MATH578A ASSIGNMENT

Jiawei Huang 02/26/2020

Problem 1

Implement the Boyer-Moore algorithm.

There are two .cpp files(**boyermoore.cpp, boyermoorecount.cpp**). The main part of these cpp files contain a Boyer Moore class with implementation of Boyer Moore algorithm.

The output of the first program(**boyermoore.cpp**) is the number of occurrences of the pattern in the text, and the output of the second program(**boyermoorecount.cpp**) is the number of occurrences, the number of character comparisons, and the time required, in seconds to the appropriate precision (excluding reading the input and pre-processing).

1. Identify a vertebrate species with a name (could be genus if needed, or informal name) that starts with the same letter as your name (either first or last). You will use the full genome of this species as the "text" in your tests. Indicate in your report which species you selected, where you obtained the genome, how large is the genome, and how many sequences are in the FASTA file.

   Answer:

Species	Rhinoceros hornbill
Size	1.1G
Genome source	NCBI
Sequence length	1,065,782,791

Table 1: General information about selected vertebrate species

Sequence	Time(s)	Matching	Comparision
ACACACACACACACACACACACACACACACACACACAC	1	2349	50387413
GAGAGAGAGAGAGAGAGAGAGAGAGAGAGAGAGAGAGAGA	1	59	48918958
CAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGCAGC	2	23	73289689
GACGACGACGACGACGACGACGACGACGACGACGACGACG	1	0	66832793
ACAGACAGACAGACAGACAGACAGACAGACAGACAGACAG	2	0	74054328
AACGAACGAACGAACGAACGAACGAACGAACGAACGAACG	1	0	66345049
AAGCAAGCAAGCAAGCAAGCAAGCAAGCAAGCAAGCAAGC	1	0	72652057
CCACCAGGGG	5	721	219537967
GGAGGACCCC	5	895	219815407

Table 2: BM algorithm on selected species

2. Using the human genome (hg38) as the text, run your program with a random 50bp portion of a human Alu element as the pattern. In your report, provide the 50bp you used, along with the run time,number of occurrences of this 50bp and total character comparisons.

   Answer:

Sequence	Time(s)	Matching	Comparision
ACCTCTTATTTAGTGGACAGACAGTAACTAGTTGAGAAGACAGGGGATTT	15	6	679699622

Table 3: human genome 50 bp sequence testing

Problem 2

Let T be a text string of length m and let S be a multiset of n characters. The problem is to find all substrings in T of length n that are formed by the characters of S. For example, let $S = (a, a, b, c)$ and $T = abahgcabah$. Then $caba$ is a substring of T formed from the characters of S. Give a solution to this problem that runs in $O(m)$ time. The method should also be able to state, for each position i, the length of the longest substring in T starting at i that can be formed from S.

Answer: Method based on sliding window. Creating a map containing the letters in multiset S with frequency counts of the letters be the value, call it target.

Start by processing the first n letters in text. For each such letter if it appears as a key in the target dictionary decrease the corresponding value by 1. The goal is to drive all target values to 0. Define discrepancy to be the sum of the absolute values of the values after processing the first window of m letters. Then move the n-length window 1 space to the right, check the letter preceeding the left window side(this letter just moves out of the window) and the letter at the right end of the window(just be covered by the window) and maintain target and discrepancy.

