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 $x_p, \ldots, x_q$ and $y_r, \ldots, y_s$ 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 bottom left corner of $j$ is above and to the right of the top 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 $j$ if rectangle $i$ can follow rectangle $j$ in some chain
- Let $v(i)$ be the maximum weight of a chain that ends in rectangle $i$

Algorithm:

- $v(1) = v(n)$ for all vertices $j$
- Topologically sort $G$; if $i$ before $j$, there is no edge $(i, j)$, i.e., $i$ cannot follow $j$ in a chain
- Updating $v(i)$ can only affect $v(a)$ for $a > i$
- For all vertices $j$ in order:
  - $v(j) = \max(v(i))$ where edge $(i, j)$ exists

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

Example

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 achieved
- 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
  
  ```
  \text{sort } I \\
  V \leftarrow I
  \text{for } i = 1 \text{ to } 2n \\
  \quad \text{if } i \text{ is left of segment } j, \text{ set } v(j) \leftarrow w(j) + V \\
  \quad \text{if } i \text{ is right of segment } j, \text{ set } V \leftarrow \max(v(j), V) \\
  \end{array}
  \quad \text{[entering } j] \\
  \quad \text{[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 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 $I$ 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 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 $(l_j, v_j, h_j)$, where:
  - $h_j$: high $y$ 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 $l_j < l_k$.
- 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 $v(j) > v(i)$ and $h_i > h_j$, rectangle $j$ is not in the list $L$.

$v(k) = w(k) + v_j$
Restrictive rectangle

If

\[ y_i \geq y_j \text{ and } z_i \geq z_j \]

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

If

\[ i \in L \text{ and } j \text{ is more restrictive than } i \]

then \( j \in L \)

But what if…

If \( i \in L \) then \( y_i \leq v_i \)

\[ v(k) = v(k) + v_j \]

\[ y_i = v_i \]

\[ x_i \leq v_i \]

\[ z_i \leq v_i \]

\[ a_i \leq v_i \]

\[ \text{not exited yet} \]

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:

\[ h_i < h_j \Rightarrow v_i < v_j \]
Therefore...

The value of \( v(k) \) is computed correctly as
\[
v(k) = w(k) + v(j)
\]
where \( j \in L \) is closest to \( k \) with \( l_j < l_k \) 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

\[
L \leftarrow \emptyset
\]
for \( j = 1 \) to \( 2n \) begin
If \( j \) is left of rectangle \( k \) then find highest \( l_j < l_k \) in \( L \)
\[
w(k) = w(k) + v(j)
\]
If \( j \) is right of rectangle \( k \) then find highest \( l_j < l_k \) in \( L \)
\[
w(k) = w(k) + v(j)
\]
If \( v(k) > v(j) \) then insert \( k \) in \( L \)
delete all entries \( j \) from \( L \) with \( l_j > l_k \) and \( v(j) \leq v(k) \)
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 takes \( O(n \log n) \) time
- Keep \( L \) as a balanced binary search tree sorted by \( l_j \), e.g., AVL tree
- Searching \( L \):
  - Either for highest \( l_j < l_k \) or for highest \( l_j \leq l_k \), 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 $(i, v(i), k)$ and are successive because $L$ is sorted by increasing order of $v(i)$.
  - Therefore, successively examine $L$, starting after $(i, v(i), k)$ until the first $(i, v(i), j)$ with $v(i) > v(j)$ is found.
  - Successor operation takes $O(\log n)$ time.
  - Deletion operation takes $O(\log n)$ time.
  - The total time needed for all deletions is $O(n \log n)$.