Introduction to Read-Based Alignment

Michael C. Zody, Ph.D.
Workshop on Genomics
Cesky Krumlov
May 25, 2022
Aligning to a Reference

• Aligning sequences is a classic problem
  – Early bioinformatic problem
  – Very similar to older text matching problems

• Several algorithms exist
  – Tradeoffs of speed versus accuracy, sensitivity

• Sequencing throughput creates new problems
  – Short reads have less information than long seqs
  – Data volume requires faster processing per read
Example of alignment

Read:

TCAACTCTGCCAACACCTTCTTCTCCAGGAAGCACTCCTGGATTTCCCTCTTGCCAACAAGATTCTGGGAGGGCA

Genome:

ATAAAATGGCCAAAATTAACTAGAAGGTGAGTAGAAACTTAAATAAACTAATTACCATTGATGAGAAAAAAATC
TGCCACTGAAAAAGGCACCCCGGTCCAGAGGGTTTCATGAGCGGGAACCTGTAGAAGACCTTTCAATTCAACTCTGC
CAACACCTTCTCCTCCAGGAAGCACTCCTGGATTTCCCTCTTGCCAACAAGATTCTGGAGGGGCAGCTCCTCCA
ACATGCCCCCAACAGCTCTCTGCAGACATATCATATCATATCATATCATATCTTCCATACCATAACTGCCATGCCATACA
Example of alignment

Read:

TCAACTCTGCAACACCTTCTCCTCAGGAAGCACTCCTGGATTTCCCTCTTGCCAACAAGATTCTGGGAGGGCA

Genome:

ATAAAATGGCCAAAATTAACCTAGAAGGTGATGGAAGAACCTTAAATAACTAATTACATGGATGAGAAAAAAAATCTGCCACTGAAAAAGGCACCCGGTTCCAGAGGTTTTCATGAGGGGAACTGAGAAACCTTTCGAAT **TCAACTCTGC**

CACACAACCTTCTCTCCAGGAAGCACTCCTGGATTTCCCTCTTGCCAACAAAGATTCTGGAAGGCA**GCTCCTCAA**

ACATGCCCAACAGCTCTCTGCAGACATATCATATCATATCATATCATATATCTTCCATACCATAACTGCCATGCCATA**CA**
How Would You Find That?

- Brute force comparison
- Smith-Waterman
- Suffix Tree
- Burrows-Wheeler Transform
- Hashing/Minimizers
Brute Force Method

TCGATCC

?  

GACCTCATCGATCCCACTG
Brute Force Method

TCGATCC
×
GACCTCATCGATCCCACTG
TCGATCC
\times
GACCTCATCGATCCCACTG
Brute Force Method

TCGATCC

GACCTCATCGATCCCACTG
Brute Force Method

GACCTCA TCGATCC CACTG
Simplistic Scoring Scheme:
+1 match if moving diagonally
-1 mismatch if moving diagonally
-1 gap if moving hor. or vert.

(no penalty for terminal gaps)

<table>
<thead>
<tr>
<th>^ G A C C T C A T C G A T C C C C C A C T G</th>
<th>0 -1 -2 0 2 1 1 0 1 3 3 2 3 5 7 6 5 4 3 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -1 -1 1 1 0 1 0 2 4 3 2 4 6 5 4 3 2 1 0</td>
<td>0 -1 0 0 -1 0 -1 -1 3 2 1 1 5 4 3 2 1 0 1 0</td>
</tr>
<tr>
<td>0 -1 0 0 -1 0 -1 -1 3 2 1 1 5 4 3 2 1 0 1 0</td>
<td>0 0 1 0 -1 -2 0 2 1 0 2 4 3 2 1 0 1 0 -1 -2</td>
</tr>
<tr>
<td>0 1 0 -1 -2 -1 1 1 0 1 3 2 1 1 1 0 -1 -2 -1 0</td>
<td>0 -1 0 -1 0 0 2 1 0 2 1 0 0 2 1 0 -1 0 0 0 0</td>
</tr>
<tr>
<td>0 -1 0 -1 0 0 2 1 0 2 1 0 0 2 1 0 -1 0 0 0 0</td>
<td>0 -1 -1 -1 -1 1 0 -1 1 0 -1 -1 -1 0 -1 -1 -1 -1 1 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>^ G A C C T C A T C G A T C C C C C A C T G</td>
<td></td>
</tr>
</tbody>
</table>
Suffix Tree

