CS512 Spell Checker - Spring 2019

Desai, Arthkumar
Jin, Dapeng
Shi,Mo
Luo,Wenhao

In this project we implemented a spell-checker program that ①helps the user to check if all input words are correctly spelled and ②give recommendation for incorrect words based on the Trie-tree data structure. We first constructed our Trie-tree using the Kaggle Urban Dictionary. Then for each word not existed in the tree, we implemented an algorithm called Edit-Distance to find the most possible words that the user probably wanted. In this way spell checking and word recommendation are achieved successfully and conveniently.

I. PROJECT DESCRIPTION

Spell checking is generally a very useful function in many different cases for texting. For instance, sending an email, writing an essay, etc. In daily usage we've already seen some mature implementation like Google Doc and Microsoft Word, but what if it's not supported for some email systems or text editors? That's where our project idea is originated: to extract the spell-checking function out as an independent program that can be used separately with any text editor. In this case, if the spell-checking function is not supported originally, our program would be helpful.

To achieve this, we need our program to be efficient and capable – the program needs to return the expected results correctly in a short time. So among different ways of implementing spell-checker, we chose a data structure called Trie-Tree, in which each path from the root (which is empty) to a leaf will be one particular word. In this way, all words could be traversed from the root and any word not in the tree will be considered as incorrectly spelled. Using the Trie-Tree structure this searching procedure can be achieved very fast, which guarantees our requirement of efficiency.

Besides spell-checking, we also want our program to have the ability of correcting words for the user, so we need to predict which word/words have the maximal possibility being the correct answer for the user. To achieve this, we implemented an algorithm called Edit-Distance, which computes the minimum number of letters that need to be changed to get the correct word. For any incorrect word, we'll compute this value for any word in the tree that has the same prefix with the input, and return the top 5 words with the smallest value. Finally the user can pick up his desiring answer from the list returned.

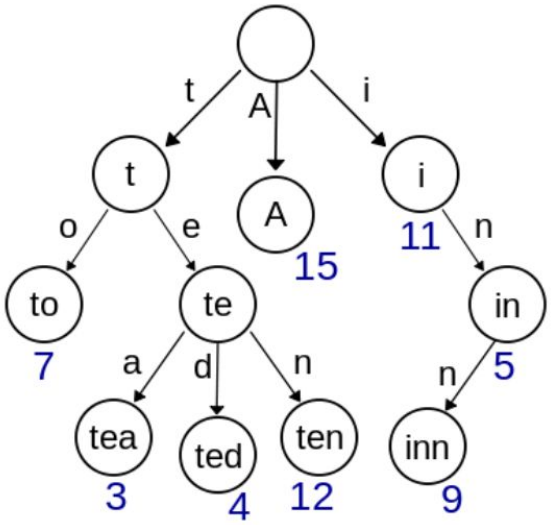
There were main blocks we faced during the implementation. The first one is if the prefix is too short or wrong, the program may not be able to return a list with the correct answer in it (or the list may be too large since the prefix is too short). The second one is the performance of our program is strongly related to the quality of the training data set. We'll specify each of them in the following sections.

The project majorly contains the following stages: finding the proper training data set, constructing the Trie-Tree & implementing the Edit-Distance Algorithm, and estimation. For each stage we used a week to achieve our milestone, and for the last week we implemented a user interface to put our program into good shape for using.

A. Stage1 - The Requirement Gathering Stage.

We want to build a spell checker application using a Trie Tree algorithm. The spell checker is a very useful tool for the applications when someone is writing an email, commenting on social networking sites like facebook or instagram. Spell checker can be incorporated into many such applications where user wants to write something. For example, when I was writing this description, I had a hard time remembering the spellings but spell checker in google docs helped me in saving time.

There are multiple ways of implementing spell checking. In our program we choose to achieve it with a Trie Tree, which is a tree structure that contains letters as nodes. Starting from the root node to a leaf, each branch represents a specific word string. For example look at the following tree.



The picture above shows a trie tree with 6 branches that represent 6 different words. From the root node (which should always be empty) if we connect all letters one by one from the top to the bottom, we'll get a specific word. Notice that some words start with the same letter/letters, so parts of their branches will be in common.

Why we want to use Trie Tree algorithm? When there are other solutions using conventional data structure. Let's consider the following solutions and analyse the complexity.

Using List.

