1 Pattern matching in BWT

Frequently need to search for pattern: i.e. does \( P \) occur in \( T \)? Suppose \( T \) is compressed. Trivially we can decompress \( T \) and then use ST/SA etc. But this needs compress/decompress many times and it can be slow to do so. A natural question is that performing pattern match in *compressed* text. Here, we ignore what is done for compression of the BWT transformed text and only look at BWT itself. That is, can we search pattern in BWT transformed text (without doing backward BWT)?

Recall two things:

1. BWT and SA are closely related (\( M \) array corresponds well to SA order). All the suffixes of the text \( T[1\ldots n] \) prefixed by a pattern \( P[1\ldots p] \) occupy a contiguous portion (subarray) of suffix array.

2. That subarray has starting position \( s_p \) and ending position \( e_p \), where \( s_p \) is actually the lexicographic position of the string \( P \) among the ordered sequence of text suffixes.

Our first step is counting how many times \( P \) occurs in \( T \). That is, we will find the segment of suffix array that matching \( P \). Example: let \( P = ssi \), and the matches form a segment in BWT’s \( M \). Start with a trivial case: \(|P| = 1\). How many times it \((P[1])\) occurs in \( T \)? This is easy: \( C[P[1]] + 1 \ldots C[P[1]] + 1 \) in \( F \). Now how about the length of pattern is more than 1 (say 2, \( P[1..2] \))? Use \( s_p = C[P[1]] + 1 \) as the lower bound, \( e_p = C[P[1]] + 1 \) is the upper bound when only looking at \( P[1] \). Can still find how many contain \( P[2] \) (i.e. we know how often they occur in \( T \)). E.g. \( P = si \), \( T = mississippi \), there are four \( i \), but how often \( si \) occurs? Key: we only consider cases where \( i \) is at \( F \) while \( s \) is at \( L \) (why? can always rotate the string to that position for each occurrence of \( si \)). That is, need to find which have \( s \) at the last column, while have \( i \) at the first column. First, \( si \) must occur consecutively in \( M \). So again we need to compute a range (lower and upper bounds) of where \( si \) occurs. Also, this region must occur within the region starting with \( s \).

Define: \( Occur(c,1,i) \) is the number of occurrences of character \( c \) occurs in the first \( i \) positions of the last column \( L \). Then the number of occurrence of \( si = Occur(s,1,e_p) - Occur(s,1,s_p-1) \).

Now where does pattern \( si \) start in \( M \)? That is, after rotation, \( s \) is now back to the first column (i.e. \( si \) are together), then \( si \) will occupy a segment of consecutive positions. But where are the portion? It starts right at \( C[s] + 1 + Occur(s,1,s_p-1) \)! Why? The same observation: after rotation, all rows ending with \( s \) will now group together. But their order appearing in the rotated \( M \) is still same! Since there are \( Occur(s,1,s_p-1) \) appearance of \( c \) that does not begin with \( i \), then the next one should begin with \( i \). Now we know the region of pattern must end with \( C[s] + 1 + Occur(s,1,s_p-1) - Occur(s,1,e_p) \). Example: \( P = si \), for \( P[2]=i \), \( s_p = 2 \), \( e_p = 5 \). For \( P[1] = s \), \( Occur(s,1,1) = 0 \), \( Occur(s,1,5) = 2 \), \( C[s] = 8 \). So \( s_p = 8 + 0 + 1 = 9 \), \( e_p = 8 + 2 = 10 \).

