The Burrows Wheeler transform and the FM index
Burrows–Wheeler (bzip2)

Currently best algorithm for text compression

High level:

• Apply the Borrows-Wheeler transform
• Use move-to-front to translate the sorted characters to small integers
• Use Huffman coding
 Shelby : \( S \):

\[
S = \text{abraca}
\]

 Shelby : \( I \):

יצירת מטריצה \( M \) שבנויה ממלאי החזרות הציקליות של \( S \):

\[
M = 
\begin{array}{cccccc}
\text{a} & \text{b} & \text{r} & \text{a} & \text{c} & \text{a} \\
\text{b} & \text{r} & \text{a} & \text{c} & \text{a} & \# \\
\text{r} & \text{a} & \text{c} & \text{a} & \# & \text{a} \\
\text{a} & \text{c} & \text{a} & \# & \text{a} & \text{b} \\
\text{a} & \text{c} & \text{a} & \# & \text{a} & \text{b} \\
\text{c} & \text{a} & \# & \text{a} & \text{b} & \text{r} \\
\text{c} & \text{a} & \# & \text{a} & \text{b} & \text{r} \\
\text{a} & \# & \text{a} & \text{b} & \text{r} & \text{a} \\
\text{a} & \# & \text{a} & \text{b} & \text{r} & \text{a} \\
\# & \text{a} & \text{b} & \text{r} & \text{a} & \text{c} \\
\# & \text{a} & \text{b} & \text{r} & \text{a} & \text{c} \\
\text{a} & \text{b} & \text{r} & \text{a} & \text{c} & \text{a} \\
\end{array}
\]
<table>
<thead>
<tr>
<th>F</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>#</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>b</td>
<td>r</td>
</tr>
<tr>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>r</td>
<td>a</td>
</tr>
</tbody>
</table>
Claim: Every column contains all chars.

You can obtain F from L by sorting
The “a’s” are in the same order in L and in F, Similarly for every other char.
From L you can reconstruct the string (backwards)

What is the last char of S?
From $L$ you can reconstruct the string $L$.

What is the last char of $S$? $a$
From L you can reconstruct the string: L # a r a c b a c r a b ca
From L you can reconstruct the string

aca
**Analysis so far**

Sorting is equivalent to computing the suffix array.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>r</th>
<th>a</th>
<th>c</th>
<th>a</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>#</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>L</td>
<td>a</td>
<td>#</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
</tr>
</tbody>
</table>

Decoding is also linear time.
Two useful arrays

- We do not really need them for decoding but they will be useful later.
- $C[\cdot]$ denotes an array of length $|\Sigma|$ such that $C[c]$ contains the total number of text characters which are alphabetically smaller than $c$.
- $Occ(c,q)$ denotes the number of occurrences of character $c$ in the prefix $L[1,q]$ of the transformed text $L$. 
Two useful arrays

\[
\begin{array}{c|c|c}
\text{mississippi} & L & C \\
\text{mississippi pi} & i & 0 \\
\text{mississippi} & p & i \\
\text{mississippi} & s & m \\
\text{mississippi} & s & p \\
\text{mississippi} & i & s \\
\text{mississippi} & i & s \\
\text{mississippi} & i & s \\
\text{mississippi} & i & s \\
\end{array}
\]

\[
L \rightarrow \text{occ}(s, 4) = 2
\]
The LF mapping

\[ \text{LF}(i) = C[L[i]] + \text{Occ}(L[i], i) \]

\[ \text{LF}(10) = C[s] + \text{Occ}(s, 10) = 12 \]

\[ L(10) = F(12) = s \]
The LF mapping

Say \( T[k] = j^{th} \) char of \( L \)

Where is \( T[k-1] \)?

\( T[k-1] = L[LF[j]] \)
Compression?

All we did so far is a rather strange permutation of the text?
Why is it good?

