BWT: forward and backward transforms

Burrows-Wheeler transform (BWT) is one of the most important developments in string algorithms during the past twenty years. Our presentation here is based on the paper: “Opportunistic Data Structures with Applications”, Paolo Ferragina and Giovanni Manziniy, FOCS 2000.

Given a text, consider all rotations (regardless left or right rotate) and sort these rotations. Example, $S = \text{mississippi}$. Commonly used trick: add a $ to end. Dollar sign is the single smallest (i.e. alphabetically smaller than any other symbol). These are the rotations: mississippi$, ississippi$m, ssissippi$mi, sissippi$mis,issippi$miss,isisippsissi,ippi$mississ, ppi$mississi,pi$mississip,i$mississipp,$mississippi.

Now question: for any $S$, which rotation is the smallest? Always $S$! But not sure which one is largest (not necessarily $S$). Sort:

1: $\text{mississippi}$
2: $i\text{mississipp}$
3: $ippi\text{mississ}$
4: $issippi\text{miss}$
5: $ississippi\text{m}$
6: $\text{mississippi}$
7: $pi\text{mississip}$
8: $ppi\text{mississi}$
9: $sippi\text{missis}$
10: $\text{mississippi}$
11: $sippi\text{missi}$
12: $\text{mississippi}$

The forward BWT is, given text $S$, generate the last column $L$. Here, we can not explicitly create all rotations, which leads to $O(n^2)$ time. Simple observation: rotations in BWT correspond exactly to the order of sorted suffixes. That is, take the portion from left till the $ of each rotation, that is exactly the suffixes. Thus, we simply construct suffix array SA. Say $SA[i] = k$, then that letter in $L$ is $S[k-1]$.

Now, why bother performing BWT? Short answer: BTW allows easier compression (with simple techniques like run-length encoding, BTF, etc). BWT is currently one of most powerful compression (and also fast).

Now we introduce the backward BWT. Our first observation is, Every column is a permutation of original $S$. BWT: output the last column $L$. In the above example, it would output $L = \text{ipssm}$pissii. We call first column $F$ (a permutation of $L$): $\text{iiiimppssss}$. Note $F$ is sorted already. An interesting observation is: given $L$, can reconstruct the original $S$. This is not obvious: $L$ usually looks quite different from $S$.

Now suppose we know $L$. Obviously, we can construct $F$ by sorting $L$. Here is the first question: can you find the last char of $S$ from the given $L$? The answer is: $L[1]$. Make sure you understand why this is the case. But how to find other parts of $S$? Let us look at what is the second last of $S$. Observation: suppose there is one row $i$, where we know $F[i]$ and $L[i]$. Then in some rotation, we will have $L[i]$ and $F[i]$ together like: $\ldots L[i]F[i]\ldots$. The next observation is critical to BWT. Let $F[i] = c$. Consider all rotations in the rotation matrix $M$ that ending with $c$. For example, consider $F[1] = i$. There are four rotations ending with $i$. Now imagine we right-rotate each of these four rotations by 1. Then they are now starting with $i$ (and still in $M$). The key is: their relative ranking within is unchanged! Why? Suppose there are two rotations ending with $i$ and the prefix parts are $S_1$ and $S_2$ respectively. That is, these rotation are $S_1i$ and $S_2i$. Suppose $S_1 < S_2$ (and thus the first rotation proceeds the second in $M$).
Now after right-rotation-by-1, the two rotations become: \( iS_1 \) and \( iS_2 \), where we still have \( iS_1 < iS_2 \).

Now return to the original question: how to continue after knowing the last of \( S \) is \( L[1] = i \)? There are 4 rotations ending with \( i \), with \( L[1] \) being the first. So there are four rotations starting with \( i \) (after right-rotation by 1), and the order is still the same. That is, \( iS \) is the right rotation (by 1) of original \( S \). So, the second to last character is \( L[2] = p \). Continue: there are two rotations ending with \( p \) (with \( M[2] \) being the first). So the right-rotation-1 is the first row in \( M \) starting with \( p \) (i.e. \( M[7] \)).

So The third-to-last is \( L[7] = p \). We can continue and obtain the entire \( S \) from \( L \).

Now the detailed algorithm for backward BTW.

1. Compute the array \( C[1, K] \) where \( k \) is the size of alphabet. \( C[c] \) is equal to the number of occurrences of characters \( $, 1, 2, \ldots, c-1 \) in the text \( S \). Notice that \( C[c] + 1 \) is the position of the first occurrence of \( c \) in \( F \) (if any). That is, \( C[c] \) is equal to the number of times all characters smaller than \( c \) in \( S \).

2. Define the LF-mapping \( LF[1 \ldots n + 1] \) as follows \( LF[i] = C[L[i]] + r_i \), where \( r_i \) equals the number of occurrences of character \( L[i] \) in the prefix \( L[1; i] \).

3. Reconstruct the original text \( S \) backward as follows: set \( s = 1 \) and \( S[n] = L[1] \) then, for each \( i = n - 1 \) downto 1, do \( s = LF[s] \) and \( T[i] = L[s] \). Here, \( s \) is the index of current position into \( L \) array; \( i \): which one position to recover in \( T \).

Example: how this algorithm runs for \( S=\text{mississippi$} \). Step 1 gives \( C[$, i, m, p, s] = [0, 1, 5, 6, 8] \).
