The Burrows-Wheeler Transform and Bioinformatics

J. Matthew Holt

April 1st, 2015

Outline

- Recall Suffix Arrays
- The Burrows-Wheeler Transform
- The FM-index
- Pattern Matching
- Multi-string BWTs
- Merge Algorithms

Recall Suffix Arrays

- Rotate
- Sort

Index (N)	Rotations	Suffix Array
0	ACACGGACA$	$ACACGGACA
1	CACGGACA$A	A$ACACGGAC
2	ACGGACA$AC	ACA$ACACGG
3	CGGACA$ACA	ACACGGACA$
4	GGACA$ACAC	ACGGACA$AC
5	GACA$ACACG	CA$ACACGGA
6	ACA$ACACGG	CACGGACA$A
7	CA$ACACGGA	CGGACA$ACA
8	A$ACACGGAC	GACA$ACACG
9	$ACACGGACA	GGACA$ACAC

The suffix array for string “ACACGGACA$”
“$” is just an end-of-string character

Recall Suffix Arrays (cont.)

- **N** - number of bases
- **k** - pattern length
- Space complexity: **O(N*log(N))** bits
  - Stored as offsets into original string
  - N offsets that require log(N) bits per value
- Search time: **O(k*log(N))** operations
  - Binary search require O(log(N)) string comparisons
  - Each string comparison requires O(k) symbol comparisons
- Problem:
  - O(N*log(N)) is too large when strings are billions of characters long

The Burrows-Wheeler Transform

- Burrows & Wheeler, 1994
- BWTs are permutations of the original string
- Implicit suffix array
  - “Last symbol” in suffix
  - “Previous symbol” to suffix

Index (N)	Rotations	Suffix Array	BWT
0	ACACGGACA$	$ACACGGAC**A**	**A**
1	CACGGACA$A	A$ACACGGA**C**	**C**
2	ACGGACA$AC	ACA$ACACG**G**	**G**
3	CGGACA$ACA	ACACGGACA**$**	**$**
4	GGACA$ACAC	ACGGACA$A**C**	**C**
5	GACA$ACACG	CA$ACACGG**A**	**A**
6	ACA$ACACGG	CACGGACA$**A**	**A**
7	CA$ACACGGA	CGGACA$AC**A**	**A**
8	A$ACACGGAC	GACA$ACAC**G**	**G**
9	$ACACGGACA	GGACA$ACA**C**	**C**

BWT algorithm

BWT (string text)

table$_i$ = Rotate(text, i) for i = 0..len(text)-1

sort table alphabetically

return (last column of the table)

```
tarheel$        $tarheel
arheel$t        arheel$t
rheel$ta        eel$tarh
heel$tar        el$tarhe
eel$tarh        heel$tar
el$tarhe        l$tarhee
l$tarhee        rheel$ta
$tarheel        tarheel$
```

BWT(“tarheels$”) = “ltherea$”

BWT in Python

```
def BWT(s):
    # create a table, with rows of all possible rotations of s
    rotation = [s[i:] + s[:i] for i in xrange(len(s))]
    # sort rows alphabetically
    rotation.sort()
    # return (last column of the table)
    return "".join([r[-1] for r in rotation])
```

Inverting a BWT

- A property of a transform is that there is no information loss and they are invertible.

inverseBWT(string *s*)
  add *s* as the first column of a table strings
  repeat length(s)-1 times:
    sort rows of the table alphabetically
    add *s* as the first column of the table
  return (row that ends with the 'EOF' character)

l	l$	l$t	l$ta	l$tar	l$tarh	l$tarhe	l$tarhee
t	ta	tar	tarh	tarhe	tarhee	tarheel	tarheel$
h	he	hee	heel	heel$	heel$t	heel$ta	heel$tar
e	ee	eel	eel$	eel$t	eel$ta	eel$tar	eel$tarh
r	rh	rhe	rhee	rheel	rheel$	rheel$t	rheel$ta
e	el	el$	el$t	el$ta	el$tar	el$tarh	el$tarhe
a	ar	arh	arhe	arhee	arheel	arheel$	arheel$t
$	$t	$ta	$tar	$tarh	$tarhe	$tarhee	$tarheel

