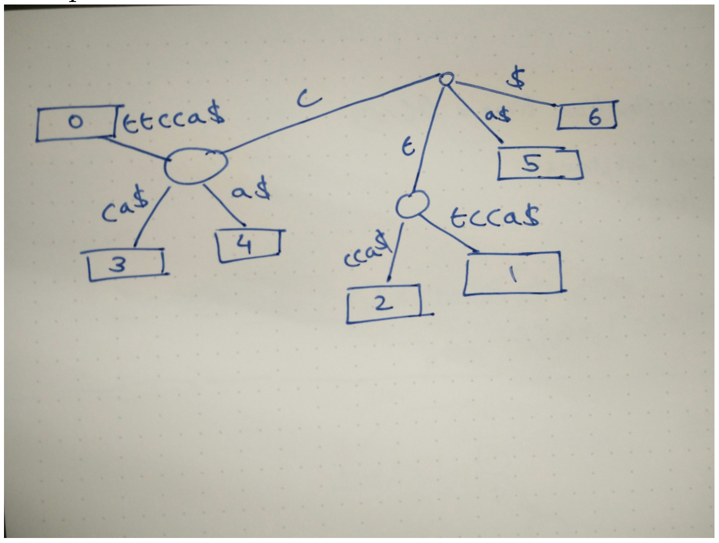
Computational Genomics: Sequences
Homework 3

Srinivas Suresh

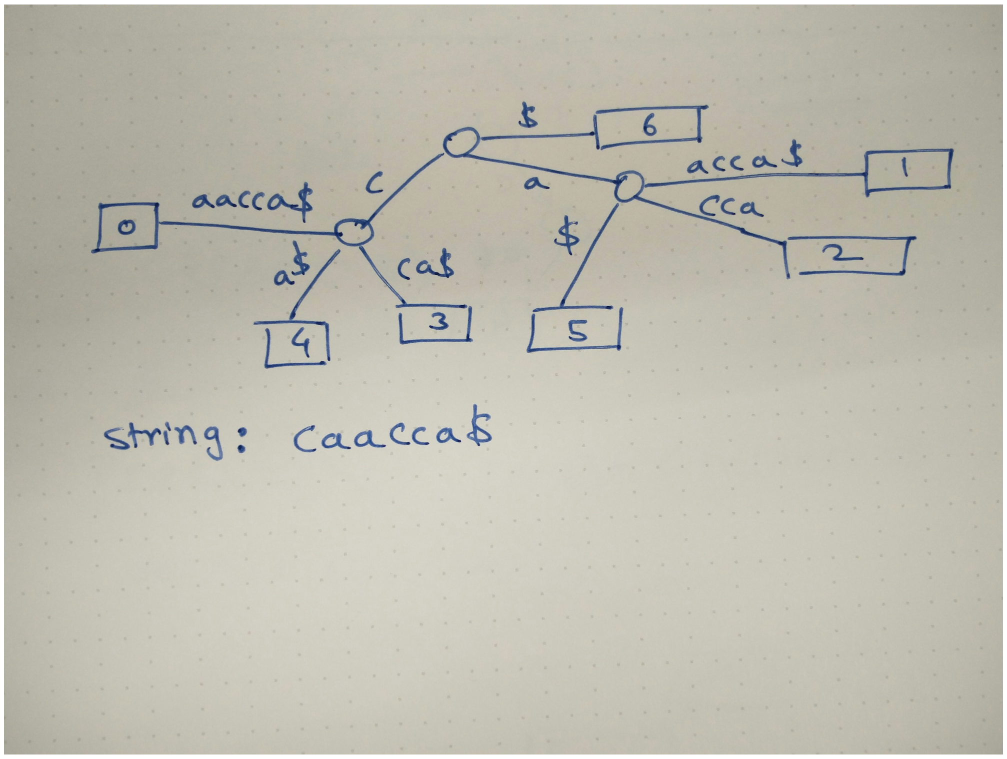
October 12, 2017

Answers

1. answer1.py > inputfile < outputfile
2. answer2.py > inputfile < outputfile
3. (a) This is possible as sometimes you might end up replacing the character with itself
   (b) This is also possible as everytime you might end up replacing the character with something other than itself
   (c) This is not possible as you don't edit the list more than n times
4. Completed tree below