- Sort words and store them into list.
- We can build the customized binary search to search the word
- Analysis → sort words take O(N Log(N))
    - Traverse words to store them takes O(N) as appending take O(N)
    - Customized Binary Search → O(M log(N)), where M is the length of the term.
    - Overall, the worst time is O(N log(N)) and Space Complexity O(N)

Using Dict/Sets

- As dicts uses the hashing function, the storing and lookup of a work takes constant time O(1). But, if we want to make suggestions then this algorithm will not work efficiently. It requires the whole word in the hash table.

Our Trie Tree

- Our implementation of the trie tree uses dict. Hence, the time complexity for the lookup of a particular letter is O(1). Also, if the word size is m, then the time complexity of traversing the word is O(m).

With the trie tree implemented at the back-end, at the front-end there will be an interface for different types of users to interact with our program. Basically our interface will contain an area for the input contents, a button to get the output, and another area to display the output. In this way we're able to show the results for users concisely.

Working Scenarios:

For our program there could be three types of users, which are customer, administrator, and builder. A customer is just the person who makes use of our program for specific functions. An administrator is the person who can check the current dictionary built by our trie tree and update it. And the builder has the accessibility of building a new trie tree based on different data resources, like a new version of dictionary. For each type of user we come up with two scenarios where the user interact with our program.

1. **Customer**

Scenario 1:

- **Description:** A person is currently trying to write a cover letter for his job application, which requires high level of precision in words. If checked word by word manually, with some possibility that the applicant couldn't find some of the mistakes or just simply neglected them. That's where our program could help.
- **System Data Input:** The whole contents of the letter which can be treated as a long string that contains different words.
- **Input Data Types:** Long string with letters and other contents (white spaces, punctuations and possibly numbers).
- **System Data Output:** A list of incorrectly spelled words.
- **Output Data Types:** List of strings.

Scenario 2:

- **Description:** A user is trying to make some comments on social websites like Facebook or Instagram and he suddenly forgot how to spell the word "sufficiently". What he does remember is that the word he wants to use starts with "suffi", but he just cannot remember the rest part. In this case he could use our program to find the word he's possibly looking for.
- **System Data Input:** The prefix of the user's target word.
- **Input Data Types:** String that only contains letters.
- **System Data Output:** A list of all words starting with the prefix input by the user.
- **Output Data Types:** List of strings.

2. **Administrator**

Scenario 1:

- **Description:** The administrator wants to check if a particular word / several words are in the current trie tree or not.
- **System Data Input:** A single word or a list of target words.
- **Input Data Types:** String or a list of strings. (If it's a single string it can be transformed to a list with one single element automatically)
- **System Data Output:** Results of the existence for each target word.
- **Output Data Types:** Bool variable(s) or strings that represent the existence of the words.

Scenario 2:

- **Description:** Recently a new word is getting popular on the Internet that is created by someone online and is not included in any existing dictionary. Our administrator wanted to find if the word exists in the current tree. If not, then add this word to the tree automatically.
- **System Data Input:** The word that the administrator want to add.
- **Input Data Types:** A string of letters.
- **System Data Output:** The results that the word is added to the tree successfully / the word already exists.
- **Output Data Types:** Strings correspond to the results above.

3. **Builder**

Scenario 1:

- **Description:** The builder wants to build a new version of spell checker based on another dictionary.
- **System Data Input:** The file that contains the whole new dictionary.
- **Input Data Types:** .csv file / .lsx file
- **System Data Output:** The result of building the new tree successfully.
- **Output Data Types:** String corresponds to the result above.

Scenario 2:

- **Description:** The builder wants to update the current spell checker based on the updated version of the original dictionary.
- **System Data Input:** The file that contains the updated version of dictionary.
- **Input Data Types:** .csv file / .lsx file
- **System Data Output:** The result of updating the tree successfully.
- **Output Data Types:** String corresponds to the result above.