Inverting in Python

```
def inverseBWT(s):
    # initialize table from s
    table = [c for c in s]
    # repeat length(s) - 1 times
    for j in xrange(len(s)-1):
        # sort rows of the table alphabetically
        table.sort()
        # insert s as the first column
        table = [s[i]+table[i] for i in xrange(len(s))]
# return (row that ends with the 'EOS' character)
return table[[r[-1] for r in table].index('$')]
```

BWT Compression

- Compression
  - Tendency to form long runs
  - Run-length encoding (RLE)
- Can be stored as:
  ACG$C3AGC
- Real dataset (Mouse DNA-seq):
  - 200 Giga-bases
  - 20 GB using RLE
  - ~10% of original size

Index (N)	Suffix Array	BWT
0	$ACACGGAC**A**	**A**
1	A$ACACGGA**C**	**C**
2	ACA$ACACG**G**	**G**
3	ACACGGACA**$**	**$**
4	ACGGACA$A**C**	**C**
5	CA$ACACGG**A**	**A**
6	CACGGACA$**A**	**A**
7	CGGACA$AC**A**	**A**
8	GACA$ACAC**G**	**G**
9	GGACA$ACA**C**	**C**

FM-Index

- Ferragina & Manzini, 2005
- Enables fast exact searches
- Takes advantage of “last-first”relationship between BWT and suffix array
  - See colors on right
  - First “A” in BWT corresponds to first suffix starting with “A”

Index (N)	Suffix Array	BWT
0	$ACACGGACA	A
1	A$ACACGGAC	C
2	ACA$ACACGG	G
3	ACACGGACA$	$
4	ACGGACA$AC	C
5	CA$ACACGGA	A
6	CACGGACA$A	A
7	CGGACA$ACA	A
8	GACA$ACACG	G
9	GGACA$ACAC	C

FM-index (cont.)

- A - alphabet size
- FM-index
  - (N+1)*A values
  - F[*i*][*c*] stores the number of times symbol *c* occurs <u>before</u> index *i*
- Offset array (O)
  - A values
  - O[*c*] stores the index of the first suffix starting with symbol *c*

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGAC**A**	**A**	0	0	0	0
1	A$ACACGGA**C**	**C**	0	1	0	0
2	ACA$ACACG**G**	**G**	0	1	1	0
3	ACACGGACA**$**	**$**	0	1	1	1
4	ACGGACA$A**C**	**C**	1	1	1	1
5	CA$ACACGG**A**	**A**	1	1	2	1
6	CACGGACA$**A**	**A**	1	2	2	1
7	CGGACA$AC**A**	**A**	1	3	2	1
8	GACA$ACAC**G**	**G**	1	4	2	1
9	GGACA$ACA**C**	**C**	1	4	2	2
10	—	**—**	**1**	**4**	**3**	**2**
Offset (O)	—	**—**	**0**	**1**	**5**	**8**

Find Predecessor

- The predecessor of index *i:*
  c = BWT[i]
  predec = O[c]+F[i][c]
- Predecessor of index 1
  c = BWT[1] = ‘C’
  predec = O[‘C’]+F[1][‘C’] = 5+0 = 5
- Predecessor of index 8
  c = BWT[8] = ‘G’
  predec = O[‘G’]+F[8][‘G’] = 8+1 = 9
- Time to find predecessor: **O(1)**
- String recovery:
  - Start at 0
  - Repeatedly find predecessor until back at 0
  - O(N) time to get original string back

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGAC**A**	**A**	0	0	0	0
1	A$ACACGGA**C**	**C**	0	1	0	0
2	ACA$ACACG**G**	**G**	0	1	1	0
3	ACACGGACA**$**	**$**	0	1	1	1
4	ACGGACA$A**C**	**C**	1	1	1	1
5	CA$ACACGG**A**	**A**	1	1	2	1
6	CACGGACA$**A**	**A**	1	2	2	1
7	CGGACA$AC**A**	**A**	1	3	2	1
8	GACA$ACAC**G**	**G**	1	4	2	1
9	GGACA$ACA**C**	**C**	1	4	2	2
10	—	**—**	**1**	**4**	**3**	**2**
Offset (O)	—	**—**	**0**	**1**	**5**	**8**

