

Souvenir's Booth

Using Auto Complete

Abhinav Garg(101303004)
Akshit Arora(101303012)
Chahak Gupta(101303041)
1/12/2015

Submitted To-

Ms. Tarunpreet Bhatia

Acknowledgement

We take this opportunity to express my profound gratitude and deep regards to Ms. Tarunpreet Bhatia for her cordial support, valuable information and guidance, which helped us in completing this task through various stages while I did face many problem regarding my project "Souvenir's Booth". She explained the concepts of algorithmic techniques in an illustrious manner along with providing the necessary guidance through links. This work simply would not have reached up to such extent without her rational guidance.

Lastly, I thank almighty, my family members and friends for their constant encouragement without which this assignment would not be possible.

INDEX

Sr. No.	Title	Page Number
1.	Abstract	3
2.	Introduction	4
3.	Objective	5
4.	Literature Review	6
5.	Algorithmic Techniques Used	6
6.	Code	7
7.	Schema of Database	10
8.	Asymptotic Analysis	11
9.	Working Examples	12
11.	Conclusion	14
12.	Contribution to Society	14
13.	References	16

Abstract

Auto Complete is the most important feature and essential tool when it comes to accessing web search engines such as Google, Yahoo, and YouTube etc. Although it is intensively used on internet. Auto-complete features in many offline activities as well. Finest example can be smart phones. Not only smart phones provide most accurate words but they also predict the words based on the priorities of the users like previous search history. Form the utilities of this features in wide application made us to analyze this feature and implement this feature in most efficient way possible. We implemented auto complete using Trie and analysed the time complexity of this algorithm.

Introduction

Over the previous couple of years, auto-complete has exploded up all over the web. Facebook, YouTube, Google, Bing, MSDN, LinkedIn and lots of new websites all try to complete your phrase as soon as you start typing.

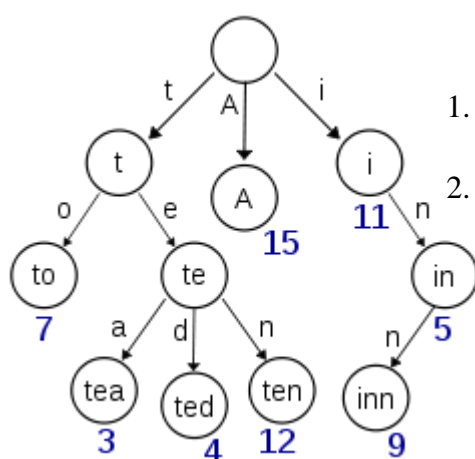
Autocomplete speeds up human-computer interactions when it correctly predicts words being typed. It works best in domains with a limited number of possible words (such as in command), when some words are much more common (such as when addressing an e-mail), or writing structured and predictable text (as in source code editors).

Many autocomplete algorithms learn new words after the user has written them a few times, and can suggest alternatives based on the learned habits of the individual user done by using machine learning.

Here, we implement auto complete in a game. The player gets an ordered set of alphabets called POX. The player needs to use those starting alphabets and make a word. Any word in the English language starting with POX may or may not exist. If there doesn't exist any such word then he has to use the maximum possible string from POX starting from its first letter. If he is not able to make a word with POX but such a word exists in the English language then he is awarded zero points. If he is able to make any such word then he'll be given points according to the letters in the word. Now Sukriti wants his friend Abhinav who is a programmer to make a program that helps him find the possible words so that she can choose from them.

We use a dictionary as data store for storing the words. When we type few letters of a word, we can do a regular expression match each time through the dictionary text file and print the words that starting the letters.

Trie data structure is a tree with each node consisting of one letter as data and pointer to the next node. It uses Finite Deterministic automation. That means, it uses state to state transition.



Words are paths along this tree and the root node has no characters associated with it.

1. The value of each node is the path or the character sequence leading up to it.
 2. The children at each node are ideally represented using a hash table mapping the next character to the child nodes.
- It's also useful to set a flag at every node to indicate whether a word/phrase ends there.

The insertion cost has a linear relationship with the string length. Now let's define the methods for listing all the strings which start with a certain prefix. The idea is to traverse down to the node representing the prefix and

from there do a breadth first search of all the nodes that are descendants of the prefix node. We check if the node is at a word boundary from the flag we defined earlier and append the node's value to the results. A caveat of the recursive approach, is you might run into stack depth problems when the maximum length of a string reaches tens of thousands of characters.

