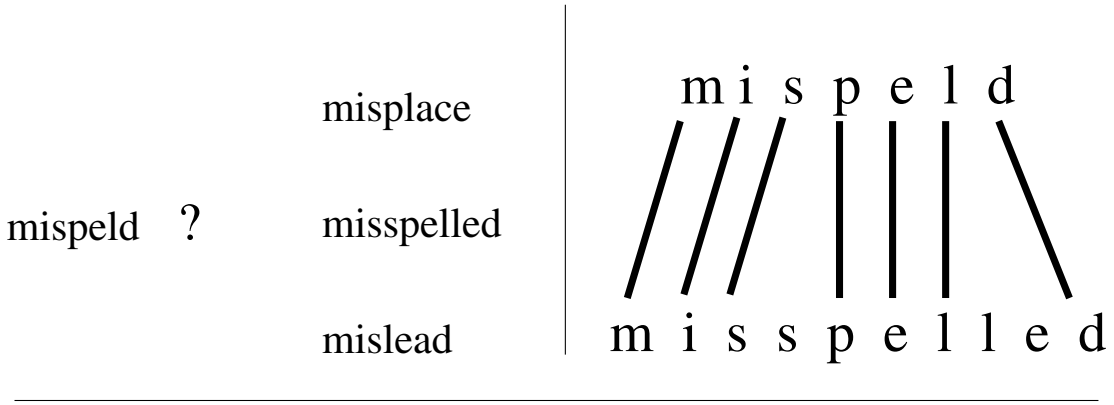

18.4 Approximate String Matching

Input description: A text string t and a pattern string p.

Problem description: What is the minimum-cost way to transform t to p using insertions, deletions, and substitutions?

Discussion: Approximate string matching is a fundamental problem because we live in an error-prone world. Spelling correction programs must be able to identify the closest match for any text string not found in a dictionary. By supporting efficient sequence similarity (homology) searches on large databases of DNA sequences, the computer program BLAST has revolutionized the study of molecular biology. Suppose you were interested in a particular gene in man, and discovered that it is similar to the hemoglobin gene in rats. Likely this new gene also produces hemoglobin, and any differences are the result of genetic mutations during evolution.

I once encountered approximate string matching when evaluating the performance of an optical character-recognition system. We needed to compare the answers produced by our system on a test document with the correct results. To improve our system, we needed to identify (1) which letters were getting misidentified and (2) gibberish when the program found letters that didn't exist. The solution was to do an alignment between the two texts. Insertions and deletions corresponded to gibberish, while substitutions signaled errors in our recognizer. This same principle is used in file difference programs, which identify the lines that have changed between two versions of a file.

When no errors are permitted, our problem reduces to exact string matching, which is presented in Section 18.3 (page 628). Here, we restrict our discussion to matching with errors.

Dynamic programming provides the basic approach to approximate string matching. Let $D[i,j]$ denote the cost of editing the first i characters of the pattern string p into the first j characters of the text t. The recurrence follows because we must have done *something* with the tail characters p_i and t_j. Our only options are matching / substituting one for the other, deleting p_i, or inserting a match for t_j. Thus, $D[i,j]$ is the minimum of the costs of these possibilities:

1. If $p_i = t_j$ then $D[i-1,j-1]$ else $D[i-1,j-1]$ + substitution cost.
2. $D[i-1,j]$ + deletion cost of p_i.
3. $D[i,j-1]$ + deletion cost of t_j.

A general implementation in C and more complete discussion appears in Section 8.2 (page 280). Several issues remain before we can make full use of this recurrence:

- *Do I match the pattern against the full text, or against a substring?* – The boundary conditions of this recurrence distinguishes between algorithms for string matching and substring matching. Suppose we want to align the full pattern against the full text. Then the cost of $D[i,0]$ must be that of deleting the first i characters of the pattern, so $D[i,0] = i$. Similarly, $D[0,j] = j$.

  Now suppose that the pattern can occur anywhere within the text. The proper cost of $D[0,j]$ is now 0, since there should be no penalty for starting the alignment in the jth position of the text. The cost of $D[i,0]$ remains i, because the only way to match the first i pattern characters with nothing is to delete all of them. The cost of the best substring pattern match against the text will be given by $\min_{k=1}^{n} D[m,k]$.

- *How should I select the substitution and insertion/deletion costs?* – The basic algorithm can be easily modified to use different costs for insertion, deletion, and the substitutions of specific pairs of characters. Which costs you should use depend on what you are planning to do with the alignment.