How to continue if \( P \) has three or more chars? Repeat the above procedure by working from the last char first and move backwards. This works because for each suffix of \( T \), we ensure the range is properly computed for that suffix. Then the above procedure then considers one more char in front of the suffix; our previous argument ensures the new suffix longer by 1 must also be properly computed. Example: \( P = sssi \). For \( P[2..3], s_p = 9 \), \( e_p = 10 \), \( Occur(s,1,8) = 2 \), \( Occur(s,1,10) = 4 \). So \( s_p = 8 + 2 + 1 = 11 \) (recall \( C[s]=8 \)), \( e_p = 8 + 4 = 12 \). How about \( P = pssi \)? \( Occur(p,1,10) = 2 \), \( Occur(p,1,12) = 2 \), \( C[p] = 6 \). Then \( s_p = 6 + 2 + 1 = 9 \), \( e_p = 6 + 2 = 8 \). This does not make sense: \( s_p \) must be at least as large as \( e_p \) if \( P \) occurs in \( T \). When this happens, this means there is no occurrence of \( P = pssi \).

Now complete algorithm (correctness omitted):
1. $c = P[k], i = k$, where $k$ is the length of $P$

2. $s_p = C[c] + 1, e_p = C[c + 1]$

3. while $((s_p \leq e_p) \text{ and } (i \geq 2))$ do

4. \quad $c = P[i - 1]$;

5. \quad $s_p = C[c] + Occ(c, 1, s_p - 1) + 1$

6. \quad $e_p = C[c] + Occ(c, 1, e_p)$

7. \quad $i \leftarrow i - 1$

8. endwhile

9. if $(e_p < s_p)$ then output “not found” else return “found $(e_p - s_p + 1)$ occurrences”.

Running time: for pattern with $m$ chars, it considers each of $m$ chars one time. Assume $Occ(c, 1, i)$ takes constant time. Then the whole algorithm runs in $O(n)$. That is, running time is proportional to the length of pattern.

Now can we obtain $Occ(c, 1, i)$ in constant time? The idea is preprocessing $T$: can not directly count each time. Naively, store $Occ[c, i]$ array for each $c$ and $i$. But this is too space-intensive (taking $O(\text{alphabet size} \ast n)$ space). With some clever tricks, this can be reduced to almost $O(n)$. The basic idea is: divide $T$ into pieces; for $k$-th piece, store $Occ_k[c, i]$ as the number of occurrence of $c$ in the first $i$ chars of $k$-th piece; also maintain $Occ_1[k, c]$ as the number of occurrence of char $c$ in the first $k$ pieces (this avoids going through each of the many pieces). Now on computing for a given position $i$: say each piece is of $L$ letter long, first find which segment it ends: $k = [i/L]$. Then: $Occ_1[k, c] + Occ_k[c, i - kL]$ is the number of occurrences of $c$ from position 1 to $i$, which can be obtained in constant time. We choose $L = \log(n)$. The time to build $Occ$ and $Occ_1$ is: $O(n/L \ast L + n/L \ast L) = O(n)$. Space: there are $n/L$ pieces, each piece needs space of $O(L \ast \log L)$ (because the values cannot be larger than $L$). $Occ_1$ takes $n/L \ast |c| = |c| \ast n/\log n$. Detailed analysis omitted.

Now how to determine the true locations (which suffixes) for these $e_p - s_p + 1$ occurrences? Remember what we have is the range of rows in BWT. Say between 5 to 10 rows in BWT matrix $M$. But these are not the true positions in text. If we really need (often we do), we must convert these row indices to the original position in $T$. Note if we have suffix array at hand, we are done: recall the suffix array order is exactly the same as $M$. But this is not feasible: we want to work on the BWT space instead of suffix array. The idea is to store which suffix corresponds to which row to a subset of BWT rows. That is, if we want to figure out which suffix for one of these rows whose true position is already pre-computed, we immediately know. But what about those not stored (say a BWT row $k$ without pre-computed true position)? Idea: look to find a nearby pre-stored row by using LF-mapping to find rotated row by growing the pattern (i.e. $T[s - 1..n]$ for position $s$). Note: each BWT row corresponds to a particular rotation of $T$. The basic idea is: by applying one LF-mapping, you find the position of the suffix starting point+1 (with one more char in front). Do this until you meet a pre-stored one (this works if there is sufficiently large number of suffix stored).