Alignment of Long Sequences

BMI/CS 776
www.biostat.wisc.edu/bmi776/
Spring 2011
Mark Craven
craven@biostat.wisc.edu

Goals for Lecture

the key concepts to understand are the following
• how large-scale alignment differs from the simple case
• the canonical three step approach of large-scale aligners
• using suffix trees to find MUMs (alignment seeds)
• using tries and threaded tries to find alignment seeds
• constrained dynamic programming to align between/around anchors
• using sparse DP to find a chain of local alignments
Pairwise Large-Scale Alignment: Task Definition

Given
- a pair of large-scale sequences (e.g. chromosomes)
- a method for scoring the similarity of a pair of characters

Do
- construct global alignment: identify matches between sequences as well as various non-match features

Large Scale Alignment Example:
Mouse Chr6 vs. Human Chr12
Why the Problem is Challenging

- sequences too big to make $O(n^2)$ dynamic-programming methods practical
- long sequences are less likely to be colinear because of rearrangements
  - initially we'll assume colinearity
  - we'll consider rearrangements in next lecture

General Strategy

1. perform pattern matching to find seeds for global alignment
2. find a good chain of anchors
3. fill in remainder with standard but constrained alignment method

Figure from: Brudno et al. Genome Research, 2003
## Comparison of Large-Scale Alignment Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Pattern matching</th>
<th>Alignment Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUMmer</td>
<td>suffix tree - MUMs</td>
<td>LIS variant</td>
</tr>
<tr>
<td>AVID</td>
<td>suffix tree - exact &amp; wobble matches</td>
<td>Smith-Waterman variant</td>
</tr>
<tr>
<td>LAGAN</td>
<td>$k$-mer trie, inexact matches</td>
<td>sparse DP</td>
</tr>
</tbody>
</table>

### The MUMmer System

*Delcher et al., Nucleic Acids Research, 1999*

**Given**: genomes $A$ and $B$

1. find all maximal, unique, matching subsequences (MUMs)
2. extract the longest possible set of matches that occur in the same order in both genomes
3. close the gaps
Step 1: Finding Seeds in MUMmer

- maximal unique match (MUM):
  - occurs exactly once in both genomes $A$ and $B$
  - not contained in any longer MUM

Genome $A$: tcgatcGACGATCGGGGCTAGATCGAATAACGAGAGCATAA
Genome $B$: gcattaGACGATCGGGGCTAGATCGAATAACGAGAGCATAA

- key insight: a significantly long MUM is certain to be part of the global alignment

Suffix Trees

- substring problem:
  - given text $S$ of length $m$
  - preprocess $S$ in $O(m)$ time
  - such that, given query string $Q$ of length $n$, find occurrence (if any) of $Q$ in $S$ in $O(n)$ time

- suffix trees solve this problem, and others
Suffix Tree Definition

• a suffix tree $T$ for a string $S$ of length $m$ is tree with the following properties:
  – rooted and directed
  – $m$ leaves, labeled 1 to $m$
  – each edge labeled by a substring of $S$
  – concatenation of edge labels on path from root to leaf $i$ is suffix $i$ of $S$ (we will denote this by $S_{i...m}$)
  – each internal non-root node has at least two children
  – edges out of a node must begin with different characters

Suffixes

$S = \text{“banana$”}$

suffixes of $S$

$\$
$a$
$na$
$ana$
$nana$
$anana$
$banana$
Suffix Tree Example

- \( \mathcal{S} = \text{“banana$”} \)
- add ‘$’ to end so that suffix tree exists (no suffix is a prefix of another suffix)

Solving the Substring Problem

- assume we have suffix tree \( T \)
- \( \text{FindMatch}(Q, T) \):
  - follow (unique) path down from root of \( T \) according to characters in \( Q \)
  - if all of \( Q \) is found to be a prefix of such a path return label of some leaf below this path
  - else, return no match found
Solving the Substring Problem

\[ Q = \text{nan} \]

\[ Q = \text{anab} \]

return 3

return no match found

MUMs and *Generalized* Suffix Trees

- build one suffix tree for both genomes \( A \) and \( B \)
- label each leaf node with genome it represents

Genome A: \( ccacg\# \)

Genome B: \( cct\$ \)

each internal node represents a repeated sequence

each leaf represents a suffix and its position in sequence
MUMs and Suffix Trees

- unique match: internal node with 2 children, leaf nodes from different genomes
- but these matches are not necessarily maximal

