Goals for Lecture

Key concepts

• using tries and threaded tries to find alignment seeds
• using sparse dynamic programming (DP) to find a chain of local alignments
• constrained dynamic programming to align between/around anchors
LAGAN: Three Main Steps

1. Pattern matching to find seeds for global alignment
2. Find a good chain of anchors
3. Fill in with standard but constrained alignment

General

LAGAN

1. Threaded tries to obtain seeds
2. Sparse dynamic programming for chaining
3. Dynamic programming for gap filling

Brudno et al. Genome Research, 2003
Step 1: Finding Seeds in LAGAN

• *Degenerate k-mers*: matching $k$-long sequences with a small number of mismatches allowed

• By default, LAGAN uses 10-mers and allows 1 mismatch

\[
\text{cacg cgcgctacat acct}
\]
\[
\text{acta cgcggtacat cgta}
\]
Finding Seeds in LAGAN

• Example: a trie to represent all 3-mers of the sequence gaacgcgacct

• One sequence is used to build the trie
• The other sequence (the query) is “walked” through to find matching $k$-mers
Allowing Degenerate Matches

• Suppose we’re allowing 1 base to mismatch in looking for matches to the 3-mer acc; need to explore green nodes
LAGAN Uses Threaded Tries

- In a *threaded trie*, each leaf for word $w_1...w_k$ has a back pointer to the node for $w_2...w_k$
Traversing a Threaded Trie

• Consider traversing the trie to find 3-mer matches for the query sequence: accgt

• Usually requires following only two pointers to match against the next k-mer, instead of traversing tree from root for each
Step 1b: Chaining Seeds in LAGAN

- can chain seeds $s_1$ and $s_2$ if
  - the indices of $s_1 >$ indices of $s_2$ (for both sequences)
  - $s_1$ and $s_2$ are near each other
- keep track of seeds in the “search box” as the query sequence is processed

Figure from: Brudno et al. *BMC Bioinformatics*, 2003
Step 2: Chaining in LAGAN

- use *sparse dynamic programming* to chain local alignments
The Problem: Find a Chain of Local Alignments

Each local alignment has a weight

FIND the chain with highest total weight

\[(x,y) \rightarrow (x',y')\]

requires

\[x < x'\]

\[y < y'\]
Sparse DP for rectangle chaining

- 1,..., N: rectangles
- (h_j, l_j): y-coordinates of rectangle j
- w(j): weight of rectangle j
- V(j): optimal score of chain ending in j
- L: list of triplets (l_j, V(j), j)

- L is sorted by l_j: smallest (North) to largest (South) value
- L is implemented as a balanced binary tree
Sparse DP for rectangle chaining

Main idea:

- Sweep through x-coordinates
- To the right of b, anything chainable to a is chainable to b
- Therefore, if $V(b) > V(a)$, rectangle a is “useless” for subsequent chaining
- In L, keep rectangles $j$ sorted with increasing $l_j$-coordinates $\Rightarrow$ sorted with increasing $V(j)$ score
Sparse DP for rectangle chaining

Go through rectangle x-coordinates, from lowest to highest:

1. When on the leftmost end of rectangle i:
   a. j: rectangle in L, with largest $l_j < h_i$
   b. $V(i) = w(i) + V(j)$

2. When on the rightmost end of i:
   a. k: rectangle in L, with largest $l_k \leq l_i$
   b. If $V(i) > V(k)$:
      i. **INSERT** $(l_i, V(i), i)$ in L
      ii. **REMOVE** all $(l_j, V(j), j)$ with $V(j) \leq V(i) \& l_j \geq l_i$
1. When on the leftmost end of rectangle $i$:
   a. $j$: rectangle in $L$, with largest $l_j < h_i$
   b. $V(i) = w(i) + V(j)$

2. When on the rightmost end of $i$:
   a. $k$: rectangle in $L$, with largest $l_k \leq l_i$
   b. If $V(i) > V(k)$:
      i. **INSERT** $(l_i, V(i), i)$ in $L$
      ii. **REMOVE** all $(l_j, V(j), j)$ with $V(j) \leq V(i)$ & $l_j \geq l_i$
Time Analysis

1. Sorting the x-coords takes $O(N \log N)$

2. Going through x-coords: $N$ steps

3. Each of $N$ steps requires $O(\log N)$ time:
   - Searching $L$ takes $\log N$
   - Inserting to $L$ takes $\log N$
   - All deletions are consecutive, so $\log N$ per deletion
   - Each element is deleted at most once: $N \log N$ for all deletions
     - Recall that INSERT, DELETE, SUCCESSOR, take $O(\log N)$ time in a balanced binary search tree
Constrained Dynamic Programming

- if we know that the $i^{th}$ element in one sequence must align with the $j^{th}$ element in the other, we can ignore two rectangles in the DP matrix.
Step 3: Computing the Global Alignment in LAGAN

- given an anchor that starts at \((i, j)\) and ends at \((i', j')\), LAGAN limits the DP to the unshaded regions
- thus anchors are somewhat flexible

Figure from: Brudno et al. *Genome Research*, 2003
Step 3: Computing the Global Alignment in LAGAN

## Comparing MUMmer and LAGAN

<table>
<thead>
<tr>
<th></th>
<th>Baboon</th>
<th>Chimpanzee</th>
<th>Mouse</th>
<th>Rat</th>
<th>Cow</th>
<th>Pig</th>
<th>Cat</th>
<th>Dog</th>
<th>Chicken</th>
<th>Zebrafish</th>
<th>Fugu</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exons</strong></td>
<td>232</td>
<td>176</td>
<td>230</td>
<td>230</td>
<td>224</td>
<td>174</td>
<td>176</td>
<td>182</td>
<td>68</td>
<td>48</td>
<td>150</td>
<td>1914</td>
</tr>
<tr>
<td><strong>MUMmer (%)</strong></td>
<td>100</td>
<td>100</td>
<td>8</td>
<td>9</td>
<td>40</td>
<td>44</td>
<td>47</td>
<td>37</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>41</td>
</tr>
<tr>
<td><strong>human exons</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>covered by ≥</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>90% alignment</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LAGAN (%)</strong></td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>99</td>
<td>100</td>
<td>100</td>
<td>99</td>
<td>99</td>
<td>88</td>
<td>77</td>
<td>98</td>
</tr>
<tr>
<td><strong>human exons</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>covered by ≥</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>90% alignment</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparing MUMmer and LAGAN

1. Pattern matching to find seeds for global alignment
2. Find a good chain of anchors
3. Fill in with standard but constrained alignment

**MUMmer**
1. Suffix trees to obtain MUMs
2. Longest Increasing Subsequence
3. Smith-Waterman, recursive MUMmer

**LAGAN**
1. k-mer trie to obtain seeds
2. Sparse dynamic programming
3. Dynamic programming