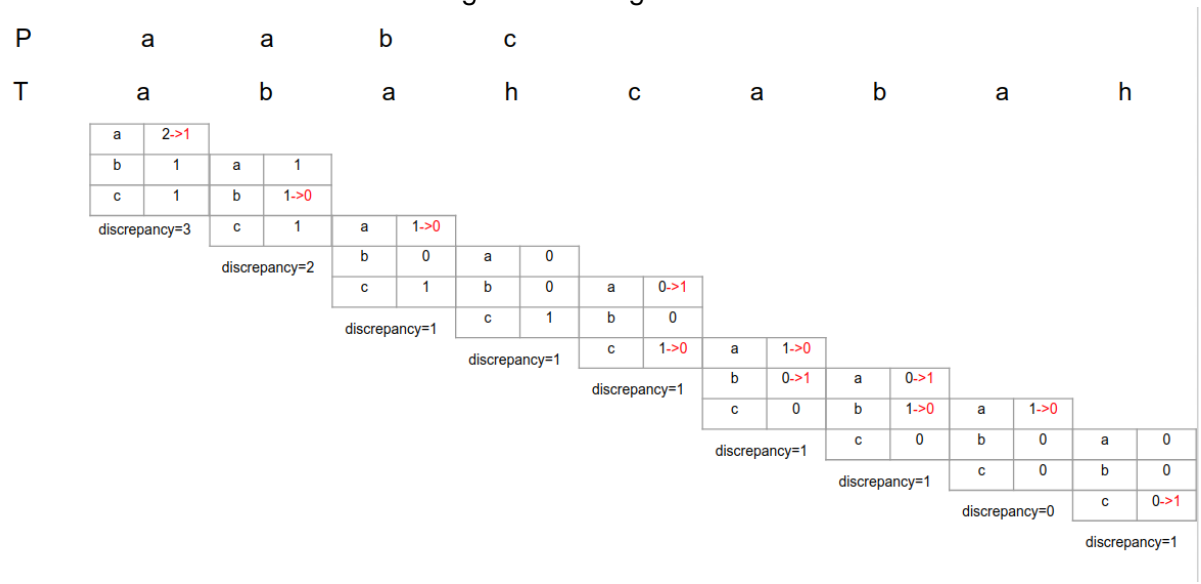
Listing 1: Pseudocode to find substring in text

```
input: s1: pattern, s2: text
output: list of start positions of substrings in s2, which are formed by letters in s1

m <- length of s1, n <- length of s2
if m > n
    return no matching

occurrence <- {c:n} # c is each character in s1, n is the difference between the first m characters in s2 and s1
discrepancy <- sum(n for c:n in occurrence)
match = []
for i <- 2 to n - m + 1
    if discrepancy==0
```

```
13          add i to match
14      if s2(i-1) in occurrence.keys
15          occurence[s2(i-1)] --
16      if s2(i + m - 1) in occurrence.keys
17          occurence[s2(i+m-1)] ++
18  return match
```

Figure 1 is a brief annotation about the process.

Figure 1: sliding window



Since there are no nested loops and each pass through the main loop involves just a few dictionary lookups, comparisons, addition and subtractions, the overall algorithm is linear.

For each position i, the length of the longest substring in T starting at i that can be formed from S can be calculated, from position i, length increases until come to a letter not in S.

Problem 3

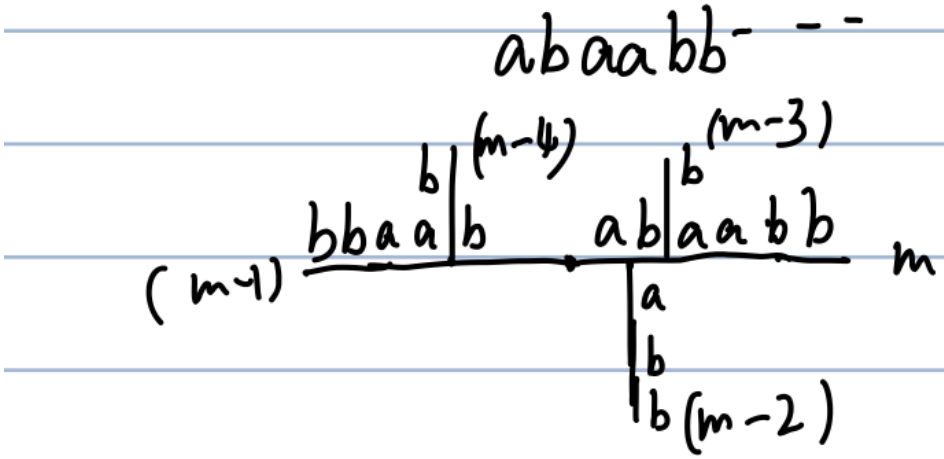
Construct an infinite family of strings over a fixed alphabet, where the total length of the edge-labels on their suffix trees grows faster than $O(m)$ (m is the length of the string). That is, show that linear-time suffix tree algorithms would be impossible if edge-labels were written explicitly on the edges.

