

Notes on space saving alignment computation

Vineet Bafna

1 Note:

These lecture notes are meant to supplement the lecture, have been written in haste, and may contain minor errors. Please send email to vbafna@cs.ucsd.edu if you find an error. The first person to locate each specific error gets a credit of 0.5%.

2 Introduction

First, recall the basic alignment strategy for aligning two strings $s[1 \dots n]$, and $t[1 \dots m]$. As input, we are given strings s, t , and also, a column scoring function \mathcal{C} that scores any pair of symbols (including '-') that are aligned together.

Let $S(i, j)$ denote the best score of an alignment of the prefix strings $s[1 \dots i]$, and $t[1 \dots j]$. We initialize as follows:

$$S(i, j) = \begin{cases} 0 & i = j = 0 \\ \mathcal{C}(s_i, '-') + S(i - 1, j) & \text{if } j = 0 \\ \mathcal{C}('-', t_j) + S(i, j - 1) & \text{if } i = 0 \end{cases} \quad (1)$$

and for $i > 0, j > 0$

$$S(i, j) = \max \begin{cases} S(i - 1, j - 1) + \mathcal{C}(s_i, t_j) & (\text{Also, } M(i, j) = \swarrow) \\ S(i - 1, j) + \mathcal{C}(s_i, '-') & (\text{Also, } M(i, j) = \uparrow) \\ S(i, j - 1) + \mathcal{C}('-', t_j) & (\text{Also, } M(i, j) = \leftarrow) \end{cases} \quad (2)$$

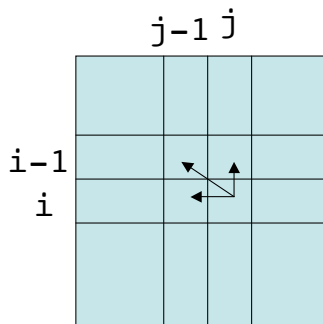


Figure 1: schematic of the dynamic programming computations. Note that in considering row i , we only need row $i - 1$. So two m -length arrays should suffice.

Figure 1 clarifies this pictorially. Note that in considering row i , we only need row $i - 1$. Using modulo 2

arithmetic, we have

$$\begin{aligned}
 i_1 &= i \bmod 2 \\
 i_2 &= (i - 1) \bmod 2 \\
 S(i_1, j) &= \max \begin{cases} S(i_2, j - 1) + \mathcal{C}(s_i, t_j) \\ S(i_2, j) + \mathcal{C}(s_i, '-') \\ S(i_1, j - 1) + \mathcal{C}('- ', t_j) \end{cases}
 \end{aligned}$$

As i_1, i_2 alternate between taking values 0, 1, two arrays suffice. However, the values in $M(i, j)$ can no longer be maintained within linear space. Hirschberg came up with a trick that will allow you to trade-off space for some time.

2.1 Reverse alignment

Recall that $S(i, j)$ denotes the score of an optimal alignment of the *prefix* strings $s[1..i], t[1..j]$. In this formulation, $S(0, 0) = 0$ and $S(n, m)$ denotes the score of the optimal alignment, where n, m are the lengths of s, t respectively.

We can also do the alignment on *suffix* strings. Let $S^r(i, j)$ denote the score of the optimal alignment of the suffix strings $s[i + 1 \dots n], t[j + 1, \dots, m]$. Now, $S^r(n, m) = 0$, and $S^r(0, 0)$ is the opt alignment score. Convince yourself that S^r can be computed analogous to S in $O(nm)$ time.

1. Describe the recurrences for computing S^r .

2.2 Computing Mid-points

Consider the optimum alignment. Clearly, there is a column such that the left part is an optimal alignment of $s[1..n/2]$ and $t[1..j]$ for some j . Likewise, the right half is an optimal alignment of $s[n/2 + 1, n]$ and $t[j + 1, \dots, m]$.

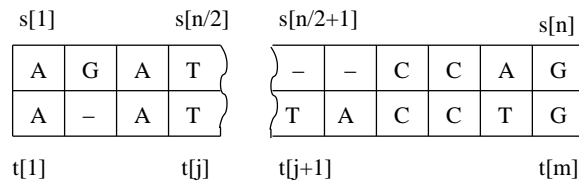


Figure 2: Computing the mid-point of an optimum alignment

The space saving algorithm has the following idea:

Align(s,t)

(* Compute the opt alignment of substrings s, t *)

1. Compute the coordinate j that splits the string t .
2. Recursively align the two halves, and concatenate.

This can be made more precise.

2.3 Analysis

We need to show two things. First, if we can compute the split j for string t in linear ($O(m)$) space, we can do the entire computation in $O(m)$ space. However, this is simple. Note that the score of the left half of the alignment is exactly $S[n/2, j]$, and the right half of the alignment (suffix strings) is $S^r[n/2, j]$. Therefore, if we split t at position j , the total alignment score will be exactly $S[n/2, j] + S^r[n/2, j]$. If the alignment is optimal, it should have the highest score over all choices of j . Therefore,

$$j = \arg \max_{1 \leq j \leq m} S[n/2, j] + S^r[n/2, j]$$

Thus the computation is clearly $O(m)$ space. What about time. It takes $O(nm)$ time to compute S, S^r . When we recurse, the two sub-parts have size $nj/2$ and $n(m-j)/2$, so the total 'size' of the recursive problem $nm/2$. In other words,

$$T(nm) = nm + T(nm/2) \leq nm \left(1 + \frac{1}{2} + \frac{1}{2^2} + \dots \right) = 2nm$$

To see the last equation, let

$$R = 1 + \frac{1}{2} + \frac{1}{2^2} + \dots$$
$$2(R - 1) = 1 + \frac{1}{2} + \frac{1}{2^2} + \dots = R$$

implying that $R = 2$. Thus, we double the running time, but reduce space requirements from $O(nm)$ to $O(n+m)$. This is critical when aligning long genomic strings.