Computational Biology
Lecture 19: Chaining local alignments
Saad Mneimneh

Having found the maximal matches between x and y, we can put some of them together to form a chain. These maximal matches can be considered as high scoring local alignments. The chain then represents a global alignment (not necessarily optimal). Therefore, this is one way of obtaining a sub-optimal global alignmnent based on high scoring chucks (local alignments).

If the chaining of the local alignments can be done efficiently, for instance in better than $O(n^2)$ time, then we have a faster global alignment algorithm that produces a sub-optimal alignment, but a one that is good enough because it contains many high scoring local alignments.

How can we chain those local alignments? These local alignments (obtained as matches) have different lengths, and each has the form $(x_a...x_b, y_c...y_d)$. Therefore, each local alignment can be represented as a square in two dimensions as the figure below shows.

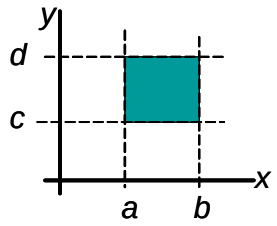

Figure 1: Local alignment (match) as a square

Two local alignments (squares) can be chained if the bottom left corner of one is above and to the right of the top right corner of the other. We will generalize the problem a little bit. Instead of the squares, we will consider rectangles. Each rectangle i has a weight $w(i)$. A rectangle i can be viewed now as a *gapped* local alignment with score $w(i)$. A chain (global alignment) is therefore a subset of the rectangles orderd in a way such that each rectangle j in the chain is above and to the right of the previous rectangle i; we say that rectangle j follows rectangle i in the chain. With the different weights of rectangles, we would like to obtain the chain that maximizes the sum of weights. Our goal is to obtain such a chain with a running time better than $O(n^2)$.

A simple solution, but $O(n^2)$ time

We present here a simple solution to the problem that requires an $O(n^2)$ running time. We construct a directed acyclic graph $G = (V, E)$ as follows:

- V contains one vertex i for every rectangle i
- E contains a directed edge (i, j) if rectangle j can follow rectangle i in some chain, i.e. if the bottom left corner of j is above and to the right of the top right corner of i.

Let $v(j)$ be the maximum weight of a chain that ends in rectangle j. Then the weight of the optimal chain is $\max_j v(j)$, since the optimal chain has to end in one of the rectangles.

An important observation is that $v(j)$ is independent of the weights of all rectangles i such that $(i, j) \notin E$ (j cannot follow i in any chain). Therefore, $v(j) = w(j) + \max_i v(i)$, i.e. the maximum weight of a chain ending in rectangle j is the weight of j plus the maximum weight of a chain ending in some rectangle i such that j could follow i. This suggests a simple algorithm for computing all the v_j's.

Algorithm

$v(j) \leftarrow w(j)$ for all j
topologically sort G [*if i before j, there is no edge (j, i), i.e. i cannot follow j in a chain*]
for all $j \in V$ in order
  do $v(j) \leftarrow w(j) + \max_{i|(i,j)\in E} v(i)$

The rectangle i with maximum $v(i)$ is the end of the optimal chain, and we can obtain the chain by keeping pointers and tracing back.

The following figure shown an example of running the algorithm where the chain of rectangles 2, 4, 9 is the optimal chain with weight $5 + 3 + 7 = 15$.

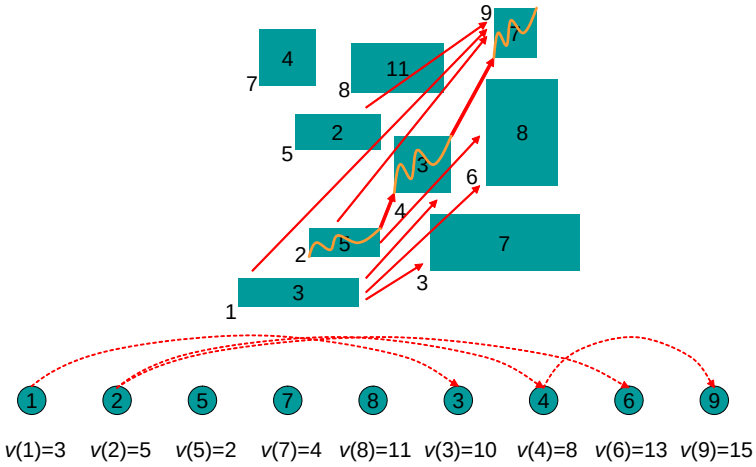

Figure 2: Example

Topological sort can be done in linear time in the size of the graph (i.e. number of vertices and edges in G); therefore in $O(n^2)$ time, where n is the number of rectangles. Similarly, updating $v(i)$ for all i takes $O(n^2)$ time since each updates takes $O(n)$ time. As a result, we have an $O(n^2)$ algorithm.