Find *k*-mer

- All searches occur in reverse order
  - Start with full BWT range (0, N)
  - Restrict by one symbol at a time
- Find *k*-mer “ACA”
  - Initialize to full range (“”)
    low, high = 0, 10
  - Find occurrences of “A”
    low = O[‘A’]+F[low][‘A’] = 1+0 = 1
    high = O[‘A’]+F[high][‘A’] = 1+4 = 5
  - Find occurrences of “CA”
    low = O[‘C’]+F[low][‘C’] = 5+0 = 5
    high = O[‘C’]+F[high][‘C’] = 5+2 = 7
  - Find occurrences of “ACA”
    low = O[‘A’]+F[low][‘A’] = 1+1 = 2
    high = O[‘A’]+F[high][‘A’] = 1+3 = 4

Index (N)	Suffix Array (not stored)	BWT	FM-index (F)			
			$	A	C	G
0	$ACACGGAC**A**	**A**	0	0	0	0
1	A$ACACGGA**C**	**C**	0	1	0	0
2	ACA$ACACG**G**	**G**	0	1	1	0
3	ACACGGACA**$**	**$**	0	1	1	1
4	ACGGACA$A**C**	**C**	1	1	1	1
5	CA$ACACGG**A**	**A**	1	1	2	1
6	CACGGACA$**A**	**A**	1	2	2	1
7	CGGACA$AC**A**	**A**	1	3	2	1
8	GACA$ACAC**G**	**G**	1	4	2	1
9	GGACA$ACA**C**	**C**	1	4	2	2
10	—	**—**	**1**	**4**	**3**	**2**
Offset (O)	—	**—**	**0**	**1**	**5**	**8**

Find *k*-mer (cont.)

- Time complexity - O(*k*)
  - Requires O(*k*) lookups
  - Search time only dependent on length of *k*-mer
  - Does not depend on BWT (data) size!!!

```
def find(pattern, FMindex):
    lo = 0
    hi = len(Fmindex)
    for l in reversed(pattern):
        lo = O[l] + F[lo][l]
        hi = O[l] + F[hi][l]
    return lo, hi
```

Application of Exact Pattern Matching

- Alignment
  - Bowtie (2009) and BWA (2009)
  - Build a BWT of the reference genome (~2-3 GB)
  - Align:
    - Given a 100 base pair read
    - Cut into smaller (i. e. four 25-mer) pieces
    - Exact search for the pieces separately - very fast using BWT
    - Use local alignment to account for errors
- Bowtie2 (2011) and Tophat2 (2013) are still very prominent and fast aligners

BWTs and String Collections

- We have a BWT of single string
- What if you have multiple strings?
  - Concatenate strings together with some overhead?
  - Build each one as a separate BWT and query each one?

Multi-string BWTs

- MSBWT - a BWT containing a string collection instead of just a single string
- Earliest: Mantaci *et al.* (2005), used concatenation approach
- Bauer *et al.* (2011) - proposed version we will discuss today

MSBWT Construction

- Naive Construction:
  - Create all rotations for all strings in the collection
  - Sort all rotations together (Suffix Array)
  - Store the last symbols in each suffix
- Strings are “cyclic”
  - Getting the predecessor always gets a suffix from the same string
  - Impossible to “jump” from one string to another

Index	Rotations	Suffix Array	MSBWT
0	ACCA$	$ACC**A**	**A**
1	CCA$A	$CAA**A**	**A**
2	CA$AC	A$AC**C**	**C**
3	A$ACC	A$CA**A**	**A**
4	$ACCA	AA$C**A**	**A**
5	CAAA$	AAA$**C**	**C**
6	AAA$C	ACCA**$**	**$**
7	AA$CA	CA$A**C**	**C**
8	A$CAA	CAAA**$**	**$**
9	$CAAA	CCA$**A**	**A**

The multi-string BWT for strings “ACCA$” and “CAAA$”.

