UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Computer Science

COS212: Practical 6

Release: Monday 8 April 2019, 18:00
Deadline: Tuesday 9 April 2019, 18:00

Objectives

This practical has the following objectives:

- To observe the similarities between trees and graphs.
- To implement a Trie using a graph structure.
- To utilize the Trie data structure for approximate string matching.

Instructions

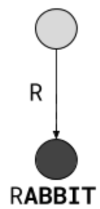
Complete the task below. Certain classes have been provided for you in the *files* zip archive of the practical. You have also been given a main file which will test some code functionality, but it is by no means intended to provide extensive test coverage. You are encouraged to edit this file and test your code more thoroughly. Remember to test boundary cases. Upload **only** the given source files with your changes in a zip archive before the deadline. Please comment your name **and** student number in at the top of each file.

Introduction

You will be required to implement a *trie* using a graph structure. Since every tree is a graph, but not every graph is a tree, the resulting data structure will be a tree with the addition that connections between nodes contain extra information such as a label.

The Trie will have a root node that never contains a key. If the Trie is empty, then the root node will have no edges going from it. Edges and their labels will be used to distinguish keys from each other.

Figure 1: Trie with the key "RABBIT"



By following edges from the root node, common prefixes can be found. Figure. 1 shows the Trie with one key: "RABBIT". The root node has one edge labeled "R". The edge has a target node of the node containing the key "RABBIT". Since there is not a common prefix the only edge in the Trie is the edge from root with the label "R".

Figure 2: After adding the key "CAT" to the trie

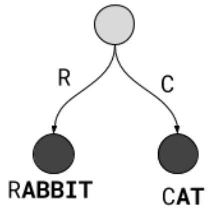


Figure. 2 follows from Figure. 1 with the key "CAT" added. There are no common prefixes so only a single edge from the root is added.

Figure 3: After adding the key "RABIES" to the trie

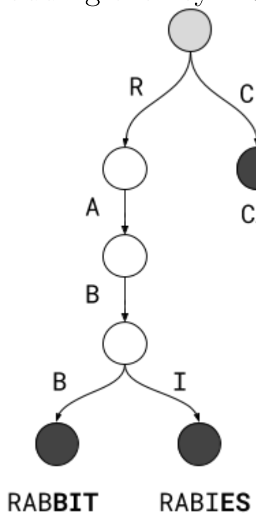


Figure. 3 follows from Figure. 2 with the key "RABIES" added. This time "RAB" becomes a common prefix so nodes and edges are created. Since "RABBIT" and "RABIES" have the same prefix of "RAB", the edges labeled "B" and "I" were created to distinguish the two keys from each other.

For convenience the full key is stored in the key nodes. For efficiency it is possible to store only the suffix. Nodes that contains keys are indicated with the *hasKey* flag and are dark coloured in the figures. The *edges* array should be sorted alphabetically according to the edge label.

For information on tries, see section 7.2 in the textbook. For information on graphs see section 8.1 and 8.3 in the textbook. The trie in this practical will function similar to the trie described in the textbook, however it will be represented using nodes and edges. With the graph representation we lose the random access provided by the implicit relationship between characters in the alphabet and their indices in the array of pointers to sub trees. For the purpose of this practical that compromise is acceptable.

Since tries provide efficient searching and retrieval they are a good data structure for applications such as autocorrect. You are also required to implement a function to return a word from the trie that is closest to a given word in terms in Edit Distance. The edit distance indicates the minimum number of insert/substitute/delete operations to edit a string to be the same as another string.

Task 1: [22]

Implement the following methods in the *Trie* Class according to the given specification:

`void insert(String key)`

Insert the given key into the trie.

`Boolean contains(String key)`

Return true if the key exists in the trie, otherwise false.

`String closestMatch(String str, Integer maxDistance)`

Return the closest string in the trie using Levenshtein distance. Only consider distances that are in the range [0, maxDistance]. If no match could be found in the given range, return null. If multiple strings have the same Levenshtein distance, return the string that comes first alphabetically.

The Levenshtein distance function [1] is provided to you and has the following signature:

`int Distance.LD(String s, String t)`

You may use your own helper functions to assist in implementing the specification. However you may not modify any of the given method signatures.

Submission

You need to submit your source files on the Assignment website (assignments.cs.up.ac.za). All methods need to be implemented (or at least stubbed) before submission. Place all the source files including a makefile in a zip or tar/gzip archive named uXXXXXXXX.zip or uXXXXXXXX.tar.gz where XXXXXXXX is your student number. There should be no folders in your archive. You have 24 hours to finish this practical, regardless of which practical session you attend. Upload your archive to the *2019 Prac 6 - Tuesday* slot on the Assignment website. Submit your work before the deadline. **No late submissions will be accepted!**

References

[1] Michael Gilleland. Levenshtein distance. http://people.cs.pitt.edu/~kirk/cs1501/Pruhs/Fall2006/Assignments/editdistance/Levenshtein%20Distance.htm.