Burrows-Wheeler Transform and FM Index

Ben Langmead

You are free to use these slides. If you do, please sign the guestbook (www.langmead-lab.org/teaching-materials), or email me (ben.langmead@gmail.com) and tell me briefly how you’re using them. For original Keynote files, email me.
Reversible permutation of the characters of a string, used originally for compression

How is it useful for compression? How is it reversible? How is it an index?

Burrows-Wheeler Transform

```python
def rotations(t):
    """ Return list of rotations of input string t """
    tt = t * 2
    return [ tt[i:i+len(t)] for i in xrange(0, len(t)) ]

def bwm(t):
    """ Return lexicographically sorted list of t’s rotations """
    return sorted(rotations(t))

def bwtViaBwm(t):
    """ Given T, returns BWT(T) by way of the BWM """
    return ''.join(map(lambda x: x[-1], bwm(t)))
```

```python
>>> bwtViaBwm("Tomorrow_and_tomorrow_and_tomorrow$
'w$wwdd__nn oo aattTmmmrrrrrrrooo__ooo'

>>> bwtViaBwm("It_was_the_best_of_times_it_was_the_worst_of_times$
's$esttssfftteww_hhmmboottttt_ii__woeaaressIi______'

>>> bwtViaBwm('in_the_jingle_jangle_morning_Ill_come_following_you$
'u_gleeengj_m1h1_nnnnt$nwj__lggIolo_iiiiarfcmylo_o0_"
```

Python example: [http://nbviewer.ipython.org/6798379](http://nbviewer.ipython.org/6798379)
Burrows-Wheeler Transform

Characters of the BWT are sorted by their right-context. This lends additional structure to BWT(T), tending to make it more compressible.

Figure 1: Example of sorted rotations. Twenty consecutive rotations from the sorted list of rotations of a version of this paper are shown, together with the final character of each rotation.

Burrows-Wheeler Transform

BWM bears a resemblance to the suffix array

BWM(T)

<table>
<thead>
<tr>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a $</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b a $</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a b a $</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b a a b a $</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>b a $</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>b a a b a $</td>
</tr>
</tbody>
</table>

SA(T)

Sort order is the same whether rows are rotations or suffixes
Burrows-Wheeler Transform

In fact, this gives us a new definition / way to construct BWT(T):

\[
BWT[i] = \begin{cases} 
T[SA[i] - 1] & \text{if } SA[i] > 0 \\
\$ & \text{if } SA[i] = 0
\end{cases}
\]

“BWT = characters just to the left of the suffixes in the suffix array”
Burrows-Wheeler Transform

def suffixArray(s):
    """ Given T return suffix array SA(T). We use Python's sorted function here for simplicity, but we can do better. """
    satups = sorted(((s[i:], i) for i in xrange(0, len(s))))
    # Extract and return just the offsets
    return map(lambda x: x[1], satups)

def bwtViaSa(t):
    """ Given T, returns BWT(T) by way of the suffix array. """
    bw = []
    for si in suffixArray(t):
        if si == 0: bw.append('$')
        else: bw.append(t[si-1])
    return ''.join(bw) # return string-ized version of list bw

>>> bwtViaSa("Tomorrow_and_tomorrow_and_tomorrow$")
'w$wwdd__nnoooaattTmmmmrrrrrooo__ooo'

>>> bwtViaSa("It_was_the_best_of_times_it_was_the_worst_of_times$")
's$esttssfftteww_hhmmbootttt_ii__woeeaaressIi_______'

>>> bwtViaSa('in_the_jingle_jangle_morning_Ill_come_following_you$')
'u_gleeeengj_mlhl_nnnnt$nwj__lggIolo_iiiiarfcmylo_oo_'

Python example: http://nbviewer.ipython.org/6798379
Burrows-Wheeler Transform

How to reverse the BWT?

BWM has a key property called the LF Mapping...
Burrows-Wheeler Transform: T-ranking

Give each character in $T$ a rank, equal to # times the character occurred previously in $T$. Call this the **$T$-ranking**.

$$a_0 \ b_0 \ a_1 \ a_2 \ b_1 \ a_3 \ \$$$

Now let’s re-write the BWM including ranks...
Burrows-Wheeler Transform

BWM with T-ranking:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>F</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
</tr>
<tr>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
</tr>
<tr>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
</tr>
<tr>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
</tr>
<tr>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
</tr>
<tr>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
</tr>
<tr>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
</tr>
</tbody>
</table>

Look at first and last columns, called F and L

And look at just the a's

a's occur in the same order in F and L. As we look down columns, in both cases we see: a₃, a₁, a₂, a₀
Burrows-Wheeler Transform

BWM with T-ranking:

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a_0 b_0 a_1 a_2 b_1 a_3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a_3 a_0 b_0 a_1 a_2 b_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a_1 a_2 b_1 a_3 a_0 b_0$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a_2 b_1 a_3 a_0 b_0 a_1$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$a_0 b_0 a_1 a_2 b_1 a_3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b_1 a_3 a_0 b_0 a_1 a_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$b_0 a_1 a_2 b_1 a_3 a_0$</td>
<td></td>
</tr>
</tbody>
</table>

Same with $b$s: $b_1, b_0$
Burrows-Wheeler Transform

*Reversible permutation* of the characters of a string, used originally for compression

How is it useful for compression?  How is it reversible?  How is it an index?

