Department of Electronics

Data Structures and Algorithms Assessment

Predictive Text

Y3839090

January 17, 2017

Contents

1 What is a predictive Text system and how do they work

A predictive text system's (also known as auto-complete or word completion) role is to take a partial word and return a list of suggestions. They use some form of dictionary of known words to offer these suggestions.

1.1 How users interact with the program

Users interact with predictive text systems via some kind of text input device. Most users interact with some kind of predictive text system on a daily basis on either their PC or mobile device. The UI for both desktop and mobile devices follow the same basic pattern. The user begins by entering text normally via a keyboard or touch screen and once there are a sufficient number of letters for the system to make a reasonable guess (often two letters), the system presents the user with a list of predicted words. The user is then able to press a key to select which of the suggestions they want to use or continue typing and ignore the systems suggestions. The word they were typing replaces the partial word so that the user can move on the the next word.

1.2 Behaviour of predictive text systems

Predictive text system often return result that fall into a few broad categories.

The input. One of the options in a predictive text system is always to keep the partial word that you already have. This is often achieved by returning the input as one of the items in the suggestion list. This behaviour can also happen if the partial word is actually a valid word from the predictive text systems dictionary.

A word prefixed by the partial word. The most common results from predictive text systems are words that are prefixed by the partial word the user has entered (e.g. "hel" may return ["hel*p*","hel*l*","hel*lo*"]).

A word that shares a prefix with the partial word. (e.g. "applez" may return ["apple","apples","app"])

More sophisticated system may also use frequency analysis techniques, lexicographical distance algorithms and consider context to make smarter suggestions. These are out side of the scope of this project (Our dictionary doesn't contain any frequency data or phrase data).

2 Requirements

Now that a predictive text systems behaviour has been defined, a set of success criteria can be derived.

1. The system must use ANSI C .
2. The system must compile with minGW (gcc)
3. The system must compile and exicute correctly on the university lab machines in PT108.
4. The system must load a word dictionary from file. *(provided file words.txt)*

5. The system must be capable of adding the loaded words into a data structure to store them whilst the program is running
6. The system must store these words in a space efficient data structure.
7. The system must be able to access these words in a time efficient manner.
8. The system must be able to check to see if a partial word is contained in that data structure.
9. The system must be able to check to see if a partial word is a valid word from the dictionary
10. The system must be able to make suggestions of words that are prefixed by a partial word.
11. The system must be able to make suggestions of words that share a common prefix with a partial word.
12. The system must let the user enter a string of text.
13. The system must present the users with suggestions as they are typing (preferred) or the user should be able to trigger a suggestion mode where they will be presented with a set of suggestions.
14. The system should let the user select the one of these suggestions.
15. The system should replace the partial word with the selected word.
16. The system could be extended to deal with punctuation.
17. The system could be extended to deal with Capitalisation.

3 Data Structures

A predictive text program needs a to be able to access a set of common words. Storing and accessing these words efficiently is one of the challenges in creating a fast predictive text engine. I decided to use a trie[2] structure.

3.1 Trie

3.1.1 What is a trie

The trie is a tree-like data structure often used for storing strings. Each node in the trie represents a single letter in a string. Each node in a trie has a fixed number of child nodes like a binary search tree. Unlike the binary search tree the trie does not have two child nodes, it has one for each letter of the alphabet. This allows the trie nodes to not store their value because their position determines it. A node's children share a common prefix, that prefix is the value of there parent node.

There are many features of tries that make them a good choice for a predictive text system. One of them is that values can be look up by their prefixes. Predictive text systems use a

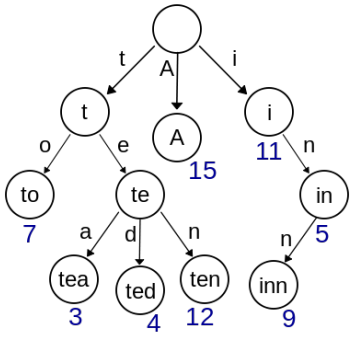

Figure 1: A trie for keys "A","to", "tea", "ted", "ten", "i", "in", and "inn"[1].

partial word (a prefix) to search for possible full words. So it's important that prefix lookups are fast and efficient. [TODO:: ADD MORE STUFF HERE]

3.1.2 Time complexity of a trie

The search time for a value in a trie is $O(len(string))$, where $len(string)$ is the length of the word to look up[3]. This means that searches in a trie are linear with respect to the number of items in the trie. A predictive text function relies upon looks. Every time a user presses a key the system will preform at least one lookup and so it is vital that these searches be fast.

The time complexity of deletion in a trie is also linear. Deletions time complexity is $O(|A|len(string))$, where $len(string)$ is the length of the word to look up and $|A|$ is the size of the alphabet[3]. In the use case as a Predictive text engine deletions will be very rare or non-existent. A simple implementation of predictive text may not allow the user to delete words form the dictionary at all.

The time complexity of insertion in a trie is the same as for deletions. Insertion time complexity is $O(|A|len(string))$, where $len(string)$ is the length of the word to look up and $|A|$ is the size of the alphabet[3]. When used in a predictive text engine most of the insertion into the trie will be made when loading the dictionary from file.

3.1.3 Space complexity of a trie

The space complexity of a trie is $O(|A|\Sigma_i len(w_i))$ where there are strings w_0 to w_n and $|A|$ is the size of the alphabet [3]. This is saying that the space complexity of a trie is equal to the size of the alphabet by the total length of all the strings. Expressed in therms of n this becomes $O(|A| * n * l)$ where $|A|$ is the size of the alphabet and l is the average length of the words. In this form it is easy to see that the space required to store is governed by a $O(n)$ relationship ship.

References

[1] Booyabazooka (based on PNG image by Deco). Modifications by Superm401. *Example of a trie*. [Online; accessed 13-January-2017]. 2006. URL: `https://en.wikipedia.org/wiki/File:Trie_example.svg`.

[2] Peter Brass. *Advanced Data Structures*. Cambridge University Press, 2008. Chap. 8.1, pp. 336–356.

[3] Peter Brass. *Advanced Data Structures*. Cambridge University Press, 2008. Chap. 8.1, p. 341.