GACCTCA\textcolor{red}{TGATC}CC\textcolor{red}{C}ACTG

A

C

G

T

G
Suffix Tree

GACCTCATCGATCCCACTG

A

C

T

G

A

T

C

G

C

T

C

A

T

C

G

A

T

C

C

A

T

G

$
Burrows-Wheeler Transform

GACCTCATCGATCCCACTG$  ACCTCATCGATCCCACTG$G
ACCTCATCGATCCCACTG$G  ACTG$GACCTCATCGATCCC
CCTCATCGATCCCACTG$GA  ATCCCACTG$GACCTCATCG
CTCATCGATCCCACTG$GAC  ATCGATCCCACGTG$GACCTC
TCATCGATCCCACTG$GACC  CACTG$GACCTCATCGATCC
CATCGATCCCACTG$GACCT  CATCGATCCCACTG$GACCT
ATCGATCCCACTG$GACCTC  CCACCTG$GACCTCATCGAT
TCGATCCCACTG$GACCTCA  CCCACCTG$GACCTCATCGAT
CGATCCCACTG$GACCTCA  CCTCATCGATCCCACTG$GA
GATCCCACTG$GACCTCATG  CGATCCCACTG$GACCTCAT
ACCTCATCGATCCCACTG  CTCATCGATCCCACTG$GAC
TCCACCTG$GACCTCATCGA  CTG$GACCTCATCGATCCC
CCACCTG$GACCTCATCGA  GACCTCATCGATCCCACTG$
CCACCTG$GACCTCATCGA  GATCCCACTG$GACCTCAT
CAGT$GACCTCATCGATCC  G$GACCTCATCGATCCCACT
ACTG$GACCTCATCGATCC  TCAATCGATCCCACTG$GACC
CTG$GACCTCATCGATCC  TCGATCCCACTG$GACCTCA
TG$GACCTCATCGATCC  TG$GACCTCATCGATCCAC
G$GACCTCATCGATCC  $GACCTCATCGATCCAC
$GACCTCATCGATCC  $GACCTCATCGATCCACTG
• Start with the transform column
• My read starts with a T, so I want rows with Ts in them
• This column gives me all the single nucleotide counts
• Sort the single nucleotide counts to get the alphabetically first column
• Now these two columns give me all the dinucleotide counts
• Sort those to get the alphabetically first two columns
• Now there is only one place my read can match
Start with the transform column
Count all the characters, sort them, and store the count of lower characters

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>12</td>
<td>15</td>
<td>19</td>
</tr>
</tbody>
</table>

This gives the positions of all the bases in the first column (because it’s sorted)
• Take the query sequence TCGATCC
• Start at the end and use the count table to look up the position of the last base in the first column

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>12</td>
<td>15</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

• The last column comes immediately before the first column
• Find all the rows of the last column with the next to last base
• Take the query sequence TCGATCC
• The order of a given character in the last column matches the order of the same instance of that character in the first column
• The 3\textsuperscript{rd}-5\textsuperscript{th} Cs in the last column precede Cs in the first column, so we now want the 3\textsuperscript{rd}-5\textsuperscript{th} Cs in column 1

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>12</td>
<td>15</td>
<td>19</td>
</tr>
</tbody>
</table>

• Now we take the next character and look for Ts in the last column (the 2\textsuperscript{nd} T)
• Take the query sequence **TCGATCC**

• The second T is preceded by the 3\textsuperscript{rd} A

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>12</td>
<td>15</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>
FM Index

- Take the query sequence **TCGATCC**
- The third A is preceded by the 2\textsuperscript{nd} G

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>12</td>
<td>15</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

```
ACCT
ACTG
ATCC
ATCG
CACT
CATC
CCAC
CCCA
CCTC
CGAT
CTCA
CTG$
GACC
GATC
G$GA
TCAT
TCCC
TCGA
TG$G
$GAC
```

```
G
C
G
C
G
C
T
C
T
C
T
C
A
A
A
A
A
```
FM Index

- Take the query sequence **TCGATCC**
- The second G is preceded by the 6\textsuperscript{th} C

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>12</td>
<td>15</td>
<td>19</td>
</tr>
</tbody>
</table>
FM Index

- Take the query sequence **TCGATCC**
- The sixth C is preceded by the 3\(^{rd}\) T

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>12</td>
<td>15</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>
FM Index

- Take the query sequence **TCGATCC**
- And we’re done

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>T</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>12</td>
<td>15</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>

- To find the position in the genome, we keep a separate index of positions for a sparse set of rows in the table and then just walk through the transform to the nearest indexed row
"Hashing" by Visual Example

Read:

TCAACTCTGCCAACACCTTCTCTCCAGGAAGCACTCTGGATTTCCCCTTTGCAACAAGATTCTGGGAGGGCAGCTCCTCCA

Genome:

