max planck institut
informatik

Burrows-Wheeler transform, with applications!

Algorithms on Strings

Paweł Gawrychowski

July 27, 2013

Outline

Today we will continue talking about compression.

Burrows-Wheeler transform

Based on the suffix array. Not very nice in theory, but useful in practice.

OK, this is a theory course, so we will try to at least mention one or two nice theorems.

BWT is not a compression method in itself. Given a string $w[1..n]$, it produces a (probably different) string of length $n + 1$.

For `mississippi`, the transform returns `ipssm$pissii`. Hmm.

So why do we care? Well, the idea is that the transformed string will be easier to compress with some simple tools. So, we first apply the transform, then use some simple compression scheme. Hopefully, the result will be better than applying the simple compression scheme directly on the original string.

For this to work, the transform needs to be efficiently **reversible**.

Take your string, append a special character \$, and construct all cyclic shifts of the resulting word of length $|w| + 1$.

$w =$ mississippi\$

mississippi\$	\$mississipp	\$
ississippi\$m	i\$mississip	i\$
ssissippi\$mi	ippi\$missis	ippi\$
sissippi\$mis	issippi\$mis	issippi\$
issippi\$miss	ississippi\$	ississippi\$
ssippi\$missi	mississippi	mississippi\$
sippi\$missis	pi\$mississi	pi\$
ippi\$mississ	ppi\$mississ	ppi\$
ppi\$mississi	sippi\$missi	sippi\$
pi\$mississip	sissippi\$mi	sissippi\$
i\$mississipp	ssippi\$miss	ssippi\$
\$mississippi	ssissippi\$m	ssissippi\$

Take your string, append a special character \$, and construct all cyclic shifts of the resulting word of length $|w| + 1$.

$w =$ mississippi\$

mississippi\$	\$mississipp	\$
ississippi\$m	i\$mississip	i\$
ssissippi\$mi	ippi\$missis	ippi\$
sissippi\$mis	issippi\$mis	issippi\$
issippi\$miss	ississippi\$	ississippi\$
ssippi\$missi	mississippi	mississippi\$
sippi\$missis	pi\$mississi	pi\$
ippi\$mississ	ppi\$mississ	ppi\$
ppi\$mississi	sippi\$missi	sippi\$
pi\$mississip	sissippi\$mi	sissippi\$
i\$mississipp	ssippi\$miss	ssippi\$
\$mississippi	ssissippi\$m	ssissippi\$

The last column of the sorted array is the Burrows-Wheeler transformed text.

$W =$ mississippi$

mississippi$	$mississippi	$
ississippi$m	i$mississipp	i$
ssissippi$mi	ippi$mississ	ippi$
sissippi$mis	issippi$miss	issippi$
issippi$miss	ississippi$m	ississippi$
ssippi$missi	mississippi$	mississippi$
sippi$missis	pi$mississip	pi$
ippi$mississ	ppi$mississi	ppi$
ppi$mississi	sippi$missis	sippi$
pi$mississip	sissippi$mis	sissippi$
i$mississipp	ssippi$missi	ssippi$
$mississippi	ssissippi$mi	ssissippi$

The last column of the sorted array is the Burrows-Wheeler transformed text.

$W =$ mississippi$

mississippi$	$mississippi	$
ississippi$m	i$mississipp	i$
ssissippi$mi	ippi$mississ	ippi$
sissippi$mis	issippi$miss	issippi$
issippi$miss	ississippi$m	ississippi$
ssippi$missi	mississippi$	mississippi$
sippi$missis	pi$mississip	pi$
ippi$mississ	ppi$mississi	ppi$
ppi$mississi	sippi$missis	sippi$
pi$mississip	sissippi$mis	sissippi$
i$mississipp	ssippi$missi	ssippi$
$mississippi	ssissippi$mi	ssissippi$

How to compute the transform?

Lemma

BWT can be computed in linear time by constructing the suffix array of $w\$$.

So, that's easy. But can we reverse the transform efficiently, too?

First column of the sorted array is called F, and the last column is called *L*. So, BWT is really L.

We define the L-to-F mapping.

LF[i] = j

We shift the word in the i-th row of the sorted array by one to the right and locate the j-th row of the sorted array containing the result.

Take the first row, which contain `$mississippi`. Shift it by one to get `i$mississipp`. The result is in the second row, so LF[1] = 2.

