Introduction to the Burrows-Wheeler Transform and FM Index

Ben Langmead, Department of Computer Science, JHU

November 24, 2013

1 Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) is a way of permuting the characters of a string T into another string $BWT(T)$. This permutation is reversible; one procedure exists for turning T into $BWT(T)$ and another exists for turning $BWT(T)$ back into T. The transformation was originally discovered by David Wheeler in 1983, and was published by Michael Burrows and David Wheeler in 1994 [1].

The BWT has two main applications: compression and indexing. We will discuss both. First we discuss the transformation from T to $BWT(T)$.

1.1 BWT via the BWM

T denotes the string we would like to transform, and $m = |T|$ (the length of T). We assume that T ends with a *terminator* character, denoted \$. We define \$ to be a character that does not appear elsewhere in T, and which is lexicographically prior to all other characters. In the case of DNA strings, for example, the alphabet order with \$ might be $\$ < A < C < G < T$.

Take $T = abaaba\$$. First, we write down the *rotations* of T: the distinct strings we can make from T by repeatedly taking a character from one end and sticking it on the other:

$$\begin{array}{ccccccc}
\$ & a & b & a & a & b & a \\
a & \$ & a & b & a & a & b \\
b & a & \$ & a & b & a & a \\
a & b & a & \$ & a & b & a \\
a & a & b & a & \$ & a & b \\
b & a & a & b & a & \$ & a \\
a & b & a & a & b & a & \$
\end{array}$$

By writing them stacked vertically, we've created an $m \times m$ matrix. Now we sort the rows of the matrix lexicographically (i.e. alphabetically):

$$\begin{array}{ccccccc}
\$ & a & b & a & a & b & a \\
a & \$ & a & b & a & a & b \\
a & a & b & a & \$ & a & b \\
a & b & a & \$ & a & b & a \\
a & b & a & a & b & a & \$ \\
b & a & \$ & a & b & a & a \\
b & a & a & b & a & \$ & a
\end{array}$$

This is the Burrows-Wheeler Matrix ($BWM(T)$). The final column of $BWM(T)$, read from top to bottom, is $BWT(T)$. So for $T = abaaba\$$, $BWT(T) = abba\$aa$.

1.2 BWT via the suffix array

The Burrows-Wheeler Matrix seems to be related to the suffix array: to make a suffix array of T, $SA(T)$, we sort T's suffixes, and to make $BWM(T)$, we sort T's rotations. The relationship is clearer when we write them side by side:

```
BWM:      SA: Suffixes given by SA:
$abaaba    6 $
a$abaab    5 a$
aaba$ab    2 aaba$
aba$aba    3 aba$
abaaba$    0 abaaba$
ba$abaa    4 ba$
baaba$a    1 baaba$
```

They correspond to the same ordering. Look at, for example, where the $s appear in each row of the comparison. So another way of defining $BWT(T)$ is via the suffix array $SA(T)$. Let $BWT[i]$ denote the character at 0-based offset i in $BWT(T)$ and let $SA[i]$ denote the suffix at 0-based offset i in $SA(T)$.

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

Here is a Python implementation of this method for building $BWT(T)$.

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

def bwt(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
```

1.3 Burrows-Wheeler Transform in compression

How is the Burrows-Wheeler Transform useful for compression? First, it's reversible. Transformations used in compression must be reversible to allow both compression and decompression. Second, characters with similar *right-contexts* in T tend to come togehter in $BWT(T)$. This can, for instance, bring several occurrences of the same character together in a tight bunch. This is hard to see in small examples; in the following example, this bunching is more obvious:

