 516–523, 1996.
- [77] A. C. Yao and F. F. Yao. A general approach to d -dimensional geometric queries. In *Proc. 17th Ann. ACM Sympos. Theory Comput.*, pages 163–168, 1985.

Computational Geometry: Generalized Intersection Searching

	64.1	Geometric Intersection Searching Problems	64-1
		Generalized Intersection Searching	
	64.2	Summary of Known Results	64-3
		Axes-Parallel Objects • Arbitrarily-Oriented Objects • Problems on the Grid • Single-Shot Problems	
	64.3	Techniques	64-4
		A Transformation-Based Approach • A Sparsification-Based Approach • A Persistence-Based Approach • A General Approach for Reporting Problems • Adding Range Restrictions	
	64.4	Conclusion and Future Directions	64-15
	64.5	Acknowledgment	64-16
Prosenjit Gupta <i>International Institute of Information Technology</i>			
Ravi Janardan <i>University of Minnesota</i>			
Michiel Smid <i>Carleton University</i>			

64.1 Geometric Intersection Searching Problems

Problems arising in diverse areas, such as VLSI layout design (Chapter 52), database querying (Chapter 60), robotics, and computer graphics (Chapter 54) can often be formulated as *geometric intersection searching problems*. In a generic instance of such a problem, a set, S , of geometric objects is to be preprocessed into a suitable data structure so that given a query object, q , we can answer efficiently questions regarding the intersection of q with the objects in S . The problem comes in four versions, depending on whether we want to report the intersected objects or simply count their number—the *reporting* version and the *counting* version, respectively—and whether S remains fixed or changes through insertion and deletion of objects—the *static* version and the *dynamic* version, respectively. In the dynamic version, which arises very often owing to the highly interactive nature of the above-mentioned applications, we wish to perform the updates more efficiently than simply recomputing the data structure from scratch after each update, while simultaneously maintaining fast query response times. We call these problems *standard* intersection searching problems in order to distinguish them from the *generalized* intersection searching problems that are the focus of this chapter. Due to their numerous applications, standard intersection searching problems have been the subject of much study and efficient solutions have been devised for many of them (see, for instance, [4, 13] and the references therein).

The efficiency of a standard intersection searching algorithm is measured by the space used by the data structure, the query time, and, in the dynamic setting, the update time. In a counting problem, these are expressed as a function of the input size n (i.e., the size of S); in a reporting problem, the space and update time are expressed as a function of n ,

whereas the query time is expressed as a function of both n and the output size k (i.e., the number of intersected objects) and is typically of the form $O(f(n) + k)$ or $O(f(n) + k \cdot g(n))$, for some functions f and g . Such a query time is called *output-sensitive*.

64.1.1 Generalized Intersection Searching

In many applications, a more general form of intersection searching arises: Here the objects in S come aggregated in disjoint groups and of interest are questions regarding the intersection of q with the groups rather than with the objects. (q intersects a group if and only if it intersects some object in the group.) In our discussion, it will be convenient to associate with each group a different color and imagine that all the objects in the group have that color. Then, in the *generalized reporting* (resp., *generalized counting*) problem, we want to report (resp., count) the distinct colors intersected by q ; in the dynamic setting, an object of some (possibly new) color is inserted in S or an object in S is deleted. Note that the generalized problem reduces to the standard one when each color class has cardinality 1.

We give two examples of such generalized problems: Consider a database of mutual funds which contains for each fund its annual total return and its beta (a real number measuring the fund's volatility). Thus each fund can be represented as a point in two dimensions. Moreover, funds are aggregated into groups according to the fund family they belong to. A typical query is to determine the families that offer funds whose total return is between, say, 15% and 20%, and whose beta is between, say, 0.9 and 1.1. This is an instance of the generalized 2-dimensional range searching problem. The output of this query enables a potential investor to initially narrow his/her search to a few families instead of having to plow through dozens of individual funds (all from the same small set of families) that meet these criteria. As another example, in the Manhattan layout of a VLSI chip, the wires (line segments) can be grouped naturally according to the circuits they belong to. A problem of interest to the designer is determining which circuits (rather than wires) become electrically connected when a new wire is added. This is an instance of the generalized orthogonal segment intersection searching problem.

One approach to solving a generalized problem is to try to take advantage of solutions known for the corresponding standard problem. For instance, we can solve a generalized reporting problem by first determining the objects intersected by q (a standard reporting problem) and then reading off the distinct colors. However, the query time can be very high since q could intersect $\Omega(n)$ objects but only $O(1)$ distinct colors. For a generalized reporting problem, we seek query times that are sensitive to the number, i , of distinct colors intersected, typically of the form $O(f(n) + i)$ or $O(f(n) + i \cdot g(n))$, where f and g are polylogarithmic. (This is attainable using the approach just described if each color class has cardinality $O(1)$. On the other hand, if there are only $O(1)$ different color classes, we could simply run a standard algorithm on each color class in turn, stopping as soon as an intersection is found and reporting the corresponding color. The real challenge is when the number of color classes and the cardinalities of the color classes are not constants, but rather are (unknown) functions of n ; throughout, we will assume this to be the case.) For a generalized counting problem, the situation is worse; it is not even clear how one can extract the answer for such a problem from the answer (a mere count) to the corresponding standard problem. One could, of course, solve the corresponding reporting problem and then count the colors, but this is not efficient. Thus it is clear that different techniques are needed.

In this chapter, we describe the research that has been conducted over the past few years on generalized intersection searching problems. We begin with a brief review of known results and then discuss a variety of techniques for these problems. For each technique,

we give illustrative examples and provide pointers to related work. We conclude with a discussion of possible directions for further research.

64.2 Summary of Known Results

Generalized intersection searching problems were introduced by Janardan and Lopez in [23]. Subsequent work in this area may be found in [2, 3, 6–8, 16, 18–22, 29]. (Some of these results are also reported in two Ph.D. theses [17, 30].) In this section, we give a broad overview of the work on these problems to date; details may be found in the cited references.

64.2.1 Axes-Parallel Objects

In [23], efficient solutions were given for several generalized reporting problems, where the input objects and the query were axes-parallel. Examples of such input/query pairs considered include: points/interval in \mathbb{R}^1 ; line segments/segment, points/rectangle, and rectangles/rectangle, all in \mathbb{R}^2 ; and rectangles/points in \mathbb{R}^d , where $d \geq 2$ is a constant. Several of these results were further extended in [18] to include counting and/or dynamic reporting, and new results were presented for input/query pairs such as intervals/interval in \mathbb{R}^1 , points/quadrant in \mathbb{R}^2 , and points/rectangle in \mathbb{R}^3 . Furthermore, a new type of counting problem, called a *type-2 counting problem* was also introduced, where the goal was to count for each color intersected the number of objects of that color that are intersected. In [6], improved solutions were given for counting and/or reporting problems involving points/interval in \mathbb{R}^1 , points/rectangle in \mathbb{R}^2 , and line segments/segment in \mathbb{R}^2 .

