University of Glasgow | Department of Computing Science

ALGORITHMICS

ASSESSED EXERCISE

Suffix Tree Applications

State & Strategy

Author:
Garry SHARP
0801585s

Supervisors:
Dr. D. MANLOVE

February 15, 2013

Contents

1 Task 1 (Search Suffix Tree)

The task here is implement a method searchSuffixTree in the class SuffixTreeAppl which searches over a global instance of a SuffixTree known as t with the search term being an array of bytes x.

1.1 State

My solution works in an effective manner to search the tree in O(n) time. It should be noted that the solutions to Task 2, Searching for all occurrences relies heavily on this method. This means that there is a very small amount(almost negligable) amount of redundant code when searching (sets a global variable).

1.2 Solution

Firstly, lets examine the variables that are used. I keep track of an x index (initialised to 0), a node index (index of the character being compared at the node), a boolean called match (which is set to true/false depending on whether there is a match), an integer startLocation (initialised to -1) as well as a SuffixTreeNode called currentNode (initialised to the child of the root of the tree).

My solution compares the relevant values of x with the relevant values held at the node. If all characters are matched successfully (and there are more in subsequent nodes) then I perform the same check at the child of that node. If all characters have been matched and no more are left then the loop terminates.

If there is a mismatch at any characters then I examine the sibling that node until, either there is another match (is a mismatch occurs at that node it looks at the sibling, otherwise if fully matched goes to the child) or a mismatch at which point the loop exits.

A variable t2Node is set throughout and this is so that Task 2 can be completed by recycling the code here in Task 1. This node is eventually set to the final node that contains leafs of the instances of the search term.

If a match does occur and startLocation is not already set, a variable startLocation is set to it's left label of the currentNode. Otherwise is a mismatch occurs startLabel is reset (set to -1).

The program terminates by returning a Task1Info object (with position start-Location which will be -1 if not found) to the main method which then displays it accordingly. The search executes in O(n) time. This solution uses no external helper methods.

2 Task 2 (All Occurrences)

The task here is very similar to Task 1, but with the result being equal to every instance of the search term in the text (as opposed to just a single occurrence), specifically the starting indices of every occurrence.

2.1 State

This works in an effective manner by reusing the search functionality implemented in Task 1. This is done by calling the searchSuffixTree method and then using the t2Node (which is set to the branch node from which all decendent leaf nodes contain the occurrences). The solution executes in O(n) time (plus build time).

2.2 Solution

As mentioned above, searchSuffixTree is called and t2Node is assigned accordingly. From here on I use a recursive method on the child of the node until there are no more siblings. I basically look at all nodes, where if there is no child (ie. node is a leaf) I add the value to a global variable called occurrences, otherwise (if leaf) I recurse down the child of the currentnode and after move to the sibling.

Finally I iterate through the value of occurences and add each of the suffixes of each node to the Task2Info object which I then return. The recursive call to iterate through the values is getLeafDecendants.

3 Task 3 (Longest Repeating Substring)

The task here is to look through a tree and find the longest repeating substring, or to put it another way, the longest multiple occurring sequence of characters. A repeating substring exists if a node has two leaf nodes that are it's children (or siblings of the child in this implementation). To get the LONGEST repeating substring, the node with the the greatest path depth must be found. This is the value of the highest ancestor's (first item in the list's) leftLabel subtracted from the last node in the list's (currentNode's) right label (provided that currentNode is valid ie. has 2 leaf children).

3.1 State

The solution detailed works as expected when tested over the data sets provided.

3.2 Solution

The solution consists of several parts which I will detail individually. Firstly I should state that my solution is not ideal as it does not visit each node only once, however, in compromise, the readability offered by this is greatly improved.

As explained above there are several schools of thought in calculating this, firstly there must be a way of checking if a node is valid (ie. has at least 2 children that are leaf nodes). To aid readability this is done in the method isValidBranch which takes a SuffixTreeNode as an argument and returns true or false if it is "valid" (has at least 2 leaf node children).