```
>>> bwt("tomorrow and tomorrow and tomorrow")
'wwwdd  nnoooaatttmmmrrrrrrooo  $ooo'
```

This makes $BWT(T)$ more compressible. For example, we could take $BWT(T)$ and shrink it (reversibly) with *run-length encoding* (RLE). Software tools for compression and decompression employ various methods to shrink $BWT(T)$, including move-to-front transforms, run-length encoding, Huffman coding, and arithmetic coding. The popular `bzip2` compression tool [3] uses these and other methods.

1.4 Reversing the Burrows-Wheeler Transform with the LF Mapping

We said the BWT is reversible, but it's far from obvious at first glance how to reverse it. Recall that $BWT(abaaba\$) = abba\$aa$. It seems at first glance that information about which a in $BWT(T)$ corresponds to which a in T has been lost.

But the BWT has an important property called the *LF Mapping*. Consider again the BWM for $T = abaaba\$$:

$$\begin{array}{ccccccc}
\$ & a & b & a & a & b & a \\
a & \$ & a & b & a & a & b \\
a & a & b & a & \$ & a & b \\
a & b & a & \$ & a & b & a \\
a & b & a & a & b & a & \$ \\
b & a & \$ & a & b & a & a \\
b & a & a & b & a & \$ & a
\end{array}$$

We re-write T, this time giving each character (except \$) a subscript: $T = a_0b_0a_1a_2b_1a_3\$$. The subscript equals the number of times that character occurs previously in T. The first occurrence of a becomes a_0, the second occurrnce of c becomes c_1, etc. We call the subscript the character's "rank." We don't rank \$ since there's only one.

Now we re-write the BWM including the ranks. Ranks don't affect lexicographical order. E.g. a_0 and a_{999} are equal as far as the ordering of the rows of $BWM(T)$ is concerned.

$$\begin{array}{ccccccc}
\$ & a_0 & b_0 & a_1 & a_2 & b_1 & a_3 \\
a_3 & \$ & a_0 & b_0 & a_1 & a_2 & b_1 \\
a_1 & a_2 & b_1 & a_3 & \$ & a_0 & b_0 \\
a_2 & b_1 & a_3 & \$ & a_0 & b_0 & a_1 \\
a_0 & b_0 & a_1 & a_2 & b_1 & a_3 & \$ \\
b_1 & a_3 & \$ & a_0 & b_0 & a_1 & a_2 \\
b_0 & a_1 & a_2 & b_1 & a_3 & \$ & a_0
\end{array}$$

The LF Mapping states: the i^{th} occurrence of a character c in the last column has the same rank as the i^{th} occurrence of c in the first column. E.g. look at the as in the last column above, starting at the top. They have ranks 3, 1, 2 and 0 in that order. Now look at the ranks of the a in the first column above; they have the same order: 3, 1, 2, 0. Same for the bs.

Why does this property hold? Let M be $BWM(T)$ for some T. Let M' be the matrix obtained by rotating all the rows of M to the right by one position. The first column of M' equals the last column of M.

Pick any character c and compare the ranks of the cs in the first column of M and the ranks of the cs in the first column of M'. The ranks appear in the same order because the sort treats those rows identically; in M the rows are sorted starting at the first position, and in M' the rows are tied with respect to their first position and sorted starting at the second position. Because of this, and because the first column of M' equals the last column of M, the LF Mapping property follows.

Now let's see how to reverse the BWT. First, let's re-rank the characters. Before we ranked them according to how many times the same character occurred previously in T. Call this the *T-ranking*. Now we re-rank according to how many times the same character occurred previously in $BWT(T)$. Call this the *B-ranking*. Here is $BWT(T)$ with B-ranks: $a_0 b_0 b_1 a_1 \$ a_2 a_3$.

How are ranks represented? Let's define an array *rank* parallel to $BWT(T)$ that stores the rank of each character. Here is an illustation of the first and last columns of $BWM(T)$, along with *rank*.

F	L	*rank*
\$	a	0
a	b	0
a	b	1
a	a	1
a	\$	0
b	a	2
b	a	3

Informally, to recover T from $BWT(T)$: start in the first row, which must have \$ in the first column. Rows are rotations of T, so the *last* column of the first row contains the character to the left of \$ in T: a in this case. We know from the *rank* array that this is the a of rank 0: a_0. Now how to recover the character to the left of a_0? We can do this if we know which row begins with a_0. But the LF Mapping tells us this. Because a_0 has rank 0, it must correspond to the first a in the first column, i.e. the a in the second row. So we advance to the second row. Now we look in the last column and *rank* array and see that b_0 precedes a_0. b_0 corresponds to the first b in the first column, so we advance to the sixth row. We proceed in this way until we reach the row with \$ in the last column. In this example, we would visit the rows in this order, assuming the first row has index 0: (0, 1, 5, 3, 2, 6, 4), and we successfully recreate the original string from right to left: $a_3 b_1 a_1 a_2 b_0 a_0 \$$.

A Python implementation is below. For now, we assume it's reasonable to pre-calculate the *rank* array for T. If T is very long, this is not reasonable, since *rank* will (usually) take much more memory than $BWT(T)$. Methods like `bzip2` compensate by compressing the text in relatively small *blocks* and decompressing a block at a time.

