CIS 320: Algorithms October 4, 2021

Lecture 9

Lecturer: Aaron Roth *Scribe: Aaron Roth*

Dynamic Programming III: Sequence Alignment

Suppose you try and remember how to spell "beurocracy", but as always, fail. You type it into your favorite text editor and it suggests the replacement 'bureaucracy". How did it know? It figured out that the string you typed in was *close* to the word you wanted: For example, you could line them up as follows:

```
beuro__cracy
b_ureaucracy
```

Each letter in the top is "matched" to the letter in the word below it. The _ symbols represent a "gap" in the matching – for example, the first "e" in the top string is not matched to anything in the bottom string. Some of the letters are matched onto their copies — for example, the "b" above is matched to a "b" below. But some are not — for example, the "o" is matched to an "e". We might consider scoring such a matching with a cost — no cost for matching a letter with its match, but a positive cost for a mismatch (maybe a lower cost for pairs of letters that are easy to mis-type, and a higher cost for other pairs), and some other cost for unmatched letters. But there are many (infinitely many!) ways to insert gaps into words and match the resulting letters. Given *costs* for different kinds of mismatches, how can we find the cost of the cheapest such matching? The asymptotic running time of such algorithms is less important in the domain of spell checking, in which words are generally very short — but becomes much more important when we are talking about matching the strings of base pairs making up chromosomes, which can have lengths reaching into the hundreds of millions. This is the kind of thing that computational biologists do when trying to reproduce evolutionary trees from the genomes of present-day animals.

To formalize the problem, we imagine that we are given two strings of (possibly) different lengths, X and Y. We write $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$ where x_i, y_j represent characters in some common alphabet.

Definition 1 (Matching) *A matching $M \subseteq \{1, \ldots, m\} \times \{1, \ldots, n\}$ is a set of ordered pairs such that each index appears at most once: i.e. for each pair $(i, j), (i', j') \in M$, $i \neq i'$ and $j \neq j'$. An index $i \in [m]$ is said to be* unmatched *if $(i, j) \notin M$ for all j. Similarly, an index $j \in [n]$ is said to be unmatched if $(i, j) \notin M$ for all i.*

Not all matchings are legal *alignments* however:

Definition 2 (Alignment) *A matching M is an alignment if for all $(i, j), (i', j') \in M$, if $i < i'$ then $j < j'$. Visually, this means that if we draw lines between the matched pairs of symbols in the strings, the lines will not cross.*

For example, the intuitive alignment:

```
stop_
_tops
```

is formalized as the alignment $M = \{(2, 1), (3, 2), (4, 3)\}$.

Now, in order to formalize the problem of finding the *optimal* alignment between two strings, we need to define the *cost* of an alignment.

Definition 3 *Given two strings $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$ and an alignment M, the* cost *of M is the sum of its* gap *and* mismatch *costs:*

$$C(M) = G(M) + MM(M)$$

Here,

$$G(M) = \delta \cdot (|\{i \in [m] : i \text{ is unmatched}\}| + |\{j \in [n] : j \text{ is unmatched}\}|)$$

is the gap cost, which pays a gap penalty δ *for each unmatched position of* X *and* Y*, and*

$$MM(M) = \sum_{(i,j) \in M} \alpha_{x_i y_j}$$

is the sum of the mismatch costs $\alpha_{x_i y_j}$ *for each matched pair of characters. Generally we assume that for each character* p*,* $\alpha_{p.p} = 0$ *— i.e. we suffer no mismatch cost for pairing a character in* X *with the identical character in* Y*.*

The parameters δ and $\{\alpha_{p,q}\}$ are inputs to the problem. Given these and a pair of strings X, Y, we seek to find the alignment M of minimum cost.

To design a dynamic programming algorithm, we need to identify a recursive solution to our problem. Suppose we are given two strings $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$. It may be that the last character in each string is matched: $(x_m, y_n) \in M$. In this case, we should continue and recursively find the minimum cost alignment of what remains of each string: $X' = x_1, \ldots, x_{m-1}$ and $Y = y_1, \ldots, y_{n-1}$. But what if $(x_m, y_n) \notin M$ — how do we make progress? Here is an important but simple observation:

Lemma 4 *Let* M *be any alignment of* X *and* Y*. If* $(m, n) \notin M$*, then either* X_m *or* y_n *is unmatched in* M*.*

Proof Suppose otherwise. Then there are indices $i < m$ and $j < n$ such that $(m, j), (i, n) \in M$. Since this is an alignment and we have $i < m$, we must also have $n < j$. But this is a contradiction, since n is the last index of Y. ■

Another way to restate this lemma which directly suggests a recursive formulation is as follows. In an optimal alignment between $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$, at least one of the following must be true:

1. $(m, n) \in M$, or
2. x_m is unmatched, or
3. y_n is unmatched.

Now we define some notation: Let $\text{OPT}(i, j)$ denote the minimum cost of any alignment between the string prefixes $X_i = x_1, x_2, \ldots, x_i$ and $Y_j = y_1, y_2, \ldots, y_j$. Our lemma above implies the following recursive identity:

$$\text{OPT}(i, j) = \min \left(\alpha_{x_i y_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1) \right)$$

where each of the three terms in the minimum correspond to the three cases above. With the recursion in hand, we're almost ready to write down our algorithm — in fact, we don't really have any design choices left. All that remains is to work out the base cases, which come from the observation that the best way to line up an i character string with a 0 character string is with i gaps: For all i, we have $\text{OPT}(i, 0) = \text{OPT}(0, i) = i \cdot \delta$. This is enough for our algorithm:

Algorithm 1 Alignment(X, Y)

Initialize an $m \times n$ array A.
Let $A[i, 0] = i\delta$ for each i.
Let $A[0, j] = j\delta$ for each j.
for $i = 1$ to m **do**
  for $j = 1$ to n **do**
    Let:

$$A[i, j] = \min\left(\alpha_{x_i y_j} + A[i-1, j-1], \delta + A[i-1, j], \delta + A[i, j-1]\right)$$

  end for
end for
Return $A[m, n]$

Once it is written in this form, the algorithm is easy to analyze. Its correctness follows from the fact that the cost for an alignment is additive and from the recursion we derived from Lemma 4. The running time can be seen to be $O(m \cdot n)$ from the fact that we perform $m \cdot n$ iterations of our inner loop, and each takes constant time. Just as we did in the previous lectures, given the array A produced by a run of Alignment, we can recover the optimal alignment itself by tracing back the recursion, which also takes $O(m \cdot n)$ time.