18.417 Introduction to Computational Molecular Biology

Lecture 7: September 30, 2004 — Scribe: Mark Halsey
Lecturer: Ross Lippert — Editor: Rob Beverly

Divide and Conquer: More Efficient Dynamic Programming

Introduction

We have seen both global and local alignment problems in previous lectures. Briefly:

- Global Alignment: We are given:
  - Two strings **v** and **w**
  - An indel (insert/delete) penalty σ
  - A match/mismatch scoring matrix δ

  The goal is to find an alignment, including indels, matches and mismatches of the two strings such that the score is maximized. There may be multiple optimal solutions.
- Local Alignment: Unfortunately, in many instances there will only be a portion of the two strings that is highly conserved. Thus, the problem is to find substrings of **v** and **w** that are highly similar while ignoring the remainder of the strings. The input to the problem is the same as in global alignment. The output should be substrings of **v** and **w** whose global alignment is maximal among all possible substrings of **v** and **w**.

We have used dynamic programming to design efficient algorithms to solve these problems. This lecture considers two correct improvements to the alignment problem:

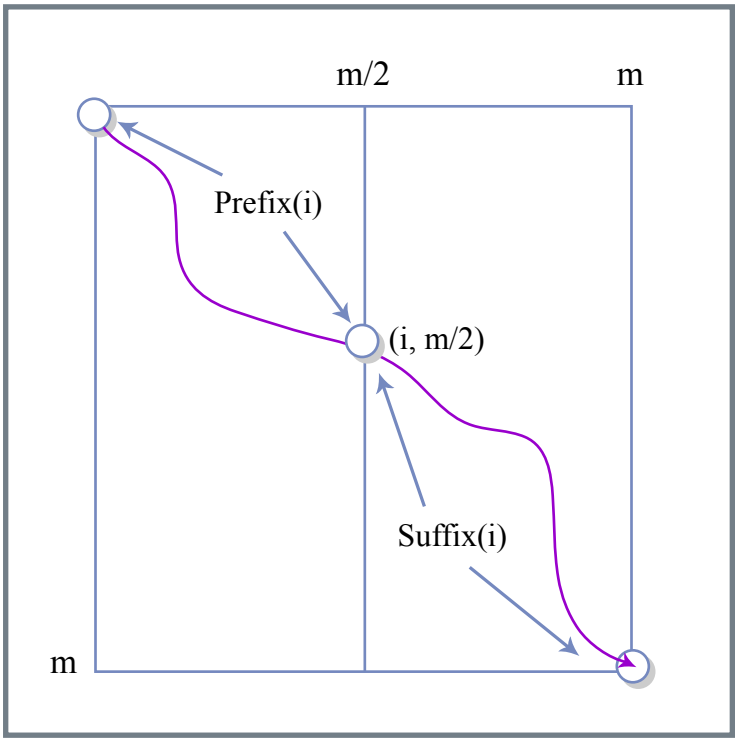
- Space-Efficient Sequence Alignment: computing the alignment solution again in $O(nm)$ but with only $O(\min(\{n, m\})) = O(n)$ (linear) space.
- Block Alignment and the Four-Russians Speedup: computing alignment in $O(\frac{n^2}{logn})$ time.

Space-Efficient Sequence Alignment

The space complexity of the algorithms we have seen previously is proportional to the number of vertices in the edit graph, i.e. $O(nm)$. Observe however that the only values needed to compute the alignment scores in column j in the DP table are the scores in column $j-1$. Therefore only two columns worth of space are required to compute the best score which is $O(n)$. However, recall that we use the **b** matrix to store backtracking pointers in order to reconstruct the longest path in the edit graph. **b** is an nxm matrix, so some clever insight is needed to bring the space needs down.

Consider the edit graph. Any optimal alignment from $(0,0)$ to (n,m) $m$ust pass through the middle column $\frac{m}{2}$. We will show that we can find the point i at which the optimal alignment passes through the middle column, i.e. the point $(i, \frac{m}{2})$ without knowing the longest path in the edit graph.