Timeline & Labor Division

Our project consists mainly 4 stages:

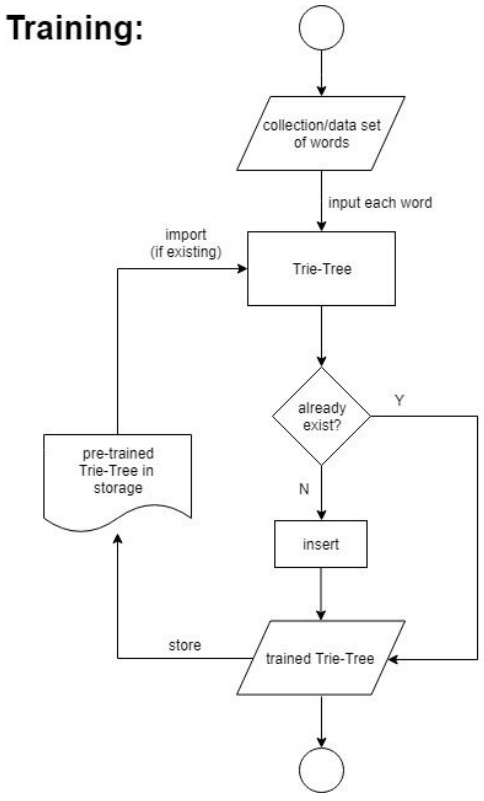
1. Training data sets finding
2. Program/Algorithm implementation
3. User interface designing
4. Testing and evaluation

The implementation will be achieved stage by stage and each stage will be completed within one week. Stage 2 and stage 3 can be done simultaneously with one member dealing with the interface, one working on the program structure and one implementing the algorithm. As for stage 1 and stage 4 all members will work together as well as the documentation and presentation.

B. Stage2 - The Design Stage.

Based on the scenarios described in stage 1, we'll demonstrate our program with data flow diagrams focus on two aspects: the training part and the testing/using part – which is the program's main function of spell checking and word recommendation. The training flow diagram describes the procedures for the administrator to build the tree structure based on any specific training data, while the testing/using flow diagram describes the underlying flow of data processing after receiving the user input, and how the expected result is returned.

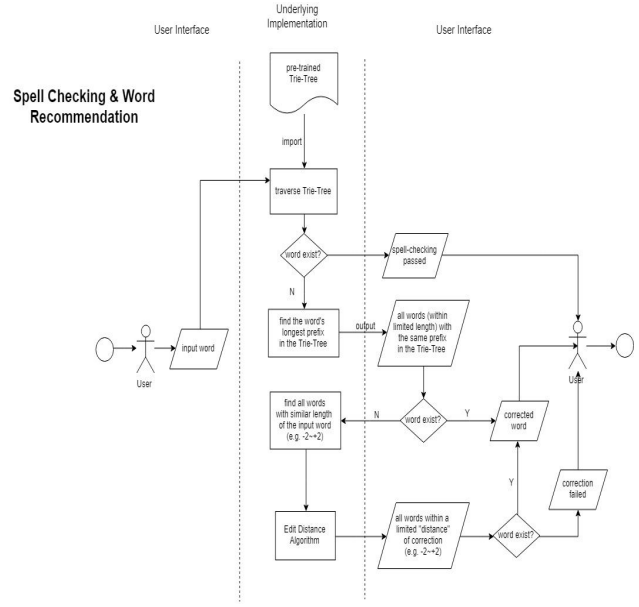
Training: the admin will choose a collection of huge amounts of words as the training data set (e.g. an authoritative dictionary). Each word will go through the trie-tree structure in the program. If the word does not exist in the current tree, the program will automatically add the word into the tree based on the insert function of the trie-tree structure. After all words have gone through this process the tree will be well trained, and our program will be able to store the current tree on local device, so that the next time any user/admin opens the program the previous trained tree can be imported directly without training again.



One integrity constraint of this training process is that the tree does not have the ability of automatic correction checking for the input. i.e. any input not existed before will

be added into the tree no matter if the word is correct. This suggests when training the trie-tree, the admin must be enough careful about choosing the proper data. More specifically, Oxford Dictionary may be a better choice compared to Urban Dictionary, since it provides higher level of authority.

Spell Checking & Word Recommendation: our program contains majorly two parts: the front end and the back end. At the front end, the user will input any word / series of words that he wants to check. At the back end the program will first import the pre-trained trie-tree automatically, and then traverse the tree structure to see if the input words exist. If all words already exist in the current tree, it means the input passed the spell checking successfully, and the program will return an inform about the success for the user. If not, the program will traverse the tree again to find the prefix of the incorrect word that already exist in the tree – as a way of finding from which letter the typo possibly appeared. This procedure will return a sub-tree that contains all words that start with the same "correct prefix" with the input word. This is our first phase of spell checking based on the prefix. If the expected word exists in this returned list, the user will choose the word he wants and then spell checking will be finished. If still not, the program will enter phase 2 of checking: the program will find all words that has similar length to the input word (i.e., the number of letters for the word and the input differs within a specific threshold). For each word in this list being the possible correct result, the program will compute the number of letters that need to be changed to make the input correct based on the Edit-Distance Algorithm – we define this number as the "distance" for correction. After all the distances are computed, the program will define the top 5 words with the smallest distance as the most possible result and return to the user.



