Assignment 3: Spell Checker
Programming Report

s5343798 & s5460484

Algorithms and Data Structures 2024

1 Problem Description

As the title implies, we need to write a spell checker program. This program will need to scan a piece of text, find the words that are misspelt, print them out, and give a count of how many words in the text are wrong.

The program's input consists of two parts, separated by an exclamation mark "!". The first part of the input is a list of correctly spelt words that are separated by a new line. This list of words will be used to create a dictionary of correct words, which are then used for spell-checking the second part of the input that consists of the text that needs to be spell-checked. The text to be checked can be text of any sort, including capital letters and symbols.

2 Problem Analysis

Firstly, we must decide how to represent the data structure of the dictionary in the program. Here, a simple and easy implementation would be to create a list of all the correctly spelt words. However, this solution is crude at best, as it has to traverse the entire dictionary list to check every word in the text. This results in a time complexity of O(n^2).

To reduce this time complexity to a more desirable O(n), O($log(n)$), or even better O(1), we need to narrow the search space(the number of words that need to be checked). By using a (Standard) Trie, we can employ a heuristic search due to the nature of the list. A Trie is based on a Linked List data structure where each element refers to the next element in the list. For the entire dictionary, the Trie allows us to define a root element (the first letter of a word) and multiple "children" of that root element (the second, third, etc. letters).

This structure allows us to narrow the check for every word by first checking if the first letter matches, then whether that letter has a "child" letter that matches or not and so forth until it reaches the end of a word. If no "children" letters match, the word is misspelt.

This brings our time complexity down to O($nlog(n)$), which is much better than the original O(n^2). Although creating the Trie data structure still only takes O(n) time, the time it takes to check the text has greatly been reduced.

After implementing the Trie data structure, the rest of the program is reasonably simple to solve. First, we must add all the words in the first part of the input to the Trie. Thereafter, the program will go line by line and check if each word exists in the Trie; if they do not, that word is printed and the incorrect word counter is updated.

3 Program Design

For the implementation of the Trie data structure, we opted to use two classes; the first represents each node in the list (class name: *TrieNode*), and the second represents the list itself (class name: *StandardTrie*) found in the *standard_trie.py* file. This implementation removes the need to create the empty root node for the Trie manually.

The *insert* function works by checking if the letter of a word is not in the list already and adding it, then moving on till the end of the word is reached and an empty element is added.

The *term_exists* function works similarly to the *insert* function, but instead of adding a letter, it returns false. At the end of the word, if there aren't any empty elements, it returns false. This avoids words like "ap" being seen as current even though those letters are a part of the word "apple."

In the file *speller_trie.py*, an object allowed_words is created to which the words in the first part of the input are added. Thereafter, we use the *split* function to split the words on each line into a list following the splitter variable schema. The program then iterates over each word and checks if they are in the dictionary, if not, it prints the word and updates the unknown_word_count variable.

Finally, once the entire text has been checked, the program prints how many words were misspelt.

4 Evaluation of the Program

For testing, we used the given example inputs provided to us, as well as the test cases provided by Themis. For performance: In a small input text, the program was rapid, and in a large input text (for example, the English dictionary and a novel), the program was very efficient at processing the text quickly. We would imagine that in the best case, the program takes a few milliseconds, and in the worst case, a few seconds.

Our program passed all test cases on Themis.

5 Extension of the Program

Having only an insert function and term_exists function limits the usability of our custom standard trie class. Hence, we created an additional remove function: This takes in one string argument - term - and returns and removes the term from the trie.

Solving this problem required splitting the function into two processes: Iterating down the trie to reach the ending of the given term, and then working backwards to remove necessary characters. The reason we went with this approach was that in a standard trie, there may be words with similar roots, e.g. app, apple, applet. If we want to remove "apple" then, we would only want to remove up to "app-", to allow for "app" and "applet" to still exist on the trie. Likewise, if we want to remove "app", we should only remove the indicator (ending) that "app" exists on the trie, so that "apple" and "applet" still exist on the trie.

For the design: First, the program iterates down the trie, building a reference list for future backtracking. Next, the program removes the ending indicator (None key) for the given term and then begins iterating back up. In this loop, each key is removed until either the programme finds a child with an ending indicator, in which the function ends, or the loop ends, ending the function too.

For testing, there were no formal cases we used - we designed simple and short inputs for quick, iterative testing. So, it is difficult to evaluate the efficiency of this program. However, this program has a time complexity of O(n), so it is somewhat efficient.

6 Process Description

We worked on the program together through an online call, bouncing back and forth in designing and testing. The work on the report was divided in two, with one handling the first half (up to 3) and the other the second half (up to 7). The distribution of labor was equal.

