CSCI 311 Project Report

Jonathan Basom, Sebastian Ascoli, Khoi Lam, Lawrence Li

Algorithm Description

Our algorithm revolves around finding the edit distance for the user input word and doing a search in the trie tree data structure, which contains all correct words. From there, we find several candidate words within the specified edit distance. After that, the program will use one of the three algorithms (most prevalent candidate, the longest common subsequence, highest frequency) to determine the best choice within the set of candidate words. This algorithm will be selected by the user before the program runs.

A step by step algorithm is described below:

Preparation: Generate the trie tree data structure to store all the words from the given dictionary in txt.

Section One: Steps for generate a set of possible words candidates

1. Get the user input word
2. Generate a set of possible forms of the user input word with an edit distance of d
3. Add those possible word forms together as a list.
4. For each possible word form in the list, try to find a set of candidate words from the trie dictionary. These sets of word matches are further combined together.
5. In the end, it returns a list of all matching words in our trie tree data structure dictionary that can be obtained.

Section Two: Find out the best word match from the list of word candidates

Choice 1: Traverse through the list of word candidates and find the word that appears mostly frequently in the list. Choose that word as the best candidate word for replacement.

Choice 2: For each of the word candidates in the list, use the longest common subsequence algorithm. The algorithm will return a similarity score. Find the word with the maximum similarity, and choose that word as the best candidate for replacement.

Choice 3: Each word is loaded into the trie in the order of most to least frequent as determined by the n-gram frequency analysis of Google's Trillion Word Corpus.

Therefore, each word will contain a unique value representing how frequently it is used in the English language. The candidate word with the lowest value, or the highest rank, will be selected as the replacement word.

Runtime analysis for the choices

For choice 1 as described above, it traverses through the entire list to calculate the number of appearances of each word, and then traverses the appearance list to find out the most common word that appeared in the candidate word list. Runtime will be O(n + m) where n is the length of the candidate word list, and m is the length of the frequency list where m <= n. This will become O(n).

For choice 2 as described above, the longest common subsequence algorithm, implemented with dynamic programming, costs O(n * m) where n is the length of the first word and m is the length of the second word.

For choice 3 as described above, the algorithm traverses through the entire list of n possible words to determine the rank of each of their frequencies. To find the frequencies, we must search for each word in the trie. Since searching is O(key length) then the total runtime is O(n * n.length).

Traversing the Trie (Inserting/Searching): O(key length) as described below in the data structures section

Generating the edit distance: $O(l^d)$ where l=length of word and d=distance. Since we only generate up to edit distance = 2, the exponential growth is not an issue. Thus, our program is only $O(l^2)$

MatchAll: The upper bound is O(26^n) where n=the number of “question marks”. However, since it only searches through words (or subwords) that exist in the trie the complexity will be a lot lower in practice.

Data structure

In storing all the words for the dictionary, we used trie data structure, which is an efficient information retrieval data structure. Every node of Trie consists of multiple branches, where each branch represents a possible character of keys. The data structure needs to mark the last node of every key as the end of the word node. If a node is marked as the end of the word node, then that node will also contain the word’s frequency rank.

Inserting a key into Trie is where every character of the input key is inserted as an individual Trie node. If the input key is new or is an extension of the existing key, we will need to construct non-existing nodes of the key, and mark the end of the word for the last node. If the input key is a prefix of the existing key in Trie, we will mark the last node of the key as the end of a word.

For searching a key, we only compare the characters and move down. If we reach the end of the word for the last node, then the key exists in the trie. In the second case, the search terminates without examining all the characters of the key, since the key is not present in the trie.

Source of Word List