Characters with the same (right) context appear together
Move to front

<table>
<thead>
<tr>
<th>L</th>
<th>i</th>
<th>p</th>
<th>s</th>
<th>s</th>
<th>s</th>
<th>m</th>
<th>#</th>
<th>p</th>
<th>i</th>
<th>s</th>
<th>s</th>
<th>s</th>
<th>i</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Replace each char in L with the number of distinct char’s seen since its last occurrence.
- Usually will result in a lot of consecutive 0’s and clusters of small numbers
**Move to front - How?**

- Keep an array/list $MTF[1,\ldots,|\Sigma|]$, initially say sorted.

- Replace each char by its index and move it to the front.

\[
L: \begin{array}{cccccccccc}
\text{i} & \text{p} & \text{s} & \text{s} & \text{m} & \# & \text{p} & \text{i} & \text{s} & \text{s} & \text{i} & \text{i} \\
1 & 3 & 4 & 0 & 4 & 4 & 3 & 4 & 4 & 0 & 1 & 0
\end{array}
\]

\[
\begin{array}{cccc}
\# & \text{i} & \text{m} & \text{p} & \text{s} \\
0 & 1 & 2 & 3 & 4
\end{array}
\]

\[
\begin{array}{cccc}
\text{i} & \# & \text{m} & \text{p} & \text{s} \\
0 & 1 & 2 & 3 & 4
\end{array}
\]

\[
\begin{array}{cccc}
\text{p} & \text{i} & \# & \text{m} & \text{s} \\
0 & 1 & 2 & 3 & 4
\end{array}
\]

\[
\begin{array}{cccc}
\text{s} & \text{p} & \text{i} & \# & \text{m} \\
0 & 1 & 2 & 3 & 4
\end{array}
\]

And so on…
Final step

• Finish the encoding by applying a prefix code such as Huffman’s to the integer sequence produced by MTF
Variations

• Different prefix codes
• Use run-length encoding: BWT + MFT + RLE + prefix
Run length encoding

Replace each sequence $0^m$ by $m+1$ encoded in binary using 2 new symbols 0,1

Example

1. $0 \rightarrow 1+1 = 2 \rightarrow 10 \rightarrow 01 \rightarrow 0$
2. $00 \rightarrow 2+1 = 3 \rightarrow 11 \rightarrow 11 \rightarrow 1$
3. $000 \rightarrow 3+1 = 4 \rightarrow 100 \rightarrow 001 \rightarrow 00$
4. $0000 \rightarrow 4+1 = 5 \rightarrow 101 \rightarrow 101 \rightarrow 10$
5. $0000000 \rightarrow 7+1 = 8 \rightarrow 1000 \rightarrow 0001 \rightarrow 000$

$MTF(L) = 1000221000023$

$RLE(MTF(L)) = 1002211023$

$RLE(MTF(L))$ is defined over $\{0,1,1,2,\ldots,|\Sigma|-1\}$
Run length encoding

You can retrieve as follows

00110 = b_0b_1b_2b_3b_4

Real # of ones is (1b_{k-1} ... b_3b_2b_1b_0)_2-1=(101100)_2-1

Replace b_i by (b_i+1)2^i zeros

Total number of zeros:

\[
\sum_{i=0}^{k-1} (b_i + 1)2^i = \sum_{i=0}^{k-1} b_i2^i + \sum_{i=0}^{k-1} 2^i = \left( \sum_{i=0}^{k-1} b_i2^i + 2^k \right) - 1
\]
A prefix code

For $i=1$ to $|\Sigma|-1$

- $\lceil \log(i+1) \rceil$ zeroes
- Followed by a binary representation of $i+1$ which takes $1+\lfloor \log(i+1) \rfloor$ bits
A prefix code

\[
\begin{align*}
\text{bin}(1) & \rightarrow 010 \\
\text{bin}(2) & \rightarrow 011 \\
\text{bin}(3) & \rightarrow 00100 \\
\text{bin}(4) & \rightarrow 00101 \\
\text{bin}(5) & \rightarrow 00110 \\
\text{bin}(6) & \rightarrow 00111 \\
\text{bin}(7) & \rightarrow 0001000
\end{align*}
\]
How good is the compression?