MSBWT and FM-index

- Identical Definition
- Find *k*-mer “CA”
  - Initialize to full range (“”)
    low, high = 0, 10
  - Find occurrences of “A”
    low = O[‘A’]+F[low][‘A’] = 2+0 = 2
    high = O[‘A’]+F[high][‘A’] = 2+5 = 7
  - Find occurrences of “CA”
    low = O[‘C’]+F[low][‘C’] = 7+0 = 7
    high = O[‘C’]+F[high][‘C’] = 7+2 = 9

Index	Suffix Array	MSBWT	FM-index		
			$	A	C
0	$ACC**A**	**A**	0	0	0
1	$CAA**A**	**A**	0	1	0
2	A$AC**C**	**C**	0	2	0
3	A$CA**A**	**A**	0	2	1
4	AA$C**A**	**A**	0	3	1
5	AAA$**C**	**C**	0	4	1
6	ACCA**$**	**$**	0	4	2
7	CA$A**C**	**C**	1	4	2
8	CAAA**$**	**$**	1	4	3
9	CCA$**A**	**A**	2	4	3
10	—	—	**2**	**5**	**3**
Offset (O)	—	—	**0**	**2**	**7**

MSBWT Merging

- Given two MSBWTs, can we merge them into a single structure?
  - Improved compression if data is similar
  - Single search through all data
  - Fundamental operation in many data structures
- Core concept:
  - MSBWT is an implicit suffix array
  - Suffix array is a sorted list of suffixes
  - Merge two sorted lists (easy!)

Merging - Overview

Inputs

Suffix	BWT_0
$ACCA	A
A$ACC	C
ACCA$	$
CA$AC	C
CCA$A	A

Suffix	BWT_1
$CAAA	A
A$CAA	A
AA$CA	A
AAA$C	C
CAAA$	$

Algorithm (??)

Interleave
0
1
0
1
1
1
0
0
1
0

Output

Suffix	BWT
$ACCA	A
$CAAA	A
A$ACC	C
A$CAA	A
AA$CA	A
AAA$C	C
ACCA$	$
CA$AC	C
CAAA$	$
CCA$A	A

Calculating the Interleave

- Holt & McMillan (2014)
- Algorithm Intuition:
  - Initialize interleave as concatenation of BWTs
  - Most-significant-symbol radix sort on implicit suffix array
  - Converges to correct interleave

Full Example

Inputs

Suffix	BWT_0
$ACCA	A
A$ACC	C
ACCA$	$
CA$AC	C
CCA$A	A

Suffix	BWT_1
$CAAA	A
A$CAA	A
AA$CA	A
AAA$C	C
CAAA$	$

Initial

I_0	S_0	B_0
0	$ACCA	A
0	A$ACC	C
0	ACCA$	$
0	CA$AC	C
0	CCA$A	A
1	$CAAA	A
1	A$CAA	A
1	AA$CA	A
1	AAA$C	C
1	CAAA$	$

$

A

C

Iter 1

I_1	S_1	B_1
0	$ACCA	A
1	$CAAA	A
0	A$ACC	C
0	ACCA$	$
1	A$CAA	A
1	AA$CA	A
1	AAA$C	C
0	CA$AC	C
0	CCA$A	A
1	CAAA$	$

Iter 2

I_2	S_2	B_2
0	$ACCA	A
1	$CAAA	A
0	A$ACC	C
1	A$CAA	A
1	AA$CA	A
1	AAA$C	C
0	ACCA$	$
0	CA$AC	C
1	CAAA$	$
0	CCA$A	A

Iter 3

I_3	S_3	B_3
0	$ACCA	A
1	$CAAA	A
0	A$ACC	C
1	A$CAA	A
1	AA$CA	A
1	AAA$C	C
0	ACCA$	$
0	CA$AC	C
1	CAAA$	$
0	CCA$A	A

S_x = implicit suffix array given interleave I_x, **B_x** = BWT given the interleave I_x

Merge for two BWTs of strings “ACCA$” and “CAAA$”. The correct interleave is found once no change occurs from iteration 2 to iteration 3. S_x and B_x are not actually stored in memory.