Answer: Consider such string $abaabbaaabbb...(a)_i(b)_i...$ where $(a)_i$ means repeat i times,in this way we only have alphabet size of 2, but the edge labels on the suffix tree grows faster than $\theta(m)$. The total number of character of the suffix tree is:

$$\sum_{1}^{m} = \frac{m(m+1)}{2} \sim \theta(m^2)$$

Also, if edge-labels were written explicitly on the edges, the time for the algorithm is at least as large as the size of the output, which is $\theta(m^2)$, so It cannot be linear-time.

Figure 2: infinite suffix tree with alphabet size 2



Problem 4

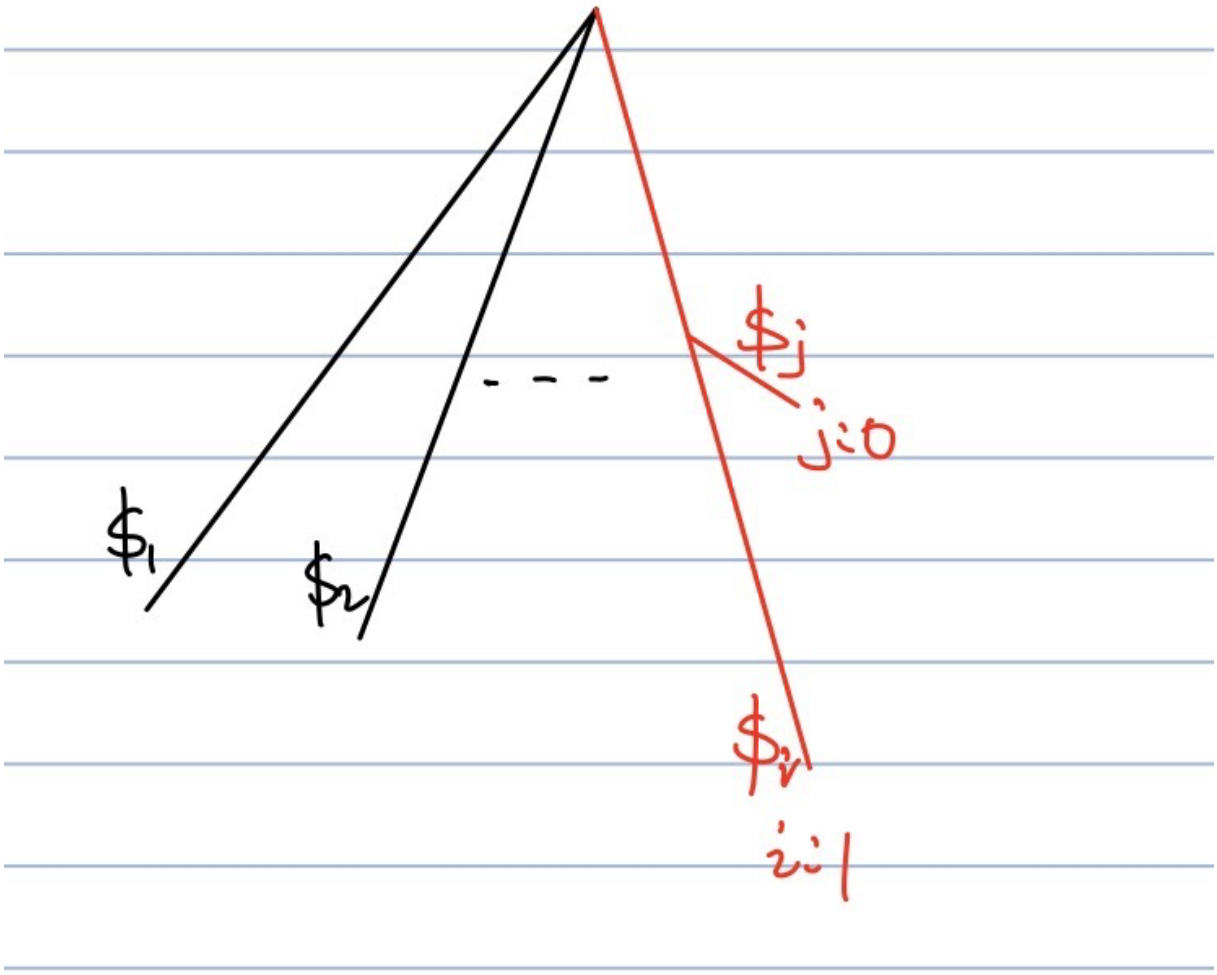
Given a set S of k strings, we want to find every string in S that is a substring of some other string in S. Assuming that the total length of all the strings is n, give an O(n)-time algorithm to solve this problem. This result will be needed in algorithms for the shortest superstring problem.

