

Rapid Global Alignments

March 9, 2009

Lecturer: Serafim Batzoglou

Scribe: Karl Uhlig

Summary

In aligning very long regions of DNA – for example, whole genomes of related species – the standard $O(N^2)$ alignment algorithms are computationally prohibitive. These algorithms are so expensive, in part, because they guarantee optimal results. However, in practice it is often the case that optimality is not strictly necessary. As a result, it is possible to relax this optimality constraint and instead produce useful alignments which are ‘very good’ or which capture some other type of information. The alignment algorithm presented in this lecture is one such technique, and is summarized roughly as follows:

1. Given two long DNA sequences, first generate a set of local alignments between the two sequences.
2. Next, find the largest ‘chain’ of these local alignments to produce the outline of a good global alignment.
3. Finally, use this chain to limit the scope of the full DP algorithm, and run regular Needleman-Wunsch on this narrow region to produce the global alignment.

Local alignments be generated quickly using existing algorithms like BLAST, and once we have an alignment chain we can a restricted version of Needleman-Wunsch in linear time (albeit with a large constant). This lecture, then, focuses on the intermediate problem of finding the optimal chain from a set of local alignments.

Motivation

In order to appreciate the usefulness of good global alignment, consider the problem of identifying important functional regions of DNA within a particular genome. It is sometimes possible to find such regions experimentally – typically, this is accomplished by modifying a region of the genome and directly observing its effects on subsequent generations of offspring.

There are several factors which complicate this process, however. For one, it is very difficult to reproduce all of the possible conditions under which a certain gene might become important. For instance, a gene might be totally inactive under normal conditions but might become crucial for survival under some starvation condition.

Furthermore, very small effects which are almost undetectable within a single generation can nevertheless have significant evolutionary consequences in the long run. As an example, suppose a particular mutation causes an organism to have (on average) 3% fewer offspring. This change might not be noticeable within the constraints of a single experiment, but 3% fewer offspring is enough to virtually eliminate the mutation in a population within 100 generations.

Global alignment of related genomes, then, provides a computational alternative to this problem. By modeling the accumulation of random mutations within a genome, alignment makes it possible to identify regions of DNA which are ‘highly conserved’ between species. The conservation of these regions is an indication that they play an important role in survival, and thus serve as good starting points for further study.

Chaining Local Alignments

Given a set of local alignments between two sequences, our task is to connect some subset of these alignments into the best overall global alignment. Intuitively, we take this approach in order to capture as much homology as possible between the sequences.

From the local alignment algorithm we know that every local alignment has some associated alignment score. Clearly, we want to maximize the sum of scores in our chain; more formally, we want to find the *heaviest weight chain* of local alignments.

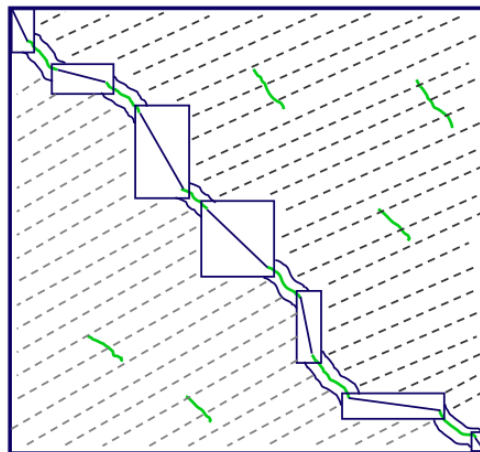


Figure 1: The optimal chain in a local alignment matrix

We will see that we can find the heaviest weight chain in $O(N \log N)$ time, where N is the number of local alignments. It is interesting to note that, in the worst case, the number of local alignments is quadratic in the length of the sequences. Therefore, the chaining algorithm described below has a worst-case running time of $O(N^2 \log N)$ in the length of the sequences. This is actually *worse* than the standard $O(N^2)$ for Needleman-Wunsch, but in practice the number of local alignments is small and the algorithm performs well on large sequences.

Longest Increasing Subsequence

To begin, we first consider the general problem of finding the *longest increasing subsequence* of a sequence x . Given a sequence (over an ordered alphabet),

$$x = x_1 x_2 \dots x_m$$

we wish to find a subsequence

$$s = s_1 s_2 \dots s_k$$

such that

$$s_1 < s_2 < \dots < s_k$$

We can find the LIS in $O(N \log N)$ time as follows:

1. Define L to be an array whose elements $L[j]$ store the smallest element x_k we have seen so far which is the final element in an increasing subsequence of length j . For example, if we are at iteration $i = 5$ and we have $L[3] = 7$, then 7 is the smallest element which ends an increasing subsequence of length 3 up to position 5 in the sequence. Also, let $B[j]$ store the index k of the element x_k in $L[j]$. Finally, let $P[i]$ store the index of the previous element in the chain ending with x_i .
2. For each element x_i in the sequence, find the index j such that $L[j - 1] < x_i \leq L[j]$, if such a position exists. Then, perform the updates

$$\begin{aligned} L[j] &\leftarrow x_i \\ B[j] &\leftarrow i \\ P[i] &\leftarrow B[j - 1] \end{aligned}$$