64.2.2 Arbitrarily-Oriented Objects

Efficient solutions were given in [23] for generalized reporting on non-intersecting line segments using a query line segment. Special, but interesting, cases of intersecting line segments, such as when each color class forms a polygon or a connected component, were considered in [3]. Efficient solutions were given in [19] for input/query pairs consisting of points/halfspace in \mathbb{R}^d , points/fat-triangle, and fat-triangles/point in \mathbb{R}^2 . (A *fat-triangle* is a triangle where each internal angle is at least a user-specified constant, hence “well-shaped”.) Some of these results were improved subsequently in [6]. In [20], alternative bounds were obtained for the fat-triangle problems within the framework of a general technique for adding range restriction capability to a generalized data structure. Results were presented in [8] for querying, with a polygon, a set of polygons whose sides are oriented in at most a constant number of different directions, with a polygon. In [30], a general method was given for querying intersecting line segments with a segment and for querying points in \mathbb{R}^d with a halfspace or a simplex. Generalized problems involving various combinations of circular objects (circles, discs, annuli) and points, lines, and line segments were considered in [21].

64.2.3 Problems on the Grid

Problems involving document retrieval or string manipulation can often be cast in the framework of generalized intersection searching. For example, in the context of document retrieval, the following problem (among others) was considered in [29]: Preprocess an array of colored non-negative integers (i.e., points on the 1-dimensional grid) such that, given two indices into the array, each distinct color for which there is a pair of points in the

index range at distance less than a specified constant can be reported efficiently. In the context of substring indexing, the following problem was considered in [16]: Preprocess a set of colored points on the 1-dimensional grid, so that given two non-overlapping intervals, the list of distinct colors that occur in their intersection can be reported efficiently. I/O efficient algorithms were given in the standard external memory model [31] for this problem. Other grid-related work in this area includes [2], where efficient solutions were given for the points/rectangle and rectangles/point problems, under the condition that the input and query objects lie on a d -dimensional grid.

64.2.4 Single-Shot Problems

In this class of problems, we are given a collection of geometric objects and the goal is to report all pairs that intersect. Note that there is no query object as such here and no notion of preprocessing the input. As an example, suppose that we are given a set of convex polygons with a total of n vertices in \mathbb{R}^2 , and we wish to report or count all pairs that intersect, with the goal of doing this in time proportional to the number of intersecting pairs (i.e., output-sensitively). If the number of polygons and their sizes are both functions of n (instead of one or the other being a constant), then, as discussed in [22], standard methods (e.g., testing each pair of polygons or computing all boundary intersections and polygon containments in the input) are inefficient. In [22], an efficient and output-sensitive algorithm was given for this problem. Each polygon was assigned a color and then decomposed into simpler elements, i.e., trapezoids, of the same color. The problem then became one of reporting all distinct color pairs (c_1, c_2) such that a trapezoid of color c_1 intersects one of color c_2 . An improved algorithm was given subsequently in [1] for both \mathbb{R}^2 and \mathbb{R}^3 . Other related work on such colored single-shot problems may be found in [7].

64.3 Techniques

We describe in some detail five main techniques that have emerged for generalized intersection searching over the past few years. Briefly, these include: an approach based on a geometric transformation, an approach based on generating a sparse representation of the input, an approach based on persistent data structures, a generic method that is applicable to any reporting problem, and an approach for searching on a subset of the input satisfying a specified range restriction. We illustrate each method with examples.

64.3.1 A Transformation-Based Approach

We first illustrate a transformation-based approach for the reporting and counting problems, which converts the original generalized reporting/counting problem to an instance of a related standard reporting/counting problem on which efficient known solutions can be brought to bear. We illustrate this approach by considering the generalized 1-dimensional range searching problem. Let S be a set of n colored points on the x -axis. We show how to preprocess S so that for any query interval q , we can solve efficiently the dynamic reporting problem, the static and dynamic counting problems, and the static type-2 counting problem. The solutions for the dynamic reporting problem and the static and dynamic counting problems are from [18]. The type-2 counting solution is from [6].

We first describe the transformation. For each color c , we sort the distinct points of that color by increasing x -coordinate. For each point p of color c , let $pred(p)$ be its predecessor of color c in the sorted order; for the leftmost point of color c , we take the predecessor to

be the point $-\infty$. We then map p to the point $p' = (p, \text{pred}(p))$ in the plane and associate with it the color c . Let S' be the resulting set of points. Given a query interval $q = [l, r]$, we map it to the grounded rectangle $q' = [l, r] \times (-\infty, l)$.

LEMMA 64.1 There is a point of color c in S that is in $q = [l, r]$ if and only if there is a point of color c in S' that is in $q' = [l, r] \times (-\infty, l)$. Moreover, if there is a point of color c in q' , then this point is unique.

Proof Let p' be a c -colored point in q' , where $p' = (p, \text{pred}(p))$ for some c -colored point $p \in S$. Since p' is in $[l, r] \times (-\infty, l)$, it is clear that $l \leq p \leq r$ and so $p \in [l, r]$.

For the converse, let p be the leftmost point of color c in $[l, r]$. Thus $l \leq p \leq r$ and since $\text{pred}(p) \notin [l, r]$, we have $l > \text{pred}(p)$. It follows that $p' = (p, \text{pred}(p))$ is in $[l, r] \times (-\infty, l)$. We prove that p' is the only point of color c in q' . Suppose for a contradiction that $t' = (t, \text{pred}(t))$ is another point of color c in q' . Thus we have $l \leq t \leq r$. Since $t > p$, we also have $\text{pred}(t) \geq p \geq l$. Thus $t' = (t, \text{pred}(t))$ cannot lie in q' —a contradiction. ■

Lemma 64.1 implies that we can solve the generalized 1-dimensional range reporting (resp., counting) problem by simply reporting the points in q' (resp., counting the number of points in q'), without regard to colors. In other words, we have reduced the generalized reporting (resp., counting) problem to the standard grounded range reporting (resp., counting) problem in two dimensions. In the dynamic case, we also need to update S' when S is updated. We discuss these issues in more detail below.

The Dynamic Reporting Problem

Our data structure consists of the following: For each color c , we maintain a balanced binary search tree, T_c , in which the c -colored points of S are stored in increasing x -order. We maintain the colors themselves in a balanced search tree CT , and store with each color c in CT a pointer to T_c . We also store the points of S' in a *balanced priority search tree* (PST) [28]. (Recall that a PST on m points occupies $O(m)$ space, supports insertions and deletions in $O(\log m)$ time, and can be used to report the k points lying inside a grounded query rectangle in $O(\log m + k)$ time [28]. Although this query is designed for query ranges of the form $[l, r] \times (-\infty, l]$, it can be trivially modified to ignore the points on the upper edge of the range without affecting its performance.) Clearly, the space used by the entire data structure is $O(n)$, where $n = |S|$.

To answer a query $q = [l, r]$, we simply query the PST with $q' = [l, r] \times (-\infty, l)$ and report the colors of the points found. Correctness follows from Lemma 64.1. The query time is $O(\log n + k)$, where k is the number of points inside q' . By Lemma 64.1, $k = i$, and so the query time is $O(\log n + i)$.

