CSCI 311 - Fall 2020:
Autocorrect Project Report: Algorithmic Structure Analysis

Jacky Lin, Minh Bui, Ethan Dunne, Tu Le

Thursday, Nov. 19, 2020

1 Structure Description

Our solution to the auto-correct problem was written in Python and uses two key classes: a HashTable2D class for storing a set of words and a Checker class for checking if a given input has misspelled words and automatically correcting them. This document will describe the structure of the algorithms these classes employ and the runtimes of them.

This project composite with 3 main parts:

1. A Checker
2. A Data Structure
3. A Program Launcher

1.1 Checker

A class provides all related function to spell check, including:

1. longest common substring that scores the similarity between target word the word to compare 2. merge sort that sorts the candidates of target words based on lcs score and frequency of the word 2. generate candidates based on common typo 3. string checker that accepts a word to check and returns the top choice and some candidates 4. file checker that checks each word in text file, replaces with the top choice of candidates, save corrected version to a new file

1.2 Data Structure

1.2.1 2D Hash Table (HashTable2D.py)

Basically, this data structure consists of 26 small hash tables. Within each hash table, the first letter of words stored is the same. So the hash function is determined by the leading letter of each input word. Other method, including search and insert are all based on the following small basic hash table

1.2.2 Hash Table (HashTable.py)

This hash table is very similar to the regular hash table, but we modify the hash function so that it is customized for the use of words in this program. We implemented based on the sum of *ASCII* encoding for each letter in input word.

Additionally, this hash table will extend dynamically according to the number of non-empty array occupying in the hash table. If the number decreases over threshold (default is 0.75), then the hash table will double its size and redistribute the element in it. In this way:

1. We prevent there is too many item in the save location of the hash table, which will increase the cost of search function.
2. We can void there is too many empty location in it.

1.3 Program Launcher

We use *run.py* to launch the program; It does the following things:

1. (First time use) Read from file, create a **wordlist**, and save it locally
2. Load the **wordlist** from previously saved file
3. Accept user input argument
4. Start checker and correct the word(s)

2 Checker Analysis

The Checker class is a Python class intended to provide a replacement word for incorrectly spelled words. It is initialized with a **wordlist** as a dictionary kind of object (specifically a custom HashTable) to use as reference for replacement words, as well as a **num_candidates** to determine the number of replacement candidates to generate when deciding whether or not to replace a word.

A Checker object is instantiated in the **run.py** main function and is used to run replacement algorithms on the user input. User input can take the form of 1 of 2 modes, string mode or file mode. Hence, the Checker class has two algorithms for checking the user input:

2.1 Algorithm 1: String Checker

The Checker class contains a method called *str_checker*() that takes in a single word in the form of a Python string. The general outline is shown below:

1. First, we check if the given word is in the *word_list*. If it is, simply return that word.
2. If it's not, then we generate a list of strings by making slight variations to the original word based on common typos. This is a method called *generate_possible_words*(). This method will be outlined later in this document.
3. We then take that list and only the ones that are actual words in our dictionary. That list is sorted based on their longest common substring score and only the top *num_candidates* candidates are kept.
4. If there are any words in the candidates list, they are returned. If those typo variations yielded no candidates, a search on the entire dictionary is done by sorting the entire list of words based on their lcs score with the target word. The top *num_candidates* candidates are returned.

To analyze this, we take a look at each step. In step (1), the time it takes to search a HashTable is normally (O(1)), but our implementation is slightly different, and so the search time will be discussed in a the analysis of that data structure.

In step (2) we call *generate_possible_words*(). This method currently creates a set of words with 1 single variation in it unions with a set of words with 2 variations in it. If we say the m = length of word, then creating the set of 1 variation possibilities is $O(m)$ because it uses every possible split of the word, and there are $m-1$ places to split it. It creates a set with a size of $C \cdot m, C \in \mathbb{R}$

Creating the list of 2 variation possibilities is a bit trickier. It must generate all of the 1 variation possibilites, then generate a set of 1 variation possibilities from each of those possibilities in the first set. Therefore, it has a runtime of $O(m \cdot m)$, or $O(m^2)$.

In step (3), we search through all of those possibilities using the *linear_search*() method. First, it loops through the set of strings given to it, which will be m^2 possibilities, and therefore take $O(m^2)$ time. The remaining set will be size m^2 in the worst case scenario where all possibilities are in the wordlist. It then performs a mergesort on the remaining set to organize it in order of decreasing LCS score. Mergesort is normally $O(nlogn)$, where n is the number of elements), but since it has to get the LCS score for each possibility with the target, it will be a little larger. Each run through of LCS between a possibility and the target word will take $O(m^2)$ because the possibilities are all just about the same length as the target word. Therefore, *merge_sort_lcs*() takes a total runtime of $O(nlogn \cdot m^2)$. Since the number of possibilities $n \approx m^2$, we can replace n with m^2 to get $O(m^4log(m^2)) = O(2 \cdot m^4log(m)) = O(m^4log(m))$. The top 5 candidates from this list are chosen ($num\_candidates = 5$ by default).

