Lecture 13: Compressed suffix array and a quick tour of two-dimensional string matching

1 Compressed suffix array (CSA)

Recall that the BWT rotation matrix corresponds to the suffix array. Plain storage of suffix array needs \( n \log_2(n) \) bits. So this motivates compressed suffix array which uses \( O(n) \) space if alphabet size is constant (like DNA). The following is from the paper “A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays” by Grossi and Vitter (2000). Recall \( SA(i) \) is the suffix index of \( i \)-th smallest suffix. Now we define its inverse: \( SA^{-1}(i) = \text{rank of suffix } i \). So, \( SA^{-1}(SA(i)) = i \). Also, \( SA(SA^{-1}(i)) = i \). Now the compressed suffix array. Definition: \( \phi(i) = SA^{-1}(SA(i) + 1) \), for \( i = 1..n - 1 \) and \( \phi(0) = SA^{-1}(0) \) for \( i = 0 \). Intuition: \( \phi(i) \) gives the rank of the next suffix after \( SA(i) \). Now, \( \phi(0) \) gives the rank of suffix 0. What is rank of suffix 1? By def of \( \phi \), it is: \( \phi(\text{suffix rank of suffix 0}) = \phi(0) \). And this can be applied again: rank of suffix 2 is \( \phi(\phi(\phi(0))) \) and so on. The reason of defining \( \phi \) is that the property of \( \phi \) allows \( O(n) \) storage. A useful property of \( \phi \) is that it has \( \Sigma \) strictly increasing (continuous) subsequences (where \( \Sigma \) is the alphabet size). Why? Claim: for \( i < j \), if \( T[SA(i)] = T[SA(j)] \), then \( \phi(i) < \phi(j) \). By definition of suffix array: \( i < j \) iff \( Suff[SA(i)] < Suff[SA(j)] \). Since \( T[SA(i)] = T[SA(j)] \), then \( Suff[SA(i)] + 1 < Suff[SA(j)] + 1 \). This is exactly what is meant by suffix ranked \( \phi(i) \) and \( \phi(j) \): \( \phi(i) < \phi(j) \).

This implies \( \phi \) values have contiguous segments (which has suffix starting with same letter since \( SA \) is well segmented). This allows more memory efficient storage of \( \phi \) as follows. For each segment of \( \phi \) with the same starting suffix symbol (which increases monotonically), we divide into two parts: the \( \log(#(c)) \)-leading bits (\( #(c) \) is the number of symbol c in text), and the rest is at most \( \log(n/#(c)) + 1 \) bits. The second part is stored as plain binary form but the first \( \log(#(c)) \) is stored more efficiently as follows: for the beginning of this segment, \( q_0(c) \) is stored plain binary (\( \alpha(c) \) is the number of symbol with starting symbol smaller than c), and each following \( q_i \) within this segment is stored as the difference between \( q_{i+1} \) and \( q_i \) using unary code (e.g. if the value is 10, then code as 10 zeros followed by a 1). Thus the total size of memory needed is \( 2\#(c) \) for this segment. Thus, the total space needed is at most \( \sum_c #(c) * (\log(n/#(c)) + 3) \). To simplify notation, define \( nH_0 \) as \( \sum_c #(c) \log(n/#(c)) \). So the total space needed is at most \( n(H_0 + 4) \). So, conclusion is: \( \phi \) array can be stored using \( O(n(H_0 + 1)) \) bits. How about retrieving these values in the compressed form? Assume one will extract values sequentially of \( \phi \), we can take care of the difference of the \( q_i \) values and that allows constant time per value. How about random access (i.e. want to find a particular \( \phi(i) \))? Well if you can afford some additional memory to use some additional data structure, then constant time is possible for random access (omitted). The additional data structure also is in the form of \( O(n(H_0 + 1)) \).