Vertex $(i, \frac{m}{2})$ partitions the edit graph into two optimal paths: $prefix(i)$ which is the optimal path from $(0,0)$ to $(i, \frac{m}{2})$ and $suffix(i)$ which is the optimal path from $(i, \frac{m}{2})$ to (n,m). This is shown graphically in Figure 7.1.



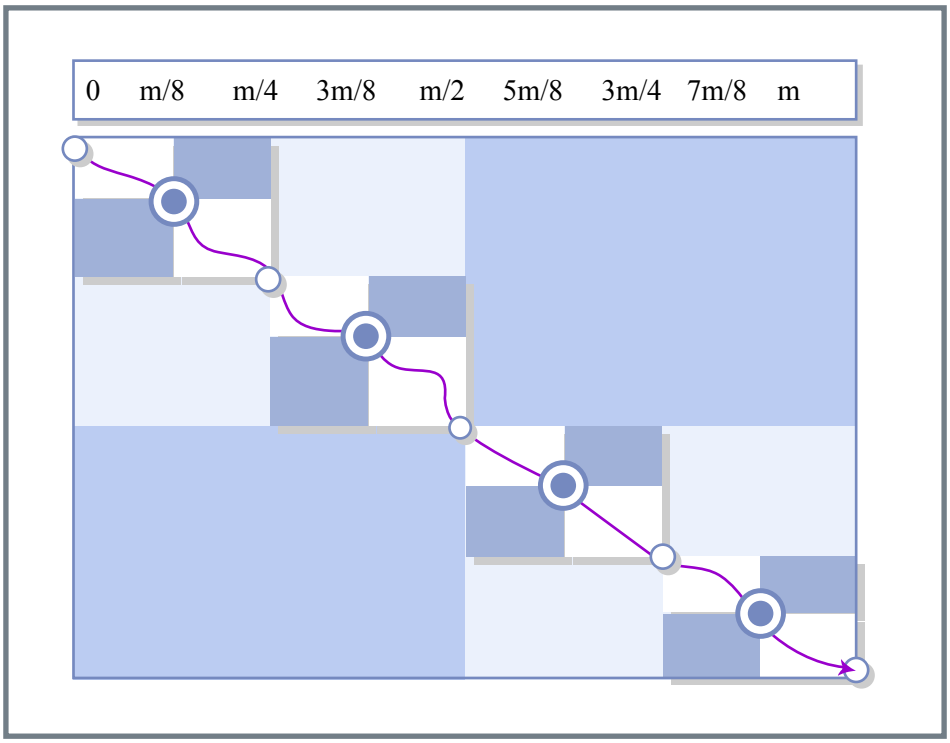
Adapted from Figure 7.1: Linear-Space Sequence Alignment

Note that the optimal alignment is simply $prefix(i) + suffix(i)$. $prefix(i)$ can be

computed by finding the score $s_{i,\frac{m}{2}}$, i.e. we compute the score in linear space as shown earlier for just the first half of the graph. To compute the $suffix$, we rely on the fact that in a DAG we can flip the direction of the edges and reverse the computation. Thus, for the second half of the edit graph (nx$\frac{m}{2}$), we can reverse edges and compute the score from (n, m) to $(i, \frac{m}{2})$.

Combining $prefix(i) + suffix(i)$ gives the score of optimal alignment that passes through $(i, \frac{m}{2})$. Because the space-efficient alignment score computation maintains column vectors, it is easy to determine $max_{0 \leq i \leq n}\,(prefix(i) + suffix(i))$. This in turn gives the optimal i which defines the optimal midpoint.

This process can be repeated by continual halving and computing the midpoint as shown in Figure 7.2. By iteratively halving and computing the optimal alignment midpoints we can reconstruct the complete optimal alignment. Note that after each halving we've reduced the time complexity of the subproblem in proportion to the area of rectangle defined by the optimal midpoints. Finding the midpoint of each rectangle requires: $area + \frac{area}{2} + \frac{area}{4} + \cdots$. Thus, the total time complexity is $O(nm)$.

