Approximate String Matching

A common task in text editing is string matching - finding all occurrances of a word in a text.

Unfortunately, many words are mispelled. How can we search for the string closest to the pattern?

Let p be a pattern string and T a text string over the same alphabet.

A k-approximate match between P and T is a substring of T with at most k differences.

Differences may be:

1. the corresponding characters may differ: KAT$\rightarrow$ CAT

2. P is missing a character from T: CAAT$\rightarrow$ CAT

3. T is missing a character from P: CT$\rightarrow$ CAT

Approximate Matching is important in genetics as well as spell checking.

A 3-Approximate Match

A match with one of each of three edit operations is:

$$P = \text{unescessaraly}$$

$$T = \text{unnecessarily}$$

Finding such a matching seems like a hard problem because we must figure out where you add *blanks*, but we can solve it with dynamic programming.

$D[i,j]$ = the minimum number of differences between $P_1, P_2, \ldots, P_i$ and the segment of T ending at j.

$D[i,j]$ is the *minimum* of the three possible ways to extend smaller strings:

1. If $P_i = \mathrm{T_j}$ then $D[i-1,j-1]$ else $D[i-1,j-1]+1$ (corresponding characters do or do not match)

2. $D[i-1,j]+1$ (extra character in text – we do not advance the pattern pointer).

3. $D[i,j-1]+1$ (character in pattern which is not in text).

Once you accept the recurrence it is easy.

To fill each cell, we need only consider three other cells, not $O(n)$ as in other examples. This means we need only store two rows of the table. The total time is $O(mn)$.

Boundary conditions for string matching

What should the value of $D[0, i]$ be, corresponding to the cost of matching the first i characters of the text with none of the pattern?

It depends. Are we doing string matching in the text or substring matching?

- If you want to match all of the pattern against all of the text, this meant that would have to delete the first i characters of the pattern, so $D[0, i] = i$ to pay the cost of the deletions.

- if we want to find the place in the text where the pattern occurs? We do not want to pay more of a cost if the pattern occurs far into the text than near the front, so it is important that starting cost be equal for all positions. In this case, $D[0, i] = 0$, since we pay no cost for deleting the first i characters of the text.

In both cases, $D[i, 0] = i$, since we cannot excuse deleting the first i characters of the pattern without cost.

SHOW FIGURE/TABLE OF DYNAMIC PROGRAMMING TABLE

What do we return?

If we want the *cost* of comparing all of the pattern against all of the text, such as comparing the spelling of two words, all we are interested in is $D[n, m]$.

But what if we want the cheapest match between the pattern anywhere in the text? Assuming the initialization for substring matching, we seek the cheapest matching of the full pattern ending anywhere in the text. This means the cost equals $\min_{1 \leq i \leq m} D[n, i]$.

This only gives the cost of the optimal matching. The actual alignment – what got matched, substituted, and deleted – can be reconstructed from the pattern/text and table without an auxiliary storage, once we have identified the cell with the lowest cost.

How much space do we need?

Do we need to keep all $O(mn)$ cells, since if we evaluate the recurrence filling in the columns of the matrix from left to right, we will never need more than two columns of cells to do what we need. Thus $O(m)$ space is sufficient to evaluate the recurrence without changing the time complexity at all.

Unfortunately, because we won't have the full matrix we cannot reconstruct the alignment, as above.

Saving space in dynamic programming is very important. Since memory on any computer is limited, $O(nm)$ space is more of a bottleneck than $O(nm)$ time.

Fortunately, there is a clever divide-and-conquer algorithm which computes the actual alignment in $O(nm)$ time and $O(m)$ space.