Suppose that a c -colored point p is to be inserted into S . If $c \notin CT$, then we create a tree T_c containing p , insert $p' = (p, -\infty)$ into the PST , and insert c , with a pointer to T_c , into CT . Suppose that $c \in CT$. Let u be the successor of p in T_c . If u exists, then we set $\text{pred}(p)$ to $\text{pred}(u)$ and $\text{pred}(u)$ to p ; otherwise, we set $\text{pred}(p)$ to the rightmost point in T_c . We then insert p into T_c , $p' = (p, \text{pred}(p))$ into the PST , delete the old u' from the PST , and insert the new u' into it.

Deletion of a point p of color c is essentially the reverse. We delete p from T_c . Then we delete p' from the PST and if p had a successor, u , in T_c then we reset $\text{pred}(u)$ to $\text{pred}(p)$, delete the old u' from the PST , and insert the new one. If T_c becomes empty in the process, then we delete c from CT . Clearly, the update operations are correct and take $O(\log n)$

time.

THEOREM 64.1 *Let S be a set of n colored points on the real line. S can be preprocessed into a data structure of size $O(n)$ such that the i distinct colors of the points of S that are contained in any query interval can be reported in $O(\log n + i)$ time and points can be inserted and deleted online in S in $O(\log n)$ time.*

For the static reporting problem, we can dispense with CT and the T_c 's and simply use a static form of the PST to answer queries. This provides a simple $O(n)$ -space, $O(\log n + i)$ -query time alternative to another solution given in [23].

The static counting problem

We store the points of S' in non-decreasing x -order at the leaves of a balanced binary search tree, T , and store at each internal node t of T an array A_t containing the points in t 's subtree in non-decreasing y -order. The total space is clearly $O(n \log n)$. To answer a query, we determine $O(\log n)$ canonical nodes v in T such that the query interval $[l, r]$ covers v 's range but not the range of v 's parent. Using binary search we determine in each canonical node's array the highest array position containing an entry less than l (and thus the number of points in that node's subtree that lie in q') and add up the positions thus found at all canonical nodes. The correctness of this algorithm follows from Lemma 64.1. The total query time is $O(\log^2 n)$.

We can reduce the query time to $O(\log n)$ as follows: At each node t we create a linked list, B_t , which contains the same elements as A_t and maintain a pointer from each entry of B_t to the same entry in A_t . We then apply the technique of fractional cascading [9] to the B -lists, so that after an initial $O(\log n)$ -time binary search in the B -list of the root, the correct positions in the B -lists of all the canonical nodes can be found directly in $O(\log n)$ total time. (To facilitate binary search in the root's B -list, we build a balanced search tree on it after the fractional cascading step.) Once the position in a B -list is known, the appropriate position in the corresponding A -array can be found in $O(1)$ time.

It is possible to reduce the space slightly (to $O(n \log n / \log \log n)$) at the expense of a larger query time ($O(\log^2 n / \log \log n)$), by partitioning the points of S' recursively into horizontal strips of a certain size and doing binary search, augmented with fractional cascading, within the strips. Details can be found in [18].

THEOREM 64.2 *Let S be a set of n colored points on the real line. S can be preprocessed into a data structure of size $O(n \log n)$ (resp., $O(n \log n / \log \log n)$) such that the number of distinctly-colored points of S that are contained in any query interval can be determined in $O(\log n)$ (resp., $O(\log^2 n / \log \log n)$) time.*

The dynamic counting problem

We store the points of S' using the same basic two-level tree structure as in the first solution for the static counting problem. However, T is now a $BB(\alpha)$ tree [32] and the auxiliary structure, $D(t)$, at each node t of T is a balanced binary search tree where the points are stored at the leaves in left to right order by non-decreasing y -coordinate. To facilitate the querying, each node v of $D(t)$ stores a count of the number of points in its subtree. Given a real number, l , we can determine in $O(\log n)$ time the number of points in $D(t)$ that have y -coordinate less than l by searching for l in $D(t)$ and adding up the count for each node of $D(t)$ that is not on the search path but is the left child of a node on the path. It should

be clear that $D(t)$ can be maintained in $O(\log n)$ time under updates.

In addition to the two-level structure, we also use the trees T_c and the tree CT , described previously, to maintain the correspondence between S and S' . We omit further discussion about the maintenance of these trees.

Queries are answered as in the static case, except that at each auxiliary structure we use the above-mentioned method to determine the number of points with y -coordinate less than l . Thus the query time is $O(\log^2 n)$. (We cannot use fractional cascading here.)

Insertion/deletion of a point is done using the worst-case updating strategy for $BB(\alpha)$ trees, and take $O(\log^2 n)$ time.

THEOREM 64.3 *Let S be a set of n colored points on the real line. S can be preprocessed into a data structure of size $O(n \log n)$ such that the number of distinctly-colored points of S that are contained in any query interval can be determined in $O(\log^2 n)$ time and points can be inserted and deleted online in S in $O(\log^2 n)$ worst-case time.*

The static type-2 problem

We wish to preprocess a set S of n colored points on the x -axis, so that for each color intersected by a query interval $q = [l, r]$, the number of points of that color in q can be reported efficiently. The solution for this problem originally proposed in [18] takes $O(n \log n)$ space and supports queries in $O(\log n + i)$ time. The space bound was improved to $O(n)$ in [6], as follows.

The solution consists of two priority search trees, PST_1 and PST_2 . PST_1 is similar to the priority search tree built on S' in the solution for the dynamic reporting problem, with an additional count stored at each node. Let $p' = (p, \text{pred}(p))$ be the point that is stored at a node in PST_1 and c the color of p . Then at this node, we store an additional number $t_1(p')$, which is the number of points of color c to the right of p .

PST_2 is based on a transformation that is symmetric to the one used for PST_1 . For each color c , we sort the distinct points of that color by increasing x -coordinate. For each point p of color c , let $\text{next}(p)$ be its successor in the sorted order; for the rightmost point of color c , we take the successor to be the point $+\infty$. We then map p to the point $p'' = (p, \text{next}(p))$ in the plane and associate with it the color c . Let S'' be the resulting set of points. We build PST_2 on S'' , with an additional count stored at each node. Let $p'' = (p, \text{next}(p))$ be the point that is stored at a node in PST_2 and c the color of p . Then at this node, we store an additional number $t_2(p'')$, which is the number of points of color c to the right of $\text{next}(p)$.

We also maintain an auxiliary array A of size n . Given a query $q = [l, r]$, we query PST_1 with $q' = [l, r] \times (-\infty, l)$ and for each color c found, we set $A[c] = t_1(p')$, where p' is the point stored at the node where we found c . Then we query PST_2 with $q'' = [l, r] \times (r, +\infty)$ and for each color c found, we report c and $A[c] - t_2(p'')$, where p'' is the point stored at the node where we found c . This works because the queries on PST_1 and PST_2 effectively find the leftmost and rightmost points of color c in $q = [l, r]$ (cf. proof of Lemma 64.1). Thus, $A[c] - t_2(p'')$ gives the number of points of color c in q .