Objectives

- 1) To optimize auto-completion of the word.
- 2) To optimize Depth first search by improving the traversal of data structure by special algorithm and analysis techniques.
- 3) To Search Word in the Trie.
- 4) Modify the default Structure of Trie and with certain tweaks improve the overall time complexity.
- 5) We substantially increased the number of word in the Trie from 4 to 45000 words and it worked correctly.

Literature Review

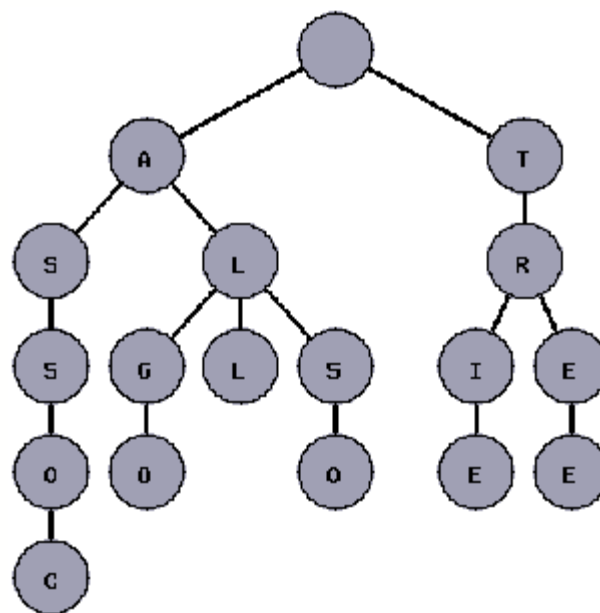
There are many algorithms and data structures to index and search strings inside a text, some of them are included in the standard libraries, but not all of them; the trie data structure is a good example of one that isn't.

Let word be a single string and let dictionary be a large set of words. If we have a dictionary, and we need to know if a single word is inside of the dictionary the tries are a data structure that can help us. But you may be asking yourself, "Why use tries if set <string> and hash tables can do the same?" There are two main reasons:

The tries can insert and find strings in $O(L)$ time (where L represent the length of a single word). This is much faster than set, but is it a bit faster than a hash table.

- The trie is a tree where each vertex represents a single word or a prefix.
- The root represents an empty string (""), the vertexes that are direct sons of the root represent prefixes of length 1, the vertexes that are 2 edges of distance from the root represent prefixes of length 2, the vertexes that are 3 edges of distance from the root represent prefixes of length 3 and so on. In other words, a vertex that are k edges of distance of the root have an associated prefix of length k .
- Let v and w be two vertexes of the trie, and assume that v is a direct father of w , then v must have an associated prefix of w .

The next figure shows a trie with the words "tree", "trie", "algo", "assoc", "all", and "also."



Note that every vertex of the tree does not store entire prefixes or entire words. The idea is that the program should remember the word that represents each vertex while lower in the tree.

Algorithmic Techniques

- For printing all possible words in a trie with given root: Backtracking Traversal (similar to DFS)
- For searching the prefix in trie: naïve algorithm used
- For insertion of new words in trie: Recursion

Code

```
#include<iostream>
#include<fstream>
#include<string>
using namespace std;
//Global variables (for the purpose of flags) =>
//found: to indicate whether the word is found or not
//len: to indicate user specified length
int found=0,len=-9999;

class node{                                     //Node class
public:
    char info; //the letter corresponding to node is stored here
    string Word;
        //for storing words (if completed) till that node
    class node* ptrs[256];
        //to hold the pointers to 256 next possible letters (ASCII)
    node() {
//node class constructor
        for(int i=0;i<256;i++){
            ptrs[i]=NULL;
        }
        info=NULL;
        Word="";
    }
};

void insertword(string word,int pos,class node * root){
//function to insert word
    if(word.length()==pos){
//if last position is reached
        root->Word=word;
//Put the final word into node
        return;
    }
```



```

        if( root-> ptrs[word[pos]]==NULL ){
//if next letter is not found
            node *newnode;
//create new node
            newnode= new node;
            newnode->info=word[pos];
//with info = letter to store
            root->ptrs[word[pos]]=newnode;
//pointer from current letter to next letter's node
            insertword(word,pos+1,root->ptrs[word[pos]]);
//call insert word function again with next position
        }
        else
            insertword(word,pos+1,root->ptrs[word[pos]]);
//if next letter to add already exists
    }
//call insert word directly

void printall(class node * root){
//function to print all possible words given root of trie
    for(int i=0;i<256;i++){
//for all 256 pointers in the provided root node
        if(root->ptrs[i]!=NULL){
//if those pointers are not null
            printall(root->ptrs[i]);
//call printall recursively at each one of them
        }
//Similar to DFS traversal

        //following code will be executed when the recursion starts
        backtracking
        if(root->Word != "" && (root->Word.length()==len && len!=-9999))
//first user specified length words will be searched
            cout<<" -> "<<root->Word<<endl;
//and printed if they exist
        else if(root->Word != "" && len== -9999)
//otherwise, all matching words are printed
        {
            cout<<" -> "<<root->Word<<endl;
            found=1;
        }
    }

}

void suggest(string key,int pos, class node * root){
//function to print suggested words