We would like to get a better time bound like $O(n \log n)$ for instance. In fact, the $O(n \log n)$ time bound can be achieved. We will first present an $O(n \log n)$ time algorithm for a simplified one dimensional problem (rectangles become segments), and then generalize it for two dimensions.

One dimension (segments), $O(n \log n)$ time

In this simplified problems the rectangles have no heights, i.e, they are segments on the x line as the figure below shows:

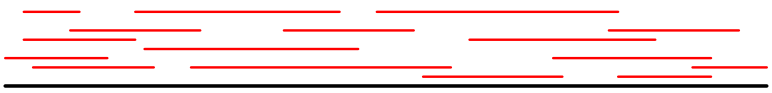

Figure 3: Segments

Therefore, a segment j can follow a segment i in some chain iff the two segments do not overlap. Assume we have n segments and let I be the list of the $2n$ end points of the n segments.

Algorithm

sort I
$V \leftarrow 0$
for $i = 1$ to $2n$
  do if $I[i]$ is left of segment j [*entering* j]
    then $v(j) \leftarrow w(j) + V$
  if $I[i]$ is right of segment j [*exiting* j]
    then $V \leftarrow \max(v(j), V)$

The value of V at the end of the algorithm is the weight of the optimal chain, and the chain itself can be obtained by a trace back strategy.

To argue for the correctness of the algorithm, the algorithm scans the segments in order from left to right. When entering a segment j, j has a potential to be the end of the chain and to contribute a weight of $w(j)$ to the maximum weight computed so far to make it $v(j) = V + w(j)$. But not until we exit segment j that this computed v_j is considered. Therefore, when exiting segment j, $v(j)$ is used as the maximum weight *unless* a better maximum V has been found before exiting j.

The running time of the algorithm is dominated by the sorting operation which takes $O(n \log n)$ time.

Two dimensions (back to rectangles), $O(n \log n)$ time

We will generalize the approach for the one dimension described above. Let I be the list of the left and right end points of the rectangles (x coordinates).

The chaining algorithm processes the entries in I in order (left to right) as in the one dimensional case. But the algorithm must also consider the y coordinates of each rectangle. Here's the idea: As we go through I, we keep a list L of some rectangles that are potential ends of the current chain (in the one dimensional case there is only one, but in two dimensions future rectangles can go up and down and we need to keep several options for the end of the current chain). Let l_j be the low y coordinate of rectangle j, and h_j be the high y coordinate of rectangle j.

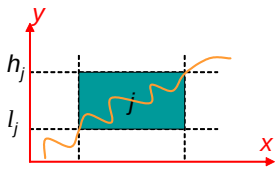

Figure 4: l_j and h_j of rectangle j

Each rectangle in L will be represented as a triple $(h_j, v(j), j)$ where h_j is as defined above, $v(j)$ is the maximum weight of a chain that ends in rectangle j, and j is the rectangle.

Entering a rectangle

When we enter a rectangle k, k has a potential to contribute $w(k)$ to the weight of the current chain. Rectangle k has to be chained to one of the rectangles in L to extend the chain. We look for the rectangle j in L that is closest to k (in the y dimension) with $h_j < l_k$. We then set $v(k) = w(k) + v(j)$. Is $v(k)$ computed as such the maximum weight of a chain ending in rectangle k? Let's see...

Assume that $v(k)$ is wrongly computed, i.e. there is a rectangle i above j in L and $v(i) > v(j)$ as illustrated below:

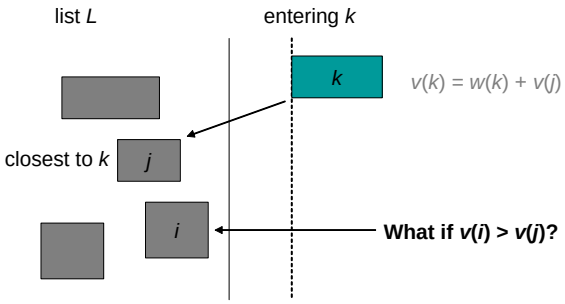

Figure 5: $v(k) \neq w(k) + v(j)$

Well, in that case, it is true for every k that if k can follow j, then k can follow i. Since $v(i) > v(j)$, the presence of j in L is not needed. Therefore we need to make sure that if $v(i) \geq v(j)$ and $h_j \geq h_i$, and $i \in L$, then $j \notin L$. In some sense j is more restrictive than i so if we have i, there is no need for j. This motivated the following definition:

Definition: If $v(i) \geq v(j)$ and $h_j \geq h_i$, then we say that rectangle j is more restrictive than rectangle i.