THEOREM 64.4 *A set S of n colored points on the real line can be preprocessed into a data structure of size $O(n)$ such that for any query interval, a type-2 counting query can be answered in $O(\log n + i)$ time, where i is the output size.*

64.3.2 A Sparsification-Based Approach

The idea behind this approach is to generate from the given set, S , of colored objects a colored set, S' —possibly consisting of different objects than those in S —such that a query object q intersects an object in S if and only if it intersects at most a constant number of objects in S' . This allows us to use a solution to a standard problem on S' to solve the generalized reporting problem on S . (In the case of a generalized counting problem, the requirement is more stringent: exactly one object in S' must be intersected.) We illustrate this method with the generalized halfspace range searching problem in \mathbb{R}^d , $d = 2, 3$.

Generalized halfspace range searching in \mathbb{R}^2 and \mathbb{R}^3

Let S be a set of n colored points in \mathbb{R}^d , $d = 2, 3$. We show how to preprocess S so that for any query hyperplane Q , the i distinct colors of the points lying in the closed halfspace Q^- (i.e., below Q) can be reported or counted efficiently. Without loss of generality, we may assume that Q is non-vertical since vertical queries are easy to handle. The approach described here is from [19].

We denote the coordinate directions by x_1, x_2, \dots, x_d . Let \mathcal{F} denote the well-known point-hyperplane duality transform [15]: If $p = (p_1, \dots, p_d)$ is a point in \mathbb{R}^d , then $\mathcal{F}(p)$ is the hyperplane $x_d = p_1x_1 + \dots + p_{d-1}x_{d-1} - p_d$. If $H : x_d = a_1x_1 + \dots + a_{d-1}x_{d-1} + a_d$ is a (non-vertical) hyperplane in \mathbb{R}^d , then $\mathcal{F}(H)$ is the point $(a_1, \dots, a_{d-1}, -a_d)$. It is easily verified that p is above (resp. on, below) H , in the x_d -direction, if and only if $\mathcal{F}(p)$ is below (resp. on, above) $\mathcal{F}(H)$. Note also that $\mathcal{F}(\mathcal{F}(p)) = p$ and $\mathcal{F}(\mathcal{F}(H)) = H$.

Using \mathcal{F} we map S to a set S' of hyperplanes and map Q to the point $q = \mathcal{F}(Q)$, both in \mathbb{R}^d . Our problem is now equivalent to: “Report or count the i distinct colors of the hyperplanes lying on or above q , i.e., the hyperplanes that are intersected by the vertical ray r emanating upwards from q .”

Let S_c be the set of hyperplanes of color c . For each color c , we compute the *upper envelope* E_c of the hyperplanes in S_c . E_c is the locus of the points of S_c of maximum x_d -coordinate for each point on the plane $x_d = 0$. E_c is a d -dimensional convex polytope which is unbounded in the positive x_d -direction. Its boundary is composed of j -faces, $0 \leq j \leq d-1$, where each j -face is a j -dimensional convex polytope. Of particular interest to us are the $(d-1)$ -faces of E_c , called *facets*. For instance, in \mathbb{R}^2 , E_c is an unbounded convex chain and its facets are line segments; in \mathbb{R}^3 , E_c is an unbounded convex polytope whose facets are convex polygons.

Let us assume that r is well-behaved in the sense that for no color c does r intersect two or more facets of E_c at a common boundary—for instance, a vertex in \mathbb{R}^2 and an edge or a vertex in \mathbb{R}^3 . (This assumption can be removed; details can be found in [19].) Then, by definition of the upper envelope, it follows that (i) r intersects a c -colored hyperplane if and only if r intersects E_c and, moreover, (ii) if r intersects E_c , then r intersects a unique facet of E_c (in the interior of the facet). Let \mathcal{E} be the collection of the envelopes of the different colors. By the above discussion, our problem is equivalent to: “Report or count the facets of \mathcal{E} that are intersected by r ”, which is a standard intersection searching problem. We will show how to solve efficiently this ray-envelope intersection problem in \mathbb{R}^2 and in \mathbb{R}^3 . This approach does not give an efficient solution to the generalized halfspace searching problem in \mathbb{R}^d for $d > 3$; for this case, we will give a different solution in Section 64.3.4.

To solve the ray-envelope intersection problem in \mathbb{R}^2 , we project the endpoints of the line segments of \mathcal{E} on the x -axis, thus partitioning it into $2n + 1$ *elementary intervals* (some of which may be empty). We build a *segment tree* T which stores these elementary intervals at the leaves. Let v be any node of T . We associate with v an x -interval $I(v)$, which is the union of the elementary intervals stored at the leaves in v 's subtree. Let $Strip(v)$ be the

vertical strip defined by $I(v)$. We say that a segment $s \in \mathcal{E}$ is *allocated* to a node $v \in T$ if and only if $I(v) \neq \emptyset$ and s crosses $\text{Strip}(v)$ but not $\text{Strip}(\text{parent}(v))$. Let $\mathcal{E}(v)$ be the set of segments allocated to v . Within $\text{Strip}(v)$, the segments of $\mathcal{E}(v)$ can be viewed as lines since they cross $\text{Strip}(v)$ completely. Let $\mathcal{E}'(v)$ be the set of points dual to these lines. We store $\mathcal{E}'(v)$ in an instance $D(v)$ of the standard halfplane reporting (resp. counting) structure for \mathbb{R}^2 given in [10] (resp. [26]). This structure uses $O(m)$ space and has a query time of $O(\log m + k_v)$ (resp. $O(m^{1/2})$), where $m = |\mathcal{E}(v)|$ and k_v is the output size at v .

To answer a query, we search in T using q 's x -coordinate. At each node v visited, we need to report or count the lines intersected by r . But, by duality, this is equivalent to answering, in \mathbb{R}^2 , a halfplane query at v using the query $\mathcal{F}(q)^- = Q^-$, which we do using $D(v)$. For the reporting problem, we simply output what is returned by the query at each visited node; for the counting problem, we return the sum of the counts obtained at the visited nodes.

THEOREM 64.5 *A set S of n colored points in \mathbb{R}^2 can be stored in a data structure of size $O(n \log n)$ so that the i distinct colors of the points contained in any query halfplane can be reported (resp. counted) in time $O(\log^2 n + i)$ (resp. $O(n^{1/2})$).*

Proof Correctness follows from the preceding discussion. As noted earlier, there are $O(|S_c|)$ line segments (facets) in E_c ; thus $|\mathcal{E}| = O(\sum_c |S_c|) = O(n)$ and so $|T| = O(n)$. Hence each segment of \mathcal{E} can get allocated to $O(\log n)$ nodes of T . Since the structure $D(v)$ has size linear in $m = |\mathcal{E}(v)|$, the total space used is $O(n \log n)$. For the reporting problem, the query time at a node v is $O(\log m + k_v) = O(\log n + k_v)$. When summed over the $O(\log n)$ nodes visited, this gives $O(\log^2 n + i)$. To see this, recall that the ray r can intersect at most one envelope segment of any color; thus the terms k_v , taken over all nodes v visited, sum to i .