Answer: Using Generalized Sufix Tree.

- Construct a suffix tree for the single string $T_1\$_1T_2\$_2...T_k\$_k$ ($O(n)$)
- Run a DFS over the suffix tree and prune the invalid suffixes, namely, get the generalized suffix tree. Leaves are tagged with $i : j$, meaning "jth suffix of string T_i". When doing DFS, annotate the original strings(tagged $i : 0$). ($O(n)$)
- Run a DFS over the selected branches and check whether there is a node on the branch whose out edge is also the original string in S and has a end $\$$ sign only. Get the string s and its sibling strings which have more than one letters on its branch. This takes $O(n)$.

Overall complexity is $O(n)$ Following figure shows the geneal idea of the algorithm. First we select branches which is tagged $i : 0$, $i = 1, 2..., k$. Then we go through these edges and try to find nodes whose out coming edge contains only end string dollar sign. Here we find string j is a substring of string i.

Figure 3: Generalized Suffix Tree to find substring



Problem 5

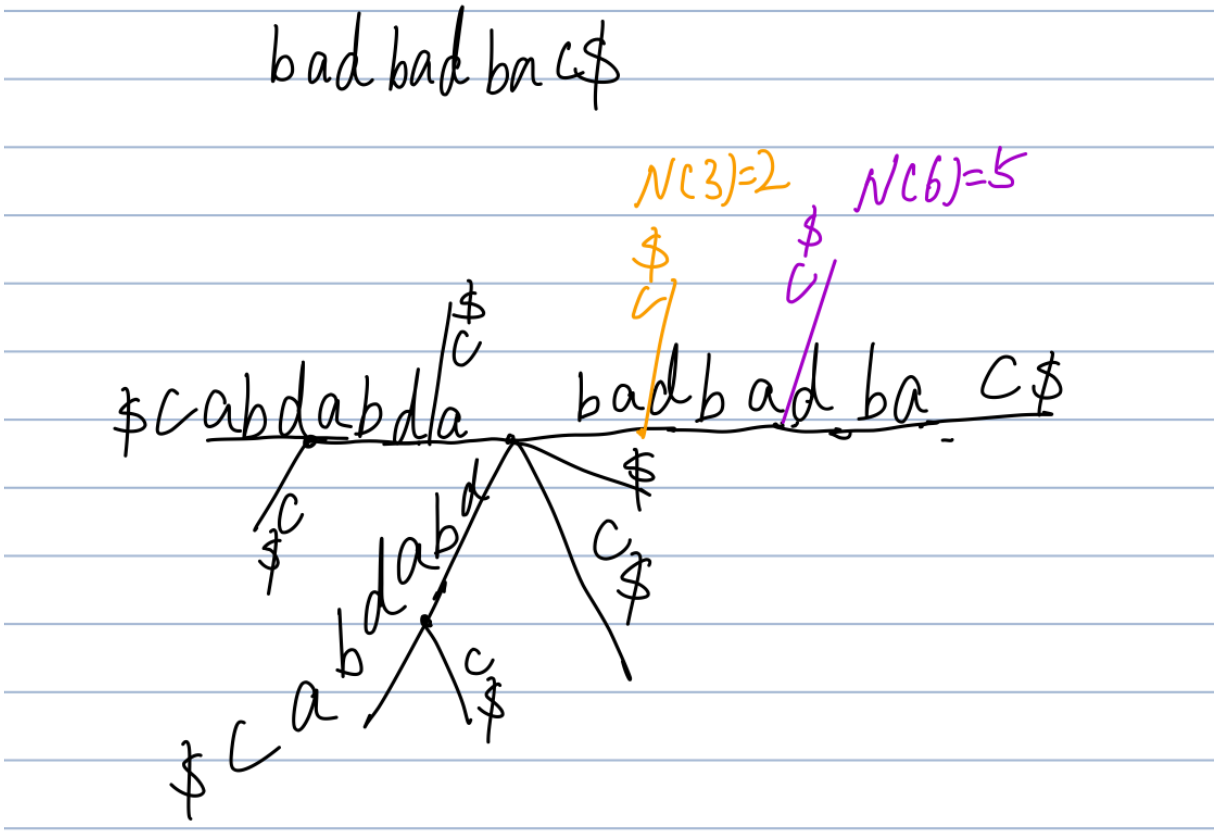
For a string S of length n, show how to compute the N(i), L(i), L'(i) and Spi values (discussed in Sections 2.2.4 and 2.3.2) in O(n) time directly from a suffix tree for S.

