Assembling Genomes

BCH394P/364C Systems Biology / Bioinformatics
Edward Marcotte, Univ of Texas at Austin

www.yourgenome.org/facts/timeline-the-human-genome-project
“If it tastes good you should sequence it... you should know what's in the genes of that species”
Wang Jun, Chief executive, BGI

The NovaSeq in the UT GSAF core generates >1.4 terabases of sequence in a 1-day run

⇒ Many millions of 75-150 bp reads
Thousands of 1,000 to 1,000,000 (ish) bp reads

A new strategy for genome sequencing

J. Craig Venter, Hamilton O. Smith and Leroy Hood

“shotgun” sequencing

“mapping”
Contemporary genome assembly is fairly complex, but at its core are assembly algorithms that grew from the shotgun concept.

Twelve quick steps for genome assembly and annotation in the classroom

Thinking about the basic shotgun concept

• Start with a very large set of random sequencing reads

• How might we match up the overlapping sequences?

• How can we assemble the overlapping reads together in order to derive the genome?
Thinking about the basic shotgun concept

• At a high level, the first genomes were sequenced by comparing pairs of reads to find overlapping reads

• Then, building a graph (i.e., a network) to represent those relationships

• The genome sequence is a “walk” across that graph

The “Overlap-Layout-Consensus” method

Overlap: Compare all pairs of reads (allow some low level of mismatches)

Layout: Construct a graph describing the overlaps

Consensus: Reconcile errors among reads along that path to find the consensus sequence
Building an overlap graph

Edge

- A → B
- A → B
- A ← B
- A ← B

Overlap

- 5' A → 3'
- A → B
- B → A
- A → B

Reads

- A 5' → B → 3'
- E → I
- F → I
- C ← D
- G ← H

Overlap graph

Simplifying an overlap graph

1. Remove all contained nodes & edges going to them

2. Transitive edge removal:
   Given A – B – D and A – D, remove A – D
Simplifying an overlap graph

3. If un-branched, calculate consensus sequence
   If branched, assemble un-branched bits and then decide
   how they fit together

“contig” (assembled contiguous sequence)
This basic strategy was used for most of the early genomes. Also useful: “mate pairs”

2 reads separated by a known distance

Contigs can be ordered using these paired reads to produce “scaffolds”

GigAssembler (used to assemble the public human genome project sequence)

Jim Kent  David Haussler

Let’s take a little walk through history to see what they did...
Whole genome Assembly: big picture

GigAssembler – Preprocessing

1. Decontaminating & Repeat Masking.
2. Aligning of mRNAs, ESTs, BAC ends & paired reads against initial sequence contigs.
   - psLayout → BLAT
3. Creating an input directory (folder) structure.
RepBase + RepeatMasker

GigAssembler: Build merged sequence contigs ("rafts")

---

**Figure 1** Two sequences overlapping end to end. The sequences are represented as dashes. The aligning regions are joined by vertical bars. End-to-end overlap is an extremely strong indication that two sequences should be joined into a contig.
Sequencing quality (Phred Score)

\[ Q = -10 \log_{10} \frac{P}{\text{Error Probability}} \]

or

\[ P = 10^{-\frac{Q}{10}} \]

Phred quality scores are logarithmically linked to error probabilities.
We’re going to skip the remaining details of GigAssembler (mainly of historical interest now) to get to the key strategy for assembling all of the various contigs and paired end reads into a genome.

GigAssembler: Build a “raft-ordering” graph

**Figure 4** Three overlapping draft clones: A, B, and C. Each clone has two initial sequence contigs. Note that initial sequence contigs $a_1$, $b_1$, and $a_2$ overlap as do $b_2$ and $c_1$.

**Figure 5** Ordering graph of clone starts and ends. This represents the same clones as in Fig. 4. (As) The start of clone A; (Ae) the end of clone A. Similarly $B_s$, $B_e$, $C_s$, and $C_e$ represent the starts and ends of clones B and C.

