Department of Electronics

Data Structures and Algorithms Assessment

Predictive Text

Y3839090

January 18, 2017

Contents

1 What is a predictive Text system and how do they work 1
1.1 How users interact with the program 1
1.2 Behaviour of predictive text systems 1

2 Requirements 1

3 Data Structures 2
3.1 Trie 2
3.1.1 What is a trie 2
3.1.2 Time complexity of a trie 3
3.1.3 Space complexity of a trie 3
3.1.4 Space vs Time 3
3.2 Alternative Data structures 3
3.2.1 Binary Search Trees vs Tries 3
3.2.2 Hash Table vs Tries 4
3.2.3 Array Lists vs Tries 4

4 Testing 5
4.1 Complexity 5
4.1.1 Time Complexity of Tries 5
4.2 Unit Testing 5
4.3 User interactions 5

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

In order to create a predictive text system, their behaviour has to be understood.

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 execute 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 should let the user delete characters with backspace.
17. The system could be extended to deal with punctuation.
18. 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

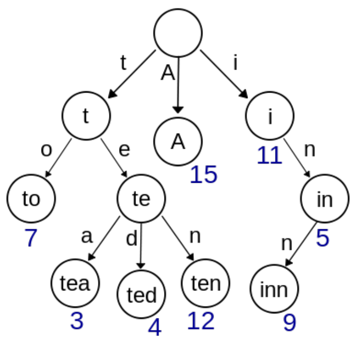

Figure 1: A trie for keys ”A”,”to”, ”tea”, ”ted”, ”ten”, ”i”, ”in”, and ”inn”[1].

be look up by their prefixes. Predictive text systems use a partial word (a prefix) to search for possible full words. So it's important that prefix look-ups 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 gives the trie a $O(1)$ search time. Meaning that searches in a trie do not depend on 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 look up 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.

3.1.4 Space vs Time

The tire data structure sacrifices a good space complexity for time complexity that doesn't depend on the number of strings that it is storing.

3.2 Alternative Data structures

This section details some alternative data structures that were be considered for storing the word list.

3.2.1 Binary Search Trees vs Tries

Both BST (Binary search trees) and Tries are ordered trees. Unlike tries each node in a BST contains a maximum of 2 children. All of a nodes children on the 'left' side have a smaller value than the node and all the nodes on the right have a large value (identical values are normal omitted from BST). BST differ from tries in that each node in a BST stores a value.

When considering space complexity BST can be shown to have a complexity of $O(n)$ [**book:BST:complexity**]. The big-O space complexity of a trie and BST is the same.

When considering search speed Tries are much faster than BST. For a balanced binary search tree the average time complexity of a search is $O(n)$. This is much worse than the trie's $O(1)$ time complexity.

Predictive text systems preform look ups via prefixes. Binary search trees don't offer any particular advantage in this area whilst tries do (All children of a node are prefixed by that node).

It is also worth noting that BST require a lot of comparisons. Comparisons of two strings is not an atomic operation and its worst case time complexity is $O(n)$.

Trie are a better choice for the purpose of a predictive text system.

3.2.2 Hash Table vs Tries

Hash tables are an associative data structure where a key can be used to look up a value and the value is used to generate the key.

Hash tables require the use of a hash function which are not constant time relative to the length of the input string.

The space complexity of a hash map is $O(n)$ which is the same as a trie.

The time complexity of look up in a hash table is $O(1)$ if there are no collisions but could be $O(n)$ in the worst case. In the bast case Tries and Hash tables have the same time complexity to look up a value. In the worst case Tries still have $O(1)$ performance which is better than the hash table.

There are also other notable conditions in which a hash table preforms worse than a trie. Consider attempting to look up a word (e.g. "zoologist") starting with a letter ("z") in a hash table or trie that contain no words starting with that letter ("z"). The hash table would evaluate the hash of that word which takes time proportional to the length of the word. Where as the trie would make a single check and know the word isn't is contained within.

A standard hash table cannot preform prefix based lockups either as similar words produce dissimilar hashes.

3.2.3 Array Lists vs Tries

Array Lists are dynamically realizable arrays. They behave exactly like a standard array, but if an addition/insert is done when the list is already full, it automatically increases its size.

The space complexity of an array list is dependant upon the number of elements in the array and the number of allocated but free elements. This means that the space complexity of an array list is $O(n)$. Array lists and tries have the same big-O space complexity. But an array list will have a smaller space foot print in practice as its space usage $n * l + k$ where as a trie uses $n * l * |A|$, where l is the average word length, k is the unused elements and $|A|$ is the alphabet size.

The time complexity of a search in an array list depends on the searching algorithm. For efficient searches the array must be sorted. So search complexity cannot be considered without considering the complexity or sorting the data.

In the use case of a predictive text system, words will be inserted infrequently but searches are done often. This means that in this use case the time complexity of sorts matter much less than the time complexity of searching. The best time complexity of a sorting algorithms is $O(n * log(n))$ and the time complexity of a binary search is $O(log(n))$. Since the words may only have to be sorted once for hundreds or thousands of searches, in this use case searching will take $O(log(n))$ time. Whilst this is a good time complexity it doesn't math the tries $O(1)$ time complexity.

The time complexity of inserting a value into an array tree is $O(1)$ to insert at the end and $O(n)$ to insert in the middle or beginning.That is if the number of items in the array list is smaller or equal to its size. But if the array has to resize this may not be the case. The performance can be $O(n)$ if the reallocation of the array fails (due to memory fragmentation) and a new array has to created in different area in memory. Because all of the items in the array have to be copied to the new array. The tries time complexity for insertion is constant time and so is better than the array lists linear time.

The time complexity of deletions in a array list is the same as insertions $O(n)$.

Array lists have some advantages over hash maps and binary search trees when it comes to prefix look up. Because the words in an array list would be sorted any words that share a prefix will be guaranteed to be neighbours. Which is similar to the trie.

4 Testing

4.1 Complexity

4.1.1 Time Complexity of Tries

4.2 Unit Testing

4.3 User interactions

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.