Construction via Merging

- Holt & McMillan (2014)
- Divide-and-conquer:
  - Build BWTs for each individual string
  - Merge until only one BWT remains
  - Good for long non-uniform strings

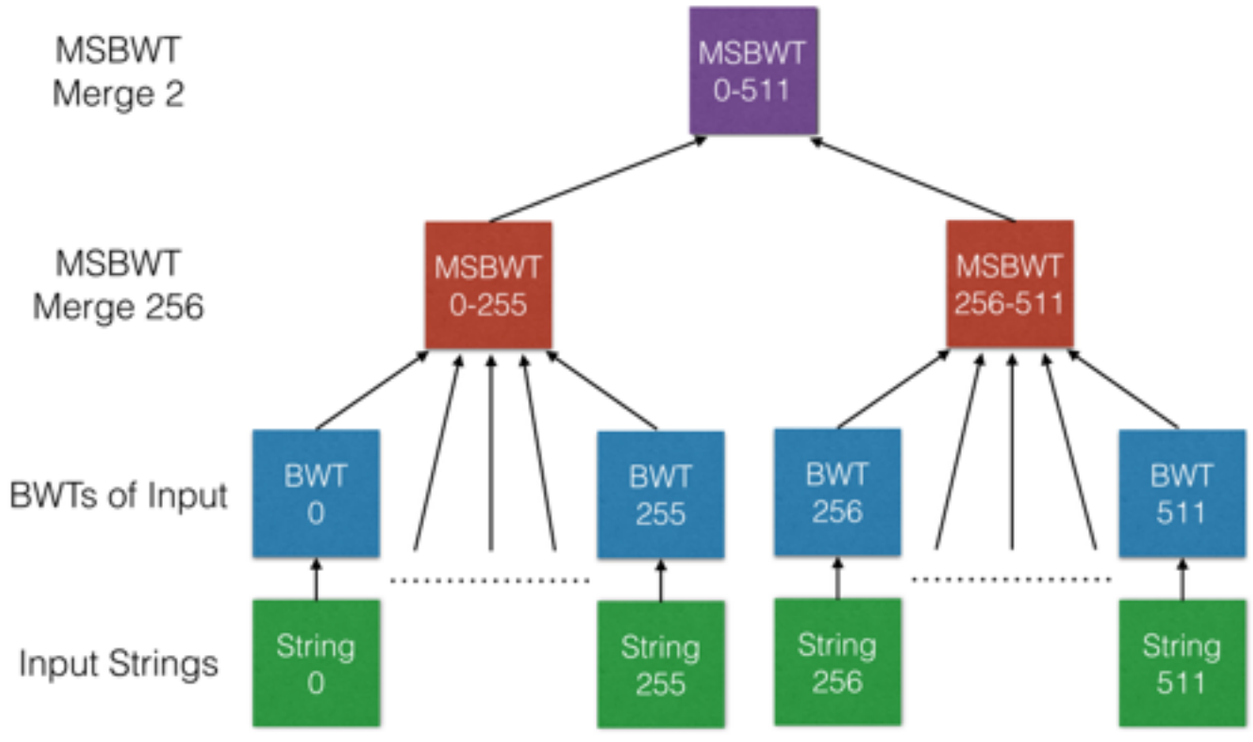


Illustration of merge for 512 strings

MSBWT Applications

- Instead of building a BWT of the reference genome, build a MSBWT of the sequenced reads
- Arbitrary exact match *k*-mer queries
  - O(*k*) time
  - Enables fast searches/counting
- Recover an arbitrary read of length *L* from MSBWT
  - O(*L*) time
  - Enables extraction of user-selected reads