5. (a) Below is the tree



(b)

6	$
5	a$
1	aacca$
2	acca$
4	ca$
0	caacca$
3	cca $

(c) lexicographically sorted rotations of T
$CAACCA
A$CAACC
AACCA$C
ACCA$CA
CA$CAAC
CAACCA$
CCA$CAA
$BWT(T)$= ACCAC$A

6. A string that has does not have contiguous chunks of repeating symbols would satisfy this requirement
eg: $abcde\$$. Would create a root node and a \$ node. And m nodes representing the m suffixes. Resulting in $m + 2$ nodes.

7. (a) Construct 1 Generalized Suffix Tree for all the strings $O(N)$, separating each string with a unique terminal.

   (b) Now we know that every string will have its own path in the suffix tree representing the string in its entireity

   (c) We perform a DFS on the tree, visiting every node making note of the deepest node in terms of label depth for every string in the set of strings. We obtain pointers to these nodes and store them in a list: $O(N)$

   (d) We iterate over this list and check if each of these nodes have a string other than the string they are the deeepest node for associated with them. If so then string for which this was the deepest node is a susbtring of another string.: $O(k)$

   (e) Thus our algorithm works in linear time. This algorithm exploits the fact that
   Every substring of a string is a prefix of some suffix

8. The Burrows-Wheeler transform sorts the permuations by their *right context*. This helps it brings similar/same characters together in runs. So, while storing, all characters need not be stored individually. One instance of a character along with the number of times it occurs is sufficient. This helps in compression.

9. (a) Sorting the last column lexicographically will give you the first column of the matrix

   (b) Yes. Every column between L and F are just permutations of the first column. so lexicographically Sorting would yield the first column

   (c) The above hold for any row as well. Therfore this is still true.

10. (a) There are multiple ways to solve this, but one way is shown below
if $BWT(S) = \mathrm{A_0A_1C_0G_0G_1G_2\$}$
then, $F = \mathrm{\$A_0A_1C_0G_0G_1G_2}$
then, $S = \mathrm{G_2G_1G_0C_0A_1A_0\$}$

(b) \$GATGAG
AG\$GATG
ATGAG\$G
G\$GATGA
GAG\$GAT
GATGAG\$
TGAG\$GA

11. $BWT(T) = \mathrm{C_0C_1T_0\$A_0G_0G_1}$
then, F = $\mathrm{\$A_0C_0C_1G_0G_1T_0}$
then T\$ $\rightarrow \mathrm{C_1A_0G_0G_1T_0C_0\$}$

12. (a) B ranked $BWT(T)$

\$	TTGAC	$\mathbf{A_0}$
$\mathbf{A_0}$	\$TTGA	$\mathbf{C_0}$
$\mathbf{A_1}$	CA\$TT	$\mathbf{G_0}$
$\mathbf{C_0}$	A\$TTG	$\mathbf{A_1}$
$\mathbf{G_0}$	ACA\$T	$\mathbf{T_0}$
$\mathbf{T_0}$	GACA\$	$\mathbf{T_1}$
$\mathbf{T_1}$	TGACA	**\$**

Reversing to get T
$\mathbf{T_1T_0G_0A_1C_0A_0\$}$

(b) In an LF mapping, the i^th^ occurrence of a character in F has the same rank as the i^th^ occurrence of that character in L. Thanks to this property we can now reassign a rank based on the order of appearance of each character in either F or L as opposed to their order in T. This ranking is known as a *B-ranking*. This sequential and ascending order of the ranks of the character greatly simplifies retrieval of T from the matrix