Implementation of Burrows-Wheeler Transform

Abhianshu Singla*

Department of Computer Science,

University of Central Florida

Abstract

This report shows the implementation of Suffix Array, Longest Common Prefix and Burrows Wheeler Transform using Wavelet Trees. In this report, the backward search algorithm is used for searching the short sequences. In addition to the exact matching search, I also implemented the inexact matching search where gaps and mismatches are allowed. Evaluation is done on very small length strings to check the accuracy of the program by looking at the string and also on medium-length string to do the performance evaluation.

Keywords: Suffix Array, Wavelet Trees, Backward Search, Burrows Wheeler Transform

1 Introduction

1.1 Suffix Array

The suffix array SA for string s of size m is a lexicographically sorted array of all suffixes of s where SA[i] = j iff the suffix of s starting at position j is the ith lexicographically smallest suffix of s. Also, let \$ be the last character of string s where \$ is not in the dictionary of the string to ensure that no suffix of S matches a prefix of another suffix of S. For the construction of Suffix Array, I used the Bucket Sorting Algorithm proposed by Manber and Myers. This algorithm is based on bucket sorting the suffixes of string s by progressively increasing the size prefixes of these suffixes. We observed the following properties while doing the bucket sorting:

1. Each suffix in the bucket has the same H-length prefix in stage H
2. The buckets are in lexicographical order, i.e., the suffix in bucket b1 is lexicographically smaller than suffixes in bucket b2 where b1 $<$ b2

Figure 1 depicts the 4 stages for string "PANAMABANANAS" with their respective buckets. Initially, the suffixes are sorted based on their first character. The bucket id and the the start index of each bucket are also maintained in each stage. Looking at stage 1, buckets has bucket id as $[1, 2, 3, 4, 5, 6, 7]$ and bucket start index as $[1, 2, 8, 9, 10, 13, 14]$. Let SA_H denote the partial suffix array for string s based on sorting the suffixes using their first H characters. In SA_H, if there are identical consecutive strings, they belong to the same bucket. The partial suffix array SA_{2H} is constructed from SA_H by considering the suffixes starting at $SA_H[1]$, $SA_H[2]$, $SA_H[3]$ and so on using them to decide how to sort buckets for the 2H stage. The start index for a bucket b in SA_{2H} is initially the same as its index in SA_H. The algorithm is as follows:

procedure BUCKETSORT
  $i = 1$
  while $i <= m$ **do**
    $j = SA_H[i]$
    if $A_j \in SingletonBucket$ **then** $SA_{2H}[i] = j$
    if $(j - H) > 0$ **then**
      Identify bucket b that A_{j-H} belongs to in SA_H
      Let k be the start index for b
      $p = k + counter(b)$
      $SA_{2H}[p] = j - H$
      $counter(b) + +$

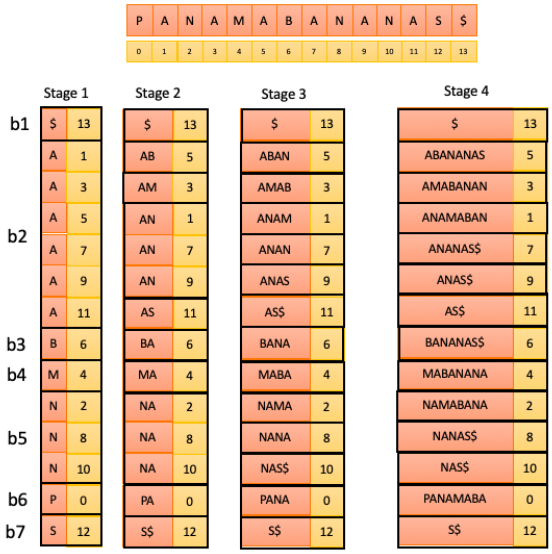


Figure 1: *Suffix Array Construction using Bucket Sort Algorithm*

Inverse of Suffix Array is denoted by R such that $R[SA[j]] = j$ and $SA[R[j]] = j$. Also, it has the same size as Suffix Array. $R[i]$ is the index in SA of the suffix starting at location i in S to facilitate constant time look up of the location in SA of a suffix. Suffix Array and Inverse of Suffix Array for String s = "PANAMABANANAS"is shown in Figure 2.

