Trie and its implementation in Auto-Correction and auto-complete

November 7, 2022

Nishad Dhuri (2021CSB1116) ,
Vipul Patil (2021CSB1141) ,
Darshan Kawale (2021CSB1080)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Monisha

Summary: This project involves a simple auto-correct algorithm, with optimal searching time using trie. A dictionary of words is being stored in a trie. The main purpose of using a trie is to reduce the searching time for any word in the dictionary. The simple autocorrect algortihm makes at most 2 edits to the user inputted word, and then arranges the valid words in increasing order of priority in a array. The last element of the array is the most probable suggested correction. We are using a basic 1-gram model. The auto-complete algorithm simply looks for the entered word in the trie, till the substring is present, and from there it traverses down all of the children and prints all valid words encountered.

---

1. Introduction

Spellchecking is the task of predicting which words in a document are misspelled. Correction is the task of substituting the well-spelled hypotheses for misspellings. Spellchecking and autocorrection are widely applicable for tasks such as wordprocessing and postprocessing Optical Character Recognition. Autocorrect is widely used in word processing softwares like Microsoft word. Auto-complete is used in mobile keyboards for predicting the complete word before it is fully typed. In our approach, we have used the Trie data structure to store all the words from a large text file. We are using Trie because of the time complexity for searching a trie. A trie can search for a string in O(L) time, where L is the length of the string, whereas a binary search tree will take O(L*logN) time, where N is the total number of words in the trie. In our case, N is very huge, hence trie provides a clear advantage.

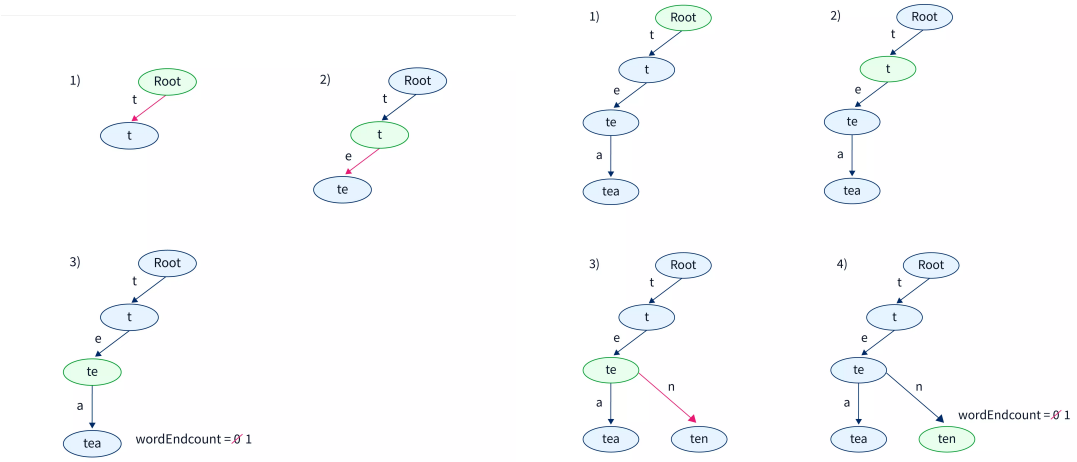
2. Explanation

Trie is a type of k-ary search tree used for storing and searching a specific key from a set. Using Trie, search complexities can be brought to optimal limit (key length). The trie node consists of an array named children of pointers to nodes of size 26, a bool value which indicates if that node is end of word or not, an integer frequency which indicates how many times the word occurs in the given text file.It also contains an int used,which indicates if the word has already been checked for autocorrect. Each word from the text file is scanned and stored in the Trie using the insert function(working explained in analysis). For Auto-Complete- Then the program reads user input and calls checker function. If the word is present in Trie, the checker function returns true,otherwise it calls printAutoSuggestions.It passes the pointer of the node where the last letter in the query string is present.Now, printAutoSuggesstions traverses down all the possible children of that node,recursively calling itself till it reaches an end of word.Then, it will print the new word where the recursion stopped. For Auto-Correct- We will check if the word entered by user is present in Trie or not.If not, we will call edits1 function. This function will generate all possible words obtained by making one change to the input word(deletion,insertion,transpose or replace).The it will pass all these words to select-suggesstions() function.Then the edits1 function will call itself again with all the newly generated words as parameters.This will give us all the words obtained after making two changes to the input word. These are again passed to the select-suggestions function. The select-suggestions function chooses all the words which are actually present in the dictionary, and then chooses the ones with the

