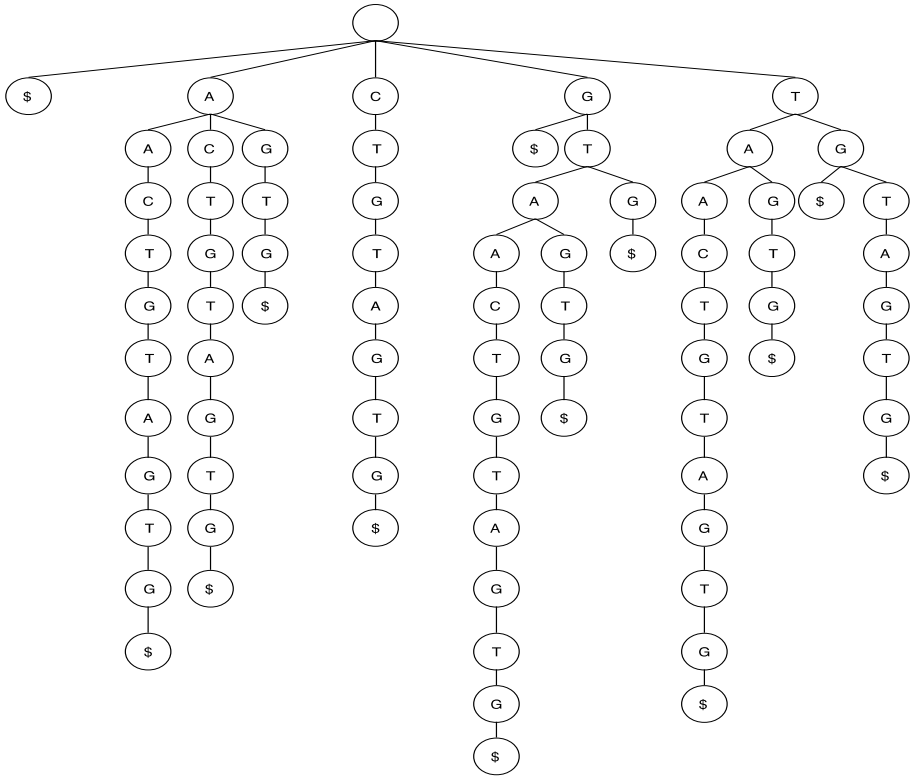
Nigel Nicholas

1155088791

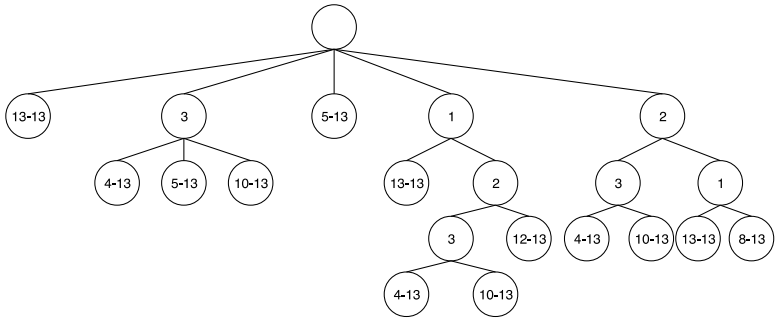
CSCI3220 Assignment 2

Sequence s = GTAACTGTAGTG$

1. (a) Sorted **Suffix Trie**:



(b) Sorted **Suffix Tree**:



(c) By Suffix tree in part b, we will construct **Suffix Array:**

Suffix	Location
$	**13**
AACTGTAGTG$	**3**
ACTGTAGTG$	**4**
AGTG$	**9**
CTGTAGTG$	**5**
G$	**12**
GTAACTGTAGTG$	**1**
GTAGTG$	**7**
GTG$	**10**
TAACTGTAGTG$	**2**
TAGTG$	**8**
TG$	**11**
TGTAGTG$	**6**

We can translate the suffix tree to suffix array easily because it is already sorted. Since all the sequence ends at leaf (all leaves go from n-13), we can really use current depth and the starting index of leaf node.

Traverse the tree with prefix order, which is from left to right precedence, going in depth. From root node, every time you traverse, add depth by 1 each time and reduce by 1 each time you return. Then get location by (*startIndex-depth-1)* , and the suffix corresponding to it will be s[*startIndex-depth-1* .. 13]. (*Note: startIndex means the initial number on leaf node)

(d) Another method to construct suffix array without using suffix tree. The result should be the same table presented in (c) part.

First from given sequence s, construct suffix array that is sorted by its location number. This can simply be done with

---

table = []

For i=1 -> length(s):

Createobject O.

O.suffix = s[i..13]

O.location = i

Append Object O to table.

Suffix	Location
GTAACTGTAGTG$	1
TAACTGTAGTG$	2
AACTGTAGTG$	3
ACTGTAGTG$	4
CTGTAGTG$	5
TGTAGTG$	6
GTAGTG$	7
TAGTG$	8
AGTG$	9
GTG$	10
TG$	11
G$	12
$	13

After getting all this, sort the table based on Object.suffix (in lexicographical order).