  The most common cost assignment charges the same for insertion, deletion, or substitution. Charging a substitution cost of more than insertion + deletion ensures that substitutions never get performed, since it will always be cheaper to edit both characters out of the string. If we just have insertion and deletion to work with, the problem reduces to *longest common subsequence*, discussed in Section 18.8 (page 650). It often pays to tweak the edit distance costs and study the resulting alignments until you find the best parameters for the job.

- *How do I find the actual alignment of the strings?* – As thus far described, the recurrence only gives the cost of the optimal string/pattern alignment, not the sequence of editing operations to achieve it. To obtain such a transcript, we can work backwards from the complete cost matrix D. We had to come from one of $D[m-1,n]$ (pattern deletion/text insertion), $D[m,n-1]$

(text deletion/pattern insertion), or $D[m-1, n-1]$ (substitution/match) to get to cell $D[m, n]$. The option which was chosen can be reconstructed from these costs and the given characters p_m and t_n. By continuing to work backwards to the previous cell, we can reconstruct the entire alignment. Again, an implementation in C appears in Section 8.2 (page 280).

- *What if the two strings are very similar to each other?* – The dynamic programming algorithm finds a shortest path across an $m \times n$ grid, where the cost of each edge depends upon which operation it represents. To seek an alignment involving a combination of at most d insertions, deletions, and substitutions, we need only traverse the band of $O(dn)$ cells within a distance d of the central diagonal. If no low-cost alignment exists within this band, then no low-cost alignment can exist in the full cost matrix.

  Another idea we can use is *filtration*, quickly eliminating the parts of the string where there is no hope of finding the pattern. Carve the m-length pattern into $d+1$ pieces. If there is a match with at most d differences, then at least one of these pieces must be an exact match in the optimal alignment. Thus, we can identify all possible approximate match points by conducting an exact multi-pattern search on the pieces, and then evaluate only the possible candidates more carefully.

- *Is your pattern short or long?* – A recent approach to string-matching exploits the fact that modern computers can do operations on (say) 64-bit words in a single gulp. This is long enough to hold eight 8-bit ASCII characters, providing motivation to design *bit-parallel algorithms*, which do more than one comparison with each operation.

  The basic idea is quite clever. Construct a bit-mask B_α for each letter α of the alphabet, such that ith-bit $B_\alpha[i] = 1$ iff the ith character of the pattern is α. Now suppose you have a match bit-vector M_j for position j in the text string, such that $M_j[i] = 1$ iff the first i bits of the pattern exactly match the $(j-i+1)$st through jth character of the text. We can find *all* the bits of M_{j+1} using just two operations by (1) shifting M_j one bit to the right, and then (2) doing a bitwise AND with B_α, where α is the character in position $j+1$ of the text.

  The *agrep* program, discussed below, uses such a bit-parallel algorithm generalized to approximate matching. Such algorithms are easy to program and many times faster than dynamic programming.

- *How can I minimize the required storage?* – The quadratic space used to store the dynamic programming table is usually a more serious problem than its running time. Fortunately, only $O(\min(m, n))$ space is needed to compute $D[m, n]$. We need only maintain two active rows (or columns) of the matrix to compute the final value. The entire matrix will be required only if we need to reconstruct the actual sequence alignment.

We can use Hirschberg's clever recursive algorithm to efficiently recover the optimal alignment in linear space. During one pass of the linear-space algorithm above to compute $D[m, n]$, we identify which middle-element cell $D[m/2, x]$ was used to optimize $D[m, n]$. This reduces our problem to finding the best paths from $D[1, 1]$ to $D[m/2, x]$ and from $D[m/2, x]$ to $D[m/2, n]$, both of which can be solved recursively. Each time we remove half of the matrix elements from consideration, so the total time remains $O(mn)$. This linear-space algorithm proves to be a big win in practice on long strings, although it is somewhat more difficult to program.

- *Should I score long runs of indels differently?* – Many string matching applications look more kindly on alignments where insertions/deletions are bunched in a small number of runs or gaps. Deleting a word from a text should presumably cost less than a similar number of scattered single-character deletions, because the word represents a single (albeit substantial) edit operation.

  String matching with *gap penalties* provides a way to properly account for such operations. Typically, we assign a cost of $A + Bt$ for each indel of t consecutive characters, where A is the cost of starting the gap and B is the per-character deletion cost. If A is large relative to B, the alignment has incentive to create relatively few runs of deletions.