highest frequency in the file.It then arranges the word in a array of specified size in increasing order of their frequencies(it also calls a sorting function for this purpose).

3. Analysis

The time complexity for storing all the words in the trie is O(N*avgL)(avgL is the average length of the words in the trie). The time complexity for searching and inserting for a string in Trie is O(L). We can explain these time complexities with the help of some examples.[2]
INSERTION -



(a) Inserting "tea". (b) Inserting "ten".

Figure 1: Insertion in trie

Inserting "tea"-
Figure 1[a] shows the steps for insertion of the string tea in the Trie. In every step, the node in which the operations are done is marked with green. We will refer to this node as the current node. 1. The root node was created. Every index of the children pointer array is NULL. The current node initially will be the root node. The first character of the word to be inserted is t. In the current node, we check whether the t-a th index of the children array is NULL or not. Right now the t-a th index of the children array is NULL. We will now point this index to a new node created using allocate-node() function. We will make the new node created as the current node. Since letter t is not the end of the string tea we will not set isEOW as true for current node.
2. The second character of the word is e. In the current node, we will check whether the e-a th index of the NULL or not. As e-a th index of the childNode array is NULL we will point it to a newly created node. We will make the new node as current and proceed to the next step. As the string te is not the end of the string tea we will not set isEOW of current node to true.
3. In step 3 we will move to the third character of the input string, i.e a. Since the 0th index of current node is NULL, we will point to a newly created node. We will make the new node the current node. However, as the string tea is the end of the string tea we will set isEOW to true for the current node.
Inserting "ten"-
Figure 1[b] shows the steps for inserting ten in the trie
1. Initially, the current node is the root node. The first character of the input string is t. As the t-a th index of the children pointer array is not NULL we will make it the current node.
2. Now we check for e. As the e-a th index of the current node is also not null we will set is as the current node.
3. Now we check for n. As the n-a th index of the current node is NULL we will point it to a newly created node, and set it as the current node. Since n is the last character, we will set isEOW to true for the current node.
SEARCH-