In words, for each i in the sequence, we check to see if the element x_i is the best choice (after seeing the first i elements) to be the final element in some increasing subsequence

of x . If multiple choices exist, then clearly the ‘best’ choice is the smallest such element, since it is this element which gives us the best chance to further extend the subsequence. Finally, after every element has been considered, the last value in L holds the final element in the LIS of x . We can use the backpointer arrays in order to reconstruct the actual subsequence.

It must be the case the the elements of L are in sorted order. Because of this, we can perform a binary search at step (2) and thus complete each iteration in $O(\log N)$ time. This gives a total running time of $O(N \log N)$.

Heaviest Weight Chain

The key concept from the LIS problem is that each time we consider an element x_i , we determine whether x_i could possibly be part of a ‘best’ increasing subsequence of x of length j . If so, we then eliminate from consideration any elements which have been rendered useless by x_i .

Using this same logic, we can extend the idea of the longest increasing subsequence to solve the alignment chaining problem. Suppose we have N local alignments numbered $1..N$ with scores $w(i)$; each alignment is bounded by a rectangle with left edge l_i , right edge r_i , top edge t_i , and bottom edge b_i . Note that we can chain two alignments i and j exactly when $r_i < l_j$ and $b_i < t_j$ (rectangle j is down and to the right of rectangle i).

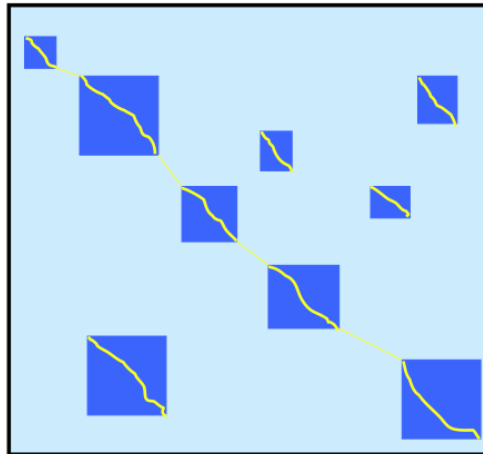


Figure 2: Local alignments defined by rectangles in the DP matrix.

Next, define $V(i)$ to be the largest total weight of any chain ending with rectangle i , and let L in this case be an array of tuples $(b_j, V(j), j)$. With these definitions, the general algorithm is given by:

1. Sweep from left to right and bottom to top in the alignment matrix.

2. When we hit the left edge l_i of some rectangle i , we find the rectangle j in L with the largest $b_j < t_i$. Set $V(i) = V(j) + w(i)$.
3. When we hit the right edge r_i of rectangle i , we find the rectangle j in L with the largest $b_j \leq b_i$. If $V(i) > V(j)$,
 - (a) Insert $(l_i, V(i), i)$ into L .
 - (b) Remove all tuples $(l_j, V(j), j)$ from L where $V(j) \leq V(i)$ and $b_j \geq b_i$.
4. The heaviest weight chain ends with the rectangle i in L with the largest value $V(i)$.

Let's examine this algorithm a little more closely. First, note that we only ever add a rectangle to L after sweeping past its right edge. This fact implies that any time we examine the left edge of a rectangle i , L only contains rectangles which completely precede rectangle i (from left to right).

Using this fact, at step (2) we can now see that $b_j < t_i$ ensures that rectangle j can be chained to rectangle i (this is true since the right edge of every rectangle j in L precedes i 's left edge). Thus, step (2) could be written as "find the (vertically) *lowest* rectangle in L which can be chained to rectangle i , and chain the two together". Using Figure 3 as an example, rectangle B satisfies this property.

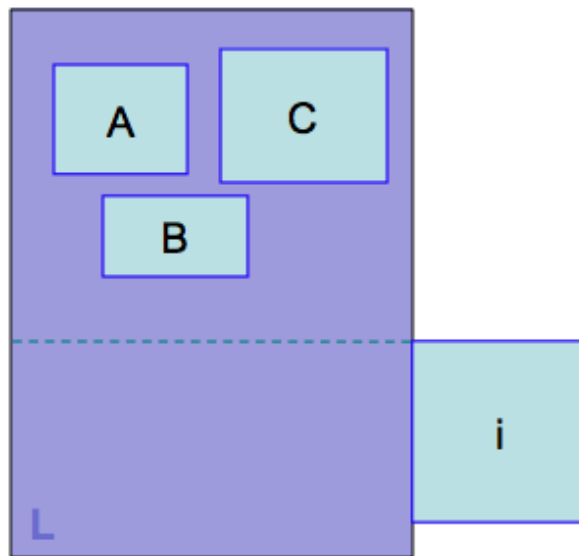


Figure 3: Choosing among rectangles in L .