Theorem (FM 05):

\[ \text{prefix}(\text{RLE}(\text{MTF}(\text{BWT}(T)))) = 5nH_k(T) + O(\log(n)) \]

\( H_k(T) \) is the \( k^{th} \) order empirical entropy of \( T \)

This is true simultaneously for every \( k \)
The FM index
Substring search using the BWT

(Count the pattern occurrences)

$P = si$

First step

Rows prefixed by char “i”

Occurrence count: 2

Inductive step: Given fr, lr for $P[j+1, p]$

1. Take $c = P[j]$
2. Find the first $c$ in $L[fr, lr]$
3. Find the last $c$ in $L[fr, lr]$
4. $L$-to-$F$ mapping of these chars

Available info

$C$

# 0
i 1
m 5
p 6
s 8

slide stolen from Paolo Ferragina@
Substring search using the BWT

(Count the pattern occurrences)

First step

rows prefixed by char “i”

\[ P = si \]

Available info

\[ C \]

\[ \text{fr}' = C[s] + \text{occ}(s,1) + 1 \]

\[ \text{lr}' = C[s] + \text{occ}(s,5) \]

\[ \text{fr}' = C[s] + \text{occ}(s,\text{fr}-1) + 1 \]

\[ \text{lr}' = C[s] + \text{occ}(s,\text{lr}) \]
Make a bit vector for each character

\[ \text{L} \]

\[
\begin{array}{cccccccccc}
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
\end{array}
\]

\[ \text{occ}(s, 4) = \text{rank}(4) \]

\[ \text{rank}(i) = \text{how many ones are there before position } i? \]
How do you answer rank queries?

\[ \begin{array}{cccccccccccc}
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
\uparrow
\end{array} \]

\( \text{rank}(i) = \text{how many ones are there before position } i \) ?

We can prepare a vector with all answers

\[ \begin{array}{cccccccccc}
0 & 0 & 1 & 2 & 2 & 2 & 2 & 2 & 3 & 4 & 4 & 4
\end{array} \]
Let's do it with $O(n)$ bits per character

Partition in $\frac{2n}{\log(n)}$ blocks of size $\frac{\log(n)}{2}$

Keep the answer for each prefix of the blocks

There are $\sqrt{n}$ “kinds” of blocks, prepare a table with all answers for each block
In our solution the bit vector takes $\Theta(n)$ bits and also the “additionals” take $\Theta(n)$ bits.
Can we do it with smaller overhead: so additionals would take $o(n)$?

Superblocks of size $\log^2(n)$

Each block keeps the number of one in previous blocks that are in the same superblock.
Analysis

The superblock table is of size $n / \log(n)$

The block table is of size $(\log\log(n)) * n / \log(n)$

The tables for the blocks $\sqrt{n \log(n) \log\log(n)}$

So the additionals take $o(n)$ space
Summary so far

• We can count occurrence using:
  • The array $C$
  • A bit vector per character $O(|\Sigma|n)$
  • Additional tables of size $o(n)$
Next steps

Do it without keeping the bit vectors themselves?

Can we do it while keeping only prefix(RLE(MTF(BWT(T))))?