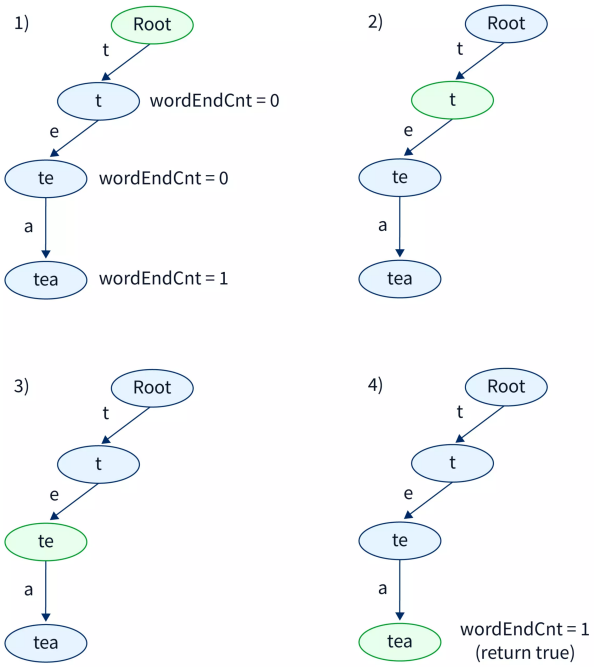
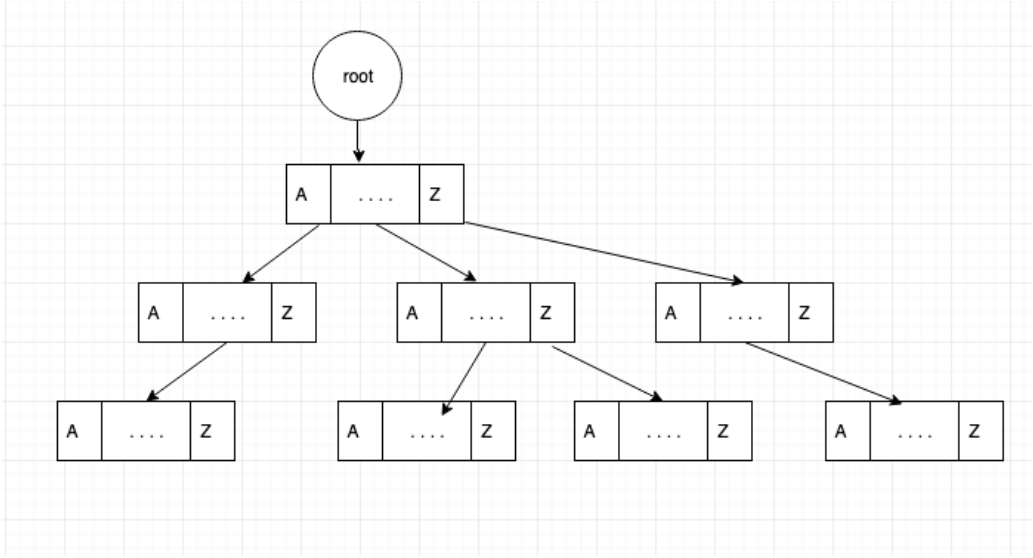


Figure 2: Search operation:"tea".

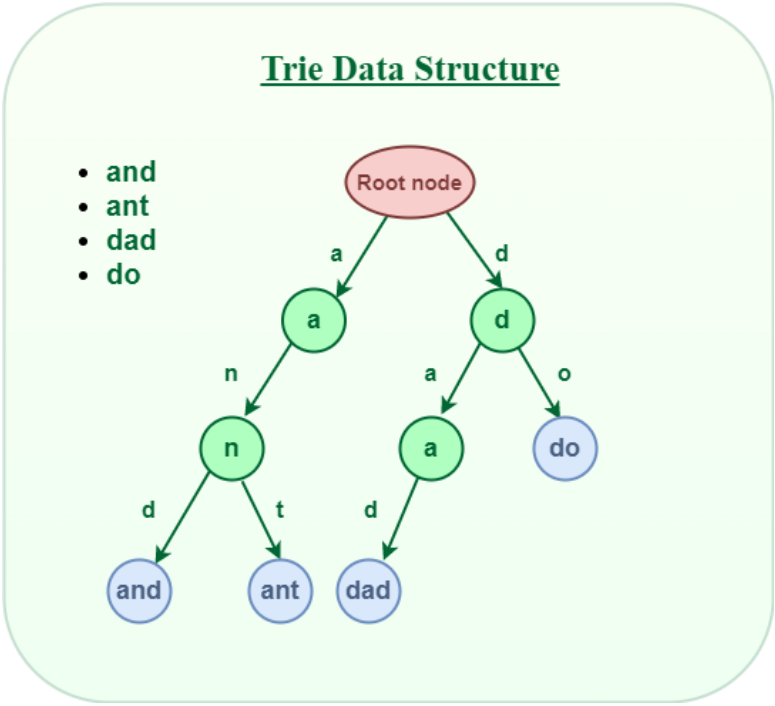
Search operation: "tea"
Figure 2 shows the steps involved in searching the string tea in the Trie. 1. Initially, the current node is the root node. The first character of the query string is t. Since the t-ath index of the children pointer array is not NULL, we will set it as the current node.
2. Now we check for e. As the e-a th index of the current node is also not NULL, we set it as the current node.
3. Then, we check for a. As the a-a index of the current node is not equal to NULL, we set it as the current node.
4. Since a is the last character of the string , we will check isEOW in the current node. Since it is true, the word is present in the string.
Search operation: "to"
1. We start with the root node and check for the first character of the query string, i.e t. As the t-a th index of the array of children pointers is not NULL, we set it as the current node.
2. In step 2 we look for the character o in the current node. However, as the o-a th index of the current node is equal to NULL. This means that the query string to is not present in the trie. Thus we return false in this case.
As we can see, the time complexity for searching or insertion is O(L), where L is the length of the string. When inserting "tea" we first visited the root node, then we went to its t-a th index child , then to the childs e-a th index child and similarly to the 0th index. We visited 4 nodes(3+1). For a string of length L we will first visit the root node, and then the subsequent children corresponding to the letters. Hence, time complexity will be O(L). Similarly, for search we will traverse the Trie in a similar manner ,hence the time complexity is same.
However, the major drawback of Trie is it's space complexity.This is because for every node, it is allocating space for all 26 children. The space complexity of Trie is 0(26*avgL*N), where N is total number of words, avgL is the average length of words.

