Qianqian Zheng

813288

5 May 2018

COMP20007 Assignment 2 Report

Data structures and algorithms used & Asymptotic time complexity

For all big O analysis here, n = number of characters in a string, avg_word_length = 5.1 (Wolfram Alpha).

For Task 3 and Task 4, dictionary and document list are both stored in hash tables.

The dictionary is stored in hash table because the most common operation performed on it is lookup. And for document, I assumed that for natural language, recurring words would be quite common. If its first occurrence has been corrected, there is no need to re-lookup the dictionary if the same error occurs again. We can always just search the word in document's hash table first and see if it has already been looked up/corrected by dictionary.

Hash table big O

I decided hash tables to be the most suitable approach for both because it has an O(1) lookup and insertion time complexity, assuming h is O(1) and spreads keys uniformly. The string hash function used is XOR hash used in lecture, which has O (n). Therefore search, lookup and deletion costs are all O(n).

For every insertion and lookup, the target (key, value) will be brought to the front of the list in order to adapt to skew access patterns.

According to the assignment specs, the dictionary is a list of strings representing correctly-spelled words. So when inserting (key, value), it will not check for existing keys.

Spell correction

For task 4, to determine whether to 1. search through the entire dictionary, or 2. generate all edits of Levenshtein distance 1 from the existing words, lookup each of them in the hash table, we have to compare the big O for each word.

For **task 1**, to calculate the Levenshtein distance for two words is O (n * m) for words lengths of n & m

```
for i from 1 to n:
for j from 1 to m:
substitute_cost = 0 if word_1[i] == word_2[j] , else it is 1
E[i, j] = min(substitution, deletion, insertion)
```

Worst scenario cost of the dictionary search : dictionary _size * Task 1 cost

So Option 1 has O (dictionary _size * n * avg_word_length)

For **task 2**, to generate all edits of Levenshtein distance 1 has O (n_results * n) = (53* n+ 26) n

*constants like 53 and 26 are kept because they are very likely to be larger than n so it is significant

```
number of results for a word of length n = n + n* NUM_ALPHABET  + (n+1)* NUM_ALPHABET = 53n + 26
For each edit: O (n) as it copies string to a different pointer and edits one char
```

The cost of generating all edits of distance 1 from a word: (53n + 26) * n

The cost of generating all edits of distance k from a list with edits of distance k - 1 : (53n + 26) ^k * n

Cost of one lookup and check in hash table : O (n) (hashing & check if two strings are identical)

Before searching for corrections foreach distance for a word, I used a function if_search_dictionary to compute the time cost for both and choose the lower cost strategy.

Task 3 & 4 Pseudocode and Asymptotic time complexity

```
        creation of hash table has O (n_elements)
        insertion in hash table has O (n)
        lookup in hash table has O (n)
```

```
for i from 0 to dictionary size:
put values from the dictionary list in dictionary hash table              O (n * n_dict)
for i from 0 to document size:
put values from the document list in document hash table                  O (n * n_doc)

for i from 0 to document size:                                    O(n_doc)
       lookup if word is in dictionary and document hash tables           O (n)

else, for i from 0 to maximum Levenshtein distance allowed if a match is not found:
       if scanning dictionary is faster than trying edits of the word            O (1)
       search dictionary for words within max lev distance        O (dict _size* n * 5.1)
       if there is a correction print and store in document hash table           o (n)

       else generate edits from the word/ existing list of edits   O ((53n + 26)^dist *n^2
       if any of the edits exist in dictionary
       print it and store in document hash table for future lookup of this word   o (n)

       free list of edits                                         O ((53n + 26)^dist
       free hash tables                                           O (n_dict + n_doc)
```

Conclusion

Overall the function has `O (n * (n_dict + n_doc) + n_doc * word lookup cost)`

For lookup cost of each word, Task 3 (check in dictionary) has O(n), so task 3 has

`O (n * (n_dict + n_doc) + n_doc * n) = O (avg_word_length * (n_dict + n_doc))`

For Task 4, lookup cost of each word is

- O(n) if it exists in dictionary/ has been corrected before
- O (dict _size* n * `avg_word_length`) if it was searched against every word in dictionary
- O ((53n + 26)^ dist * n ^ 2) if edits with lev distance “dist” were generated for it and looked up in dictionary

If the document used has very few spelling errors, cost is close to

`O (avg_word_length * (n_dict + n_doc))`

If the document used has majority long words with errors, cost is close to

`O(dictionary_size* avg_word_length ^ 2 * (n_doc))`

If the document used has majority short words with minor (1 or 2 Levenshtein distance) errors, cost is close to

`O((53* avg_word_length+ 26)^dist *avg_word_length ^2 * (n_doc) + avg_word_length*dictionary_size)`

Alternative approaches considered

- To store dictionary, I also considered using a tree structure in hope that words with close Levenshtein distance would all sorted be next to each other. However, it is expensive to create a sorted tree (n log(n)) and for this task we are doing mostly random access (log (n)) of the dictionary so tree structures are not useful.
- For hash table, separate chaining with an array may be faster than a linked list (due to the effect of CPU cache) . However I am not using array because the table needs to adapt to language patterns through inputs with Move To Front. Doing MTF in array will have O(log(n)) at least (if we use min heap) while MTF in LinkedList has consistent O(1). For the amount of lookups for this program, the CPU memory advantage is superseded by the sorting cost.