**Figure 6** Ordering graph after adding in rafts. The initial sequence contigs shown in Fig. 4 are merged into rafts where they overlap. This forms three rafts: $a_1b_1a_2$, $b_2c_1$, and $c_2$. These rafts are constrained to lie between the relevant clone ends by the addition of additional ordering edges to the graph shown in Fig. 5.
GigAssembler:
Build a “raft-ordering” graph

- Add information from mRNAs, ESTs, paired plasmid reads, BAC end pairs: building a “bridge”
  - Different weight to different data type: (mRNA ~ highest)
  - Conflicts with the graph as constructed so far are rejected.
- Build a sequence path through each raft.
- Fill the gap with N’s.
  - 100: between rafts
  - 50,000: between bridged barges

Finding the shortest path across the ordering graph using the Bellman-Ford algorithm

Find the shortest path to all nodes.

Take every edge and try to relax it ($N - 1$ times where $N$ is the count of nodes)
Find the shortest path to all nodes.

Take every edge and try to relax it ($N - 1$ times where $N$ is the count of nodes)
Find the shortest path to all nodes.

Take every edge and try to relax it (N – 1 times where N is the count of nodes)
Find the shortest path to all nodes.

Take every edge and try to relax it (N – 1 times where N is the count of nodes)

Answer: A-D-C-B-E
Modern assemblers now work a bit differently, using so-called DeBruijn graphs:

Here’s what we saw before:

In Overlap-Layout-Consensus:
- Nodes are reads
- Edges are overlaps

In a DeBruijn graph:
- Vertices are (k-1)-mers
- Edges are k-mers

Eulerian cycle
Visit each edge once
Why Eulerian?

From Leonhard Euler’s solution in 1735 to the ‘Bridges of Königsberg’ problem:

Königsberg (now Kaliningrad, Russia) had 7 bridges connecting 4 parts of the city. Could you visit each part of the city, walking across each bridge only once, & finish back where you started?

(Euler conceptualized it as a graph: Nodes = parts of city Edges = bridges)

(Visiting every edge once = an Eulerian path)

DeBruijn graph assemblers tend to have nice properties, e.g. correcting sequencing errors & handling repeats better

(Sequencing errors appear as ‘bulges’)

Removing the ‘bulges’ corrects the errors (e.g. leaves the red path)
e.g. Velvet, an example algorithm using DeBruijn graphs

Once a reference genome is assembled, new sequencing data can ‘simply’ be mapped to the reference.
Mapping reads to assembled genomes

<table>
<thead>
<tr>
<th>Program</th>
<th>Website</th>
<th>Open source</th>
<th>Handles ABI color space</th>
<th>Maximum read length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bowtie</td>
<td><a href="http://bowtie.ccb.ca/">http://bowtie.ccb.ca/</a></td>
<td>Yes</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>BWA</td>
<td><a href="http://maq.sourceforge.net/bwa-man.shtml">http://maq.sourceforge.net/bwa-man.shtml</a></td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>Maq</td>
<td><a href="http://maq.sourceforge.net">http://maq.sourceforge.net</a></td>
<td>Yes</td>
<td>Yes</td>
<td>127</td>
</tr>
<tr>
<td>Mosaic</td>
<td><a href="http://bioinformatics.bc.ca/mosaic/Mosaic">http://bioinformatics.bc.ca/mosaic/Mosaic</a></td>
<td>No</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>NovaAlign</td>
<td><a href="http://www.novocraft.com">http://www.novocraft.com</a></td>
<td>No</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td>SOAP2</td>
<td><a href="http://soap.genomics.org.cn">http://soap.genomics.org.cn</a></td>
<td>No</td>
<td>No</td>
<td>60</td>
</tr>
<tr>
<td>ZOOM</td>
<td><a href="http://www.bioinformatics.org/">http://www.bioinformatics.org/</a></td>
<td>No</td>
<td>Yes</td>
<td>240</td>
</tr>
</tbody>
</table>

The list is a little longer now! e.g. see https://en.wikipedia.org/wiki/List_of_sequence_alignment_software#Short-Read_Sequence_Alignment

Mapping strategies

![Diagram of mapping strategies](image)
Burroughs Wheeler indexing

BWT is often used for file compression (like bzip2), here used to make a fast ‘lookup’ index in a genome

BWT = ‘reversible block-sorting’

Input: SIX.MIXED.PIXIES.SIFT.SIXTY.PIXIE.DUST.BOXES

Forward BWT

Output: TEXYDST.E.IXIXIXXSMPPS.B..E.S.EUSFXDIIIIIT

This sequence is more compressible

Reverse BWT

Recovered input: SIX.MIXED.PIXIES.SIFT.SIXTY.PIXIE.DUST.BOXES

Burroughs-Wheeler transform indexing

Input

^BANANA |


Burroughs-Wheeler transform indexing

All Rotations

^BANANA | ^BANANA
| ^BANANA | ^BAN
A | ^BANAN
NA | ^BANA
ANA | ^BAN
NANA | ^BA
ANANA | ^B
BANANA | ^
### Burroughs-Wheeler transform indexing