How do we find the occurrences themselves?
Storing
prefix(RLE(MTF(BWT(T))))

$L$: ...

\[ \ell = \gamma \log(n) \]

\[ Z = \text{pr}(\text{RLE(MTF(BWT(T))))} : \]

0100101

Also partition into superblocks (not shown) each consisting of \( \ell \) blocks
The tables

- \( \text{NOs}[c,f] \): \#occ of \( c \) before superblock \( f \)
- \( \text{NOb}[c,i] \): \#occ of \( c \) from the beginning of the superblock and before block \( i \)
The tables

- **MTF[i]**: The MTF list at the beginning of block $i$
- **$S[c,j,Z,M]$**: #occ of $c$ up to char $j$ of a block whose compressed rep. is $Z$ and MTF at the beginning is $M$

We would like to use it by retrieving $S[c,j,Z_i,MFT[i]]$

How do we find $Z_i$?
Two more tables

- $W_s[f]$: where does the compressed part of superblock $f$ starts
- $W_b[i]$: The offset of block $i$ within its superblock
- Retrieving block $j$

\[
s = \left\lfloor \frac{j}{\ell} \right\rfloor
\]

\[
start = W_s[s] + W_b[j]
\]

\[
end = \text{if } (j \mod \ell = 0) \ W_s[s+1] - 1 \text{ else } W_s[s] + W_b[j+1] - 1
\]

\[
Z_j = Z[start, end]
\]
Tables sizes

- $W_s, W_b, NO_s, NO_b, MTF: O(n \log \log(n)/\log(n))$

- What is the size of $S$?

$S[c,j,Z,M]: \#occ$ of $c$ up to char $j$ of a block whose compressed rep. is $Z$ and the MTF at the beginning is $M$

$$|Z| = \left(1 + 2 \left\lfloor \log |\Sigma| \right\rfloor \right)\ell = \left(1 + 2 \left\lfloor \log |\Sigma| \right\rfloor \right)\gamma \log(n)$$

Choose $\gamma$ such that $|Z| \leq \delta \log(n)$

So the # of possible $Z$'s is $\leq n^{\delta}$

$\Rightarrow$ So the size of $S$ is $O(n^{\delta}\ell \log(\ell))$
Find the occurrences

• We find fr, lr, so we know that there are \( lr \)-fr+1 occurrences
• But how do we find the positions of these occurrences?
• If we had the suffix array it would have been easy
The LF mapping..

\[ LF(i) = C[L[i]] + \text{Occ}(L[i], i) \]

\[ LF(9) = C[s] + \text{Occ}(s, 9) = 11 \]

⇒ The previous suffix is at row 11
The LF mapping...

\( LF(i) = C[L[i]] + \text{Occ}(L[i], i) \)

\( LF(11) = C[i] + \text{Occ}(i, 11) = 4 \)

\( \rightarrow \) The previous suffix is at row 4
Finding the occurrences

We can keep going until we get to the first prefix (and record where the first prefix is in the array)

By counting steps we discover the position
Finding the occurrences

• This may take too much time...

• Remember the positions of a subset of the suffixes in the suffix array:

Mississippi....

\[ \log^{1+\varepsilon}(n) \]
Finding the occurrences

<table>
<thead>
<tr>
<th>#mississippi</th>
<th>i#mississipp</th>
<th>ippi#mississ</th>
<th>#mississippi</th>
<th>i#mississi</th>
<th>ippi#mississi</th>
<th>#mississippi</th>
<th>i#mississ</th>
<th>ippi#mississi</th>
</tr>
</thead>
<tbody>
<tr>
<td>→ issipipi#misi</td>
<td>→ ississippi#mississippi</td>
<td>→ pi#mississippi</td>
<td>→ ppi#mississippi</td>
<td>→ sipippi#mississippi</td>
<td>→ sissippi#mississippi</td>
<td>→ sissippi#mississippi</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These suffixes are equally spaced along the text

But in the suffix array they are in arbitrary positions

Keep then in a dictionary structure like a hash table
Summary

Can find each occurrence in $O(\log^{1+\varepsilon}(n))$ time

Additional space:

$$O\left(\frac{n}{\log^{1+\varepsilon}(n)} \log(n)\right) = O\left(\frac{n}{\log^\varepsilon(n)}\right)$$

Time to find an occurrence:

$$O(\log^{1+\varepsilon}(n))$$