---

Burrows-Wheeler Transform: LF Mapping

LF Mapping: The $i^{th}$ occurrence of a character $c$ in $L$ and the $i^{th}$ occurrence of $c$ in $F$ correspond to the same occurrence in $T$

However we rank occurrences of $c$, ranks appear in the same order in $F$ and $L$
Burrows-Wheeler Transform: LF Mapping

Why does the LF Mapping hold?

Why are these Rs in this order relative to each other?

They’re sorted by right-context

Why are these Rs in this order relative to each other?

They’re sorted by right-context

Occurrences of c in F are sorted by right-context. Same for L!

Whatever ranking we give to characters in T, rank orders in F and L will match
### Burrows-Wheeler Transform: LF Mapping

**BWM with T-ranking:**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
</tr>
<tr>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
</tr>
<tr>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
</tr>
<tr>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
</tr>
<tr>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
</tr>
<tr>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
<td>a₀</td>
<td>b₀</td>
<td>a₁</td>
</tr>
<tr>
<td>b₀</td>
<td>a₁</td>
<td>a₂</td>
<td>b₁</td>
<td>a₃</td>
<td>$</td>
</tr>
</tbody>
</table>

We’d like a different ranking so that for a given character, ranks are in ascending order as we look down the F / L columns...
Burrows-Wheeler Transform: LF Mapping

BWM with B-ranking:

\[
\begin{array}{cccccc}
F & & & & & L \\
$ & a_3 & b_1 & a_1 & a_2 & b_0 & a_0 \\
a_0 & $ & a_3 & b_1 & a_1 & a_2 & b_0 \\
a_1 & a_2 & b_0 & a_3 & $ & a_3 & b_1 \\
a_2 & b_0 & a_0 & $ & a_3 & b_1 & a_1 \\
a_3 & b_1 & a_1 & a_2 & b_0 & a_0 & $ \\
b_0 & a_0 & $ & a_3 & b_1 & a_1 & a_2 \\
b_1 & a_1 & a_2 & b_0 & a_0 & $ & a_3 \\
\end{array}
\]

Ascending rank

\[F\] now has very simple structure: a $, a block of a\text{\text{\text{s}}} with ascending ranks, a block of b\text{\text{\text{s}}} with ascending ranks
Burrows-Wheeler Transform

Which BWM row begins with $b_1$?

Answer: row 6

Skip row starting with $\$ (1 row)
Skip rows starting with $a$ (4 rows)
Skip row starting with $b_0$ (1 row)
Say $T$ has $300$ $A$s, $400$ $C$s, $250$ $G$s and $700$ $T$s and $\$ < A < C < G < T$

Which BWM row (0-based) begins with $G_{100}$? (Ranks are B-ranks.)

Skip row starting with $\$ (1 row)
Skip rows starting with $A$ (300 rows)
Skip rows starting with $C$ (400 rows)
Skip first 100 rows starting with $G$ (100 rows)

Answer: row $1 + 300 + 400 + 100 = \text{row 801}$
Burrows-Wheeler Transform: reversing

Reverse BWT(T) starting at right-hand-side of $T$ and moving left

**Start** in first row. $F$ must have $. $L$ contains character just prior to $:$ $a_0$

$a_0$: LF Mapping says this is same occurrence of $a$ as first $a$ in $F$. **Jump** to row **beginning** with $a_0$. $L$ contains character just prior to $a_0$: $b_0$.

Repeat for $b_0$, get $a_2$
Repeat for $a_2$, get $a_1$
Repeat for $a_1$, get $b_1$
Repeat for $b_1$, get $a_3$
Repeat for $a_3$, get $,$ done

Reverse of chars we visited $= a_3b_1a_1a_2b_0a_0$ $= T$
Burrows-Wheeler Transform: reversing