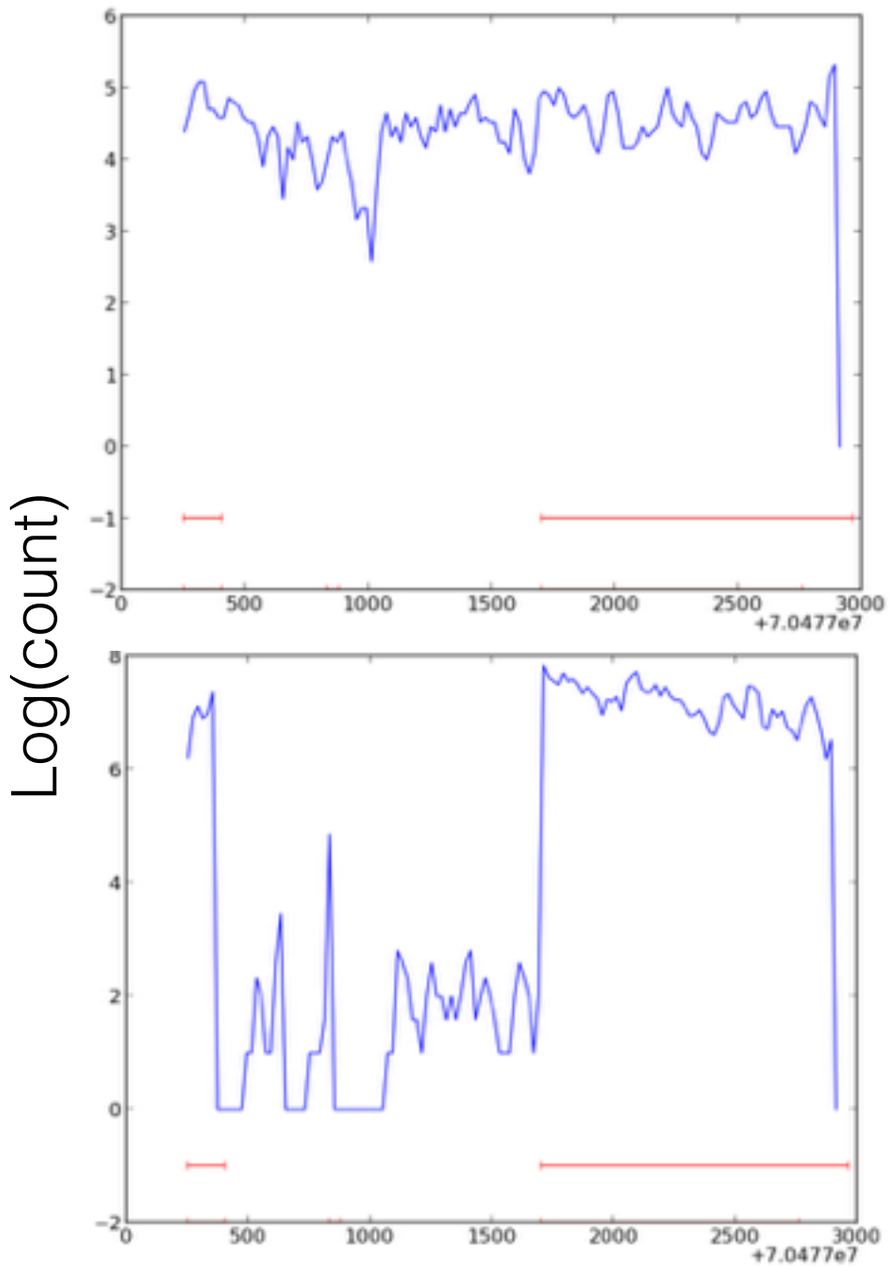
K-mer Search & Extraction

- Basic utilization
  - Search for all reads with a given *k*-mer
  - Extract all reads with that *k*-mer or the reverse-complement of the *k*-mer
  - Build a consensus

k-mer search & consensus builder
Green - *k*-mer query
Red - forward reads
Blue - reverse complemented reads

Reference-based Searches

- Given a reference genome and region of that genome
- Split reference into *k*-mers
- Count the abundance of each *k*-mer and plot
- Fast - O(*k*) time per *k*-mer
- Similar to a post-alignment pileup