For the counting problem, the query time at v is $O(m^{1/2})$. It can be shown that if v has depth j in T , then $m = |\mathcal{E}(v)| = O(n/2^j)$. (See, for instance, [12, page 675].) Thus, the overall query time is $O(\sum_{j=0}^{O(\log n)} (n/2^j)^{1/2})$, which is $O(n^{1/2})$. ■

In \mathbb{R}^3 , the approach is similar, but more complex. Our goal is to solve the ray-envelope intersection problem in \mathbb{R}^3 . As shown in [19], this problem can be reduced to certain standard halfspace range queries in \mathbb{R}^3 on a set of triangles (obtained by triangulating the E_c 's.) This problem can be solved by building a segment tree on the x -spans of the triangles projected to the xy -plane and augmenting each node of this tree with a data structure based on partition trees [25] or cutting trees [24] to answer the halfplane queries. Details may be found in [19].

THEOREM 64.6 *The reporting version of the generalized halfspace range searching problem for a set of n colored points in \mathbb{R}^3 can be solved in $O(n \log^2 n)$ (resp. $O(n^{2+\epsilon})$) space and $O(n^{1/2+\epsilon} + i)$ (resp. $O(\log^2 n + i)$) query time, where i is the output size and $\epsilon > 0$ is an arbitrarily small constant. The counting version is solvable in $O(n \log n)$ space and $O(n^{2/3+\epsilon})$ query time.*

Additional examples of the sparsification-based approach may be found in [23]. (An example also appears in the next section, enroute to a persistence-based solution of a generalized problem.)

64.3.3 A Persistence-Based Approach

Roughly speaking, we use persistence as follows: To solve a given generalized problem we first identify a different, but simpler, generalized problem and devise a data structure for it that also supports updates (usually just insertions). We then make this structure partially persistent [14] and query this persistent structure appropriately to solve the original problem.

We illustrate this approach for the generalized 3-dimensional range searching problem, where we are required to preprocess a set, S , of n colored points in \mathbb{R}^3 so that for any query box $q = [a, b] \times [c, d] \times [e, f]$ the i distinct colors of the points inside q can be reported efficiently. We first show how to build a semi-dynamic (i.e., insertions-only) data structure for the generalized versions of the quadrant searching and 2-dimensional range searching problems. These two structures will be the building blocks of our solution for the 3-dimensional problem.

Generalized semi-dynamic quadrant searching

Let S be a set of n colored points in the plane. For any point $q = (a, b)$, the *northeast quadrant* of q , denoted by $NE(q)$, is the set of all points (x, y) in the plane such that $x \geq a$ and $y \geq b$. We show how to preprocess S so that for any query point q , the distinct colors of the points of S contained in $NE(q)$ can be reported, and how points can be inserted into S . The data structure uses $O(n)$ space, has a query time of $O(\log^2 n + i)$, and an amortized insertion time of $O(\log n)$. This solution is based on the sparsification approach described previously.

For each color c , we determine the c -maximal points. (A point p is called c -maximal if it has color c and there are no points of color c in p 's northeast quadrant.) We discard all points of color c that are not c -maximal. In the resulting set, let the predecessor, $pred(p)$, of a c -colored point p be the c -colored point that lies immediately to the left of p . (For the leftmost point of color c , the predecessor is the point $(-\infty, \infty)$.) With each point $p = (a, b)$, we associate the horizontal segment with endpoints (a', b) and (a, b) , where a' is the x -coordinate of $pred(p)$. This segment gets the same color as p . Let S_c be the set of such segments of color c . The data structure consists of two parts, as follows.

The first part is a structure \mathcal{T} storing the segments in the sets S_c , where c runs over all colors. \mathcal{T} supports the following query: given a point q in the plane, report the segments that are intersected by the upward-vertical ray starting at q . Moreover, it allows segments to be inserted and deleted. We implement \mathcal{T} as the structure given in [11]. This structure uses $O(n)$ space, supports insertions and deletions in $O(\log n)$ time, and has a query time of $O(\log^2 n + l)$, where l is the number of segments intersected.

The second part is a balanced search tree CT , storing all colors. For each color c , we maintain a balanced search tree, T_c , storing the segments of S_c by increasing y -coordinate. This structure allows us to dynamically maintain S_c when a new c -colored point p is inserted. The general approach (omitting some special cases; see [18]) is as follows: By doing a binary search in T_c we can determine whether or not p is c -maximal in the current set of c -maximal points, i.e., the set of right endpoints of the segments of S_c . If p is not c -maximal, then we simply discard it. If p is c -maximal, then let s_1, \dots, s_k be the segments of S_c whose left endpoints are in the southwest quadrant of p . We do the following: (i) delete s_2, \dots, s_k from T_c ; (ii) insert into T_c the horizontal segment which starts at p and extends leftwards upto the x -coordinate of the left endpoint of s_k ; and (iii) truncate the segment s_1 by keeping only the part of it that extends leftwards upto p 's x -coordinate. The entire operation can be done in $O(\log n + k)$ time.

Let us now consider how to answer a quadrant query, $NE(q)$, and how to insert a point

into S . To answer $NE(q)$, we query \mathcal{T} with the upward-vertical ray from q and report the colors of the segments intersected. The correctness of this algorithm follows from the easily proved facts that (i) a c -colored point lies in $NE(q)$ if and only if a c -maximal point lies in $NE(q)$ and (ii) if a c -maximal point is in $NE(q)$, then the upward-vertical ray from q must intersect a segment of S_c . The correctness of \mathcal{T} guarantees that only the segments intersected by this ray are reported. Since the query can intersect at most two segments in any S_c , we have $l \leq 2i$, and so the query time is $O(\log^2 n + i)$.

Let p be a c -colored point that is to be inserted into S . If c is not in CT , then we insert it into CT and insert the horizontal, leftward-directed ray emanating from p into a new structure T_c . If c is present already, then we update T_c as just described. In both cases, we then perform the same updates on \mathcal{T} . Hence, an insertion takes $O((k+1)\log n)$ time.

What is the total time for n insertions into an initially empty set S ? For each insertion, we can charge the $O(\log n)$ time to delete a segment s_i , $2 \leq i \leq k$, to s_i itself. Notice that none of these segments will reappear. Thus each segment is charged at most once. Moreover, each of these segments has some previously inserted point as a right endpoint. It follows that the number of segments existing over the entire sequence of insertions is $O(n)$ and so the total charge to them is $O(n \log n)$. The rest of the cost for each insertion ($O(\log n)$ for the binary search plus $O(1)$ for steps (ii) and (iii)) we charge to p itself. Since any p is charged in this mode only once, the total charge incurred in this mode by all the inserted points is $O(n \log n)$. Thus the time for n insertions is $O(n \log n)$, which implies an amortized insertion time of $O(\log n)$.

LEMMA 64.2 Let S be a set of n colored points in the plane. There exists a data structure of size $O(n)$ such that for any query point q , we can report the i distinct colors of the points that are contained in the northeast quadrant of q in $O(\log^2 n + i)$ time. Moreover, if we do n insertions into an initially-empty set then the amortized insertion time is $O(\log n)$.

Generalized semidynamic 2-dimensional range searching

Our goal here is to preprocess a set S of n colored points in the plane so that for any axis-parallel query rectangle $q = [a, b] \times [c, d]$, we can solve the semi-dynamic reporting problem efficiently.