$W =$ mississippi$

		i	LF[i]
mississippi$	$mississippi	1	2
ississippi$m	i$mississipp	2	7
ssissippi$mi	ippi$mississ	3	9
sissippi$mis	issippi$miss	4	10
issippi$miss	ississippi$m	5	6
ssippi$missi	mississippi$	6	1
sippi$missis	pi$mississip	7	8
ippi$mississ	ppi$mississi	8	3
ppi$mississi	sippi$missis	9	11
pi$mississip	sissippi$mis	10	12
i$mississipp	ssippi$missi	11	4
$mississippi	ssissippi$mi	12	5

Now look at L[1], L[LF[1]], L[LF[LF[1]]],...

$w =$ mississippi$

		i	LF[i]
mississippi$	$mississippi	1	2
ississippi$m	i$mississipp	2	7
ssissippi$mi	ippi$mississ	3	9
sissippi$mis	issippi$miss	4	10
issippi$miss	ississippi$m	5	6
ssippi$missi	mississippi$	6	1
sippi$missis	pi$mississip	7	8
ippi$mississ	ppi$mississi	8	3
ppi$mississi	sippi$missis	9	11
pi$mississip	sissippi$mis	10	12
i$mississipp	ssippi$missi	11	4
$mississippi	ssissippi$mi	12	5

Now look at L[1], L[LF[1]], L[LF[LF[1]]],...

$w =$ mississippi$

		i	LF[i]
mississippi$	$mississippi	1	2
ississippi$m	i$mississipp	2	7
ssissippi$mi	ippi$mississ	3	9
sissippi$mis	issippi$miss	4	10
issippi$miss	ississippi$m	5	6
ssippi$missi	mississippi$	6	1
sippi$mississ	pi$mississip	7	8
ippi$mississ	ppi$mississi	8	3
ppi$mississi	sippi$missis	9	11
pi$mississip	sissippi$mis	10	12
i$mississipp	ssippi$missi	11	4
$mississippi	ssissippi$mi	12	5

Now look at L[1], L[LF[1]], L[LF[LF[1]]],...

$W =$ mississippi$

		i	LF[i]
mississippi$	$mississippi	1	2
ississippi$m	i$mississipp	2	7
ssissippi$mi	ippi$mississ	3	9
sissippi$mis	issippi$miss	4	10
issippi$miss	ississippi$m	5	6
ssippi$missi	mississippi$	6	1
sippi$missis	pi$mississip	7	8
ippi$mississ	ppi$mississi	8	3
ppi$mississi	sippi$missis	9	11
pi$mississip	sissippi$mis	10	12
i$mississipp	ssippi$missi	11	4
$mississippi	ssissippi$mi	12	5

Now look at L[1], L[LF[1]], L[LF[LF[1]]],...

Observation

$$w[|w| - i] = \mathrm{L}[\mathrm{LF}^i[1]]$$

See the whiteboard.
So, having the array LF allows us to actually reconstruct w. But where do we get this array from?

- $C[x]$ is the the total number of all characters $c < x$ in the text.
- $Occ(x, i)$ is the number of occurrences of the letter x in $\mathrm{L}[1..i]$.

Lemma

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

See the whiteboard.

This actually gives a linear time reconstruction algorithm if we can implement $Occ(x, i)$ efficiently. For that, observe that we only need $Occ(\mathrm{L}[i], i)$, which can be computed in a single left-to-right scan over L. More precisely, we scan and maintain a table counting the occurrences of each letter seen so far.

Theorem

BWT and its reverse can be both computed in linear time.

OK, but why should you care about BWT? There are (at least) two reasons:

- it compresses well,
- it can be used to efficiently count and locate all occurrences of a given pattern.

Let's start with the first reason.

The compression works as follows:

- compute the Burrows-Wheeler transform bwt(w),
- encode the result using move-to-front scheme mtf(bwt(w)),
- use run-length encoding to replace the runs of 0's,
- compress the result using a zeroth order entropy coder, say Huffman coding.

Move-to-Front

Go through the sequence from left to right. Maintain a permutation of the alphabet. For each character in the sequence, output its rank in the current permutation, and move it to the front (in the permutation).

The compression works as follows:

- compute the Burrows-Wheeler transform bwt(w),
- encode the result using move-to-front scheme mtf(bwt(w)),
- use run-length encoding to replace the runs of 0's,
- compress the result using a zeroth order entropy coder, say Huffman coding.