It is clear that we can chain rectangles B and i , but it is less clear that this is the absolute best chain that can end with rectangle i . In order to see why this must be the case, consider the last part of step (3). After we insert a rectangle i into L , we remove all *lower* rectangles j in L whose total weight $V(j)$ is not larger than $V(i)$. We do this because any

rectangle k which we can chain to rectangle j , we can also chain to rectangle i . Therefore, if $V(i) \geq V(j)$, rectangle j is 'useless' and should be removed from consideration. This means that lower rectangles in L must have higher weights – consequently, step (2) does indeed produce the highest weight chain ending in rectangle i .

Finally, we need to verify that we only add a rectangle to L when it could potentially be part of the heaviest weight chain. The condition in step (3) that $V(i) > V(j)$ ensures this property: $b_j \leq b_i$ means that rectangle j can be chained to every rectangle which i can be chained to (the reverse, however, is not true). Therefore, we should only add i to L if $V(i) > V(j)$, since otherwise rectangle i would be useless.

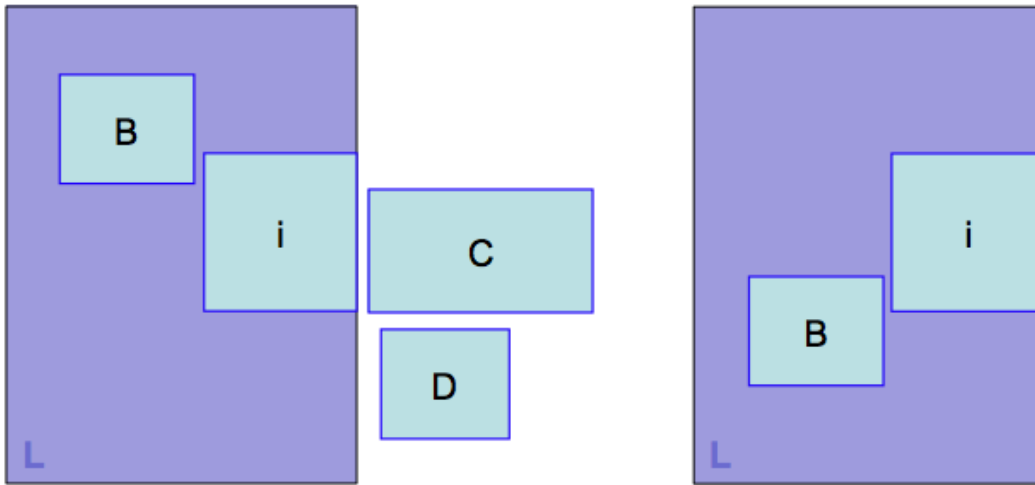


Figure 4: Possible configurations at step (3) of the chaining algorithm. On the left, if $V(i) \leq V(B)$, then i is useless. If $V(i) > V(B)$, then both rectangles must be considered (for example, C can be chained to B but not i). On the right, if $V(i) \geq V(B)$, then B is useless and is eliminated at step (3b).

To summarize, when we examine the left edge of a rectangle i , we can use L to compute the heaviest weight chain which ends with i . When we examine the right edge of i , we can use L to determine whether i is potentially useful. If it is (i.e. it could contribute to heaviest weight chain), we add it to L and remove any rectangles in L which are *rendered* useless by i .

Complexity Analysis

It may not be obvious, but the heaviest weight chain algorithm runs in the same $O(N \log N)$ time as the LIS problem. A general analysis is given below:

1. We must sort the rectangles by x - and y -coordinates. This can be done in $O(N \log N)$ time.

2. In step (2) of each iteration, we must search for a rectangle in L based on b_j -coordinate. If we manage L as a balanced binary tree sorted by b_j -coordinate, we can perform this search in $O(\log N)$ time.
3. When we hit the right edge at step (3) of the algorithm, we need to again search L by b_j coordinates. Next, if $V(i) > V(j)$ we need to perform the insertion and removal steps. Inserting into a binary tree requires $O(\log N)$ time, and it turns out that the removal stage also requires $O(\log N)$ time. This is true because the property that lower rectangles in L must have larger chain weights means that L is also sorted by $V(j)$ value. Therefore, searching for rectangles with $V(j) \leq V(i)$ and $b_j \geq b_i$ can be done in $O(\log N)$ time.

Overall, we need $O(N \log N)$ to sort, and $O(\log N)$ for each of N iterations. This gives the desired $O(N \log N)$ running time.