```
def rankBwt(bw):
    """ Given BWT string bw, returns a parallel list of B-ranks.  Also
        returns tots, a mapping from characters to # times the
        character appears in BWT. """
    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 a map from characters to the range of cells in the first
        column containing 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
    t = "$"
    while bw[rowi] != '$':
        c = bw[rowi]
        t = c + t
        rowi = first[c][0] + ranks[rowi]
    return t
```

Quick example of this implementation in action:

```
>>> bw = bwt("Tomorrow_and_tomorrow_and_tomorrow")
>>> bw
'w$wwdd__nnoooaattTmmmrrrrrrooo__ooo'
>>> reverseBwt(bw)
'Tomorrow_and_tomorrow_and_tomorrow$'
```

2 The FM Index

In 2000, six years after the BWT was published, Paolo Ferragina and Giovanni Manzini published a paper [2] describing how the BWT, together with some small auxilliary data structures, can be used as a space-efficient index of T. It generally uses less than half the space required to store m

integers. They named it the *FM Index*. Just as the LF Mapping was the key to understanding how the BWT is reversible, it's also the key to how it can be used as an index.

Let's start by considering just the first column (F) and last column (L) of the BWM, as well as the *rank* array.

F	L	*rank*
\$	a	0
a	b	0
a	b	1
a	a	1
a	\$	0
b	a	2
b	a	3

We will refine this to obtain the FM Index.

3 Searching

Say we are searching for occurrences of a string $P = aba$. Like the suffix array, the BWM is sorted. This implies that any rows having P as a prefix will be consecutive.

We proceed first by finding the rows beginning with the shortest proper suffix of P: a in this case. The first column is part of our index, so this is trivial. These are the rows in the 0-based range [1, 5). Let's visualize this in the context of the whole matrix:

F						L	*rank*
\$	a	b	a	a	b	a	0
a	\$	a	b	a	a	b	0
a	a	b	a	\$	a	b	1
a	b	a	\$	a	b	a	1
a	b	a	a	b	a	\$	0
b	a	\$	a	b	a	a	2
b	a	a	b	a	\$	a	3

Remember: even though we just drew the entire matrix, our index so far contains just F, L and *rank*.

Now we must find all rows beginning with the next-longest proper suffix of P: ba. Observe the shaded characters in the L above. We see two bs, indicating there are two instances where a is preceeded by b. Also, the LF Mapping and *rank* array tell us which rows have ba as a prefix: the rows beginning with b_0 and b_1; i.e. the first two rows in the "b section".

F						L	$rank$
\$	a	b	a	a	b	a	0
a	\$	a	b	a	a	b	0
a	a	b	a	\$	a	b	1
a	b	a	\$	a	b	a	1
a	b	a	a	b	a	\$	0
b	a	\$	a	b	a	a	2
b	a	a	b	a	\$	a	3

Now we find rows beginning with the final suffix, aba. Again we look at the shaded characters in the last column. We see that the occurrences of ba are preceded by a_2 and a_3, giving us the range of rows prefixed by P:

F						L	$rank$
\$	a	b	a	a	b	a	0
a	\$	a	b	a	a	b	0
a	a	b	a	\$	a	b	1
a	b	a	\$	a	b	a	1
a	b	a	a	b	a	\$	0
b	a	\$	a	b	a	a	2
b	a	a	b	a	\$	a	3

This is called *backwards matching*. In short, we apply the LF Mapping repeatedly to find the range of rows prefixed by successively longer proper suffixes of P until the range becomes empty, in which case P does not occur in T, or until we run out of suffixes. If we run out of suffixes, the size of the range equals the number of times P occurs in T.

Here is a Python implementation:

```
def countMatches(bw, p):
    """ Given BWT(T) and a pattern string p, return the number of times
        p occurs in T. """
    ranks, tots = rankBwt(bw)
    first = firstCol(tots)
    l, r = first[p[-1]]
    i = len(p)-2
    while i >= 0 and r > l:
        c = p[i]
        # scan from left, looking for occurrences of c
        j = l
        while j < r:
            if bw[j] == c:
                l = first[c][0] + ranks[j]
                break
            j += 1
        if j == r:
            l = r
            break # no occurrences -> no match
        r -= 1
        while bw[r] != c:
            r -= 1
        r = first[c][0] + ranks[r] + 1
        i -= 1
    return r - l
```

And some example output:

```
>>> bw = bwt("Tomorrow_and_tomorrow_and_tomorrow")
>>> bw
'w$wwdd__nnoooaattTmmmrrrrrrooo__ooo'
>>> countMatches(bw, "tomorrow")
2
>>> countMatches(bw, "Tomorrow")
1
>>> countMatches(bw, "tomorrow")
2
>>> countMatches(bw, "omorrow")
3
>>> countMatches(bw, "and")
2
>>> countMatches(bw, "r")
6
>>> countMatches(bw, "o")
9
>>> countMatches(bw, "xyz")
0
```

A drawback is that, in each step, we are *scanning* a range of elements in L. This is $O(m)$ (where $m = |T|$).

We can make this $O(1)$ by augmenting the rank array in the following way. Instead of a $m \times 1$ *rank* array, we store a $m \times |\Sigma|$ *rankAll* matrix. In each row of *rankAll*, we store an integer for each character of the alphabet equal to the number of times that character has been observed up to and including that position in $BWT(T)$. For example:

			rankAll	
F	L	\$	a	b
\$	a	0	1	0
a	b	0	1	1
a	b	0	1	2
a	a	0	2	2
a	\$	1	2	2
b	a	1	3	2
b	a	1	4	2

Now, instead of scaning the last column, we can simply look up the appropriate character of *rankAll* at the left and right extremes of our range. If there is no difference between the two lookups, then the character does not occur. If there are one or more occurrences of the character, the lookups will give the ranks of those occurrences.

Here is a Python implementation of this refinement:

```
def rankAllBwt(bw):
    """ Given BWT string bw, returns a map of lists.  Keys are
        characters and lists are cumulative # of occurrences up to and
        including the row. """
    tots = {}
    rankAll = {}
    for c in bw:
        if c not in tots:
            tots[c] = 0
            rankAll[c] = []
    for c in bw:
        tots[c] += 1
        for c in tots.iterkeys():
            rankAll[c].append(tots[c])
    return rankAll, tots