Answer:

1. $N(i) : N(i)$ is the length of the longest suffix of substring $S[1..i]$ which is also a suffix of full string S.
   First reverse string S and get S^r and then construct suffix tree based on S^r. And then find internal nodes on branch $S^r[1..n]$. $N(i)$ is the number of common elements between $S^r[1..n]$ and sub-branch whose length is i. If there is no sub-branch with length i, then $N(i) = 0$. Consider string $cabdabdab$, the reverse string is $badbadbac$. And N array is $[0, 0, 2, 0, 0, 5, 0, 0, 0]$, we can get it from the reversed suffix tree.

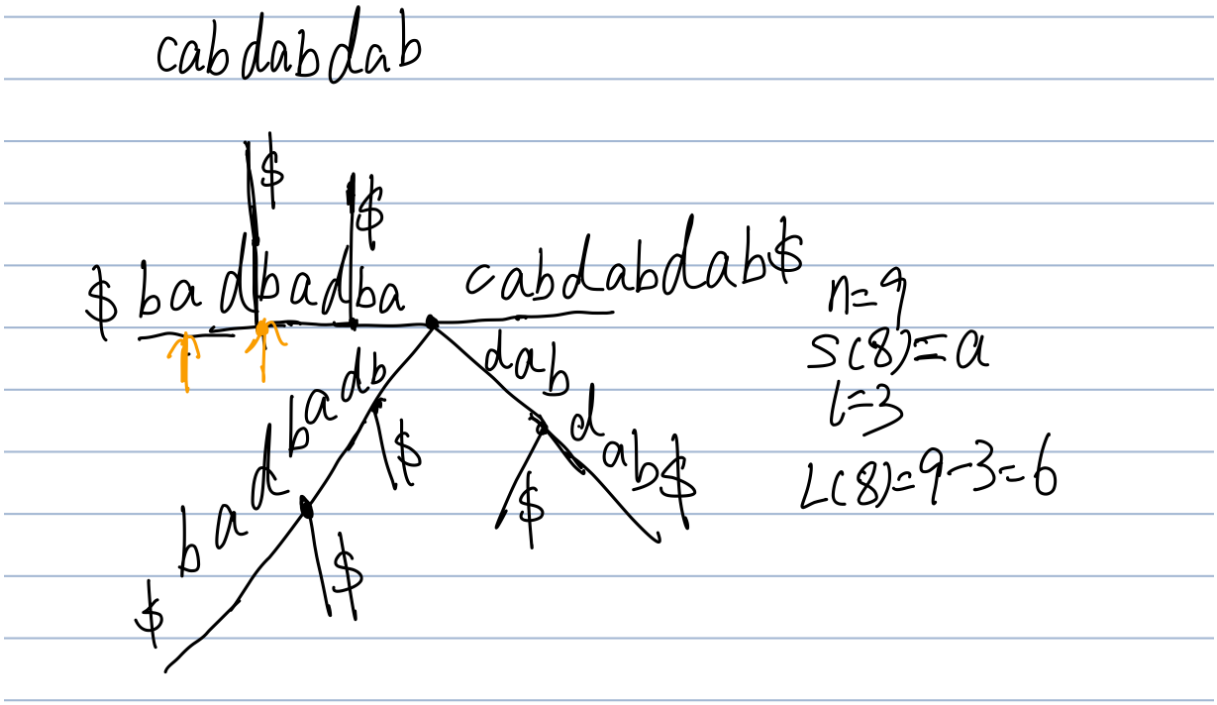
2. $L(i) : L(i)$ is the largest position less than n such that $S[i..n]$ matches a suffix of $S[1..L(i)]$