1.2 Longest Common Prefix

Longest Common Prefix array stores the length of the longest common prefixes between consecutive pair of suffixes in SA, i.e., $SA[i]$ and $SA[i + 1]$. LCP array has a length of m-1 where m is the size of the string s. To construct LCP array, we need m interations and each iteration computes the lcp between suffix $S[i, m]$ and its right neighbor in the suffix array $S[SA[R[i]+1], m])$. Let l be the lcp of S[i,m] and S[j,m]. If $l > 0$, then lcp for suffix beginning at $s[i + 1]$ will be at-least l-1 because the relative order of characters remain same. If we delete the first character from both suffixes, then at least l-1 characters will match. LCP array is also shown in Figure 2. The algorithm of construction LCP array from Suffix Array, inverse Suffix Array and String s is as follows:

procedure LONGESTCOMMONPREFIX
  $LCP[R[0]] = charComp(s[0 : m], s[SA[R[0]+1] : m])$
  $l = LCP[R[0]]$
  i = 1
  while $i < m - 1$ **do**

*e-mail:abhianshu@knights.ucf.edu

6: **if** $l == 0$ **then**
7: $LCP[R[i]] = charComp(s[i : m], s[SA[R[i] + 1] : m])$
8: $l = LCP[R[i]]$
9: **else**
10: $LCP[R[i]] = charComp(s[SA[R[i] + 1] + l : m], s[i + l : m]) + (l - 1)$
11: $l = LCP[R[i]]$

1.3 Burrows Wheeler Transform

The BWT of String S and its Suffix Array SA is defined as a string S_{bwt} which is obtained by sequentially traversing SA and concatenating the characters that precede each suffix, i.e., for each $1 < i <= m$, $S_{bwt}(i) = S(SA[i] - 1)$. Also, the base case is $S_{bwt}(1) = S(m)$. BWT of String "PANAMABANANAS" is shown in Figure 2. BWT is used to compress the strings, Since it is a reversible transformation, so we don't need to store any additional data.

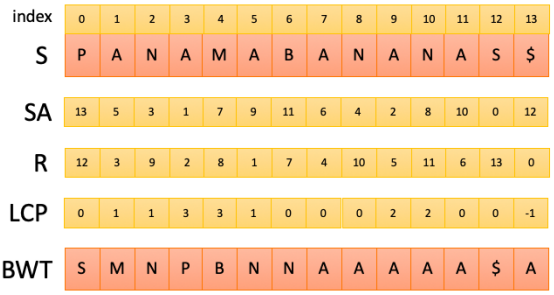


Figure 2: *Suffix Array, Inverse of Suffix Array and LCP array*

1.4 Searching using FM-Index

A FM-Index is an index which is constructed on string S and it is based on the Burrows-Wheeler Transform of S. It is a self-index which means it can replace the original string S. It is used to identify the interval in Suffix Array that contains a query pattern P. Let C(c) denote the number of occurrences in S of characters alphabetically smaller than c, and function Occ(L,i,c) denote the number of occurrences of character c in BWT[1,i] where BWT and SA represents the Burrows Wheeler Transform and Suffix Array of String S respectively. The algorithm is as follows:

1: **procedure** BACKWARDSEARCH
2: $i = |P|, sp = 1, ep = |S|$
3: **while** $sp <= ep$ and $i >= 1$ **do**
4: $c = P[i]$
5: $sp = C[c] + Occ(L, sp - 1, c) + 1$
6: $ep = C[c] + Occ(L, ep, c)$
7: $i = i - 1$
8: **if** $ep < sp$ **then**
9: Return $notFound$
10: **else**
11: Return $< sp, ep >$

1.5 Wavelet Trees

A wavelet tree is a data structure which iteratively partitions a string into two parts until the part is left with the same character over the sub-sequence. It is used to answer range queries efficiently which is used in Burrows Wheeler Transform to compute the occurrences of a character in a range. The root of this tree is the starting String s which contains characters in the range $[a, b]$. Every node($S[1, n]$) in this tree has a bit vector($B_v[1, n]$) which has the value $B_v[i] = 0$ if $S[i] <= (a+b)/2$, otherwise $B_v[i] = 1$. Also, create two children wavelet trees for this node - left wavelet tree and right wavelet tree. The left wavelet tree has the characters in the range $[a, (a + b)/2]$ and right wavelet tree has the characters in the range $[(a + b)/2 + 1, b]$.

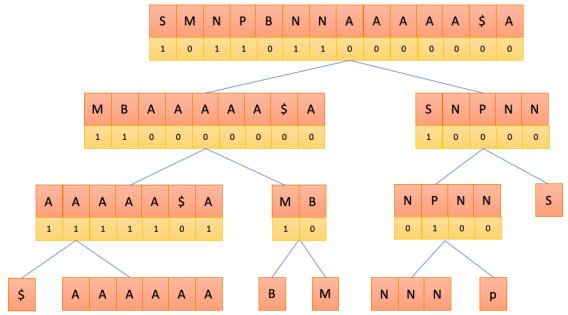


Figure 3: *Wavelet Tree for Burrows Wheeler String "SMNPBNNAAAAA$A". Yellow boxes shows the bit vectors and orange boxes shows the String.*

1.6 Inexact Search

The inexact algorithm is a recursive algorithm to search for the SA intervals of sub-strings of S that match the query string P with no more than z differences (mismatches or gaps). Essentially, this algorithm uses backward search to sample distinct sub-strings. Below is the algorithm for inexact search.