Move-to-Front

Go through the sequence from left to right. Maintain a permutation of the alphabet. For each character in the sequence, output its rank in the current permutation, and move it to the front (in the permutation).

Zeroth entropy coder is a compression method achieving the zeroth order entropy bound.

Zeroth order entropy H_0

For a text $w[1..n]$ over an alphabet $\{c_1, c_2, \ldots, c_\sigma\}$ we define n_i to be the number of occurrences of c_i and the entropy as:

$$H_0(w) = \sum_i \frac{n_i}{n} \log \frac{n}{n_i}$$

It tells you how well you can compress the text if you assign a unique **code** to each letter. For letters that occur lots of times, the code should be short, for letters that occur just once in a while the code can be longer. To allow unique decoding, the code should be **prefix-free**.

That's the zeroth order entropy. One can generalize it to the k-th order entropy, where the code of each letter can actually depend on the previous k letters of the text. So, the compressor is allows to have some limited memory.

k-th order entropy H_k

Let w_s is the concatenation of all character of w following the occurrences of s. Then:

$$H_k(w) = \frac{1}{n} \sum_{s \in \Sigma^k} |w_s| H_0(w_s)$$

Theorem [Manzini 01]

Without the run-length improvement, the compressed representation is of size

$$8|s|H_k(s) + \frac{2}{25}|s|$$

Theorem [Manzini 01]

With the run-length improvement, the compressed word is of size

$$(5+\epsilon)|s|H_k^*(s) + \mathcal{O}(1)$$

where $\epsilon \approx 10^{-2}$, and the definition of the entropy is **slightly** modified.

The reason for the modification is that for $\Sigma = \{a\}$ the entropy is always zero. But, we do need around $\log |w|$ bits, right?

Now let's look at the second reason, namely let's think how to count all occurrences of a given pattern using the BWT of a string.

...it's almost like the suffix array, so should be possible, right?

So let's make our life more interesting. We don't want to augment our structure too much. In other words, the additional space should be $o(|w|)$.

Backwards search

All entries in the sorted array starting with a given pattern p form a contiguous range. So, we will try to compute the range, then its length gives us the number of occurrences.

We compute the range for all suffixes of p, namely $p[|p|..|p|]$, $p[|p|-1..|p|]$, ..., $p[1..|p|]$.

- The range for $p[|p|..|p|]$ is simply $[C[p[|p|]]+1, C[p[|p|]+1]]$. (p[|p|]+1 denotes the successor of p[|p|] in the alphabet, we add an artificial entry in the array C so that everyone has one)
- Given the range for $p[i+1..|p|]$, we should compute the range for $p[i..|p|]$.

Say that the range for $p[i+1..|p|]$ was $[a, b]$, and the range for $p[i..|p|]$ should be $[a', b']$. Then $[a', b']$ is clearly inside the range corresponding to all shift starting with $p[i]$, which is $[\mathrm{C}[p[i]]+1, \mathrm{C}[p[i]+1]]$. But how do we determine which of these shifts start with $p[i]p[i+1..|p|]$?

We have this information already! They correspond to shift starting with $p[i+1..|p|]$ such that the corresponding letter in L is $p[i]$. So, the new range is $[\mathrm{C}[p[i]] + Occ(p[i], a-1), \mathrm{C}[p[i]] + Occ(p[i], b)]$.

$w =$ mississippi\$, $p = ssi$

mississippi\$	\$mississippi
ississippi\$m	i\$mississipp
ssissippi\$mi	ippi\$mississ
sissippi\$mis	issippi\$miss
issippi\$miss	ississippi\$m
ssippi\$missi	mississippi\$
sippi\$missis	pi\$mississip
ippi\$mississ	ppi\$mississi
ppi\$mississi	sippi\$missis
pi\$mississip	sissippi\$mis
i\$mississipp	ssippi\$missi
\$mississippi	ssissippi\$mi

Now we only have to show how to implement any $Occ(x, i)$. If $|\Sigma| = \mathcal{O}(1)$, this can actually be done in constant time after adding a **sublinear** additional data to the compressed representation of the text.

s OK, the sublinear actually means $\mathcal{O}(|w|\frac{\log\log|w|}{\log|w|})$. At a very high level the idea is use a two-level Four Russian trick: split the input data into large blocks of size $\log^2|w|$, then split each large block into small blocks of size $\log|w|$.