Figure 4: Suffix Tree of badbadbac



Construct suffix tree of S, and start with $S[i]$ branch, if there is a node at a distance $n-i+1$ from root on this branch. Then choose next node on this branch and calculate the distance l between the two nodes, if there is not an extra node on this branch, make l be the distance from the node to end. Then $L(i) = n - l$, if there is not a node at $n - i + 1$, then $L(i) = 0$. For example, we want to calculate $L(8)$ of following string $S = cabdabdab$. Here $S(8) = a, n = 9$, we first look at branch start with a, and we find there is a branch of $ab\$$, which means $L(8)$ is not 0, and we find next node and the distance between the two nodes $l = 3$, so $L(8) = n - l = 9 - 3 = 6$

Figure 5: L$_arrayfromSuffixTree$



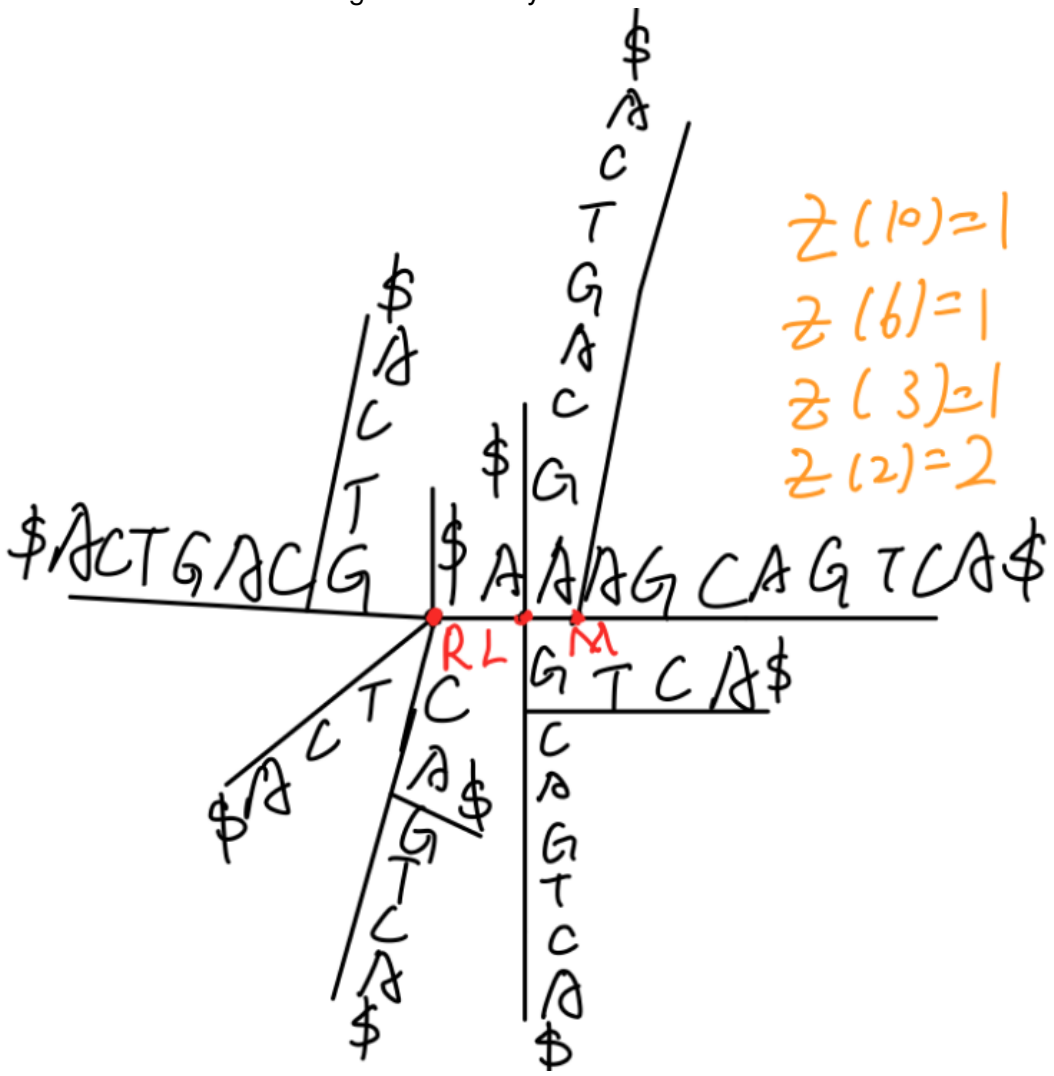
3. $L'(i) : L'(i)$ is the largest position less than n such that $S[i..n]$ matches a suffix of $S[1..L(i)]$ and the character preceding that suffix is not equal to $S[i-1]$.

   Similar to the calculation of $L(i)$, first start with $S[i]$ branch and see whether there is a branch at $n-i+1$, if no extra branch, set $L'(i) = 0$, else find distance between next node or end to this node, but this time we should compare the character preceding the suffix with $S[i-1]$, if they are equal, ignore this node and go to next node, if there is not an extra node, set $L'(i) = 0$