Another way to visualize reversing BWT(T):

\[ T: a_3 b_1 a_1 a_2 b_0 a_0 \]$
Burrows-Wheeler Transform: reversing

```python
def rankBwt(bw):
    """ Given BWT string bw, return parallel list of B-ranks. Also returns"""
    """tots: map from character to # times it appears. """
    tots = dict()
    ranks = []
    for c in bw:
        if c not in tots: tots[c] = 0
        ranks.append(tots[c])
        tots[c] += 1
    return ranks, tots

def firstCol(tots):
    """ Return map from character to the range of rows prefixed by"""
    """the character. """
    first = {}
    totc = 0
    for c, count in sorted(tots.iteritems()):
        first[c] = (totc, totc + count)
        totc += count
    return first

def reverseBwt(bw):
    """ Make T from BWT(T) """
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0  # start in first row
    t = '$'  # start with rightmost character
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t  # prepend to answer
        # jump to row that starts with c of same rank
        rowi = first[c][0] + ranks[rowi]
    return t
```

Calculate B-ranks and count occurrences of each char

Make concise representation of first BWM column

Do reversal

Python example: http://nbviewer.ipython.org/6860491
Burrows-Wheeler Transform: reversing

```python
def reverseBwt(bw):
    ''' Make T from BWT(T) '''
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0  # start in first row
    t = '$'  # start with rightmost character
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t  # prepend to answer
        # jump to row that starts with c of same rank
        rowi = first[c][0] + ranks[rowi]
    return t
```
Burrows-Wheeler Transform

We’ve seen how BWT is useful for compression:

Sorts characters by right-context, making a more compressible string

And how it’s reversible:

Repeated applications of LF Mapping, recreating $T$ from right to left

How is it used as an index?
FM Index

FM Index: an index combining the BWT with a few small auxiliary data structures

“FM” supposedly stands for “Full-text Minute-space.” (But inventors are named Ferragina and Manzini)

Core of index consists of $F$ and $L$ from BWM:

$F$ can be represented very simply (1 integer per alphabet character)

And $L$ is compressible

Potentially very space-economical!

FM Index: querying

Though BWM is related to suffix array, we can’t query it the same way.

We don’t have these columns; binary search isn’t possible.
FM Index: querying

Look for range of rows of BWM(T) with $P$ as prefix

Do this for $P$'s shortest suffix, then extend to successively longer suffixes until range becomes empty or we’ve exhausted $P$

$$P = \text{aba}$$

Easy to find all the rows beginning with $a$, thanks to $F$'s simple structure
We have rows beginning with a, now we seek rows beginning with ba

Use LF Mapping. Let new range delimit those bs

Now we have the rows with prefix ba
We have rows beginning with $ba$, now we seek rows beginning with $aba$

Now we have the rows with prefix $aba$
Now we have the same range, [3, 5), we would have got from querying suffix array

Unlike suffix array, we don’t immediately know where the matches are in T...
FM Index: querying

When $P$ does not occur in $T$, we will eventually fail to find the next character in $L$:

$$P = bba$$

Rows with $ba$ prefix

No bs!
FM Index: querying

If we *scan* characters in the last column, that can be very slow, $O(m)$

$$P = \textbf{aba}$$

$F$  $L$
---  
$\$ a b a a b a_3$ 
$a_0$ $\$ a b a a b_1$ 
$a_1$ a b a $\$ a b_0$ 
$a_2$ b a $\$ a b a_1$ 
$a_3$ b a a b a $\$ 
$b_0$ a $\$ a b a a_2$ 
$b_1$ a a b a $\$ a_0$

Scan, looking for bs
FM Index: lingering issues

(1) Scanning for preceding character is slow

(2) Storing ranks takes too much space

(3) Need way to find where matches occur in $T$:

```python
def reverseBwt(bw):
    """ Make T from BWT(T) ""
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0
    t = "$"
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t
        rowi = first[c][0] + ranks[rowi]
    return t
```
Is there an O(1) way to determine which bs precede the as in our range?

Idea: pre-calculate # as, bs in \( L \) up to every row:

We infer \( b_0 \) and \( b_1 \) appear in \( L \) in this range

O(1) time, but requires \( m \times |\Sigma| \) integers
Another idea: pre-calculate # as, bs in L up to some rows, e.g. every 5th row. Call pre-calculated rows checkpoints.

<table>
<thead>
<tr>
<th>F</th>
<th>L</th>
<th>Tally</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>a</td>
<td>1 0</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td>3 2</td>
</tr>
<tr>
<td>b</td>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

Lookup here succeeds as usual

Oops: not a checkpoint

But there’s one nearby

To resolve a lookup for character c in non-checkpoint row, scan along L until we get to nearest checkpoint. Use tally at the checkpoint, adjusted for # of cs we saw along the way.
FM Index: fast rank calculations

Assuming checkpoints are spaced $O(1)$ distance apart, lookups are $O(1)$

What’s my rank?
$482 + 2 - 1 = 483$

What’s my rank?
$439 - 2 - 1 = 436$

Tally

<table>
<thead>
<tr>
<th>L</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>482</td>
<td>432</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>488</td>
<td>439</td>
<td></td>
</tr>
</tbody>
</table>
FM Index: a few problems

Solved! At the expense of adding checkpoints \(O(m)\) integers to index.