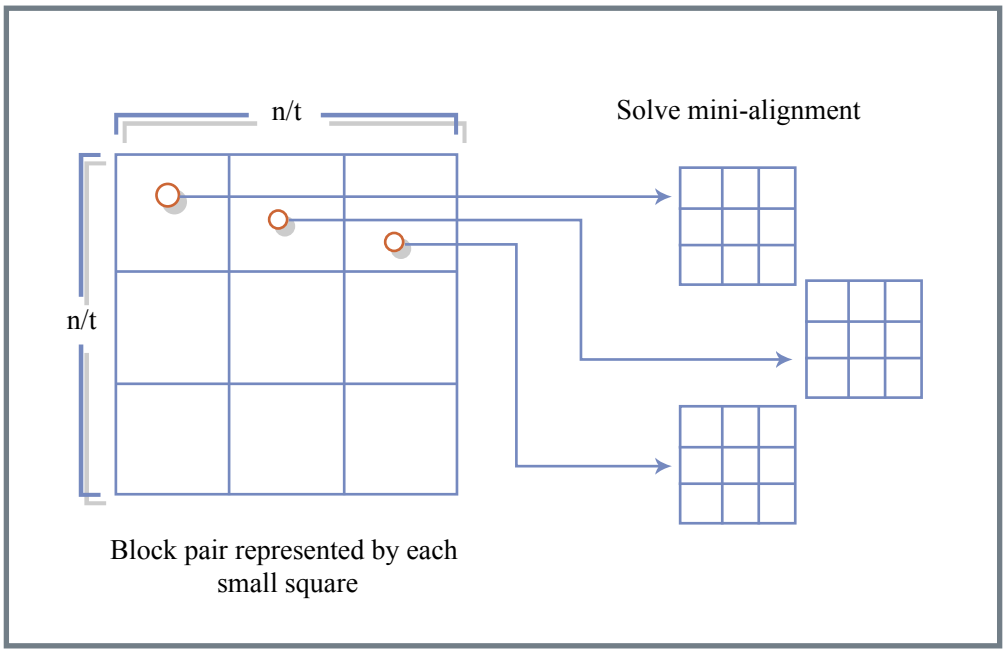


Adapted from Figure 7.2: Iteratively Computing the Optimal Alignment Midpoints

Block Alignment and the Four-Russians Speedup

The time complexity of the dynamic programming global alignment algorithm we've studied previously was $O(n^2)$. In this section we examine a trick to speedup the algorithm to sub-quadratic time. Note that no non-trivial lower bound exists for global alignment and an $O(nlogn)$ would likely revolutionize bioinformatics. We will begin by examining the block alignment problem in conjunction with the Four-Russians speedup. The next section extends the intuition here to longest common subsequence (LCS) speedup.

Consider our two strings to align: **v** and **w**. Without loss of generality, assume that $n = |v| = |w|$ and are divisible by some t. We can partition v and w into $\frac{n}{t}$ chunks of size t. This partitioning leads to the edit graph of Figure 7.3.



Adapted from Figure 7.3: Partitioning the Edit Graph into Mini-Alignments

If we were to solve the "mini-alignment" of each *txt* sub-grid, we could then perform *b*lock alignment of the blocks defined by the partitioning. In other words we construct a path that includes going through a block (from the top left to the bottom right) or along the edges of a block. Thus we are restricting entry and exit to the corners of blocks.

The block alignment problem is:

- Given: Two strings **v** and **w** partitioned into blocks of size t

- Output: The block alignment of **v** and **w** with the maximum score

Let $\beta_{i,j}$ be the alignment score for the (i, j) block. The recurrence for the block alignment algorithm is:

$$s_{i,j} = max \begin{cases} s_{i-1,j} - \sigma_{block} \\ s_{i,j-1} - \sigma_{block} \\ s_{i-1,j-1} + \beta_{i,j} \end{cases}$$

where σ_{block} is the indel block penalty. Since the indices of the recurrence vary from 0 to $\frac{n}{t}$, we have an $O(\frac{n}{t}^2)$ algorithm. But computing each block score $\beta_{i,j}$ requires solving $\frac{n}{t}^2$ mini-alignments of size txt which amounts to $O(n^2)$ time. Therefore we have not yet achieved any speed improvement.