Our solution is based on the quadrant reporting structure of Lemma 64.2. We first show how to solve the problem for query rectangles $q' = [a, b] \times [c, \infty)$. We store the points of S in sorted order by x -coordinate at the leaves of a $BB(\alpha)$ tree T' . At each internal node v , we store an instance of the structure of Lemma 64.2 for NE -queries (resp., NW -queries) built on the points in v 's left (resp., right) subtree. Let $X(v)$ denote the average of the x -coordinate in the rightmost leaf in v 's left subtree and the x -coordinate in the leftmost leaf of v 's right subtree; for a leaf v , we take $X(v)$ to be the x -coordinate of the point stored at v .

To answer a query q' , we do a binary search down T' , using $[a, b]$, until either the search runs off T' or a (highest) node v is reached such that $[a, b]$ intersects $X(v)$. In the former case, we stop. In the latter case, if v is a leaf, then if v 's point is in q' we report its color. If v is a non-leaf, then we query the structures at v using the NE -quadrant and the NW -quadrant derived from q' (i.e., the quadrants with corners at (a, c) and (b, c) , respectively), and then combine the answers. Updates on T' are performed using the amortized-case updating strategy for $BB(\alpha)$ trees [32]. The correctness of the method should be clear. The space and query time bounds follow from Lemma 64.2. Since the amortized insertion time of the quadrant searching structure is $O(\log n)$, the insertion in the $BB(\alpha)$ tree takes amortized time $O(\log^2 n)$ [32].

To solve the problem for general query rectangles $q = [a, b] \times [c, d]$, we use the above approach again, except that we store the points in the tree by sorted y -coordinates. At each internal node v , we store an instance of the data structure above to answer queries of the form $[a, b] \times [c, \infty)$ (resp. $[a, b] \times (-\infty, d]$) on the points in v 's left (resp. right) subtree. The query strategy is similar to the previous one, except that we use the interval $[c, d]$ to search in the tree. The query time is as before, while the space and update times increase by a logarithmic factor.

LEMMA 64.3 Let S be a set of n colored points in the plane. There exists a data structure of size $O(n \log^2 n)$ such that for any query rectangle $[a, b] \times [c, d]$, we can report the i distinct colors of the points that are contained in it in $O(\log^2 n + i)$ time. Moreover, points can be inserted into this data structure in $O(\log^3 n)$ amortized time.

Generalized 3-dimensional range searching

The semi-dynamic structure of Lemma 64.3 coupled with persistence allows us to go up one dimension and solve the original problem of interest: Preprocess a set S of n colored points in \mathbb{R}^3 so that for any query box $q = [a, b] \times [c, d] \times [e, f]$ the i distinct colors of the points inside q can be reported efficiently.

First consider queries of the form $q' = [a, b] \times [c, d] \times [e, \infty)$. We sort the points of S by non-increasing z -coordinates, and insert them in this order into a partially persistent version of the structure of Lemma 64.3, taking only the first two coordinates into account. To answer q' , we access the version corresponding to the smallest z -coordinate greater than or equal to e and query it with $[a, b] \times [c, d]$.

To see that the query algorithm is correct, observe that the version accessed contains the projections on the xy -plane of exactly those points of S whose z -coordinate is at least e . Lemma 64.3 then guarantees that among these only the distinct colors of the ones in $[a, b] \times [c, d]$ are reported. These are precisely the distinct colors of the points contained in $[a, b] \times [c, d] \times [e, \infty)$. The query time follows from Lemma 64.3. To analyze the space requirement, we note that the structure of Lemma 64.3 satisfies the conditions given in [14]. Specifically, it is a pointer-based structure, where each node is pointed to by only $O(1)$ other nodes. As shown in [14], any modification made by a persistent update operation on such a structure adds only $O(1)$ amortized space to the resulting persistent structure. By Lemma 64.3, the total time for creating the persistent structure, via insertions, is $O(n \log^3 n)$. This implies the same bound for the number of modifications in the structure, so the total space is $O(n \log^3 n)$.

To solve the problem for general query boxes $q = [a, b] \times [c, d] \times [e, f]$, we follow an approach similar to that described for the 2-dimensional case: We store the points in a balanced binary search tree, sorted by z -coordinates. We associate with each internal node v in the tree the auxiliary structure described above for answering queries of the form $[a, b] \times [c, d] \times [e, \infty)$ (resp. $[a, b] \times [c, d] \times (-\infty, f]$) on the points in v 's left (resp. right) subtree. (Note that since we do not need to do updates here the tree need not be a $BB(\alpha)$ tree.) Queries are done by searching down the tree using the interval $[e, f]$. The query time is as before, but the space increases by a logarithmic factor.

THEOREM 64.7 Let S be a set of n colored points in 3-space. S can be stored in a data structure of size $O(n \log^4 n)$ such that for any query box $[a, b] \times [c, d] \times [e, f]$, we can report the i distinct colors of the points that are contained in it in $O(\log^2 n + i)$ time.

Additional applications of the persistence-based approach to generalized intersection problems can be found in [18, 19, 21].

64.3.4 A General Approach for Reporting Problems

We describe a general method from [21] for solving any generalized reporting problem given a data structure for a “related” standard decision problem.

Let S be a set of n colored geometric objects and let q be any query object. In preprocessing, we store the distinct colors in S at the leaves of a balanced binary tree CT (in no particular order). For any node v of CT , let $C(v)$ be the set of colors stored in the leaves of v 's subtree and let $S(v)$ be the set of those objects of S colored with the colors in $C(v)$. At v , we store a data structure $DEC(v)$ to solve the following *standard decision* problem on $S(v)$: “Decide whether or not q intersects any object of $S(v)$.” $DEC(v)$ returns “true” if and only if there is an intersection.

To answer a generalized reporting query on S , we do a depth-first search in CT and query $DEC(v)$ with q at each node v visited. If v is a non-leaf node then we continue searching below v if and only if the query returns “true”; if v is a leaf, then we output the color stored there if and only if the query returns “true”.

THEOREM 64.8 *Assume that a set of n geometric objects can be stored in a data structure of size $M(n)$ such that it can be decided in $f(n)$ time whether or not a query object intersects any of the n objects. Assume that $M(n)/n$ and $f(n)$ are non-decreasing functions for non-negative values of n . Then a set S of n colored geometric objects can be preprocessed into a data structure of size $O(M(n) \log n)$ such that the i distinct colors of the objects in S that are intersected by a query object q can be reported in time $O(f(n) + i \cdot f(n) \log n)$.*

Proof We argue that a color c is reported if and only if there is a c -colored object in S intersecting q . Suppose that c is reported. This implies that a leaf v is reached in the search such that v stores c and the query on $DEC(v)$ returns “true”. Thus, some object in $S(v)$ intersects q . Since v is a leaf, all objects in $S(v)$ have the same color c and the claim follows.

For the converse, suppose that q intersects a c -colored object p . Let v be the leaf storing c . Thus, $p \in S(v')$ for every node v' on the root-to- v path in CT . Thus, for each v' , the query on $DEC(v')$ will return “true”, which implies that v will be visited and c will be output.