```

```

        if(root->ptrs[key[pos]] != NULL){
//if node is the last position of given key
            suggest(key,pos+1,root->ptrs[key[pos]]);
//call suggest with next position
        }
        else{
            printall(root);
//when last node, print all words below it
        }
    }

int main(){
    ifstream in("wordlist.txt");
//input file "wordlist.txt"
    string word,current="",key;
    char ch;
    node *root;
//root node of trie defined
    root = new node;
    while(in){
        in>>word;
//file parsed and all words input to trie
        insertword(word,0,root);
    }
    in.close();
//file stream closed
    cout<<endl<<"Trie has been created successfully!"<<endl;
//trie created
    cout<<"Enter the starting letters of the word : ";
    cin>>key;
//input key from user
    cout<<"Do you know the length of the word?(y/n)";
//input required word length(if any)
    cin>>ch;
    sos:
    if(ch=='y')
    {
        cout<<"Enter the length\n";
        cin>>len;
        if(len<=0) {
//if user enters incorrect string length
            cout<<"Please enter a length greater than 0.\n";
            goto sos;
        }
    }
}

```

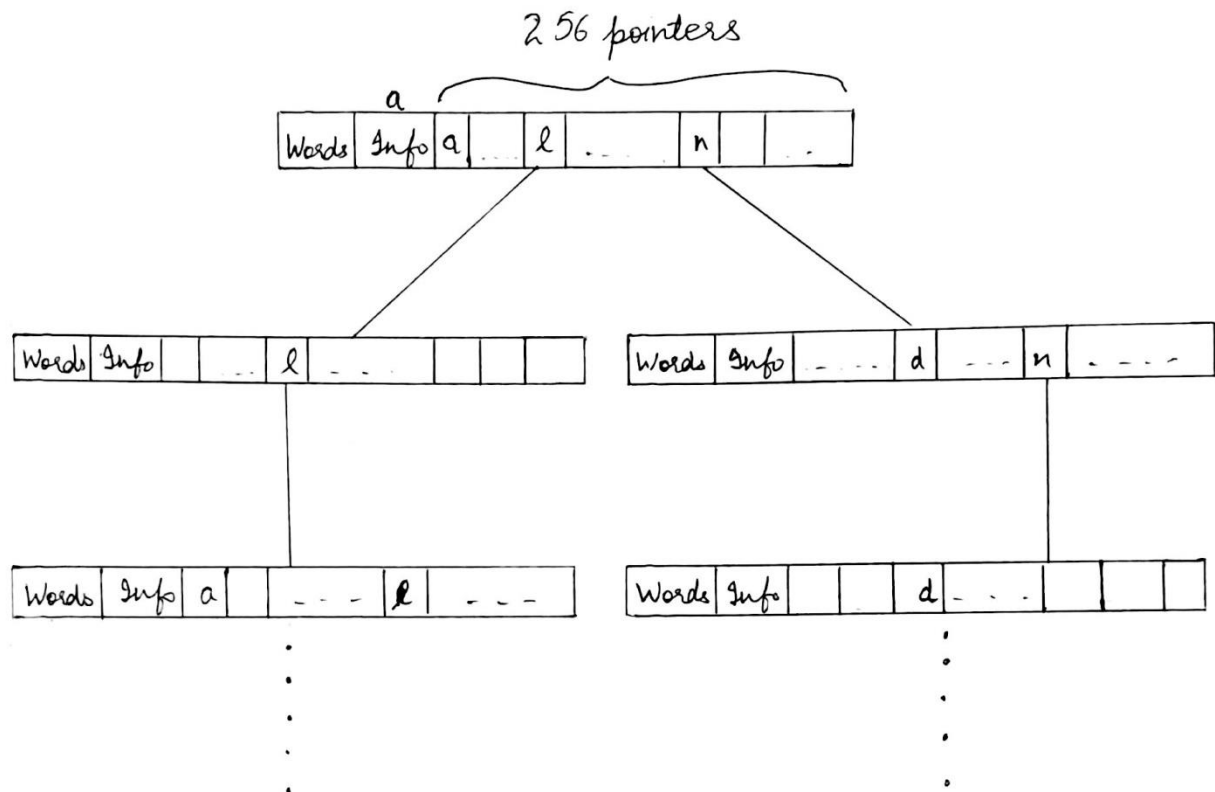
```

        cout<<endl<<"Possible suggestions are :"<<endl;
        suggest(key,0,root);
//suggest function call
        if(found && ch=='y')
        {
            cout<<"No words of specified length found";
//if no string found
        }
        return 0;
}