At this point, $O(m^4log(m))$ is our largest cost out of these 3 steps and so it dominates the runtime. However, in the case that none of the possibilities were in the wordlist and therefore no candidates were found, step (4) has us sort the entire wordlist based on each words LCS score with the target word. This feature takes a long time but can easily be removed if it seems unnecessary.

2.2 Algorithm 2: File Checker

The Checker class contains a method called $file\_checker()$ which takes a path to a text file as a parameter and runs the *str_checker*() method on each word in the file. It creates a new file with all the auto-corrections replacing each word. The $file\_checker()$ method loops through a list of the the lines and corrects each word in the line by passing it as an argument to the *split_helper*() method. The *split_helper*() method splits the line on a given delimiter, so we first call it to split on punctuation, the we split it by white space. It loops through every word in the line and runs the he *str_checker*() method on it. If there are k words in a file, and a word in the file can be at most m letters long, then the total runtime of this algorithm is $O(k \cdot m^4log(m))$.

3 Data Structure Analysis

3.1 HashTable.py

```
def hash(self, word):
    ascii_sum = 0
    for i in range(int(len(word) / 2) + 1):
        ascii_sum += ord(word[i]) - ord('a')
    return ascii_sum % self.capacity
```

This is our hash function. We chose to hash the words according to the ASCII values of their first half of characters, so that we can account for words with different length. The complexity is $O(1)$ for each word.

```
def add(self, key):
    if not self.search(key.word):
        index = self.hash(key.word)
        if len(self.array[index]) == 0:
            self.array_load += 1
        self.array[index].append(key)
        self.num_items += 1
        load_factor = self.array_load / self.capacity
        if load_factor > self.max_load:
            self.resize()
```

For this add function, we used our own type of key called Word. Word basically consists of the word represented in string and it's usage frequency. The frequency is later used in spell checker to provide suggestions to the user. The adding itself is just $O(1)$. However, since there might be a chance that the function *resize*() is involved. We set the hash table to resize if it reaches 3/4 capacity. Therefore, using the accounting method and charging each append operation \$3, leaving \$2 to the later copying operations, the amortized analysis for the *add* function in a sequence of n operations is: $O(1)$.

```
def resize(self):
    self.capacity *= 2
    newArray = [[] for _ in range(self.capacity)]
    for i in self.array:
        if len(i) != 0:
            for j in i:
                index = self.hash(j.word)
                newArray[index].append(j)
    self.array = newArray
```

This function is used when we need a larger hash table. The function basically creates a new hash table of twice the size of the original one. After that, it hashes every element onto the new hash table, making the complexity $O(n)$ where n is the number of elements in the hash table.

```
def search(self, target):
    index = self.hash(target)
    for i in self.array[index]:
        if i.word == target:
            return i
    return None
```

This *search* function uses the *hash* function to find the possible index of the target word and search within that array slot to see if there is a similar word. Therefore, the complexity is $O(1)$ since we are not looking through every element in the hash table.

3.2 HashTable2D.py

```
def __init__(self):
    self.size = 0
    self.hash_tables = [HashTable() for _ in range(26)]
    self.elements = set()
```

This method creates an array of 26 hash tables; each hash table contains words with the same initial letter. This is an $O(1)$ method because the complexity of its operation isn't changed by any input.

```
def __hash(key):
    return ord(key[0]) - ord('a')
```

This method is O(1) because it only does one calculation

```
def insert(self, key, freq):
    newKey = Word(key, freq)
    idx = self.__hash(key)
    self.size += 1
    self.hash_tables[idx].add(newKey)
```

The first line creates a new object Word which is $O(1)$. $self.\_\_hash(key)$ is $O(1)$ as explained above. $self.size+=1$ is $O(1)$. The last line $self.hash\_tables[idx].add(newKey)$ uses method $add()$ of the hash table which is $O(1)$. Thus, this method is $O(1)$.

```
def search(self, target):
    if target == '':
        return None
    idx = self.__hash(target[0])
    return self.hash_tables[idx].search(target)
```

Find the index of the correct hashtable to search is $O(1)$. The last line is a hash table at index idx in array $self.hash\_table$ calling its method $search()$. $search()$ is $O(1)$. Thus, the time complexity of the entire method is $O(1)$

```
def fill_hash_table(self, file_name):
    if not os.path.exists(file_name):
        print("FAIL TO OPEN THE FILE")
        exit(-1)
    with open(file_name) as csv_file:
        readCSV = csv.reader(csv_file, delimiter=',')
        count = 0
        for row in readCSV:
            freq = int(''.join(row[2].split(',')))
            count += 1
            print("%dth insert %s" % (count, row[0]))
            self.insert(row[0].lower(), freq)
            self.elements.add(row[0].lower())
```

This function reads a CSV file that contains the words and their frequencies, then insert them all into the hash table to create a dictionary. Since the $insert$ function generally takes $O(1)$ to complete, this $fill\_hash\_table$ function should take $O(w)$, where w is the number of words in the CSV file