(1) \(F\) \hspace{2cm} \(L\)

\[
\begin{array}{cccccc}
\$ & a & b & a & a & b \\
 a_0 & \$ & a & b & a & a \\
a_1 & a & b & a & $ & a \\
a_2 & b & a & $ & a & a \\
a_3 & b & a & a & b & a \\
b_0 & a & $ & a & b & a \\
b_1 & a & a & b & a & $ \\
\end{array}
\]

With checkpoints it’s \(O(1)\)

(2) Ranking takes too much space

```
def reverseBwt(bw):
    """ Make T from BWT(T) """
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    rowi = 0
    t = "$"
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t
        rowi = first[c][0] + ranks[rowi]
    return t
```

With checkpoints, we greatly reduce # integers needed for ranks

But it’s still \(O(m)\) space - there’s literature on how to improve this space bound
FM Index: a few problems

Not yet solved: (3) Need a way to find where these occurrences are in $T$:

If suffix array were part of index, we could simply look up the offsets

<table>
<thead>
<tr>
<th>$F$</th>
<th>$L$</th>
<th>$SA$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$ a b a a b a</td>
<td>6</td>
<td>$$</td>
</tr>
<tr>
<td>a $ a b a a b $</td>
<td>5</td>
<td>a $</td>
</tr>
<tr>
<td>a a b a $ a b</td>
<td>2</td>
<td>a a b a $</td>
</tr>
<tr>
<td>a b a $ a b a</td>
<td>3</td>
<td>a b a $</td>
</tr>
<tr>
<td>a b a a b a $</td>
<td>0</td>
<td>a b a a b a $</td>
</tr>
<tr>
<td>b a $ a b a a</td>
<td>4</td>
<td>b a $</td>
</tr>
<tr>
<td>b a a b a $ a</td>
<td>1</td>
<td>b a a b a $</td>
</tr>
</tbody>
</table>

Offsets: 0, 3

But SA requires $m$ integers
FM Index: resolving offsets

Idea: store some, but not all, entries of the suffix array

Look up for row 4 succeeds - we kept that entry of SA

Look up for row 3 fails - we discarded that entry of SA
FM Index: resolving offsets

But LF Mapping tells us that the \textbf{a} at the end of row 3 corresponds to...
...the \textbf{a} at the beginning of row 2

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{F} & \textbf{L} \\
\hline
$ & a b a a b a \\
a & $ a b a a b \\
a & a b a $ a b \\
a & b a $ a b \\
b & a a b a $ \\
b & a a b a $ \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{|c|}
\hline
\textbf{SA} \\
\hline
6 \\
2 \\
0 \\
4 \\
\hline
\end{tabular}
\end{center}

And row 2 has a suffix array value = 2

So row 3 has suffix array value = 3 = 2 (row 2’s SA val) + 1 (# steps to row 2)

If saved SA values are $O(1)$ positions apart in $T$, resolving offset is $O(1)$ time.
At the expense of adding some SA values ($O(m)$ integers) to index
Call this the “SA sample”

Need a way to find where these occurrences are in $T$:

$\begin{align*}
\$ & a b a a b a_0 \\
a_0 & \$ a b a a b b_0 \\
a_1 & a b a $ a b_1 \\
a_2 & b a $ a b a_1 \\
a_3 & b a a b a $ \\
b_0 & a $ a b a a_2 \\
b_1 & a a b a $ a_3 
\end{align*}$

With SA sample we can do this in $O(1)$ time per occurrence
FM Index: small memory footprint

Components of the FM Index:

First column \((F)\):
\[ \approx |\Sigma| \text{ integers} \]

Last column \((L)\):
\[ m \text{ characters} \]

SA sample:
\[ m \cdot a \text{ integers, where } a \text{ is fraction of rows kept} \]

Checkpoints:
\[ m \times |\Sigma| \cdot b \text{ integers, where } b \text{ is fraction of rows checkpointed} \]

Example: DNA alphabet (2 bits per nucleotide), \(T = \text{human genome}, a = 1/32, b = 1/128\)

First column \((F)\):
16 bytes

Last column \((L)\):
2 bits \(*\) 3 billion chars = 750 MB

SA sample:
3 billion chars \(*\) 4 bytes/char / 32 = \(\approx 400 \text{ MB}\)

Checkpoints:
3 billion \(*\) 4 bytes/char / 128 = \(\approx 100 \text{ MB}\)

Total < 1.5 GB
Approximate Search
BWT-based assembly

Problem: How would you use the BWT to find an overlap alignment of length $l$?