```

Schema of Databse

It is the file-based database which used to store words for the Trie. Initially we took 1000 words and inserted them into the Trie. The words in the file can be of same prefix or different prefix. As we optimized the code we could store up to 45000 words in the Trie.



Space Complexity

In worst case analysis, the dictionary will consist of words with no common prefix. At every level, 256 combinations will be possible from each node. So this will result in a lot of memory usage.

Maximum number of nodes possible at level 1= 256

Maximum number of nodes possible at level 2= 256^2

Maximum number of nodes possible at level 3= 256^3

Maximum number of nodes possible at level 4= 256^4

.

.

.

Maximum number of nodes possible at level $m=256^m$

Maximum total number of nodes in a trie with m as maximum length of word is:

$$256+256^2+256^3+256^4+\dots\dots\dots 256^m$$

$$= (256(256^m+1 - 1))/(256-1)$$

$$= (256/255) (256^{m+1} - 1)$$

$$= O(256^m)$$

So trie will have exponential space complexity in worst case complexity.

Time Complexity

The key determines trie depth. Trie will have linear time complexities in all the cases. So it is considered to be the most efficient algorithm for implementing trie data structure in spite of large space complexity.

Using trie, we can search / insert the single key in $O(M)$ time. Here, M is maximum string length.

In this project, we get the search result in $O(\text{key_length} + \sum(L))$, where key_length is input given by user and $\sum(L)$ is summation of all the string lengths starting with prefix entered by user.

Working Examples



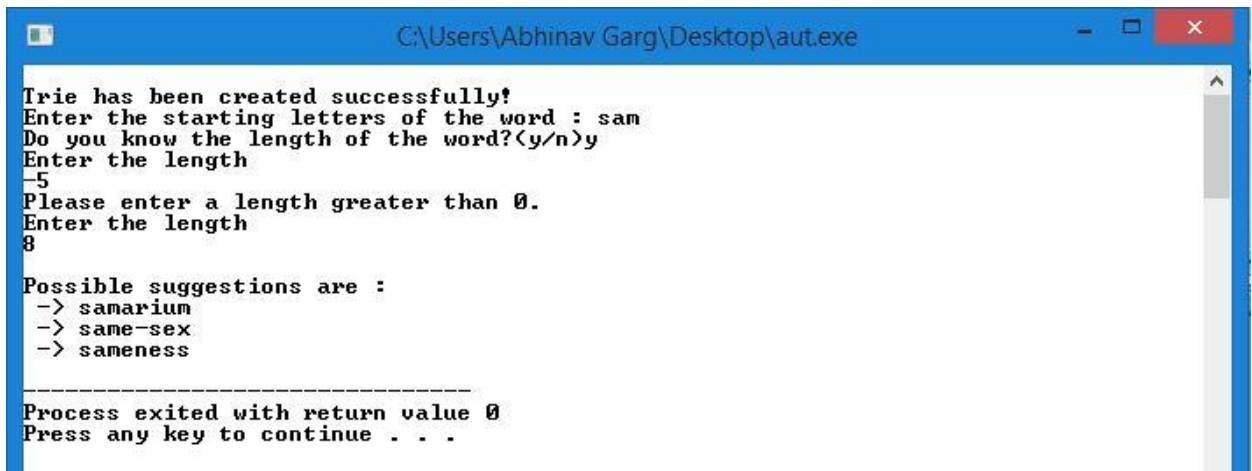
```
C:\Users\Abhinav Garg\Desktop\aut.exe

Trie has been created successfully!
Enter the starting letters of the word : abh
Do you know the length of the word?(y/n)n

Possible suggestions are :
-> abhorrence
-> abhorrently
-> abhorrent
-> abhor

-----
Process exited with return value 0
Press any key to continue . . .
```