If v_1, v_2, \dots, v_r are the nodes at any level, then the total space used by CT at that level is $\sum_{i=1}^r M(|S(v_i)|) = \sum_{i=1}^r |S(v_i)| \cdot (M(|S(v_i)|)/|S(v_i)|) \leq \sum_{i=1}^r |S(v_i)| \cdot (M(n)/n) = M(n)$, since $\sum_{i=1}^r |S(v_i)| = n$ and since $|S(v_i)| \leq n$ implies that $M(|S(v_i)|)/|S(v_i)| \leq M(n)/n$. Now since there are $O(\log n)$ levels, the overall space is $O(M(n) \log n)$. The query time can be upper-bounded as follows: If $i = 0$, then the query on $DEC(\text{root})$ returns “false” and we abandon the search at the root itself; in this case, the query time is just $O(f(n))$. Suppose that $i \neq 0$. Call a visited node v *fruitful* if the query on $DEC(v)$ returns “true” and *fruitless* otherwise. Each fruitful node can be charged to some color in its subtree that gets reported. Since the number of times any reported color can be charged is $O(\log n)$ (the height of CT) and since i colors are reported, the number of fruitful nodes is $O(i \log n)$. Since each fruitless node has a fruitful parent and CT is a binary tree, it follows that there are only $O(i \log n)$ fruitless nodes. Hence the number of nodes visited by the search is $O(i \log n)$. At each such node, v , we spend time $f(|S(v)|)$, which is $O(f(n))$ since $|S(v)| \leq n$ and f is non-decreasing. Thus the total time spent in doing queries at the visited nodes is

$O(i \cdot f(n) \log n)$. The claimed query time follows. ■

As an application of this method, consider the generalized halfspace range searching in \mathbb{R}^d , for any fixed $d \geq 2$. For $d = 2, 3$, we discussed a solution for this problem in Section 64.3.2. For $d > 3$, the problem can be solved by extending (significantly) the ray-envelope intersection algorithm outlined in Section 64.3.2. However, the bounds are not very satisfactory— $O(n^{d \lfloor d/2 \rfloor + \epsilon})$ space and logarithmic query time or near-linear space and superlinear query time. The solution we give below has more desirable bounds.

The colored objects for this problem are points in \mathbb{R}^d and the query is a closed halfspace in \mathbb{R}^d . We store the objects in CT , as described previously. The standard decision problem that we need to solve at each node v of CT is “Does a query halfspace contain any point of $S(v)$.” The answer to this query is “true” if and only if the query halfspace is non-empty. We take the data structure, $DEC(v)$, for this problem to be the one given in [27]. If $|S_v| = n_v$, then $DEC(v)$ uses $O(n_v^{\lfloor d/2 \rfloor} / (\log n_v)^{\lfloor d/2 \rfloor - \epsilon})$ space and has query time $O(\log n_v)$ [27]. The conditions in Theorem 64.8 hold, so applying it gives the following result.

THEOREM 64.9 *For any fixed $d \geq 2$, a set S of n colored points in \mathbb{R}^d can be stored in a data structure of size $O(n^{\lfloor d/2 \rfloor} / (\log n)^{\lfloor d/2 \rfloor - 1 - \epsilon})$ such that the i distinct colors of the points contained in a query halfspace Q^- can be reported in time $O(\log n + i \log^2 n)$. Here $\epsilon > 0$ is an arbitrarily small constant.*

Other applications of the general method may be found in [21].

64.3.5 Adding Range Restrictions

We describe the general technique of [20] that adds a range restriction to a generalized intersection searching problem.

Let PR be a generalized intersection searching problem on a set S of n colored objects and query objects q belonging to a class Q . We denote the answer to a query by $PR(q, S)$. To add a *range restriction*, we associate with each element $p \in S$ a real number k_p . In a range-restricted generalized intersection searching problem, denoted by TPR , a query consists of an element $q \in Q$ and an interval $[l, r]$, and

$$TPR(q, [l, r], S) := PR(q, \{p \in S : l \leq k_p \leq r\}).$$

For example, if PR is the generalized $(d - 1)$ -dimensional range searching problem, then TPR is the generalized d -dimensional version of this problem, obtained by adding a range restriction to the d th dimension.

Assume that we have a data structure DS that solves PR with $O((\log n)^u + i)$ query time using $O(n^{1+\epsilon})$ space and a data structure TDS that solves TPR for generalized (semi-infinite) queries of the form $TPR(q, [l, \infty), S)$ with $O((\log n)^v + i)$ query time using $O(n^w)$ space. (Here u and v are positive constants, $w > 1$ is a constant, and $\epsilon > 0$ is an arbitrarily small constant.) We will show how to transform DS and TDS into a data structure that solves generalized queries $TPR(q, [l, r], S)$ in $O((\log n)^{\max(u, v, 1)} + i)$ time, using $O((n^{1+\epsilon})$ space.

Let $S = \{p_1, p_2, \dots, p_n\}$, where $k_{p_1} \geq k_{p_2} \geq \dots \geq k_{p_n}$. Let m be an arbitrary parameter with $1 \leq m \leq n$. We assume for simplicity that n/m is an integer. Let $S_j = \{p_1, p_2, \dots, p_{jm}\}$ and $S'_j = \{p_{j(m+1)}, p_{j(m+2)}, \dots, p_{(j+1)m}\}$ for $0 \leq j < n/m$.

The transformed data structure consists of the following. For each j with $0 \leq j < n/m$, there is a data structure DS_j (of type DS) storing S_j for solving generalized queries of the

form $PR(q, S_j)$, and a data structure TDS_j (of type TDS) storing S'_j for solving generalized queries of the form $TPR(q, [l, \infty), S'_j)$.

To answer a query $TPR(q, [l, \infty), S)$, we do the following. Compute the index j such that $k_{p_{(j+1)m}} < l \leq k_{p_{jm}}$. Solve the query $PR(q, S_j)$ using DS_j , solve the query $TPR(q, [l, \infty), S'_j)$ using TDS_j , and output the union of the colors reported by these two queries. It is easy to see that the query algorithm is correct. The following lemma gives the complexity of the transformed data structure.

LEMMA 64.4 The transformed data structure uses $O(n^{2+\epsilon}/m + nm^{w-1})$ space and can be used to answer generalized queries $TPR(q, [l, \infty), S)$ in $O((\log n)^{\max(u, v, 1)} + i)$ time.

THEOREM 64.10 Let S , DS and TDS be as above. There exists a data structure of size $O(n^{1+\epsilon})$ that solves generalized queries $TPR(q, [l, r], S)$ in $O((\log n)^{\max(u, v, 1)} + i)$ time.

Proof We will use Lemma 64.4 to establish the claimed bounds for answering generalized queries $TPR(q, [l, \infty), S)$. The result for queries $TPR(q, [l, r], S)$ then follows from a technique, based on $BB(\alpha)$ trees, that we used in Section 64.3.3.

If $w > 2$, then we apply Lemma 64.4 with $m = n^{1/w}$. This gives a data structure having size $O(n^2)$ that answers queries $TPR(q, [l, \infty), S)$ in $O((\log n)^{\max(u, v, 1)} + i)$ time. Hence, we may assume that $w = 2$.