In addition there must be some form of iteration over the tree in order to find the greatest path depth, this is aided by a method called next. Next takes a list of SuffixTreeNodes as an argument which represent a path from the highest ancestor (a child of the root of tree t) to the currentNode (the furthest down descendant). Next determines what the next path should be and will either visit a child (adding to the list and returning it), visit siblings (removing the last element of the list and adding the sibling in it's place). Finally if all children/siblings have been visited then the last element is removed continually until either the last node in the list has a sibling OR there is no more nodes in the list (meaning all paths have been visited). The requirement of a list is because nodes do not have a .getParent() method. A local variable "visited" is featured as to prevent an infinite loop occurring by removing a node from the list and then immidiately adding it again, this is also used to allow the path to shrink by more than one node if necessary (ie. the immediate parent has no sibling). This is perhaps the most complex part of the solution (building the next method) and is what consumed the majority of my time on this specific task.

Finally a method that takes a path (explained below) is needed, which takes the current path and calculates it's depth by summing the individual lengths (rightlabel - leftlabel + 1).

The final solution basically combines all of the above methods, iterating through the tree until the path equals null (terminating the iteration as all paths have been examined), and comparing it the length of the deepest path (called bestLsrNodes in solution). if the currentPath is deeper than the current best and also has a "valid node" (node with at least 2 child leafs) as the last entry, then bestLsrNodes is updated accordingly.

Finally a Task3Info object is returned which contains the length of bestLsrNodes and two of the suffixes (or positions) of the last node in the bestLsrNodes list.

NB. Because I calulate the length of the path every time a new path is found you may consider the code inefficient, however, this was done intentionally for a number of reasons. Firstly, it aids in the logical reading of the code, improving readability and description. Secondly given the time contraints, I did not want to dwell for to long on a minor improvement in efficiency (takes approximately 350ms to execute as it is currently). The improvement could be realised by the creation of a bean/inner class which consists of a SuffixTreeNode and a depth (which is calculated in the next method). This inner class could then be used in the path/bestLrsNodes instead of just the nodes, removing the need for computing the depth later as it is stored along with the node. This was not done as the project description advices against using inner classes in the implementation of the SuffixTreeAppl class. The program still executes quickly (faster than construction) despite this.

4 Task4 (Longest Common Substring)

The task here is to output the longest common substring over two files (ie. the longest sequence of characters that is present in both some file a and also some file b.

4.1 State

The code (I think) works well as it does output what I think is the longest common string. That said, when I have compared my results to the results of peers they have been different upon occation, the longest common substring that is returned by comparing text1.txt and text2.txt is the same as the output on the handout sheet under "example output".

4.2 Solution

The code used here relies heavily upon the code detailed in Section 3. It is therefore advisable that you are familiar with this method before reading this as I will presume familiarity.

Firstly, in order to use this method, a constructor for two files must be implemented in the SuffixTree class. This is done by reading in the bytes to the first file, appending a # symbol, appending the bytes for the second file, and finally appending a $ symbol. Variables such as stringLen must here be set to the length of the total bytes -1 (as to not include the $ symbol as a character in the actual tree). No checking that either files contain a $ or # symbol is done as this is stated as a premumption in the constructor's javadoc. The tree is then built.

For the description below, file a and file b reference the first and second files' bytes that make up part of the tree respectively.

The code is almost identical to the solution as detailed in task 3, however, there is a difference in the sense that an additional check must be made AFTER checking is the node is valid (ie. contains at least two leaf nodes) and BEFORE setting the bestLsrNodes to the path if the depth is greater than what has previously been encountered. This step is simply making sure that AT LEAST ONE of the suffixes of the last node in the path is less than the length of the first file (ie. it is present in file a) and also that AT LEAST ONE OTHER suffix is greater than the length of the first file (ie. it is present in file b). If both of these conditions are met, then and only then is the bestLrsNodes updated, if this condition is not met then the two substrings are not common, ie. It is not present in BOTH files.

This solution suffers from the same minor flaw as task 3 in that it visits nodes multiple times to calculate the path depth. Other than this it appears to work as expected.