#### Sorting All Rows in Alphabetical Order

| ANANA | ^B  |
| ANA   | ^BAN |
| A     | ^BANAN |
| BANANA| ^  |
| NANA  | ^BA |
| NA    | ^BANA |
| ^BANANA|  |
|       | ^BANANA |


---

### Burroughs-Wheeler transform indexing

#### Taking Last Column

| ANANA | ^B  |
| ANA   | ^BAN |
| A     | ^BANAN |
| BANANA| ^  |
| NANA  | ^BA |
| NA    | ^BANA |
| ^BANANA|  |
|       | ^BANANA |

Burroughs-Wheeler transform indexing

### Output

<table>
<thead>
<tr>
<th>Last Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNN^AA</td>
</tr>
</tbody>
</table>


---

<table>
<thead>
<tr>
<th>Input</th>
<th>All Rotations</th>
<th>Sorting All Rows in Alphabetical Order</th>
<th>Taking Last Column</th>
<th>Output Last Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>^BANANA</td>
<td>^BANANA</td>
<td>ANANA</td>
<td>^B</td>
<td>ANANA</td>
</tr>
<tr>
<td>^BANANA</td>
<td>^BANANA</td>
<td>ANA</td>
<td>^BAN</td>
<td>ANA</td>
</tr>
<tr>
<td>A</td>
<td>^BANANA</td>
<td>A</td>
<td>^BANAN</td>
<td>A</td>
</tr>
<tr>
<td>NA</td>
<td>^BANA</td>
<td>NANA</td>
<td>^BA</td>
<td>NANA</td>
</tr>
<tr>
<td>ANA</td>
<td>^BAN</td>
<td>NANA</td>
<td>^BA</td>
<td>NANA</td>
</tr>
<tr>
<td>NANA</td>
<td>^BA</td>
<td>NANA</td>
<td>^BA</td>
<td>NANA</td>
</tr>
<tr>
<td>ANANA</td>
<td>^B</td>
<td>NANA</td>
<td>^BA</td>
<td>NANA</td>
</tr>
<tr>
<td>BANANA</td>
<td>^</td>
<td>NANA</td>
<td>^BA</td>
<td>NANA</td>
</tr>
</tbody>
</table>

BWT is remarkable because it is *reversible*.

Any ideas as how you might reverse it?

Burroughs-Wheeler transform indexing

<table>
<thead>
<tr>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNN^AA</td>
</tr>
</tbody>
</table>

### Burroughs-Wheeler transform indexing

<table>
<thead>
<tr>
<th>Add 1</th>
<th>Sort 1</th>
<th>Add 2</th>
<th>Sort 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
<td>BA</td>
<td>AN</td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>NA</td>
<td>AN</td>
</tr>
<tr>
<td>N</td>
<td>A</td>
<td>NA</td>
<td>A</td>
</tr>
<tr>
<td>^</td>
<td>B</td>
<td>^B</td>
<td>BA</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td>AN</td>
<td>NA</td>
</tr>
<tr>
<td>A</td>
<td>N</td>
<td>AN</td>
<td>NA</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Write the sequence as the last column.
Sort it...
Add the columns...
Sort those...


---

### Burroughs-Wheeler transform indexing

<table>
<thead>
<tr>
<th>Add 3</th>
<th>Sort 3</th>
<th>Add 4</th>
<th>Sort 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAN</td>
<td>ANA</td>
<td>BANA</td>
<td>ANAN</td>
</tr>
<tr>
<td>NAN</td>
<td>ANA</td>
<td>NANA</td>
<td>ANA</td>
</tr>
<tr>
<td>NA</td>
<td>A</td>
<td>^</td>
<td>NA</td>
</tr>
<tr>
<td>^BA</td>
<td>BAN</td>
<td>^BAN</td>
<td>BANA</td>
</tr>
<tr>
<td>ANA</td>
<td>NAN</td>
<td>ANAN</td>
<td>NANA</td>
</tr>
<tr>
<td>ANA</td>
<td>NA</td>
<td>^</td>
<td>ANA</td>
</tr>
<tr>
<td></td>
<td>^B</td>
<td>^BA</td>
<td>^BA</td>
</tr>
<tr>
<td>A</td>
<td>^</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Add the columns...
Sort those...
Add the columns...
Sort those...

