Approximate String Matching

An important task in text processing is string matching - finding all the occurrences of a word in the text. Unfortunately, many words in documents are mispelled (sic). How can we search for the string closest to a given pattern in order to account for spelling errors?

To be more precise, let *P* be a pattern string and *T* a text string over the same alphabet. The *edit distance* between *P*and *T* is the smallest number of changes sufficient to transform a substring of *T* into *P*, where the changes may be:

1. *Substitution* - two corresponding characters may differ: KAT $\rightarrow$ CAT.
2. *Insertion* - we may add a character to *T* that is in *P*: CT $\rightarrow$ CAT.
3. *Deletion* - we may delete from *T* a character that is not in *P*: CAAT $\rightarrow$ CAT.

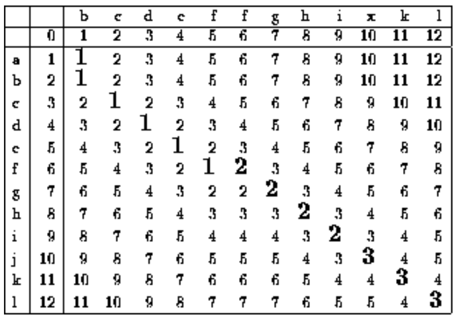
For example, *P=abcdefghijkl* can be matched to *T=bcdeffghixkl* using exactly three changes, one of each of the above types.

Approximate string matching arises in many applications, as discussed in Section . It seems like a difficult problem, because we have to decide where to delete and insert characters in pattern and text. But let us think about the problem in reverse. What information would we like to have in order to make the final decision; i.e. what should happen with the last character in each string? The last characters may be either be matched, if they are identical, or otherwise substituted one for the other. The only other options are inserting or deleting a character to take care of the last character of either the pattern or the text.

More precisely, let *D*[*i*,*j*] be 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 $(P_i = T_j)$, then *D*[*i*-1, *j*-1], else *D*[*i*-1, *j*-1]+1. This means we either match or substitute the *i*th and *j*th characters, depending upon whether they do or do not match.
2. *D*[*i*-1, *j*]+1. This means that there is an extra character in the pattern to account for, so we do not advance the text pointer and pay the cost of an insertion.
3. *D*[*i*, *j*-1]+1. This means that there is an extra character in the text to remove, so we do not advance the pattern pointer and pay the cost of a deletion.

The alert reader will notice that we have not specified the boundary conditions of this recurrence relation. It is critical to get the initialization right if our program is to return the correct edit distance. The value of *D*[0,*i*] will correspond to the cost of matching the first *i* characters of the text with none of the pattern. What this value should be depends upon what you want to compute. If you seek to match the entire pattern against the entire text, this means that we must delete the first *i* characters of the text, so *D*[0,*i*] = *i* to pay the cost of the deletions. But what if we want to find where the pattern occurs in a long text? It should not cost more if the matched pattern starts far into the text than if it is near the front. Therefore, the starting cost should 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 penalty.



		b	c	d	e	f	f	g	h	i	x	k	l
	0	1	2	3	4	5	6	7	8	9	10	11	12
a	1	**1**	2	3	4	5	6	7	8	9	10	11	12
b	2	**1**	2	3	4	5	6	7	8	9	10	11	12
c	3	2	**1**	2	3	4	5	6	7	8	9	10	11
d	4	3	2	**1**	2	3	4	5	6	7	8	9	10
e	5	4	3	2	**1**	2	3	4	5	6	7	8	9
f	6	5	4	3	2	**1**	**2**	3	4	5	6	7	8
g	7	6	5	4	3	2	2	**2**	3	4	5	6	7
h	8	7	6	5	4	3	3	3	**2**	3	4	5	6
i	9	8	7	6	5	4	4	4	3	**2**	3	4	5
j	10	9	8	7	6	5	5	5	4	3	**3**	4	5
k	11	10	9	8	7	6	6	6	5	4	4	**3**	4
l	12	11	10	9	8	7	7	7	6	5	5	4	**3**

Figure: Example dynamic programming matrix for edit distance computation, with the optimal alignment path highlighted in bold

Once you accept the recurrence, it is straightforward to turn it into a dynamic programming algorithm that creates an $n \times m$ matrix *D*, where *n* = |*P*| and *m* = |*T*|. Here it is, initialized for full pattern matching:

```
EditDistance(P,T)

        (*initialization*)

        For i = 0 to n do D[i,0] = i

        For i = 0 to m do D[0,i] = i

        (*recurrence*)

        For i = 1 to n do

                For j = 1 to m do

                        $D[i,j] = \min(D[i-1,j-1] + matchcost(s_i, t_j),$

                                D[i-1,j]+1, D[i,j-1]+1 )
```

How much time does this take? To fill in cell $D[i,j]$, we need only compare two characters and look at three other cells. Since it requires only constant time to update each cell, the total time is $O(mn)$.

The value to return as the answer to our pattern matching problem depends on what we are interested in. If we only needed the cost of comparing all of the pattern against all of the text, such as in comparing the spelling of two words, all we would need is the cost of $D[n,m]$, as shown in Figure . But what if we need to identify the best matching substring in the text? Assuming that the initialization was performed correctly for such substring matching, we seek the cheapest matching of the full pattern ending anywhere in the text. This means the cost equals $\min_{1 \leq i \leq m} D[n,i]$, i.e. the smallest cost on the last row of D.

Of course, this only gives the cost of the optimal matching, while we are often interested in reconstructing the actual alignment - which characters got matched, substituted, and deleted. These can be reconstructed from the pattern/text and table without an auxiliary storage, once we have identified the cell with the lowest cost. From this cell, we want to walk upwards and backwards through the matrix. Given the costs of its three neighbors and the corresponding characters, we can reconstruct which choice was made to get to the goal cell. The direction of each backwards step (to the left, up, or diagonal to the upper left) identifies whether it was an insertion, deletion, or match/substitution. Ties can be broken arbitrarily, since either way costs the same. We keep walking backwards until we hit the end of the matrix, specifying the starting point. This backwards-walking phase takes $O(n+m)$ time, since we traverse only the cells involved in the alignment.

The alert reader will notice that it is unnecessary to keep all $O(mn)$ cells to compute the cost of an alignment. If we evaluate the recurrence by filling in the columns of the matrix from left to right, we will never need more than two columns of cells to store what is necessary for the computation. Thus $O(m)$ space is sufficient to evaluate the recurrence without changing the time complexity at all. Unfortunately, without the full matrix we cannot reconstruct the alignment.

Saving space in dynamic programming is very important. Since memory on any computer is limited, $O(nm)$ space proves more of a bottleneck than $O(nm)$ time. Fortunately, there is a clever divide-and-conquer algorithm that computes the actual alignment in $O(nm)$ time and $O(m)$ space. This algorithm is discussed in Section .

Next Up Previous Contents Index CD Home Lecture Notes Algorithms Repository

Next: Longest Increasing Sequence **Up:** Dynamic Programming **Previous:** The Partition Problem

Algorithms
Mon Jun 2 23:33:50 EDT 1997