Bioinformatics

Lecture 21
Chaining local alignments

- Having found many maximal matches (local alignments) between $x$ and $y$ with different lengths, we would like to chain them together to maximize the sum of lengths.

- Each match $x_a$, ..., $x_b$ and $y_c$, ..., $y_d$ can be represented as a square in two dimensions.

- Two squares can be chained if the top left corner of one is below and to the right of the bottom right corner of the other.
Generalizing

- We have rectangles, each with a weight $w$

- Two rectangles $i$ and $j$ can be in the same chain if the top left corner of $j$ is below and to the right of the bottom right corner of $i$, we say $j$ follows $i$ in the chain

- We would like to find a chain with maximum weight
Simple solution

- Construct a directed acyclic graph $G$:
  - one vertex for each rectangle
  - a directed edge from vertex $i$ to vertex $j$ iff rectangle $j$ can follow rectangle $i$ in some chain

- Let $v(i)$ be the maximum weight of a chain that ends in rectangle $i$.

Algorithm:

$v(j) \leftarrow w(j)$ for all vertices $j$

topologically sort $G$ (if $i$ before $j$, there is no edge $(j, i)$, i.e. $i$ cannot follow $j$ in a chain)

updating $v(i)$ can only affect $v(j)$ for $j > i$

for all vertices $j$ in order
  $v(j) \leftarrow w(j) + \max v(i)$ where edge $e = (i, j)$ exists

the rectangle $i$ with max $v(i)$ is the end of the optimal chain and we can trace back by keeping pointers
Example

\[
\begin{align*}
\nu(1) &= 3 \\
\nu(2) &= 5 \\
\nu(5) &= 2 \\
\nu(7) &= 4 \\
\nu(8) &= 11 \\
\nu(3) &= 10 \\
\nu(4) &= 8 \\
\nu(6) &= 13 \\
\nu(9) &= 15
\end{align*}
\]
Running time

• Topological sort can be done in linear time in the number of vertices and edges of $G$; therefore in $O(n^2)$, where $n$ is the number of rectangles

• Updating $v(i)$ for all $i$ takes $O(n^2)$ time as well

• We would like a better time bound like $O(n \log n)$

• The bound $O(n \log n)$ can be achieve

• We will consider an $O(n \log n)$ time algorithm for the one dimensional problem (rectangles become segments on the $x$ line) and then generalize it for two dimensions
One dimension

We have \( n \) segments

Let \( I \) be the list of all \( 2n \) left and right end points

```
sort I
V ← 0
for i = 1 to 2n
  if \( [i] \) is left of segment \( j \), set \( v(j) \) to \( w(j) + V \) [ entering \( j \) ]
  if \( [i] \) is right of segment \( j \), set \( V \) to \( \max(v(j), V) \) [ exiting \( j \) ]
```

The value of \( V \) at the end is the weight of the optimal chain

The chain itself can be obtained by the now familiar back tracing strategy
Correctness and time

• When entering a segment $j$, $j$ has a potential to participate in the chain and contribute a $w(j)$ to the $\text{max}$ weight computed so far to make it

\[ v(j) = V + w(j) \]

• When leaving segment $j$, $v(j)$ is used as the maximum weight unless a better maximum $V$ has been found before exiting $j$

• The running time is $O(n\log n)$ dominated by the sorting operation
Two dimensions

- We will generalize the approach for the one dimension

- Let \( l \) be the list of the left and right end points of the rectangles (\( x \) coordinates)

- The chaining algorithm processes the entries in \( l \) in order (left to right) as in the one dimension case

- But the algorithm must also consider the \( y \) coordinates of each rectangle
Idea

- As we go through $l$, we keep a list $L$ of some rectangles that are possible ends for the current chain

- Let $l_j$ be the low $y$ coordinate of rectangle $j$ and $h_j$ be the high $y$ coordinate of rectangle $j$

- Each rectangle in $L$ will be represented as a triple $(l_j, v(j), j)$ where:
  - $l_j$: low $y$ coordinate of rectangle $j$
  - $v(j)$: maximum weight of a chain that ends in rectangle $j$
  - $j$: the rectangle
Entering a rectangle

- When we enter a rectangle $k$, $k$ has a potential to contribute $w(k)$ to the weight of the 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 $l_j > h_k$.

- We set $v(k) = w(k) + v(j)$.

- Is $v(k)$ computed as above the maximum weight of a chain ending in rectangle $k$? Let’s see…
Computing $v(k)$

If $k$ can follow $j$, then $k$ can follow $i$

Therefore we need to make sure that if $v(i) \geq v(j)$ and $l_j \leq l_i$, rectangle $j$ is not in the list $L$
Restrictive rectangle

If
- \( v(i) \geq v(j) \) and
- \( l_j \leq l_i \)

then we say that rectangle \( j \) is more restrictive than rectangle \( i \)

If
- \( i \in L \) and
- \( j \) is more restrictive than \( i \)

then \( j \notin L \)
But what if…

list $L$ entering $k$

$j$ is more restrictive than $i$

$j$ not exited yet

$v(k) = w(k) + v(j)$

but here $k$ cannot follow $i$ and $j$ should be used!

make sure $i$ is inserted in $L$ only when we exit $i$
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:

  \[
  \text{If } l_i > l_j, \text{ then } v(i) < v(j)
  \]
Therefore...

The value of \( \nu(k) \) is computed correctly as

\[
\nu(k) = w(k) + \nu(j)
\]

where \( j \in L \) is closest to \( k \) with \( l_j > h_k \) 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

\[ L \leftarrow \emptyset \]
\[ \text{for } i = 1 \text{ to } 2n \]
\[ \text{begin} \]
\[ \text{if } l[i] \text{ is left of rectangle } k \quad \text{[entering } k]\]
\[ \text{then} \quad \text{find lowest } l_j > h_k \text{ in } L \]
\[ v(k) = w(k) + v(j) \]
\[ \text{if } l[i] \text{ is right of rectangle } k \quad \text{[exiting } k]\]
\[ \text{then} \quad \text{find lowest } l_j \geq l_k \text{ in } L \]
\[ \text{if } v(k) > v(j) \]
\[ \text{then} \quad \text{insert } k \text{ in } L \]
\[ \text{delete all entries } j \text{ from } L \text{ with } l_j \leq l_k \text{ and } v(j) \leq v(k) \]
\[ \text{end} \]

The maximum \( v(j) \) in \( L \) is the value of the maximum weight chain
The chain can be obtained by a back tracing strategy
Analysis

• Sorting $I$ takes $O(n \log n)$ time

• Keep $L$ as a balanced binary search tree sorted by decreasing order of $l_j$, e.g. AVL tree

• **Searching $L$:**
  
  - Either for lowest $l_j > h_k$ or for lowest $l_j \geq l_k$ takes $O(\log n)$ time
  
  - The total time of search is $O(n \log n)$

• **Inserting in $L$:**
  
  - Insertion operation takes $O(\log n)$ time
  
  - The time needed for all insertions is $O(n \log n)$
Analysis (cont.)

• Deleting from $L$:
  
  – All entries to be deleted start just after $(l_k, v(k), k)$ and are successive because $L$ is sorted by increasing order of $v(j)$

  – Therefore, successively examine $L$ starting after $(l_k, v(k), k)$ until the first $(l_j, v(j), j)$ with $v(j) > v(k)$ is found

  – Successor operation takes $O(\log n)$ time

  – Deletion operation takes $O(\log n)$ time

  – The total time needed for all deletions is $O(n\log n)$