### Burroughs-Wheeler transform indexing

<table>
<thead>
<tr>
<th>Add 5</th>
<th>Sort 5</th>
<th>Add 6</th>
<th>Sort 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANAN</td>
<td>ANANA</td>
<td>BANANA</td>
<td>ANANA</td>
</tr>
<tr>
<td>NANA ^</td>
<td>ANA ^</td>
<td>NANA ^</td>
<td>ANA ^B</td>
</tr>
<tr>
<td>NA ^B</td>
<td>A ^BA</td>
<td>NA ^B</td>
<td>A ^BAN</td>
</tr>
<tr>
<td>^BANA</td>
<td>BANAN</td>
<td>^BANAN</td>
<td>BANANA</td>
</tr>
<tr>
<td>ANANA</td>
<td>NANA ^</td>
<td>ANANA ^</td>
<td>NANA ^</td>
</tr>
<tr>
<td>ANA ^</td>
<td>NA ^B</td>
<td>ANA ^B</td>
<td>NA ^BA</td>
</tr>
<tr>
<td>^BAN</td>
<td>^BANA</td>
<td>^BANA</td>
<td>^BANAN</td>
</tr>
<tr>
<td>A ^BA</td>
<td>^BAN</td>
<td>A ^BAN</td>
<td>^BAN</td>
</tr>
</tbody>
</table>

Add the columns...  Sort those...  Add the columns...  Sort those...


---

### Burroughs-Wheeler transform indexing

<table>
<thead>
<tr>
<th>Add 7</th>
<th>Sort 7</th>
<th>Add 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>BANANA</td>
<td>ANANA ^</td>
<td>BANANA ^</td>
</tr>
<tr>
<td>NANA ^B</td>
<td>ANA ^BA</td>
<td>NANA ^BA</td>
</tr>
<tr>
<td>NA ^BAN</td>
<td>A ^BANA</td>
<td>NA ^BANA</td>
</tr>
<tr>
<td>^BANANA</td>
<td>BANANA</td>
<td>^BANANA</td>
</tr>
<tr>
<td>ANANA ^</td>
<td>NANA ^B</td>
<td>ANANA ^B</td>
</tr>
<tr>
<td>ANA ^BA</td>
<td>NA ^BAN</td>
<td>ANA ^BAN</td>
</tr>
<tr>
<td>^BANAN</td>
<td>^BANANA</td>
<td>^BANANA</td>
</tr>
<tr>
<td>A ^BANA</td>
<td>^BANAN</td>
<td>A ^BANAN</td>
</tr>
</tbody>
</table>

Add the columns...  Sort those...  Add the columns...

The row with the "end of file" character at the end is the original text

Burroughs-Wheeler transform indexing

<table>
<thead>
<tr>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>^BANANA</td>
</tr>
</tbody>
</table>

The row with the "end of file" character at the end is the original text


The Burroughs-Wheeler transform leads naturally to a suffix array

Li & Durbin, doi:10.1093/bioinformatics/btp324/
The Burroughs-Wheeler transform leads naturally to a suffix array


"If string W is a substring of X, the position of each occurrence of W in X will occur in an interval in the suffix array. This is because all the suffixes that have W as prefix are sorted together."

Li & Durbin, doi:10.1093/bioinformatics/btp324/

e.g. applying BWT to construct the suffix array of GATGCGAGAGATG

The search can be even more efficient by using compression & various other extensions

http://blog.thegrandlocus.com/2016/07/a-tutorial-on-burrows-wheeler-indexing-methods
Why is this efficient?

Searching a suffix array in this way cuts the search space in half at each step, so...

A suffix array of the human genome (3.2 billion bases) takes at most

$$\log_2(3.2 \text{ billion}) + 1 = 32 \text{ steps}$$

to determine if a query sequence is present or not

There are few more steps to find all the occurrences, build an efficient real-world implementation, use compression to reduce memory and storage space, etc., but this still illustrates the massive savings in time and memory from constructing an index.

Burroughs Wheeler indexing

 Trapnell C, Salzberg SL, Nat. Biotech., 2009