Large-Scale Global Alignments

Multiple Alignments

ARACHNE: Steps to Assemble a Genome

1. Find overlapping reads

2. Merge good pairs of reads into longer contigs

3. Link contigs to form supercontigs

4. Derive consensus sequence

..ACGATTACAATAGGTT..
3. Link Contigs into Supercontigs (cont’d)

Define T: contigs linked to either A or B
Fill gap between A and B if there is a path in G passing only from contigs in T

4. Derive Consensus Sequence

Derive multiple alignment from pairwise read alignments
Derive each consensus base by weighted voting

Simulated Whole Genome Shotgun

- Known genomes: Flu, yeast, fly, Human chromosomes 21, 22
- Make "realistic" shotgun reads
- Run ARACHNE
- Align output with genome and compare

Making a Simulated Read

Simulated reads have error patterns taken from random real reads

Human 22, Results of Simulations

<table>
<thead>
<tr>
<th>Average Contig</th>
<th>N50 scaffold</th>
<th>Mean contig</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg base qual</td>
<td>4.1 Kb</td>
<td>2.7 Kb</td>
</tr>
<tr>
<td>% &gt; 2 kb</td>
<td>3 Mb</td>
<td>15 Kb</td>
</tr>
<tr>
<td>N50 scaffold</td>
<td>2.0 Kb</td>
<td>10.6 Kb</td>
</tr>
<tr>
<td>N50 contig</td>
<td>3 X/0 X</td>
<td>5 X/0.5 X</td>
</tr>
<tr>
<td>Plasmid/Cosmid cov</td>
<td>6</td>
<td>260</td>
</tr>
</tbody>
</table>
Mouse Genome

Improved version of ARACHNE assembled the mouse genome

Several heuristics of iteratively:
- Breaking supercontigs that are suspicious
- Rejoining supercontigs

Size of problem: 32,000,000 reads

Time: 15 days, 1 processor
Memory: 28 Gb

N50 Contig size: $16.3 \text{ Kb} \rightarrow 24.8 \text{ Kb}$
N50 Supercontig size: $.265 \text{ Mb} \rightarrow 16.9 \text{ Mb}$

Next few lectures

More on alignments
- Large-scale global alignment – Comparing entire genomes
  - Suffix trees, sparse dynamic programming
    - Mummer, Avid, LAGAN, Shuffle-LAGAN
- Multiple alignment – Comparing proteins, many genomes
  - Scoring, Multidimensional DP, Center/Star, Progressive alignment
    - CLUSTALW, TCOFFEE, MLAGAN

Gene recognition
- Gene recognition on a single genome
  - GENSCAN - A HMM for gene recognition
- Cross-species comparison-based gene recognition
  - TWINSCAN - A HMM
  - SLAM - A pair-HMM

Rapid Global Alignments

How to align genomic sequences in (more or less!) linear time

Motivation

- Genomic sequences are very long:
  - Human genome $= 3 \times 10^9$ - long
  - Mouse genome $= 2.7 \times 10^9$ - long
- Aligning genomic regions is useful for revealing common gene structure
  - Useful to compare regions $> 1,000,000$ long

Main Idea

Genomic regions of interest contain ordered islands of similarity
- E.g. genes
1. Find local alignments
2. Chain an optimal subset of them

Outline

- Methods to FIND Local Alignments
  - Sorting $k$-long words
  - Suffix Trees
- Methods to CHAIN Local Alignments
  - Dynamic Programming
  - Sparse Dynamic Programming
Methods to FIND Local Alignments

1. Sorting K-long words
   BLAST, BLAT, and the like
2. Suffix Trees

Finding Local Alignments:
Sorting k-long words

Given sequences x, y:
1. Write down all
   \((w, 0, i) : w = x_{i+1} \ldots x_{i+k}\)
   \((z, 1, j) : z = y_{j+1} \ldots y_{j+k}\)
2. Sort them lexicographically
3. Deduce all k-long matches between x and y
4. Extend to local alignments

Running time

- Worst case: \(O(NxM)\)
- In practice: a large value of \(k\) results in a short list of matches

Tradeoff:
Low \(k\): worse running time
High \(k\): significant alignments missed

PatternHunter:
Sampling non-consecutive positions increases the likelihood to detect a conserved region, for a fixed value of \(k\) - refer to Lecture 3

Suffix Trees

- Suffix trees are a method to find all maximal matches between two strings (and much more)

Example:
\(x = dbbdac\)

Definition of a Suffix Tree

**Definition:**
For string \(x = x_1 \ldots x_m\), a suffix tree is:
- A rooted tree with \(m\) leaves
- Leaf \(x_1, \ldots, x_m\)
- Each edge is a substring
- No two edges out of a node, start with same letter

It follows, every substring corresponds to an initial part of a path from root to a leaf
Constructing a Suffix Tree

- Naïve algorithm: $O(N^2)$ time
- Better algorithms: $O(N)$ time
  (outside the scope of this class – too technical and not so interesting)

Memory: $O(N)$ but with a sizeable constant

Naïve Algorithm to Construct a Suffix Tree

1. Initialize tree $T$: a single root node $r$
2. Insert special symbol $\$$ at end of $x$
3. For $j = 1$ to $m$
   - Find longest match of $x_1 \ldots x_m$ to $T$, starting from $r$
   - Split edge where match stops: new node $w$
   - Create edge $(w, j)$, and label with unmatched portion of $x_1 \ldots x_m$

Example of Suffix Tree Construction

1. Insert $d a b d a $
2. Insert $a b d a $  
3. Insert $b d a $  
4. Insert $d a $  
5. Insert $a $  
6. Insert $\$$

Faster Construction

Several algorithms

- $O(N)$ time,
- $O(N)$ memory with a big constant

Technical but not deep, outside the scope of this course

Optional: Gusfield, chapter 6

Memory to Store Suffix Tree

- Can store in $O(N)$ memory!
- Every edge is labeled with $(i, j)$:
  $(i, j)$ denotes $x_i \ldots x_j$
- Tree has $O(N)$ nodes

Proof:
1. #leaves $\geq$ #nodes - 1
2. #leaves = $|x|$

Application: Find all Matches Between $x$ and $y$

1. Build suffix tree for $x$, mark nodes with $x$
2. Insert $y$ in suffix tree, mark all nodes $y$ “passes from” with $y$
- The path label of every node marked both 0 and 1, is a common substring
Example of Suffix Tree Construction for x, y

```
x = d a b d a $
y = a b a d a $
```

1. Construct tree for x
2. Insert a b a d a $
3. Insert b a d a $
4. Insert a d a $
5. Insert d a $
6. Insert a $
7. Insert $  

Application: Online Search of Strings on a Database

Say a database $D = \{s_1, s_2, ..., s_n\}$
(eg. proteins)

**Question:** given new string x, find all matches of x to database

1. Build suffix tree for $\{s_1, ..., s_n\}$
2. All new queries x take $O(|x|)$ time
   (somewhat like BLAST)

Application: Common Substrings of k Strings

- Say we want to find the longest common substring of $s_1, s_2, ..., s_n$
1. Build suffix tree for $s_1, ..., s_n$
2. All nodes labeled $\{s_{i_1}, ..., s_{i_k}\}$ represent a match between $s_{i_1}$, ..., $s_{i_k}$