Now, we know if we are given a compressed suffix array, we can store it efficiently using less memory. Now another question is, how to construct CSA in the first place? A simple approach is: first construct \( SA \) and then convert to CSA. This is easy by the previous recursive relation which allows easy construction of CSA from \( SA \) (and \( SA^{-1} \)). But this still not optimal: you still need to construct \( SA \) at some point (even you only do this once). There is an algorithm allowing construction of CSA using \( O(n(H_0 + 1)) \) memory (but runs in \( O(n \log n) \) time, not linear time), which does not construct suffix array. We will not discuss this algorithm.

2 Two dimensional string matching

Given 2-dimensional array \( T[1..m_1,1..m_2] \) and a pattern \( P[1..n_1,1..n_2] \). We want to find whether \( P \) occurs in \( T \) or not (as an array). This is a natural extension of 1-dimensional case and many algorithmic
ideas from strings can apply here. Brute-force: try all $O(m_1m_2)$ positions in $T$ of possible occurrence of $P$; and test whether $P$ matches. This takes $O(m_1m_2n_1n_2)$.

One basic idea is to convert 2D to 1D (linear) in the following way. Convert $P$ to a 1D array $P'[1..n_1]$ where each $P'[i]$ is the $i$th row of $P$ (i.e. $P[i,1]..P[i,n_2]$). Similarly convert $T$ into $m_1 - n_1 + 1$ 1D array $T'(i)[1..m_1]$, where starting from line $i$, take the following $n_1$ rows, i.e. $T'(i)[j] = T[i,j]...T[i,j + n_2 - 1]$. Clearly if $P'$ (as a string) occurs in one of these $T'[i]$, we find a match. The question is how to do the matching this way.

The first algorithm is called the Zhu-Takahata algorithm. They essentially use Karp-Rabin to test occurrence. For $i$-th column of $P$, hash it (i.e. $P[1..m_1,i]$ to an integer). We also hash $T$ (the first $n_1$ rows); that is: $T[1..m_1,i]$ to an integer for each column of $T$. Note there are $n_2$ numbers for $P$ and $m_2$ for text. How to check whether $P$ may occur in $T$ starting this row? Simple: check whether the pattern created by the hashed key of $P$ occur exactly in $T$. This can be done by classic string matching algorithms such KMP (note: alphabet size can be large; but KMP works regardless of alphabet size). Once a hit is found, do brute-force comparison to verify; after this, move to the next row by rehashing as done in Karp-Rabin. Time: depends on how often we get the hash hits. Assume worst case: it is still possible to have each hit position with hash key hit and we still need to check each one. But in practice we expect this works much better.

The second algorithm is the Bird-Baker algorithm. Return to the basic idea: there are $O(m_1)$ for $T'$, and each with $m_2$ (expanded) characters. We test whether a pattern $P$ (with $n_2$ characters) occurs in this $T'$: using KMP takes $O(n+m)$ time and thus total time is: $O(m(n + m) = O(m^2)$. There is a problem, however, is how to compare expanded characters. If we assume comparing two expanded char taking $O(n)$ time, then total time will be $O(m^2n)$. This is better than $O(n^2m^2)$. Now we make one more improvement. We now set $T_i[j] = k$ if $k$ is the smallest number such that the $k$th row of $P$ matches the subrow starting at $T[i,j]$. Similarly we set $P'[i] = k$ if $k$ is the smallest number such that the $i$th row of $P$ equals the $k$th row of $P$. Now suppose $T'$ and $P'$ are known, can they allow constant time compression of two expanded chars? This is simple: given two expanded char at $T[i,j]$ and $P[k]$. Check whether $T_i[j] = P'[k]$: if yes match (that position $T[i,j]$ to right by $m$ chars matches the smallest $k'$-th pattern; and if $P'[k] = k'$, it means pattern $P_k$ is the same as pattern $P_k'$ and we know $P_k$ matches $P_k'$); otherwise there is no match. We omit the details on how $T_i$ and $P'$ are constructed.