By applying Lemma 64.4 repeatedly, we obtain, for each integer constant $a \geq 1$, a data structure of size $O(n^{1+\epsilon+1/a})$ that answers queries $TPR(q, [l, \infty), S)$ in $O((\log n)^{\max(u, v, 1)} + i)$ time. This claim follows by induction on a ; in the inductive step from a to $a + 1$, we apply Lemma 64.4 with $m = n^{a/(a+1)}$. ■

Using Theorem 64.10, we can solve efficiently, for instance, the generalized orthogonal range searching problem in \mathbb{R}^d . (Examples of other problems solvable via this method may be found in [20].)

THEOREM 64.11 Let S be a set of n colored points in \mathbb{R}^d , where $d \geq 1$ is a constant. There exists a data structure of size $O(n^{1+\epsilon})$ such that for any query box in \mathbb{R}^d , we can report the i distinct colors of the points that are contained in it in $O(\log n + i)$ time.

Proof The proof is by induction on d . For $d = 1$, the claim follows from Theorem 64.1. Let $d \geq 2$, and let DS be a data structure of size $O(n^{1+\epsilon})$ that answers generalized $(d - 1)$ -dimensional range queries in $O(\log n + i)$ time. Observe that for the generalized d -dimensional range searching problem, there are only polynomially many distinct semi-infinite queries. Hence, there exists a data structure TDS of polynomial size that answers generalized d -dimensional semi-infinite range queries in $O(\log n + i)$ time. Applying Theorem 64.10 to DS and TDS proves the claim. ■

64.4 Conclusion and Future Directions

We have reviewed recent research on a class of geometric query-retrieval problems, where the objects to be queried come aggregated in disjoint groups and of interest are questions concerning the intersection of the query object with the groups (rather than with the indi-

vidual objects). These problems include the well-studied standard intersection problems as a special case and have many applications. We have described several general techniques that have been identified for these problems and have illustrated them with examples.

Some potential directions for future work include: (i) extending the transformation-based approach to higher dimensions; (ii) improving the time bounds for some of the problems discussed here—for instance, can the generalized orthogonal range searching problem in \mathbb{R}^d , for $d \geq 4$, be solved with $O(\text{polylog}(n) + i)$ query time and $O(n(\log n)^{O(1)}n)$ space; (iii) developing general dynamization techniques for generalized problems, along the lines of, for instance, [5] for standard problems; (iv) developing efficient solutions to generalized problems where the objects may be in time-dependent motion; and (v) implementing and testing experimentally some of the solutions presented here.

64.5 Acknowledgment

Portions of the material presented in this chapter are drawn from the authors' prior publications: References [18–20], with permission from Elsevier (<http://www.elsevier.com/>), and reference [21], with permission from Taylor & Francis (<http://www.tandf.co.uk>).

References

- [1] P. K. Agarwal, M. de Berg, S. Har-Peled, M. H. Overmars, M. Sharir, and J. Vahrenhold. Reporting intersecting pairs of convex polytopes in two and three dimensions. *Computational Geometry: Theory and Applications*, 23:195–207, 2002.
- [2] P. K. Agarwal, S. Govindarajan, and S. Muthukrishnan. Range searching in categorical data: Colored range searching on grid. In *Proceedings of the 10th European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 17–28, Berlin, 2002. Springer-Verlag.
- [3] P. K. Agarwal and M. van Kreveld. Polygon and connected component intersection searching. *Algorithmica*, 15:626–660, 1996.
- [4] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. In B. Chazelle, J. E. Goodman, and R. Pollack, editors, *Advances in Discrete and Computational Geometry*, volume 223 of *Contemporary Mathematics*, pages 1–56. American Mathematical Society, Providence, RI, 1999.
- [5] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformations. *Journal of Algorithms*, 1:301–358, 1980.
- [6] P. Bozanis, N. Kitsios, C. Makris, and A. Tsakalidis. New upper bounds for generalized intersection searching problems. In *Proceedings of the 22nd International Colloquium on Automata, Languages and Programming*, volume 944 of *Lecture Notes in Computer Science*, pages 464–475, Berlin, 1995. Springer-Verlag.
- [7] P. Bozanis, N. Kitsios, C. Makris, and A. Tsakalidis. Red-blue intersection reporting for objects of non-constant size. *The Computer Journal*, 39:541–546, 1996.
- [8] P. Bozanis, N. Kitsios, C. Makris, and A. Tsakalidis. New results on intersection query problems. *The Computer Journal*, 40:22–29, 1997.
- [9] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.
- [10] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.
- [11] S. W. Cheng and R. Janardan. Efficient dynamic algorithms for some geometric intersection problems. *Information Processing Letters*, 36:251–258, 1990.
- [12] S. W. Cheng and R. Janardan. Algorithms for ray-shooting and intersection searching. *Journal of Algorithms*, 13:670–692, 1992.

- [13] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Germany, 2nd edition, 2000.
- [14] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.
- [15] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer-Verlag, New York, 1987.
- [16] P. Ferragina, N. Koudas, S. Muthukrishnan, and D. Srivastava. Two-dimensional substring indexing. In *Proceedings of the 20th ACM Symposium on Principles of Database Systems*, pages 282–288, 2001.
- [17] P. Gupta. *Efficient algorithms and data structures for geometric intersection problems*. Ph.D. dissertation, Dept. of Computer Science, Univ. of Minnesota, Minneapolis, MN, 1995.
- [18] P. Gupta, R. Janardan, and M. Smid. Further results on generalized intersection searching problems: counting, reporting and dynamization. *Journal of Algorithms*, 19:282–317, 1995.
- [19] P. Gupta, R. Janardan, and M. Smid. Algorithms for generalized halfspace range searching and other intersection searching problems. *Computational Geometry: Theory and Applications*, 5:321–340, 1996.
- [20] P. Gupta, R. Janardan, and M. Smid. A technique for adding range restrictions to generalized searching problems. *Information Processing Letters*, 64:263–269, 1997.
- [21] P. Gupta, R. Janardan, and M. Smid. Algorithms for some intersection searching problems involving circular objects. *International Journal of Mathematical Algorithms*, 1:35–52, 1999.
- [22] P. Gupta, R. Janardan, and M. Smid. Efficient algorithms for counting and reporting pairwise intersections between convex polygons. *Information Processing Letters*, 69:7–13, 1999.
- [23] R. Janardan and M. Lopez. Generalized intersection searching problems. *International Journal of Computational Geometry and Applications*, 3:39–69, 1993.
- [24] J. Matoušek. Cutting hyperplane arrangements. *Discrete & Computational Geometry*, 6:385–406, 1991.
- [25] J. Matoušek. Efficient partition trees. *Discrete & Computational Geometry*, 8:315–334, 1992.
- [26] J. Matoušek. Range searching with efficient hierarchical cuttings. *Discrete & Computational Geometry*, 10:157–182, 1993.
- [27] J. Matoušek and O. Schwarzkopf. On ray shooting in convex polytopes. *Discrete & Computational Geometry*, 10:215–232, 1993.
- [28] E. M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14:257–276, 1985.
- [29] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 657–666, 2002.
- [30] M. J. van Kreveld. *New Results on Data Structures in Computational Geometry*. Ph.D. dissertation, Dept. of Computer Science, Utrecht Univ., The Netherlands, 1992.
- [31] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33:209–271, 2001.
- [32] D.E. Willard and G.S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32:597–617, 1985.