The Four-Russians technique is to set $t = \frac{logn}{4}$ and precompute an exhaustive table of all 4^tx4^t alignments. 4^tx4^t = n total entries in the table. Computing each entry in the table requires $O(log^2 n)$ time, so to compute all n entries in the lookup table requires $O(nlog^2 n)$ time.

As noted above, the block alignment recurrence requires $O(\frac{n}{t}^2)$ time. Looking up an element in the lookup table takes $O(t)$. Therefore, given a lookup table, the block alignment algorithm takes $O(\frac{n^2}{t}) = O(\frac{n^2}{logn})$. We then add the time to compute the lookup table, but see that the overall time is dominated by the n^2 term. Therefore, the overall running time is: $O(\frac{n^2}{logn})$.

LCS and the Four-Russians Speedup

Finally, the path corresponding to the LCS does not necessarily enter and exit through the corners of blocks. In this section we turn to the more involved problem of allowing unrestricted entry and exit between blocks in the partitioned edit graph. We will rely on the intuition from the block alignment Four-Russians speedup.

Instead of performing dynamic programming on corner vertices of the blocks, we will use DP on the vertices of the edges of the blocks, ignoring the internal block vertices. This total $O(\frac{n^2}{t})$ vertices in the DP. Thus, the problem amounts to finding the alignment scores of the last row and last column of the txt blocks.

Using the Four-Russians speedup, we wish to construct a lookup table. Such a table would again include all pairs of t length strings and all pairs of possible scores for the

first row and first column. For each of these entries, the table would have precomputed scores for the last row and last column. But this table would be very large as it would include all possible scores for the first row and column. To alleviate this, we rely on the fact that the scores in the first row and column are not arbitrary. The scores must be both monotonically increasing and adjacent elements cannot differ by more than 1. Thus, the possible scores can be encoded as a vector of differences. This table is depicted in Figure 7.4.

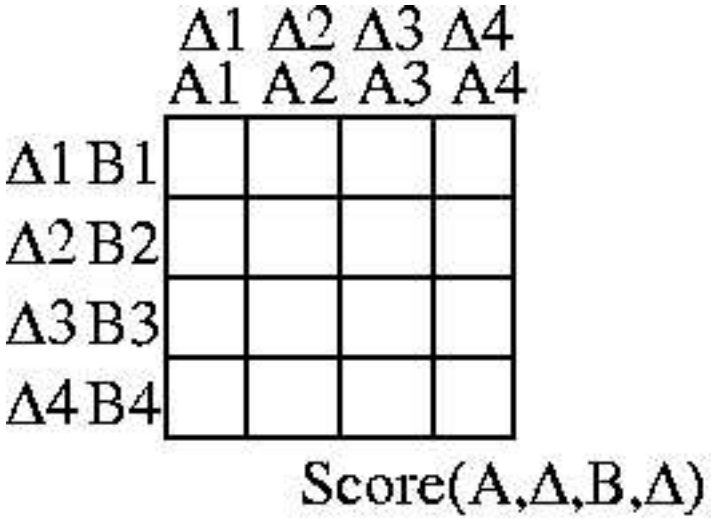


Figure 7.4: Mini-Alignment Lookup Table for LCS

As there are 2^t possible scores and 4^t possible strings, the lookup table requires $2^t 2^t 4^t 4^t = 2^{6t}$ space. Setting $t = \frac{logn}{4}$ as before makes the table of size $O(n^{1.5})$. This allows computation of the $n^{1.5}$ entries in the table to be constructed in $O(n^{1.5} log^2 n)$ time. As in the block alignment problem, this time is dominated by the DP and allows for an $O(\frac{n^2}{logn})$ algorithm.