Note that the definition above represents a little abuse of the concept. For instance, if $v(i) = v(j)$ and $h_j = h_i$, then j is more restrictive than i and i is more restrictive than j. This abuse can be removed by breaking ties with the indices

i and j, but this is not so important. What is important is that if $i \in L$ and j is more restrictive than i, then we should not insert j in L.

But what if we have the following situation? j is not in L because it is more restrictive than $i \in L$, nevertheless, j is the one that should be used.

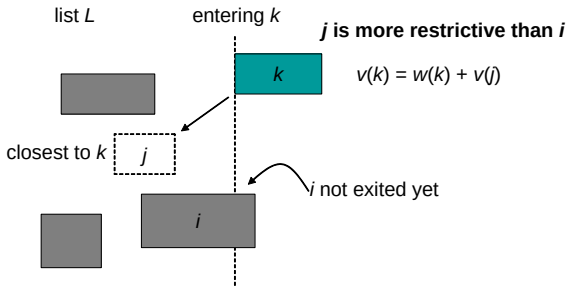

Figure 6: j is more restrictive than i, but $v(k) = w(k) + v(j)$

As illustrated above, k cannot follow i and j should be used! The problem here is that i was inserted in L too soon. If we make sure that a rectangle i is (possibly) inserted in L only when we exit i then this situation cannot occur.

Exiting a rectangle

When we exit a rectangle k, we insert it in L only if k is not more restrictive than some $j \in L$. Moreover, after we insert k, we delete from L all j that are more restrictive than k.

Therefore, L satisfies the following:
If $i \in L$ and $j \in L$, $h_i \neq h_j$ and $h_i < h_j \Leftrightarrow v(i) < v(j)$.

Entering and exiting rectangles as described above guarantees that $v(k) = w(k) + v(j)$, where $j \in L$ is closest to k with $h_j < l_k$, is computed correctly because:

- j is not more restrictive than any $i \in L$
- k can follow j because $j \in L$ means that j ends before k starts
- all j that end before k starts where considered for L

Algorithm

$L \leftarrow \emptyset$
for $i = 1$ to $2n$
do if $I[i]$ is left of rectangle k [*entering* k]
then find highest $h_j < l_k$ in L
$v(k) = w(k) + v(j)$

if $I[i]$ is right of rectangle k [*exiting* k]
then find highest $h_j \leq h_k$ in L
if $v(k) > v(i)$
then insert k in L
delete all entries j from L with $h_j \geq h_k$ and $v(j) \leq v(k)$

The maximum $v(j)$ in L is the weight of the optimal chain, and the chain can be obtained by a back tracing strategy.

Let us now analyze the running time of the algorithm. Sorting I takes $O(n \log n)$ time. We peep L as a balanced binary search tree sorted by h_j, e.g. AVL tree. Note that L will also be sorted by $v(j)$.

Searching L either for the highest $h_j < l_k$ or for highest $h_j \leq h_k$ takes $O(\log n)$ time (these correspond to predecessor operations in a binary search tree). Therefore, the total time spent on search operations is $O(n \log n)$. Similarly, an

insertion takes $O(\log n)$ time so the total time spent on insertion operations is $O(n \log n)$. Now let us look at the deletions. Although the time needed for a deletion is also $O(\log n)$, the concern here is the time spent on looking for all j's with $h_j \geq h_k$ and $v(j) \leq v(k)$ to delete them. Do we have to examine all j's? The answer is of course no. Since L is sorted by h_j, starting from h_k we investigate the successors one by one and delete each one with $v(j) \leq v(k)$. Can we end up investigating more entries than we delete? This in fact can be avoided since L is also sorted by $v(j)$. Once an entry with $v(j) > v(k)$ is found, we can stop investigating further entries. Therefore, we investigate at most one entry without deleting it. Once an entry is deleted it is never inserted back. The total time of deletion operations will therefore be $O(n \log n + n) = O(n \log n)$.

The total running time of the algorithm is $O(n \log n)$.

References

Gusfield D., Algorithms on Strings, Trees, and Sequences, Chapter 13.