def countMatches2(bw, p):
    """ Given BWT(T) and a pattern string p, return the number of times
        p occurs in T. """
    rankAll, tots = rankAllBwt(bw)
    first = firstCol(tots)
    if p[-1] not in first:
        return 0 # character doesn't occur in T
    l, r = first[p[-1]]
    i = len(p)-2
    while i >= 0 and r > l:
        c = p[i]
        l = first[c][0] + rankAll[c][l-1]
        r = first[c][0] + rankAll[c][r-1]
        i -= 1
    return r - l # return size of final range
```

4 Rank checkpoints

So far we have assumed that it's reasonable to pre-calculate and store the *ranks* array or *rankAll* matrix. But this is probably not reasonable for large T. Dividing T into blocks, as is done in compression, is not a good option since our goal is to build an index over all of T.

Instead we discard most but not all of the rows from the *rankAll* matrix. For instance, we might keep one in every 32 rows and discard the rest. We call the retained rows *rank checkpoints*. Now each time we would like to look up $rankAll[c][i]$, we are in one of two cases: (1) row i was not discarded, in which case we do the lookup as usual, or (2) row i was discarded, in which case we scan characters in L starting at i and moving forward (or backward) until we reach the next rank checkpoint. If we count x occurrences of c on our way to a checkpoint at row i', then $rankAll[c][i'] - x$ gives the same value we would have found in $rankAll[c][i] - x$.

If checkpoints are spaced at most C rows from each other, where C is a small constant, then this operation is $O(1)$. In which case, backward search for a pattern P of length n is $O(n)$ time.

5 Looking up offsets

So far we have discussed how to use the FM Index to determine whether and how many times a substring P occurs within T, but we have not discussed how to find *where* P occurrs, i.e. the substring's offset into T.

If our index included the suffix array $SA(T)$, we could simply look this up in $SA(T)$. For example, here was the range we ended up with after searching for $P = aba$ within $BWT(abaaba\$)$:

F						L	$SA(T)$
\$	a	b	a	a	b	a	6
a	\$	a	b	a	a	b	5
a	a	b	a	\$	a	b	2
a	b	a	\$	a	b	a	3
a	b	a	a	b	a	\$	0
b	a	\$	a	b	a	a	4
b	a	a	b	a	\$	a	1

$SA(T)$ tells us these matches occurred at T offsets 0 and 3.

However, the suffix array takes m integers, and we would like to use less space than that. We again use the idea of discarding most of the entries and re-creating them as needed: Instead of storing every entry of $SA(T)$, we store, say, every 4^{th}. If we want to look up $SA[3]$ but find it has been discarded (discarded rows given "-" below):

F						L	$SA(T)$
\$	a	b	a	a	b	a	6
a	\$	a	b	a	a	b	–
a	a	b	a	\$	a	b	–
a	b	a	\$	a	b	a	–
a	b	a	a	b	a	\$	0
b	a	\$	a	b	a	a	–
b	a	a	b	a	\$	a	–

We can use the LF Mapping to step one position to the left in the text:

F						L	$SA(T)$
\$	a	b	a	a	b	a	6
a	\$	a	b	a	a	b	–
a	a	b	a	\$	a	b	–
a	b	a	\$	a	b	a	–
a	b	a	a	b	a	\$	0
b	a	\$	a	b	a	a	–
b	a	a	b	a	\$	a	–

We're still in a discarded row, so we keep stepping until we reach a retained row. This happens after two more steps:

F						L	$SA(T)$
\$	a	b	a	a	b	a	6
a	\$	a	b	a	a	b	–
a	a	b	a	\$	a	b	–
a	b	a	\$	a	b	a	–
a	b	a	a	b	a	\$	0
b	a	\$	a	b	a	a	–
b	a	a	b	a	\$	a	–

$SA(T)$ at this row equals 0 and it took us 3 steps to get here, so we conclude the row we started from had offset 3.

Note that if we retain an entry of $SA(T)$ every C positions of T, where C is a small constant, then looking up the text offset associated with a row of the BWM is $O(1)$.

References

[1] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. 1994.

[2] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[3] Julian Seward. bzip2 and libbzip2, version 1.0. 5: A program and library for data compression. *URL http://www. bzip. org*, 2007.