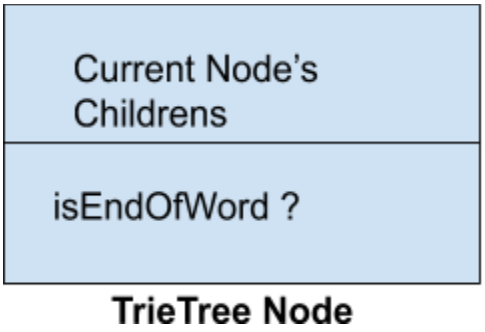
The integrity constraint of spell checking and word recommendation procedure due mostly to the number of words of processing / returning for each time. For instance, if the user's typo appears at the second letter, then the program will return all words under one specific letter, which could be a lot. To avoid this kind of situation, we may set some strategy to limit the number of words returned to the user in phase 1, but we need further test for finding the most proper limitation in order to include the possible result and not to include to many words as well.

Pseudocode for Trie Tree

Trie Tree has the following methods:

1. Insert (word)
2. Search (word)
3. autocomplete(prefix)

- **Tree Node metadata**



- Current Node's children is a dictionary. This dictionary stores the pointers to current Node's childrens which are the next characters in the word.
- isEndOfWord → 1. True, if that that is the last character in the word.
  2. Else False. That node does not represents the end of word.

Trie Tree methods:

1. **Insert (word)** → call insert if we want to insert new word into trie tree.
2. **Search (word)** → search if the word is present into the tree.
3. **autoCompletion (prefix)** → Takes the prefix and returns the list of words with the same prefix.

Inserting a Word to Trie Tree Data Structure

def insert(word):

1. Create temp pointer which points to the root of the tree.
2. Traverse the word character by character.

a. Check if the temp has a children same as the character we are pointing to in the word.
    i. If false : create a new node and add that node to the dictionary of temp node.
    ii. If True, move temp pointer to the child.

3. Make isEndOfWord = True → Flag to indicate that this node is the last for the current word.

def insert(word) :

1. Temp = tree.root
2. For char in word:
    a. If not temp.children.get(char):
        i. Temp.children[char] = TrieNode()
    b. Temp = temp.children[char]
3. temp.isEndOfWord = True

Searching if the word is correct.

def search (word) : → Boolean

1. Initialize the indicator which tells us if the word is present of no.
2. Initialize temp pointer that points to the root of the tree.
3. Traverse word character wise.

    If temp's dictionary does not contains the character :

        Return False

    Elif temp node is the end of word

        Return false

    Else

        Make temp point to the children which represents the current character.

4. Return True

def search (word) :

1. Found = true
2. Temp = tree.root
3. For char in word:
    a. If not tree.children.get(char) == false:
        i. return False
    b. temp.isEndOfWord == True and found == false:
        i. return false
    c. Temp = temp.children[char]
4. Return Found

AutoSuggestion for incorrect words

def autosuggestion (prefix) → List [suggested words list]

1. Initialize the empty string
2. Traverse the prefix until the last character
3. Make temp pointer point to the last node which represents the last character of the prefix
4. Traverse the tree and keep on adding the words into list.
5. Return the list of suggested words.

def autoSuggestion (Prefix) → List [suggested words list]

1. tempKey = " "
2. Temp = tree.root
3. For char in prefix:
    a. If not present in temp.children
        i. Return empty list
    b. Temp = temp.children[char]
    c. tempKey = tempKey + char
4. Helper(temp,tempKey)

def Helper(root, prefix):

1. If this is whole word:
    a. Add to list
2. Recursively call the the helper function by adding char.

def edit_dist(str1, str2):

1. Create a len(str1)*len(str2) matrix simplifying as (n*m).
2. Initialize the matrix.
    a. Every e(i, 0) = i
    b. Every e(0, j) = j
3. Fill the matrix.

    for i = 1 to n:

        for j = 1 to m:

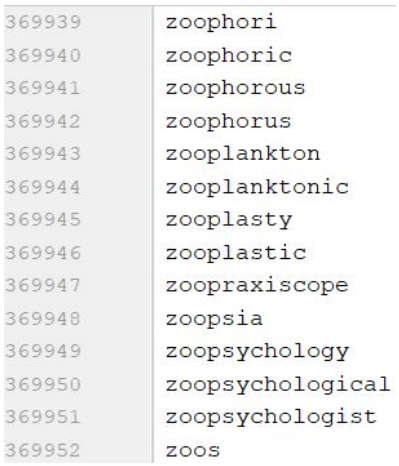
            e[i, j] = min{ e(i-1, j)+1, e(i, j-1) +1, e(i-1, j-1) + diff(x, y)} (diff(i, j) = 0 if x[i] = y[j] else 1)

C. Stage 3 - Implementation Stage

For the implementation of our spell checker using Trie Tree algorithm, we mainly used Python. To save the computation cost we serialized the train model using a python library called Pickle.

The deliverables for this stage include the following items:

• Sample small data snippet.



Our dataset contains around 370,000 words. The above figure shows how the data looks like in the file. We build our trie tree using all the words available in the dataset. Moreover, we also implemented the feature to insert the word into trie tree model.

•Sample small output

-- Input:

```
Trie is an efficient information retrievl
data structure. Using trie, search
complexities can be brought to optimal
limit (key length). If we store keys in
binary search tree, a well balanced BST will
need time proportonal to Mlog(N), where M is
maximum string length and N is nuber of keys
in tree. Using trie we can search the key in
O(M) time.Howerer, the penalty is on Trie
storage requiremets
```

-- Output

(Incorrect Word --> List of recommended words)

1. retrievl ---> [('retrieval', 1.0), ('retrieve', 1.0), ('retrievals', 2.0), ('retrieved', 2.0), ('retriever', 2.0), ('retrieves', 2.0), ('retrievable', 3.0), ('retrievably', 3.0), ('retrievers', 3.0), ('retrieving', 3.0)]
2. proportonal ---> [('proportional', 1.0), ('proportion', 3.0), ('proportionable', 3.0), ('proportionably', 3.0), ('proportionally', 3.0), ('proportionate', 3.0), ('proportioned', 3.0), ('proportioner', 3.0), ('proportions', 3.0), ('proport', 4.0)]
3. nuber ---> [('nubecula', 4.0), ('nubeculae', 5.0)]
4. howerer ---> [('however', 1.0), ('howe', 3.0), ('howea', 3.0), ('howel', 3.0), ('howes', 3.0)]
5. requiremets ---> [('requirements', 1.0), ('requirement', 2.0)]

• Data size:

The data we collected has 370,000 words and the size of the dataset in terms of disk resident is 75MB after serialization. Once the program is run, all 75MB data will be extracted to the RAM, which is one of the shortcomings of the trie tree.

• Estimation / Findings:

One interesting finding in our estimation is that the recommendation may not be satisfying if the “prefix” is defined by the program incorrectly. An example of this can be seen in the sample output 3: the expected word should be “number”, but since “nube” is a prefix already exists in the tree, all words returned will start with “nube” instead of “nu”, so the actual expected answer “number” will be ignored.

In case like this, if the actual “prefix” is defined wrongly, the program may not be able to return the correct answer. This may be one of the shortcomings of the trie tree structure. But we came up with a method to solve this: we can define a threshold for the distance returned by Edit-Distance algorithm, and only those words with a distance less than this threshold will be considered as the candidate. For the example above, since ‘nubecula’ has a distance of 4 and ‘nubeculae’ has a distance of 5, both of which are over the threshold (let’s say, 2), none of them will be considered as the candidate. So, our program will “take a step back” and try to find the expected word with the prefix of ‘nub’, and later with the prefix of ‘nu’, at which the actual word ‘number’ will be found since it has a distance within the threshold and will be considered as a candidate. After this modification, the returned list now looks like:

Nuber' ---> [('nuder', 1.0), ('number', 1.0), ('nub', 2.0), ('nuba', 2.0), ('nubby', 2.0), ('nubbier', 2.0), ('nubia', 2.0), ('nubs', 2.0), ('nude', 2.0), ('nudes', 2.0)]

And we can see now ‘number’ is provided in the list.