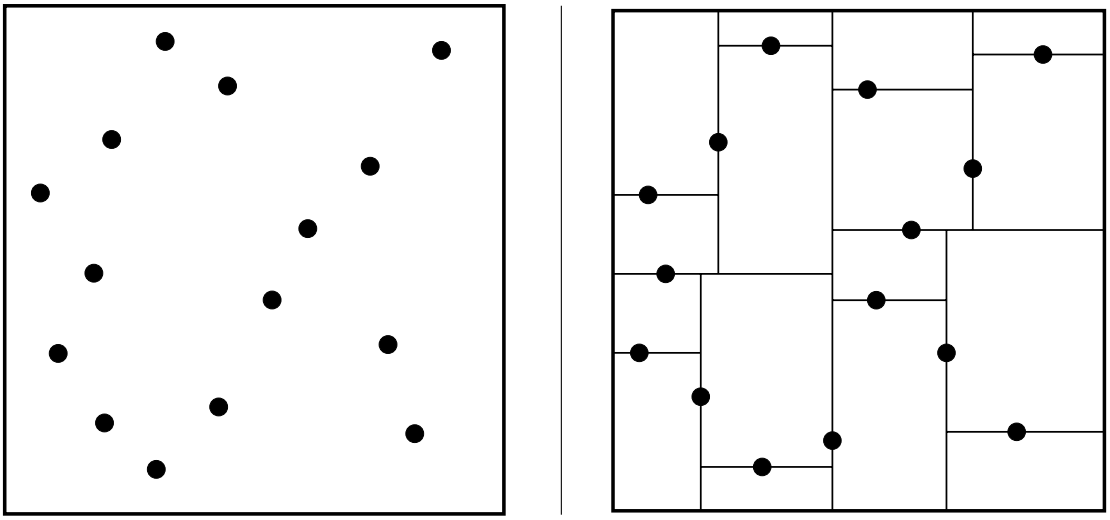

INPUT OUTPUT

12.6 Kd-Trees

Input description: A set S of n points or more complicated geometric objects in k dimensions.

Problem description: Construct a tree that partitions space by half-planes such that each object is contained in its own box-shaped region.

Discussion: Kd-tree and related spatial data structures hierarchically decompose space into a small number of cells, each containing a few representatives from an input set of points. This provides a fast way to access any object by position. We traverse down the hierarchy until we find the smallest cell containing it, and then scan through the objects in this cell to identify the right one.

Typical algorithms construct kd-trees by partitioning point sets. Each node in the tree is defined by a plane cutting through one of the dimensions. Ideally, this plane equally partitions the subset of points into left/right (or up/down) subsets. These children are again partitioned into equal halves, using planes through a different dimension. Partitioning stops after $\lg n$ levels, with each point in its own leaf cell.

The cutting planes along any path from the root to another node defines a unique box-shaped region of space. Each subsequent plane cuts this box into two boxes. Each box-shaped region is defined by $2k$ planes, where k is the number of dimensions. Indeed, the "kd" in *kd-tree* is short for "k-dimensional." We maintain

the region of interest defined by the intersection of these half-spaces as we move down the tree.

Flavors of kd-trees differ in exactly how the splitting plane is selected. Options include:

- *Cycling through the dimensions* – partition first on d_1, then $d_2, \ldots, d_k$ before cycling back to d_1.
- *Cutting along the largest dimension* – select the partition dimension to make the resulting boxes as square or cube-like as possible. Selecting a plane to partition the points in half does not mean selecting a splitter in the middle of the box-shaped regions, since all the points may lie in the left side of the box.
- *Quadtrees or Octtrees* – Instead of partitioning with single planes, use all axis-parallel planes that pass through a given partition point. In two dimensions, this means creating four child cells; in 3D, this means eight child cells. Quadtrees seem particularly popular on image data, where leaf cells imply that all pixels in the regions have the same color.
- *BSP-trees – Binary space partitions* use general (i.e. , not just axis-parallel) cutting planes to carve up space into cells so that each cell ends up containing only one object (say a polygon). Such partitions are not possible using only axis-parallel cuts for certain sets of objects. The downside is that such polyhedral cell boundaries are more complicated to work with than boxes.
- *R-trees* – This is another spatial data structure useful for geometric objects that cannot be partitioned into axis-oriented boxes without cutting them into pieces. At each level, the objects are partitioned into a small number of (possibly-overlapping) boxes to construct searchable hierarchies without partitioning objects.

