COL106: Data Structures, I Semester 2016-17

Assignment 6

The substring matching problem

October 19, 2016

In this assignment we will address a basic problem in text processing: the substring matching problem. First note that a string is an array of characters. Now, given two strings, a of length n and b of length m we say that a *is a substring of* b if there is an $i : 0 \leq i \leq m - n$ such that $a[j] = b[i + j]$ for all $j : 0 \leq j \leq n - 1$. For example "at" is a substring of "cat", "bat", and "attendance" but not of "basin".

Given a set of strings s and a query string q, the *substring matching problem* asks us to answer questions like

- Is q a substring of s?
- How many times does q occur as a substring of s? (The occurences may be overlapping or non-overlapping).
- At what position(s) of s does q occur as a substring?
- Does q' that differs by q in only k characters occur as a substring of s? (Example: the string "tag" is not a substring of the string "category" but it has only 1 character that differs with "teg" which is a substring.)

Important: For the purposes of this assignment we will do case-insensitive matching, i.e., we will treat upper and lower case letters as the same.
Sort-of important: This assignment involves a significant amount of self-study since you are required to code a solution to the substring matching problem using *suffix tries*, an efficient data structure for the problem, and this data structure will *not* be covered in class.

Very important: There are *many* implementations of suffix tries and associated algorithms available on the Web. Please do *not* copy them. We will randomly check for copying using turnitin and if we find significant matches with the database then we will take strict disciplinary action. The *minimum* punishment will be an F grade in the class. You are to write *all* code yourself.

Assignment 6 [100 marks]
Deadline: **11:55 PM, 1 November 2016**

You are expected to do the following

- Read and understand the *trie* data structure from the textbook or any other source.
- Read and understand *suffix tries* from any source. Sometimes you will find these called suffix *trees*.
- Write a Java class named `SuffixTrie` for a suffix trie containing all the methods you will need. Note that we are not specifying method names or implementation details this time, you have to work them out yourself. Although, include `performAction(String actionMessage)` method in the `SuffixTrie.java` as `checker` file calls it.
- Read in a file that contains the target string s and creates a suffix trie out of this. Note that there is a standard algorithm for this that creates a suffix trie in $\theta(n)$ time where n is the length of s. Read this algorithm and implement it.
- Handle the four kinds of queries mentioned above.

Demo Instructions: In this assignment, you have been given files: checker.java, actions.txt, input.txt and output.txt. checker.java file is our program which reads the input actions from the actions.txt file, and feeds them to SuffixTrie.java. The output.txt file contains output corresponding to the actions.txt. The input string for making suffix trie is provided in the file named input.txt. The primary task of your assignment is to give the correct answer to the query messages (described below). We will verify the output of the program during the demo.
The list of actions that you will be expected to handle:

- `makeSuffixTrie filename`: reads in a single string from the file whose name is specified and creates a suffix trie. *Everything* from the beginning of the file to the end of the file will be considered as part of the string including any special characters or spaces or line breaks.

- `isSubstring s`: Returns 1 if `s` is a substring of the string stored in the suffix trie, 0 if not and throws an exception if the suffix trie is empty (i.e. has not been built yet.)

- `numSubstrings s overlapflag`: Returns the number of copies of `s` can be found in the string stored in the suffix trie, 0 if `s` is not a substring, and throws an exception if the suffix trie is empty (i.e. has not been built yet.) If `overlapflag` is 1 then overlapping occurences are treated as different instances of the string, otherwise not, i.e., if our string is "banana" and the query string is "ana" then if `overlapflag` is 1 your program should output `2` because "ana" is found starting from locations 1 and 3 (the first location is 0). If `overlapflag` is 0 then the output is `1`.

- `posSubstrings s overlapflag`: Returns a list of locations at which a copy of `s` can be found in the string stored in the suffix trie, -1 if `s` is not a substring, and throws an exception if the suffix trie is empty (i.e. has not been built yet.) If `overlapflag` is 1 then overlapping occurences are treated as different instances of the string, otherwise not, i.e., if our string is "banana" and the query string is "ana" then if `overlapflag` is 1 your program should output `1 3` indicating that "ana" is found starting from locations 1 and 3 (the first location is 0). If `overlapflag` is 0 then the output is only `1`.

- `numFuzzySubstrings s num overlapflag`: Returns the number of locations where a copy of a string that differs from `s` in *at most* `num` locations is to be found in the string stored in the suffix trie, -1 if no such substring exists. Each unique substring matched with atmost `num` difference should be counted once. If `num` is greater than the length of `s` then an exception must be raised. An exception must also be raised if the suffix trie is empty. The role of `overlapflag` is as mentioned above.

- `posFuzzySubstrings s num overlapflag`: Returns a list of locations where a copy of a string that differs from `s` in *at most* `num` locations is

to be found in the string stored in the suffix trie, -1 if no such substring exists. If `num` is greater than the length of `s` then an exception must be raised. An exception must also be raised if the suffix trie is empty. The role of `overlapflag` is as mentioned above.