(e) Construct BWT using suffix array.

Using the sorted suffix array like the one shown in (c) we can construct the BWT b in linear time.

Now, for everything in the sorted suffix array,

Algorithm

b = " " (empty string)

s[0] = $ // to ensure that index not out of bound since array starts with 1

For entry in suffix_array_table:

    b = b + s[entry.location -1]

end for

return b

^ b here is the BWT, b.

---

result:

b=GTATAT$TAGGGC

(f) Just like the one provided in lecture notes, we can simply rotate the first letter in s to the end after $ sign, sort the string in lexicographical order, then take the ends of each string.

Algorithm:

rotation_table = [] // initialize array of rotation table

// We can assume the s to be in some sort of queue data structure to make use of inject and eject

For i in range of length(s):

    rotation_table.append(s)

    Inject(eject(s));

End for

Sorted_rotation_table = Sort(rotation_table)

b= " " //empty string

For entry in sorted_rotation_table:

    b = b + entry[13]

end for

Here is the illustration:

s = GTAACTGTAGTG$

rotations:

Rotations	Sorted Rotation
GTAACTGTAGTG$	$GTAACTGTAGT**(G)**
TAACTGTAGTG$G	AACTGTAGTG$G**(T)**
AACTGTAGTG$GT	ACTGTAGTG$GT**(A)**
ACTGTAGTG$GTA	AGTG$GTAACTG**(T)**
CTGTAGTG$GTAA	CTGTAGTG$GTA**(A)**
TGTAGTG$GTAAC	G$GTAACTGTAG**(T)**
GTAGTG$GTAACT	GTAACTGTAGTG**($)**
TAGTG$GTAACTG	GTAGTG$GTAAC**(T)**
AGTG$GTAACTGT	GTG$GTAACTGT**(A)**
GTG$GTAACTGTA	TAACTGTAGTG$**(G)**
TG$GTAACTGTAG	TAGTG$GTAACT**(G)**
G$GTAACTGTAGT	TG$GTAACTGTA**(G)**
$GTAACTGTAGTG	TGTAGTG$GTAA**(C)**

Then, **b=GTATAT$TAGGGC .** Then BWT b is the letters contained within parantheses.

As for result, the answer is exactly the same with part (e).

(g) We can use data structure to represent FM index.

We can make use of suffix array and construct b from it. Now let's construct the FM index.

b = GTATAT$TAGGGC

s = GTAACTGTAGTG$

1st column = $AAACGGGGTTTT -> taken from suffix array's first letter off of each entry

	A	C	G	T
First occurrence position in the first column (F)	2	5	6	10

a = $AAACGGGGTTTT //a is the first column

b = GTATAT$TAGGGC

Occurrence within first i rows in b	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*
A	0	0	1	1	2	2	2	2	3	3	3	3	3
C	0	0	0	0	0	0	0	0	0	0	0	0	1
G	1	1	1	1	1	1	1	1	1	2	3	4	4
T	0	1	1	2	2	3	3	4	4	4	4	4	4

Now with all the provided FM-Index we can search the query string.

Query q = GTA

Step-by-step starting from last character to first :

a. Find O[13, A] -> finding total frequency of A after 13 (which means total frequency of A in sequence s). Total 3 occurrences.
b. Find the positions of all A. Starting position A is F[A] = 2 and it goes until (F[A] + 3 -1 = 4). So position of A in the first column will be from 2 to 4.
c. Get the number of occurrences of T until the F[A]-1 which is 1. O[1,T] = 0. Then find O[4,T]= 2 which means by first to last occurrence of A in first column, there are 2Ts. 1st to 2nd occurrence appear throughout the passing of all A.
d. Start iterating all row at which the last column of A is T. This in row 2 and 4. For each input, recursively do the same thing but now search T that has last column G.

T from row 2:

a. Since this is T's first occurrence, find this T's **first** occurrence in a. F[T]+1-1 =10. At row 10, b[10] =G. This shows that the query exists in this particular search. Then, with this row O[row, G] => which is 2. Find the 2nd occurrence of G in a, which is F[G]+2-1= 7. Then Find the 7th entry in Suffix array table, SA[7]. Return SA[7] as an answer. * note index starts from 1

T from row 4:

b. Since this is T's second occurrence, find T's **second** occurrence : F[T] + 2 -1 = 11. At row 11, b[11] = G. This shows that second occurrence of G also has the query : GTA. O[11,G] = 3. Find 3rd occurrence F[G]+3-1 = 8. Therefore return SA[8] as the answer.

*Note: Using SA[n] means look into nth entry in the sorted suffix array table.

(h) No. Because if there is only 1 '$' sign appended to the sequence of length n, then after rotating everything, the first sorted rotated sequence will be one with $ in the first position.

Example: , say s is the sequence

Sequence s$-> The first in sorted rotation will always be $s.

With that, it is impossible $ will be positioned first in *b,* probability ($ at position 1) = 0. Therefore the statement that it has equal probability in all position is false, because if so, then probability of all other position is 0 : (0 x n) which contradicts the theory that ∑probability = 1.