Ideally, our partitions split both the space (ensuring fat, regular regions) and the set of points (ensuring a log height tree) evenly, but doing both simultaneously can be impossible on a given input. The advantages of fat cells become clear in many applications of kd-trees:

- *Point location* – To identify which cell a query point q lies in, we start at the root and test which side of the partition plane contains q. By repeating this process on the appropriate child node, we travel down the tree to find the leaf cell containing q in time proportional to its height. See Section 17.7 (page 587) for more on point location.
- *Nearest neighbor search* – To find the point in S closest to a query point q, we perform point location to find the cell c containing q. Since c is bordered by some point p, we can compute the distance $d(p, q)$ from p to q. Point p is likely

close to q, but it might not be the single closest neighbor. Why? Suppose q lies right at the boundary of a cell. Then q's nearest neighbor might lie just to the left of the boundary in another cell. Thus, we must traverse all cells that lie within a distance of $d(p, q)$ of cell c and verify that none of them contain closer points. In trees with nice, fat cells, very few cells should need to be tested. See Section 17.5 (page 580) for more on nearest neighbor search.

- *Range search* – Which points lie within a query box or region? Starting from the root, check whether the query region intersects (or contains) the cell defining the current node. If it does, check the children; if not, none of the leaf cells below this node can possibly be of interest. We quickly prune away irrelevant portions of the space. Section 17.6 (page 584) focuses on range search.

- *Partial key search* – Suppose we want to find a point p in S, but we do not have full information about p. Say we are looking for someone of age 35 and height 5'8" but of unknown weight in a 3D-tree with dimensions of age, weight, and height. Starting from the root, we can identify the correct descendant for all but the weight dimension. To be sure we find the right point, we must search *both children* of these nodes. The more fields we know the better, but such partial key search can be substantially faster than checking all points against the key.

Kd-trees are most useful for a small to moderate number of dimensions, say from 2 up to maybe 20 dimensions. They lose effectiveness as the dimensionality increases, primarily because the ratio of the volume of a unit sphere in k-dimensions shrinks exponentially compared to the unit cube. Thus exponentially many cells will have to be searched within a given radius of a query point, say for nearest-neighbor search. Also, the number of neighbors for any cell grows to $2k$ and eventually become unmanageable.

The bottom line is that you should try to avoid working in high-dimensional spaces, perhaps by discarding (or projecting away) the least important dimensions.

Implementations: *KDTREE 2* contains C++ and Fortran 95 implementations of *kd*-trees for efficient nearest nearest neighbor search in many dimensions. See *http://arxiv.org/abs/physics/0408067*.

Samet's spatial index demos (*http://donar.umiacs.umd.edu/quadtree/*) provide a series of Java applets illustrating many variants of *kd*-trees, in association with his book [Sam06].

Terralib (*http://www.terralib.org/*) is an open source geographic information system (GIS) software library written in C++. This includes an implementation of spatial data structures.

The 1999 DIMACS implementation challenge focused on data structures for nearest neighbor search [GJM02]. Data sets and codes are accessible from *http://dimacs.rutgers.edu/Challenges*.

Notes: Samet [Sam06] is the best reference on kd-trees and other spatial data structures. All major (and many minor) variants are developed in substantial detail. A shorter survey [Sam05] is also available. Bentley [Ben75] is generally credited with developing kd-trees, although they have the murky history associated with most folk data structures.

The performance of spatial data structures degrades with high dimensionality. Projecting high-dimensional spaces onto a random lower-dimensional hyperplane has recently emerged as a simple but powerful method for dimensionality reduction. Both theoretical [IM04] and empirical [BM01] results indicate that this method preserves distances quite nicely.

Algorithms that quickly produce a point provably close to the query point are a recent development in higher-dimensional nearest neighbor search. A sparse weighted-graph structure is built from the data set, and the nearest neighbor is found by starting at a random point and walking greedily in the graph towards the query point. The closest point found during several random trials is declared the winner. Similar data structures hold promise for other problems in high-dimensional spaces. See [AM93, AMN$^+$98].

Related Problems: Nearest-neighbor search (see page 580), point location (see page 587), range search (see page 584).