ATAAAATGGCCAAAATTAACTAGAAGGTGAGTAGAAACTTAATATACTAATTACCATTGATGAGGAAAAAATCTG
GCCACTGAAAAGGCACCGGTCCAGAGGGTTTCTAGAGCGGGAACTGTAGAAACCTTTCGAATTCAACTCTGC
CAACACCTTCCTCCCTGGAAGCACTCCTGGATTTCCCCTTTGCCCAACAAGATTCTGGGAGGGCAGCTCCTCCA
ACATGCCCCAACAGCTCTCTGCAGACATATCATATATCATATATATATATATTTCCATACCTAATACTGCCATGCCATACA
Read:

\[\text{TCAACTCTGCAACACCTTCCCTCCAGGAAGCCTGCTGGATTTCCCTCTTGCCAACAAGATTCTGGGAGGGCA}\]

Genome:

\[\text{ATAAAATGGCAAAATTAACCTGAAGGTGATGAAACCTTAAATAAATAAATTAATTACCCATTGATGAGAATCGGCAATCTCCTGCCACTGAAAAAGGCACCTCCAGAGGGTTTCATGAGCGGGAACTGTAGAAACCTTTCGAATTCAACTCTGCCAACAACCTTCCTTCCAGGAAGCCTGCTGGATTTCCCTCTTGCCAACAAGATTCTGGGAGGGCA}\]
”Hashing” by Visual Example

Read:

TCAACTCTGCCAACACCTTCTCTCAGGAAGGACCTCTGATTTCCCTCTTGCCAACAAGATTCTGGGAGGGCA

Genome:

ATAAAATGGCCAAATTTAATAGAGGTAGAAACTAAATAATTTCCATTAGGAGAAAAAATC
TGCCACTGAAAAAGGACACCGGTCCAGAGGGTTCATGAGCGGGAACCTGTAGAAACCTTTCGAATTCAACTGTC
CAACACCTTCCTCTTCTAGGAAAGCACPCTTGGATTCTCTTCTTGGCCCAAAGATTCTGGGAGGGCAGCTCTTCCCA
ACATGCCAACAGCTCTGCGACATATCATATATCATATATATATATATATATATATATATCATACGACTAGCCGATCATAC
"Hashing" Explained

• Walk the reference and build a list of words of length k (k-mers) with their positions in the sequence
  – Exhaustive method is every k-mer
  – Can do non-overlapping, partially overlapping, etc.
  – The more k-mers you store, and the smaller k is, the more sensitive the method will be
  – The fewer k-mers you store, and the larger k is, the more efficient it will be

• To align, find all the k-mers in each read and look for them in the index (or “hash”) and find their locations, then use a modified Smith-Waterman to extend and score the match
“Seeding”

• Hashing is a way to seed, but not the only way
• One can use suffix trees or bwts to seed (in fact many aligners do this); however, it is only efficient if a single seed can be extended to most of the alignment cheaply
• For a while, there was a great deal of effort expended to develop better and more efficient seeding methods
Minimizers

• A minimizer (Roberts et al., 2004) is one efficient way to seed

• Minimizers are generated as follows:
  – Slide a window of size w across the genome
  – For every position starting in w, determine the k-mer that starts at w
  – By some deterministic method, sort the k-mers in w
  – The lowest sort order k-mer is w is the minimizer of w

• Any sequence containing a window w identical to the window will produce the same minimizer, making it irrelevant to store other k-mers to match those regions

• By tuning k and w, you can adjust sensitivity and efficiency
Minimizer Example

• Minimizers in a toy example with $k = 3$ and $w = 3$
• For all $w \leq k$, it is guaranteed every position will be covered by at least one minimizer
• Although compression is small ($7/14$) in this toy example, it is given by $2 / (w+1)$, so can become quite efficient for large $w$

<table>
<thead>
<tr>
<th>Position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

```
2 3 1 0 3
3 1 0 3 2
1 0 3 2 1
0 3 2 1 0
3 2 1 0 1
2 1 0 1 2
1 0 1 2 3
0 1 2 3 3
1 2 3 3 1
2 3 3 1 0
3 3 1 0 1
```
Seed-Chain-Extend

• For long, noisy (or diverged) data, going straight from seeding to base pair resolution alignments may be inefficient
• Instead, we can form an optimal chain of seeds
• This uses a dynamic programming scheme similar to Smith-Waterman, but optimizes on minimum gap size
• If our sequences are highly similar and our minimizers are dense, we may have the complete alignment from overlapping chained minimizers
• Otherwise, we can add an extend step where we use a true Smith-Waterman global alignment between each adjacent pair of non-overlapping minimizers
Common Short Read Aligners

• Seed and Smith-Waterman extend
  – Novoalign
• BWA align gap-free
  – Bowtie
• BWA align with gaps
  – BWA aln, Bowtie2
• BWA Seed and Smith-Waterman extend
  – BWA mem
• Seed-chain-extend
  – STAR, Blasr, minimap2