4. Figures, Tables and Algorithms

4.1. Figures



(a) Every node has 26 children.



(b) Visual representation of stored data.

Figure 3: Trie

4.2. Tables

Time complexities

	Average Case	Worst Case	Best Case
Insertion	O(N)	O(N)	O(N)
Deletion	O(N)	O(N)	O(N)
Searching	O(N)	O(N)	O(1)

Table 1: Time complexities for different functions[3]

4.3. Algorithms

The algorithms used for creating a trie and searching are as follows[1]-.

Algorithm 1 Insert(Node* root,char* key, int lenstr)

```
    The insert function will take the pointer to the root node, pointer to the string to be stored and
    length of the string as arguments
 2: height=0
    index=0
 4: traverse=root
    for height < lenstr do
 6:   if key[height] == Nullcharacter then
        Break out of loop
 8:   end if
      index=char2index(key[height])
10:   if traverse! = NULL then
        if !traverse− > children[index] then
12:       traverse->children[index]=ALLOCATE-NODE()
        end if
14:     traverse=traverse->children[index]
      end if
16: end for
    if !traverse then
18:   traverse->is_EOW = truetraverse− > freq + +
20: traverse->used=0
    end if
22: Return root
```

Algorithm 2 Search(Node* root,char* key,int strlen)

1: The search function takes the pointer to root node, pointer to string and length of string as arguments
2: index=height=0
3: traverse=root
4: **for** $height < strlen$ **do**
5: index= CHAR-TO-INDEX(key[height])
6: **if** $!traverse->children[index]$ **then**
7: return false
8: **end if**
9: traverse=traverse->children[index]
10: **end for**
11: return traverse->is$_EOW$

The algorithms used for Auto-correct are mainly edits1, and select suggestions.[4] Edits1 makes one change to the user entered word, which is done by the four methods, deleting a letter,inserting new letter, swapping two letters or replacing a letter with another. All the modified words are again passed to edits1, hence we are effectively checking all the words at 2 edit distance away from the entered word. The select suggestions function chooses the correct words generated by edits1 and puts them in an array in a sorted manner based on their frequency.

Algorithm 3 edits1(Node* root,char s[],int strlen, int track)

1: Track is for knowing if we are finding edit1 or edit2 words.
2: i=0
3: **for** $i < strlen + 1$ **do**
4: s1[]=first i characters in s
5: s2[]=last strlen-i characters in s
6: string delete,transpose,replace[],inserted
7: delete= s1 +entire s2 except the first element
8: **if** strlen-i>=2 **then**
9: transpose= s1+interchange first two elements of s2
10: **end if**
11: j=0
12: **for** j<26 **do**
13: replace[j]= s1+replace first letter of s2 with character at jth position from a
14: selectsuggestions(root,replace[j],strlen,track)
15: **end for**
16: j=0
17: **for** j<26 **do**
18: inserted[j]= s1+ character at jth position from a +s2
19: selectsuggestions(root,inserted[j],strlen+1,track)
20: **end for**
21: **end for**
22: return