1: **procedure** INEXSEARCH(P,i,z,k,l)
2: **if** $z < 0$ **then** return ϕ
3: **if** $i < 0$ **then** return (k,l)
4: $I = I \cup InexSearch(P, i - 1, z - 1, k, l)$
5: **for** c `in` $\sum$ **do**
6: $K = C[c] + Occ(k - 1, c) + 1$
7: $L = C[c] + Occ(l, c)$
8: **if** $K <= L$ **then**
9: $I = I \cup InexSearch(P, i, z - 1, K, L)$
10: **if** $c == P[i]$ **then**
11: $I = I \cup InexSearch(P, i, z, K, L)$
12: **else**
13: $I = I \cup InexSearch(P, i - 1, z - 1, K, L)$
14: return I

The InexSearch algorithm takes Pattern P, $|P|$, z, 1 and $|S|$ as its input. In this algorithm, the first two calls to InexSearch is for the insertion and deletion of characters (gaps) from the Pattern P and third call is for the exact character matching. The last call to InexSearch is for the mismatch characters. This algorithm is made more efficient by computing D which gives the lower bound of the number of differences in P[0,i]. The better the D is estimated, the smaller the search space and the more efficient the algorithm is. D is calculated using the CalculateD algorithm, furthermore the BWT of the reverse string s is used to test if a sub-string of P is also a sub-string of S as it makes CalculateD operation an $O(|P|)$ rather than $O(|P|^2)$ procedure.

1: **procedure** CALCULATED(P)
2: $z = 0, j = 0$
3: **for** $i = 0$ `to` $|P| - 1$ **do**

4: **if** $P[j,i]$ is not a substring of S **then**
5: $z = z + 1, j = i + 1$
6: $D[i] = j$

With D calculated, the first check $z < 0$ in the InexRecur procedure is replaced with $z < D[i]$ to achieve a better solution.

2 Performance Evaluation

Consider a string s of size m. To construct Suffix Array using Bucket Sorting Algorithm, the total number of stages is $\log_2$m because if in stage H we sort the suffixes using H-length prefixes, then in the next stage we sort using 2H-length prefixes and then in next stage with 4H-length prefixes and so on. Also, each stage takes O(m) time to sort the suffixes in that stage. So, total time complexity for the construction of Suffix Array is O(m $\log_2$m). Furthermore, the space complexity for Suffix Array is O(m) as it just needs to store only the previous stage (history) of sorted suffixes and bucket ids and their starting index which is roughly of size 3m. If we consider the memory in terms of bits, then SA can be stored in m log$|\sum|$ bits. Inverse of Suffix Array R is computed from SA using only one traversal, so time complexity as well as space complexity for the construction of Inverse of Suffix Array is O(m).

To construct LCP array from String s and Suffix Array SA, the total time complexity is O(m). Let's look at this problem by charging the comparison between the r^{th} character in the suffix $S[i,m]$ and the corresponding character in its right neighbor suffix in SA. A comparison made in an iteration is termed successful if the characters compared are identical (i.e. contribute to the lcp being computed), otherwise it is termed as a failed comparison. As there is one failed comparison in each iteration, the total number of failed comparisons is O(m). Also, each position in the string is charged only once for a successful comparison. Thus the total number of successful comparisons is O(m). Thus, the total number of comparisons over all iterations is O(m). Hence, both the time complexity and space complexity for Constructing LCP array is O(m).