Regarding difficulty, there was not too much frustration in the assignment. Given that we had some pseudocode from the Reader, it was just a matter of implementation, although we did need to first figure out how to insert terms into the trie. After that though, the work was straightforward.

That being said, there was a small problem with interpreters: One member used Python 3.11, while the other used 3.9. While 3.11 had no issues with the given code, 3.9 had some difficulty, so this required the member to update their interpreter.

By the end of this assignment, we became more knowledgeable about data structures in how to create a custom one from scratch and how to iterate through a data structure. We also learnt how efficient tries can be when implemented properly. And finally, we learnt how to interpret pseudocode effectively.

7 Conclusions

This program efficiently solves the given problem. Even with a large input, the program will handle it in just a few seconds. Regarding whether the program is optimal, perhaps there is some room for improvement. Insertion has a time complexity of O(n) since it iterates n number of times until the program can end, and the same is the case for searching for the term. However, modifying the program to aim for a time complexity of O(1) or

O($log(n)$) may require redesigning the structure to allow a search similar to a binary search. In the end though, given the context of the problem - checking if a term exists in a trie - this may be irrelevant scope-wise.

A Program Text

A.1 standard_trie.py

This program text also includes the additional remove function.

```
"""
File: standard_trie.py
Authors:
    Marcus Persson (m.h.o.persson@student.rug.nl),
    Marinus van den Ende (m.van.den.ende.1@student.rug.nl)

Description:
    A custom class for standard tries.
"""


class TrieNode:
    """
    Represents a standard trie node.
    """

    def __init__(self):
        self.children: dict[str | None, TrieNode] = dict()


class StandardTrie:
    """
    Represents a standard trie.
    """

    def __init__(self):
        self.root = TrieNode()

    def insert(self, term: str) -> None:
        """
        Inserts a term into the trie.
        :param term: Term to insert in the trie.
        """
        trie_depth = self.root

        for char in term:
            if not trie_depth.children.get(char):
                # If the character is not present at the current trie depth,
                # we create a node
                trie_depth.children[char] = TrieNode()
            # Moving downwards
            trie_depth = trie_depth.children[char]

        # Making sure there is an ending None node
        trie_depth.children[None] = TrieNode()

    def remove(self, term: str) -> str:
        """
        Returns and removes a term from the trie.
        :param term: Term to remove from the trie.
        :return: The removed term, if it exists.
        """
```

```
        if not self.term_exists(term):
            return "No such term exists in the trie."
        trie_depth = self.root
        parents = []

        # Iterating down the trie.
        for char in term:
            parents.insert(0, trie_depth)
            trie_depth = trie_depth.children[char]

        trie_depth.children.pop(None)

        reference = [*term]
        # Iterating back up
        for parent in parents:
            if trie_depth.children.get(None):
                # If the term iterates back up to a pre-existing term,
                # end the function
                return term
            trie_depth = parent
            trie_depth.children.pop(reference[-1])
            reference.pop(-1)
        return term

    def term_exists(self, term: str) -> bool:
        """
        Checks if a term exists in the trie.
        :param term: Term to check in the trie.
        :return: True if the term exists in the trie, otherwise False.
        """
        trie_depth = self.root

        for char in term:
            if not trie_depth.children.get(char):
                # If the character does not exist at that depth,
                # return False
                return False
            # Moving downwards
            trie_depth = trie_depth.children[char]

        # Validate if the character is the ending
        return True if trie_depth.children.get(None) else False
```

A.2 speller_trie.py

```
"""
File: speller_trie.py
Authors:
    Marcus Persson (m.h.o.persson@student.rug.nl),
    Marinus van den Ende (m.van.den.ende.1@student.rug.nl)

Description:
    A spell checker, created using a standard trie.
"""
from standard_trie import *

# Step 1: Create the dictionary
allowed_words = StandardTrie()

word = input()
```

```
while word != "!":
    allowed_words.insert(word)
    word = input()

# Step 2: Validate if words exist in a given input.
# Handle non-alphabetical characters.
splitters = [".", ",", "\'", "    ", "/", "-", "\"", ":", ";", "?",
             "!", "(", ")", "1", "2", "3", "4", "5", "6", "7", "8",
             "9", "0"]

unknown_word_count = 0

word = input()
while word != "":
    for splitter in splitters:
        word = " ".join(word.lower().split(splitter))
    for w in word.split():
        if not allowed_words.term_exists(w):
            print(w)
            unknown_word_count += 1
    word = input()

# Step 3: Generate output.
print("There are", unknown_word_count, "unknown words.")
```