Sequence Alignment

Introduction

> *Trying to "line-up" two strings in the most optimal way possible, where "optimal" in this case means there's a minimal amount of gaps in the two strings, or mismatches between a pair of characters matched between the two strings.*

Suppose we are given two strings, X and Y where X consists of the sequence of symbols $x_1, ..., x_m$ and Y consists of the sequence $y_1, ...y_n$.

Consider the sets $\{1, 2, ..., m\}$ and $\{1, 2, ..., n\}$ as representing the different **positions** in the strings X and Y

A **matching** of two sets, M (where this matching uses the position sequences) is a set of ordered pairs (i, j) where each element from the set appears in at most one pair. Additionally there are no **crossings** - which is a fancy way of saying you must keep the characters of the string in order for both strings - don't pair early elements of one string with late elements of another.

Mathematically: if (i, j) and (i', j') are in the matching, and if $i < i'$ then $j < j'$.

Determining the "similarity" between two strings will be based on finding the *optimal* alignment of X and Y. Let's call M an alignment between X and Y We determine what is "optimal" based on two weights/penalties...

- $\delta > 0$ - **gap penalty.** For each position of X or Y that is not matched in M is a gap, that comes with a cost δ.
  - These gaps are represented as dashes in the matchings.
- $\alpha_{x_i y_j}$ - **mismatch cost** for a lining up of a (different) p and q. Generally, you assume $\alpha_{x_i, y_j} = 0$ when $x_i = y_j$, that being, when you have the same letter for your pairing - it costs nothing. Otherwise (when $x_i \neq y_j$), you assume some other $\alpha > 0$ which can be a constant value for *any* mismatch or variable based on some other rules.

The *cost* of M is the sum of the gap and mismatch penalties. The goal is to minimize this cost. The process of minimizing this cost is called **sequence alignment**.

The Algorithm

In an optimal alignment of M, one of the following is true;

- $(m, n) \in M$ or...
- the m^{th} position of X is not matched; or...
- the n^{th} position of Y is not matched.

In other words, either the last two symbols (x_m and y_n) in the two strings are matched, or one of them is mismatched (placed against a gap).

From here the actual *dynamic programming* solution can be built - based on the cases above being applied to cascading subproblems;

Let $\text{OPT}(i, j)$ represent the optimal (minimal cost) of aligning the *first* i characters from X and the first j characters of Y. This definition uses a recurrence relation based on the 3 possibilities above - slowly building up the overall optimal solution by using the optimal solution of smaller subsets (looking at OPT for $i - 1$ and/or $j - 1$) + the best cost for the current (i, j) pairing;

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

This recurrence relation is picking the best (lowest) cost result out of three possible cases (corresponding to the cases listed above).

$(i, j) \in M$

If x_i and y_j are matched, the cost is α_{x_i, y_j}
Then we add this to the cost of the preceding optimal solution: $\text{OPT}(i - 1, j - 1)$.

i^{th} position of X is not matched (put gap in Y)

If x_i is aligned with a gap, add the gap penalty δ

We then add $\text{OPT}(i - 1, j)$ - we don't use $j - 1$ as we used a gap, and thus haven't changed what the "current" j to be matched.

j^{th} position of Y is not matched (put gap in X)

If y_j is aligned with a gap, there is a gap penalty δ.

We then add $\text{OPT}(i, j - 1)$. Again, we don't use $i - 1$ as we put a gap there, and thus haven't changed the "current" i to be matched next.

Using this recurrence relation, we build up until we get our solution value of $\text{OPT}(m, n)$.

In Code

The algorithm is implemented using the following psuedocode;

```
Alignment(X,Y)
    Array A[0 . . . m,0... n]
    Initialize A[i, 0]= iδ for each i
    Initialize A[0, j]= jδ for each j
    For j = 1, . . . , n
        For i = 1, . . . , m
            Use the recurrence equation above to compute A[i, j]
        Endfor
    Endfor
    Return A[m, n]
```

This implementation uses a (pretty typical) 2D `DP` array. Let's call it `A`. This array is of size $(m + 1) \times (n + 1)$. (Recall that m is for X and n is for Y). This array will store the minimum alignment costs for different subproblems, where `A[i][j]` represents the cost of aligning the first i characters of X with the first j characters of Y. In other words, `A[i][j]` = $\text{OPT}(i, j)$.

For purposes of initialization, we note that $\text{OPT}(i, 0) = \text{OPT}(0, i) = i\delta$ for all i, since the only way to line up an i-letter word with a 0-letter word is to use i gaps.

Meaning that the array is of size $(m + 1) \times (n + 1)$ because we have an additional row/column representing the use of just gaps. These rows/columns are filled up incrementally by δ to denote an increasing number of used gaps.

For example, for a matching of "ACGT" with "AGCT" it would start as looking something like this... (assuming $\delta = 1$)

		A	C	G	T
	0	1	2	3	4
A	1				
G	2				
C	3				
T	4				

From here, you just fill in the rest of the DP cells based on the recurrence relation established above.

In other words, define every `A[i][j]` where;

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

> *We go left-to-right, top-to-bottom to satisfy the needed equation dependencies at each cell.*

Assuming $\delta = 1$ and $\alpha = 1$ (for any mismatch). The table for ACGT, AGCT would look something like this;

		A	C	G	T
	0	1	2	3	4
A	1	0	1	2	3
G	2	1	1	1	2
C	3	2	2	2	2
T	4	3	3	3	2

Final answer is stored in `A[m][n]`

Analysis

$\Theta(mn)$ time and space - you make a table, and fill it out linearly. (simpul as)