Genome A: `ccacg#`
Genome B: `cct$

```
represents unique match
```

```
MUMs and Suffix Trees

- to identify maximal matches, can compare suffixes following unique match nodes

Genome A: `acat#`
Genome B: `acaa$

```
the suffixes following these two match nodes are the same; the left one represents a longer match (aca)
```
Using Suffix Trees to Find MUMs

- O(n) time to construct suffix tree for both sequences (of lengths \( \leq n \))
- O(n) time to find MUMs - one scan of the tree (which is O(n) in size)
- O(n) possible MUMs in contrast to O(n^2) possible exact matches

- main parameter of approach: length of shortest MUM that should be identified (20 – 50 bases)

Step 2: Chaining in MUMmer

- sort MUMs according to position in genome A
- solve variation of Longest Increasing Subsequence (LIS) problem to find sequences in ascending order in both genomes

Figure from: Delcher et al., *Nucleic Acids Research* 27, 1999
Finding Longest Subsequence

- unlike ordinary LIS problems, MUMmer takes into account
  - lengths of sequences represented by MUMs
  - overlaps
- requires $O(k \log k)$ time where $k$ is number of MUMs

Types of Gaps in a MUMmer Alignment

1. SNP: exactly one base (indicated by ^) differs between the two sequences. It is surrounded by exact-match sequence.
   
   Genome A: `cgtcatgggcttcgtcggtg`
   Genome B: `cgtcatgggcatcgcgtcggtg`

2. Insertion: a sequence that occurs in one genome but not the other.
   
   Genome A: `cgggtaacccg.............ccctgggctgg`
   Genome B: `cgggtaacccgcttcggtgccctgggctgg`

3. Highly polymorphic region: many mutations in a short region.
   
   Genome A: `ccgcttcgcctgggctgctgcccgt`
   Genome B: `ccgcttcgccttgaccctggcctgtgcct`

4. Repeat sequence: the repeat is shown in uppercase. Note that the first copy of the repeat in Genome B is imperfect, containing one mismatch to the other three identical copies.
   
   Genome A: `cTGGGTGGGACAACGTaaaTGGGTGGGACAACGT`
   Genome B: `aTGGGTGGGACAACGTTggggggtgggggTTGGGTGGGACAACGTTa`

Figure from: Delcher et al., *Nucleic Acids Research* 27, 1999
Step 3: Close the Gaps

- SNPs:
  - between MUMs: trivial to detect
  - otherwise: handle like repeats
- inserts
  - transpositions (subsequences that were deleted from one location and inserted elsewhere): look for out-of-sequence MUMs
  - simple insertions: trivial to detect

Step 3: Close the Gaps

- polymorphic regions
  - short ones: align them with dynamic programming method
  - long ones: call MUMmer recursively w/ reduced min MUM length
- repeats
  - detected by overlapping MUMs

Figure from: Delcher et al. Nucleic Acids Research 27, 1999
The LAGAN Method
Brudno et al., Genome Research, 2003

Given: genomes A and B

anchors = find_anchors(A, B)
step 3: finish global alignment with DP constrained by anchors

find_anchors(A, B)
step 1: find local alignments by matching, chaining k-mer seeds
step 2: anchors = highest-weight sequence of local alignments
for each pair of adjacent anchors $a_1, a_2$ in anchors
  if $a_1, a_2$ are more than $d$ bases apart
    $A', B' =$ sequences between $a_1, a_2$
    sub-anchors = find_anchors($A', B'$)
    insert sub-anchors between $a_1, a_2$ in anchors
return anchors

Step 1a: Finding Seeds in LAGAN

• degenerate k-mers: matching $k$-long sequences with a small number mismatches allowed
• by default, LAGAN uses 10-mers and allows 1 mismatch

caacgcgctacatacct
actacgcggtacatcgta
Finding Seeds in LAGAN

- example: a trie to represent all 3-mers of the sequence *gaaccgacct*

- 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 $w_1...w_p$ has a back pointer to the node for $w_2...w_p$

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

Step 2: Chaining in LAGAN

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

Sparse DP for rectangle chaining

- \( 1, \ldots, N \): 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

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

requires

- \( x < x' \)
- \( y < y' \)

Each local alignment has a weight

FIND the chain with highest total weight
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 $V(j) \leq V(i)$ & $l_j \geq l_i$
Example

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


Example Alignment:
*E. Coli* O157:H7 vs. *E. coli* K-12

Figure from: Perna et al. *Nature*, 2001