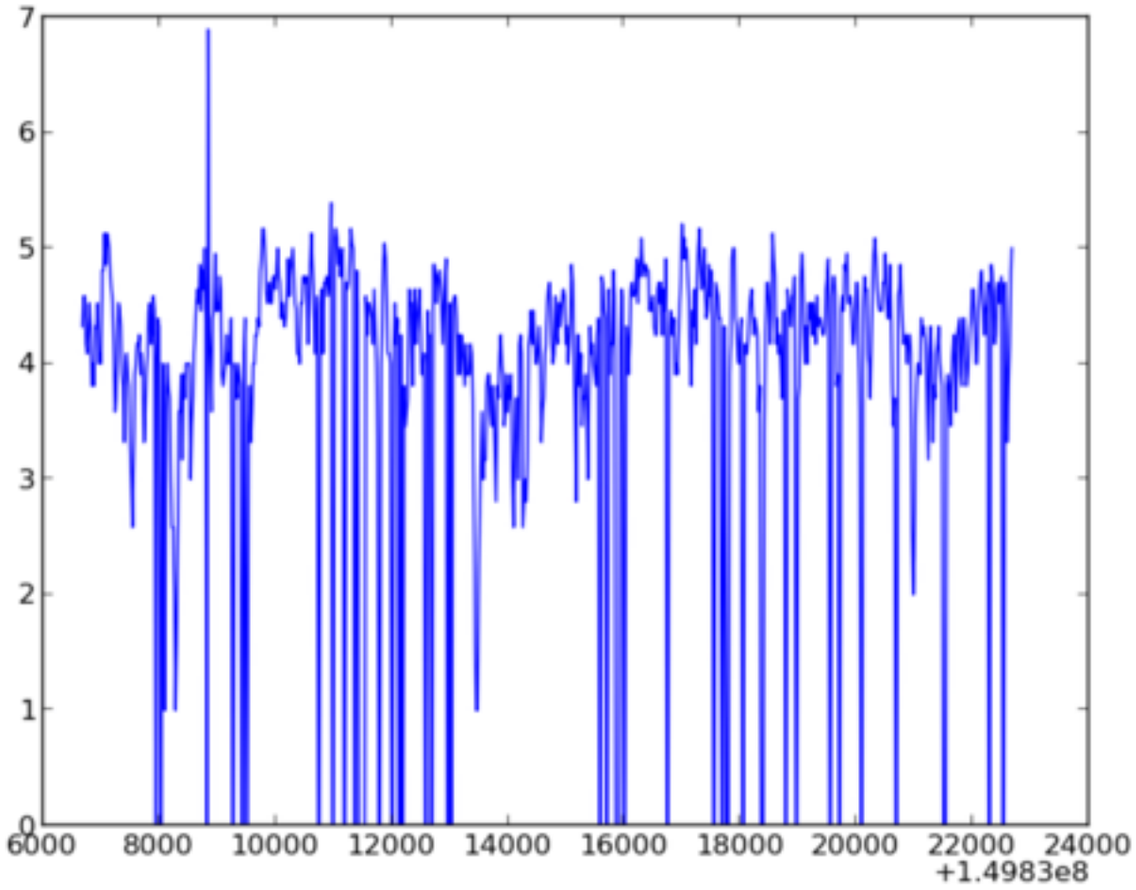


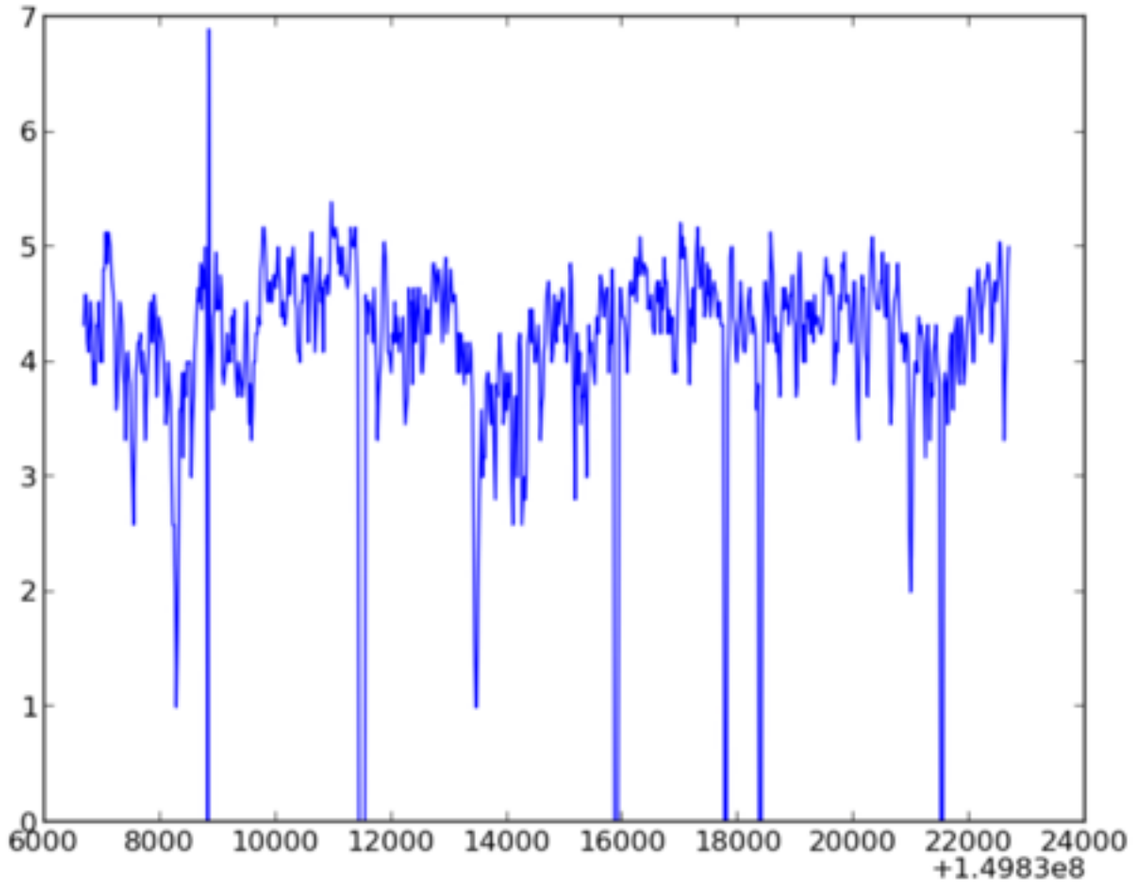
DNA-seq

RNA-seq

CAST/EiJ at *Egr3*, counting 40-mers overlapping by 20

Reference Correction





Uncorrected

Corrected

```
149,838,013: 0   TTGATGGCTCGATGCATTCATTACCTGATCACTGCTCCCG
149,838,033: 0   TTACCTGATCACTGCTCCCGTTATGTAGGGAATGGGTACA

149,838,013: 18  TTGATGGCTCGATGCATTCATTACTTGATCACTGCTCCCG
149,838,033: 17  TTACTTGATCACTGCTCCCGTTATGTAGGGAATGGGTACA
```

CAST/EiJ DNA-seq for annotated gene *Igf2*

Targeted Assembly

- *De novo* assembly given a *k*-mer target known as the "seed" *k*-mer
- Extend the seed by counting the occurrence of each possible extension
- Generates a graph extending from the seed
  - Nodes - continuous unambiguous choice of extensions (similar to a contig)
  - Edges - multiple possible choices for extension

Targeted Assembly Tool Demo

- Gene
- Mitochondria

Practical Adaptations

- Compressed BWT is small
- FM-index is not, O(A*N) for alphabet of size A and a BWT of length N
- Trade-off, space v. time:
  - Use a sampled FM-index
  - B - bin size
  - Uses O(A*N/B) values
  - Requires O(B) time per lookup (for a fixed size B, this is just a larger constant time lookup)

Summary

- Burrows-Wheeler Transform & FM-index
  - More compressible
  - Last-first relationship
  - O(k) search time for arbitrary k-mer
  - O(m) recovery for string of length m
  - MSBWT for string collections
  - Can be merged