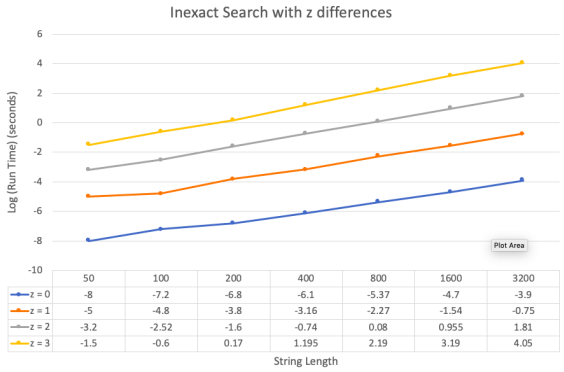
Wavelet Trees has log$|\sum|$ levels, with each level needing m bits where m is the size of the string S. So, the total space complexity for the wavelet trees is O(m log$|\sum|$) and the time complexity for the construction is also O(m log$|\sum|$) as it goes till log$|\sum|$ levels and stores the m values in the bitmap at each level if the tree is balanced. The time complexity of a single rank query in a wavelet tree is O(log$|\sum|$)) as there are only log$|\sum|$) levels in a wavelet tree. I calculated the number of bits iteratively, however, each level can use a binary rank to get the number of 1's till the desired number in O(1) time. So, total time taken by a range query can become O(log$|\sum|$)) from O(i log$|\sum|$)) where i is the index in Rank(c,i). Assuming O(log$|\sum|$)) time for rank query, I considered the occurrence function runs in O(log$|\sum|$)).

Time Complexity to compute Burrows Wheeler Transform String is O(m) as it just traverses the Suffix Array and the String S once. The backward search algorithm to find the interval of the pattern P in Suffix Array SA runs for the $|P|$(length of the pattern) iterations. The Count C(c) of character c is fetched in O(1) time. Hence, each Occurrence function can be computed in O(log$|\sum|$)) time. So, total time to locate the interval of pattern P in SA is O($|P|$ log$|\sum|$)) as it runs for the $|P|$ iterations. For the backward Search, no space is required as such apart from two pointers - sp and ep. However, the pre-processing space is O(m log$|\sum|$) for the Wavelet Trees and O(m) for BWT String and $|\sum|$ for the Count Array. Similar to the iterative version of backward search algorithm, a recursive version of this algorithm also takes O($|P|$ log$|\sum|$)) time to locate the interval of Pattern P. This recursive version is then exploited to do the inexact search by checking the gaps and mismatches. As the only modification is the introduction of the three more unions of InexSearch on the result and now we are checking for every character in the dictionary till the number of difference is 0. So, InexRecur has a time complexity of O($|\sum|^z$ $|P|$ log$|\sum|$)). Figure 4 shows the time complexity and space complexity of all the algorithms used in this paper.

Algorithm	Time Complexity	Space Complexity	Preprocessing Space
Suffix Array Construction – Bucket Sorting	m $\log_2$m	~3m	-
Inverse of Suffix Array	m	m	-
Longest Common Prefix	m	m	-
Burrows Wheeler Transform String	m	m	-
Wavelet Trees Construction	m log$\|\sum\|$	m log$\|\sum\|$	-
Rank Query	log$\|\sum\|$	-	m log$\|\sum\|$ (Wavelet Tree)
Count Query	1	-	$\|\sum\|$(Count Array)
Backward Search Algorithm for Exact Search	$\|P\|$ log$\|\sum\|$	-	m log$\|\sum\|$(Wavelet Trees) + m(BWT) + $\|\sum\|$(Count Array)
Inexact Search	$\|\sum\|^z \|P\|$ log$\|\sum\|$	$\|I\|$(Number of Intervals)	m log$\|\sum\|$(Wavelet Trees) + m(BWT) + $\|\sum\|$(Count Array)

Figure 4: *Space and Time Complexity*

The Inexact Search Algorithm is being run for strings of various lengths. As the string size is increased, run time increases exponentially for the inexact search. So, I took the logarithm of the run time to show in the below graph. The exact matching algorithm(z=0) runs in less than a second, so log values are negative. Figure 5 shows the inexact search with 0,1,2 and 3 differences. It has been observed from the figure that the log of run time is high for high value of z and the logarithm of run time increases linearly with the increase in string length.



	50	100	200	400	800	1600	3200
z = 0	-8	-7.2	-6.8	-6.1	-5.37	-4.7	-3.9
z = 1	-5	-4.8	-3.8	-3.16	-2.27	-1.54	-0.75
z = 2	-3.2	-2.52	-1.6	-0.74	0.08	0.955	1.81
z = 3	-1.5	-0.6	0.17	1.195	2.19	3.19	4.05

Figure 5: *Inexact Search with various text lengths and differences for a fixed Pattern Length of 6*

3 Conclusion

An exact search using Backward Search Algorithm and an Approximate Search developed on the recursive version of exact search is being implemented. This approximate search allows for the character insertion/deletion (gaps) from the string in addition to the character mismatches. To construct this algorithm, Suffix Array, Burrows Wheeler Transform and Wavelet Trees is also being implemented. I also computed Longest Common Prefix, although it is not used in the pattern search.

References

LI, H., AND DURBIN, R. 2009. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*.