Algorithm 4 selectsuggestions(Node* root,char* s,int lenstr,int track)

```
MAX-SUGG, track-sugg and suggested-words[MAX-SUGG] are global variables
if track-sugg<MAX-SUGG then
    p=get-count(root,s,lenstr)
    if track==0p!=0 then
        p=(p/100)+1
    end if
    if p!=0  !get-used(root,s,len-str) then
        suggested-word[track-sugg]=s
        resp-probabilities[track-sugg]=p
        track-sugg++
    end if
    update-used-value(root,s,len-str)
    if track-sugg==MAX-SUGG then
        Sort resp-probabilities in ascending order and sort suggested-words in same sense
    end if
    return
else
    p=get-count(root,s,lenstr)
    if track==0p!=0 then
        p=(p/100)+1
    end if
    if !get-used(root,s,lenstr) then
        if p>resp-probabilities[0] then
            resp-probabilities[0]=p
            suggested-words[0]=s
            Sort resp-probabilities in ascending order and sort suggested-words in same sense
        end if
        update-used-value(root,s,len-str)
    end if
end if
Final instructions
```

Next is autocomplete. The functions used for autocomplete are mainly printAutoSuggesstions, and checker. Checker is just a modified version of search which just gives the location of the last node when the string is being searched. If word is present then it will return true, otherwise it will return false and call printAutoSuggestions.[5]

Algorithm 5 Checker(Node* root,char *key,int strlen)

```
temp=root
height=0
for height<strlen do
    index=CHAR-TO-INDEX(key[height])
    if temp->children[index]==NULL then
        printAutoSuggestions(temp,key,height)
        return false
    end if
end for
if temp->is-EOW then
    return true
end if
printAutoSuggestions(temp,key,strlen);
return false
```

Algorithm 6 printAutoSuggesstions(Node* root,char* s,int len-str)

```
1: if count-sugg<=MAX-SUGG then
2:    i=height=0
3:    traverse=root
4:    if traverse->is-EOW then
5:       print(s)
6:       count-sugg++
7:    end if
8:    for i<26 do
9:       if traverse->children[i]!=NULL then
10:         temp=s
11:         s+= 'a'+i (Character at ith index from a)
12:         printAutoSuggestions(traverse->children[i],s,lenstr+1)
13:      end if
14:      i++
15:   end for
16: end if
17: return
```

5. Conclusions

We are using a trie because of the advantage it provides in terms of time complexity. The trie will require O(L)time to search for a string(where L is the length of the string),whereas a binary tree would take O(logN) time, where N is the total number of words stored. For a dictionary ,N will be very large, hence Trie is the best choice. However the main disadvantage of using a Trie is it's space complexity.The storage requirement of a trie is O(26 * L * N). Ternary search trees and hash-tables have better space-complexity.Hence ternary search trees can be better suited for large data files due to their space-efficiency.

6. Bibliography and citations

The following are the sources we used to make this project: `bibliography.bib`.

Acknowledgements

It is our esteemed pleasure to present a project on "Auto correct and auto complete using Trie data struture". Any achievement bigger and small should have a catalyst and constant encouragement and advice of valuable and noble minds. We would like to thank our professor Dr. Anil Shukla and our mentor Monisha mam for their guidance and help in our project.

References

[1] dharanendralv23. Spell checker using trie. `https://www.geeksforgeeks.org/spell-checker-using-trie/`, 2021.

[2] Rishi Raj Kalita. Trie data structure. `https://www.scaler.com/topics/data-structures/trie-data-structure/`, June 2022.

[3] Ue Kiao. Time and space complexity of trie. `https://iq.opengenus.org/time-complexity-of-trie/`.

[4] Peter Norvig. How to write a spelling corrector. `https://norvig.com/spell-correct.html`, February 2007.

[5] Hemang Sarkar. Auto-complete feature using trie. `https://www.geeksforgeeks.org/auto-complete-feature-using-trie/`, February 2022.