4. $SP(i) : SP(i)$ is the length of the longest suffix of $S[1..i]$ matching a prefix of S. We can first get Z values and then convert them into SP values.

   Z value is the distance from root to node on the original string, that is to say $Z(n-l-d+1) = d$, n is the length of original string, l is the number of letters out of the nodes, d is the distance from root to the node. For example, calculate the Z values of string $AAAGCAGTCA$. We can get the suffix tree like below. And we found 2 nodes L, M on the original string, and 3 branches out of L, 2 branches out of M. For node L, distance between root R to it is 1, there is one $\$$ branch out of L, which means $l = 0$, so $Z(10-0-1+1) = Z(10) = 1$, the second branch out of L is $GTCA\$$, so $l = 4$, $Z(10-4-1+1) = Z(6) = 1$, the third branch out of L is $GCAGTCA\$$, so $l = 7$, $Z(10-7-1+1) = Z(3) = 1$, and similarly we can get $Z(2) = 2$.

Figure 6: Z array from Suffix Tree



After getting Z values, we can then convert them into SP values. Here is the pseudocode.

Listing 2: Pseudocode to convert Z values into SP values

```
input: Z array
output: SP array

for i <- 1 to n
  sp_prime(i) <- 0
for j <- n downto 2
  i <- j + Z(j) - 1
  sp_prime(i) <- Z(j)
sp = sp_prime
for i <- n-1 downto 2
  sp(i) <- max(sp(i+1) - 1, sp_prime(i))
return sp
```