Figure 1: Output when user doesn't know length.




```
C:\Users\Abhinav Garg\Desktop\aut.exe

Trie has been created successfully!
Enter the starting letters of the word : sam
Do you know the length of the word?(y/n)y
Enter the length
-5
Please enter a length greater than 0.
Enter the length
8

Possible suggestions are :
-> samarium
-> same-sex
-> sameness

-----
Process exited with return value 0
Press any key to continue . . .
```

Figure 2: Output when user inputs non-positive length



```
C:\Users\Abhinav Garg\Desktop\aut.exe

Trie has been created successfully!
Enter the starting letters of the word : ret
Do you know the length of the word?(y/n)y
Enter the length
5

Possible suggestions are :
-> retch
-> retie
-> retro
-> retry

-----
Process exited with return value 0
Press any key to continue . . .
```

Figure 3: Output when specified length matches

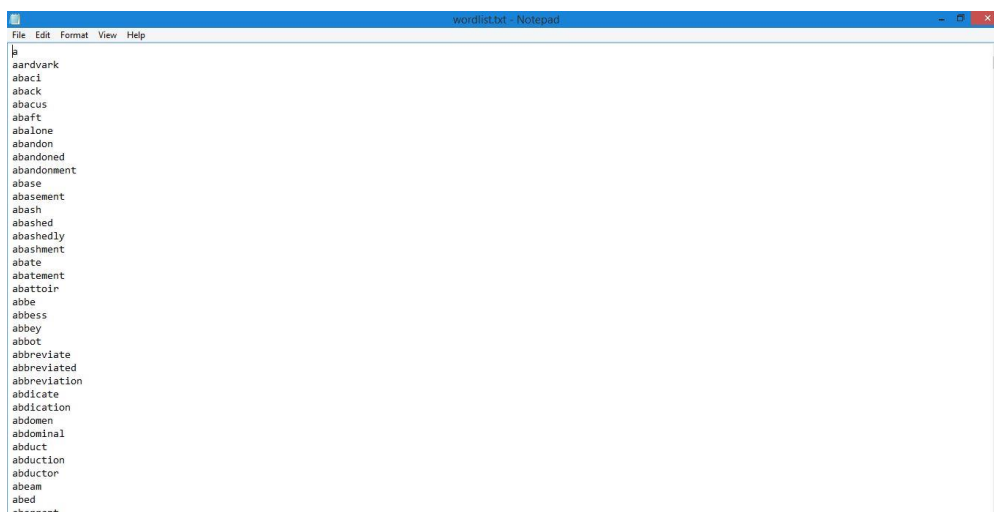


Figure 4: The dictionary

Conclusion

- Sukriti's problem of finding the desired solution was solved.
- Auto complete was successfully implemented using Trie.
- Suggests words if words of desired length are not available.
- DFS and Trie's implementation was understood.

Contribution to Society

Auto Complete speeds up human-computer interactions when it correctly predicts words being typed.

It works best in domains with a limited number of possible words (such as in command line interpreters), when some words are much more common (such as when addressing an e-mail), or writing structured and predictable text (as in source code editors).

Works for:

1. Web Browsers
2. E-Mail Programs
3. Search Engines
4. Source Code
5. Word Processors
6. Command-Line Interpreters

References

- <http://v1v3kn.tumblr.com/post/18238156967/roll-your-own-autocomplete-solution-using-tries>
- <https://www.topcoder.com/community/data-science/data-science-tutorials/using-tries/>
- <http://igoro.com/archive/efficient-auto-complete-with-a-ternary-search-tree/>
- <https://en.wikipedia.org/wiki/Autocomplete>
- <https://en.wikipedia.org/wiki/Trie>
- <http://www.geeksforgeeks.org/trie-insert-and-search/>