  String matching under such *affine* gap penalties can be done in the same quadratic time as regular edit distance. We will use separate insertion and deletion recurrences E and F to encode the cost of being in gap mode, meaning we have already paid the cost of initiating the gap:

$$V(i, j) = \max(E(i, j), F(i, j), G(i, j))$$

$$G(i, j) = V(i - 1, j - 1) + match(i, j)$$

$$E(i, j) = \max(E(i, j - 1), V(i, j - 1) - A) - B$$

$$F(i, j) = \max(F(i - 1, j), V(i - 1, j) - A) - B$$

  With a constant amount of work per cell, this algorithm takes $O(mn)$ time, same as without gap costs.

- *Does similarity mean strings that sound alike?* – Other models of approximate pattern matching become more appropriate for certain applications. Particularly interesting is *Soundex*, a hashing scheme that attempts to pair up English words that sound alike. This can be useful in testing whether two names that have been spelled differently are likely to be the same. For example, my last name is often spelled "Skina", "Skinnia", "Schiena", and occasionally "Skiena." All of these hash to the same Soundex code, *S25*.

The algorithm drops vowels and silent letters, removes doubled letters, and then assigns the remaining letters numbers from the following classes: *BFPV* gets a 1, *CGJKQSXZ* gets a 2, *DT* gets a 3, *L* gets a 4, *MN* gets a 5, and *R* gets a 6. The code starts with the first letter and contains at most three digits. Although this sounds very hokey, experience shows that it works reasonably well. Experience indeed: Soundex has been used since the 1920's.

Implementations: Several excellent software tools are available for approximate pattern matching. Manber and Wu's *agrep* [WM92a, WM92b] (approximate general regular expression pattern matcher) is a tool supporting text search with spelling errors. A recent version is available from *http://www.tgries.de/agrep/*. Navarro's *nrgrep* [Nav01b] combines bit-parallelism and filtration, resulting in running times that are more constant than *agrep*, although not always faster. It is available at *http://www.dcc.uchile.cl/~gnavarro/software/*.

TRE is a general regular-expression matching library for exact and approximate matching, which is more general than *agrep*. The worst-case complexity is $O(nm^2)$, where m is the list of the regular expressions involved. *TRE* is available at *http://laurikari.net/tre/*.

Wikipedia gives programs for computing edit (Levenshtein) distance in a dizzying array of languages (including Ada, C++, Emacs Lisp, Io, JavaScript, Java, PHP, Python, Ruby VB, and C#) Check it out at:

http://en.wikibooks.org/wiki/Algorithm_implementation/Strings/Levenshtein_distance

Notes: There have been many recent advances in approximate string matching, particularly in bit-parallel algorithms. Navarro and Raffinot [NR07] is the best reference on these recent techniques, which are also treated in other recent books on string algorithmics [CHL07, Gus97]. String matching with gap penalties is particularly well treated in [Gus97].

The basic dynamic programming alignment algorithm is attributed to [WF74], although it is apparently folklore. The wide range of applications for approximate string matching was made apparent in Sankoff and Kruskal's book [SK99], which remains a useful historical reference for the problem. Surveys on approximate pattern matching include [HD80, Nav01a]. The edit distance between two strings is sometimes referred to as the *Levenshtein distance*. Expositions of Hirschberg's linear-space algorithm [Hir75] include [CR03, Gus97].

Masek and Paterson [MP80] compute the edit distance between m- and n-length strings in time $O(mn/\log(\min\{m,n\}))$ for constant-sized alphabets, using ideas from the four Russians algorithm for Boolean matrix multiplication [ADKF70].

The shortest path formulation leads to a variety of algorithms that are good when the edit distance is small, including an $O(n \lg n + d^2)$ algorithm due to Myers [Mye86] and an $O(dn)$ algorithm due to Landau and Vishkin [LV88]. Longest increasing subsequence can be done in $O(n \lg n)$ time [HS77], as presented in [Man89].

Bit-parallel algorithms for approximate matching include Myers's [Mye99b] algorithm for approximate matching in $O(mn/w)$ time, where w is the number of bits in the computer word. Experimental studies of bit-parallel algorithms include [FN04, HFN05, NR00].

Soundex was invented and patented by M. K. Odell and R. C. Russell. Expositions on Soundex include [BR95, Knu98]. Metaphone is a recent attempt to improve on Soundex [BR95, Par90]. See [LMS06] for an application of such phonetic hashing techniques to the problem entity name unification.

Related Problems: String matching (see page 628), longest common substring (see page 650).