Algorithms for string processing. Trie and Levenshtein distance

Grigore Iulia-Anita

Faculty of Automatic Control and Computer Science
iulia_anita.grigore@stud.acs.upb.ro

1 Introduction

1.1 Describing the problem

In a world dominated by technology, strings and characters are everywhere: text messages sent online, searches made on different browsers and even in out own organism, where the DNA sequences are considered string of different symbols with a well known meaning.

This is why the need for powerful algorithms for string processing has appeared in the recent years.

Some of the real-life necessities are, for example, the spell-checking programs implemented by almost every text editor on the market or even the auto complete system offered by every online browser.

Another example, one that is not so well known but is very important in the science world and in the development of it, is the sequence alignment. In bioinformatics, this represents a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Gaps are inserted between the residues so that identical or similar characters are aligned in successive columns. [2]

1.2 Solutions

To solve the problems presented above, I chose the data structure Trie and an edit distance algorithm called Levenshtein distance. In this paper, I will compare this algorithm with others, like Hamming distance, but only to quantify the efficiency of it.

Trie

A Trie is an ordered tree data structure used to store a dynamic set or associative array where the keys are usually strings. The root is empty and every child is represented by a letter from a previously declared alphabet. To store a string in a Trie, a depth search started from the child corresponding the wanted letter is needed. Because of this implementation, the program uses less memory (in most cases). That is because instead of saving all the words individually, we just add the needed letters to the already existing prefixes. [3]

A common use of the Trie is found in the search engines to offer suggestions based on the characters you already written. For example, if you would search the word "tree" in a search engine, you can get suggestions like "tree of life" or "treehouse".

Levenshtein distance

The Levenshtein distance computes the difference of characters between two strings. This is computed using a mathematical formula witch works for deleting, inserting and replacing a character. [1]

The mathematical formula for computing Levenshtein distance is:

$$\operatorname{lev}_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{if } \min(i,j)=0 \\ \min \begin{cases} \operatorname{lev}_{a,b}(i-1,j)+1 \\ \operatorname{lev}_{a,b}(i,j-1)+1 \\ \operatorname{lev}_{a,b}(i-1,j-1)+1 \end{cases} & \text{otherwise} \end{cases} \tag{1}$$

1.3 Criteria for evaluating the solutions

To evaluate and analyze the solutions proposed by me, I created two types of tests. To check the correctness of the code, I used unit tests of general cases, but also for edge cases and to check e efficiency of the algorithm, I wrote benchmarks.

The unit tests are written with the sole purpose of checking the functionality of the Trie, without taking into account the performance of the algorithm. But, because the accuracy of the benchmarks shouldn't be influenced by the use of RAM memory, all the code is optimized from this point of view.

For the performance of the operations supported by the Trie, I created benchmarks using the American English dictionary. The size of the dictionary inserted in a Trie instance is about 4.5MB. The file is processed to obtain and compare the insertion, deletion and the search of words in the Trie. The words are inserted in random order.

For the Levenshtein distance, I wrote two different implementations, one that uses a Trie and another that uses an array to store the dictionary. I will compare the results given by the both implementations using an unit test and the performance with benchmarks that test the search of suggestions for correcting a misspelled word with words from the dictionary.

2 Presenting the solutions

2.1 Trie

A Trie is an ordered tree data structure that has a constant number of children (in this case, 26 represented by the number of letters in the dictionary), ?? where words can be saved, letter by letter, in the depth of it. ?? In my implementation, it supports insertions, deletions and searches of a word given as an argument to the function. Each node in the Trie contains a pointer to the parent node, an array of children and the number of it and a flag, *end_of_word*, that is set when the node is the end of a word from the dictionary.

To insert a word, first I have to check for an existing prefix in the Trie, and then adding (if is necessary) the remaining letters and setting the flag *end_of_word* on the last letter.

To search for a word in the Trie, it's not enough to search the sequence of letters. For it to really be an existing word, the *end_of_word* flag has to be set in the last letter of the wanted word. This way, if the word **moment** is already saved in the Trie, with the flag set only on the letter **t** and we search the word **mom**, the function *has_word* will return false.

Both the insertion and the search have a complexity of O(word_length).

To implement the deletion, I considered 4 cases:

– the word does not exist in the Trie, so there is nothing to delete
– the word has no letter in common with another word, so delete all the nodes
– the word has a prefix with another word, so delete until the *end_of_word* is set
– the word is a prefix for another word, so just set the *end_of_word* flag to false

Considering the cases listed above, the complexity varies, but the worst case scenario is, like for the insertion and search, O(word_length).

The main advantage of this data structure that the search is proportionate with the length of the word, in contrast with other data structures where it is proportionate with the number of keys known. Also, another advantage is that this structure can save other types of keys, beside strings, depending on the necessity and the dictionary.

A big disadvantage is the space needed for saving the words in the memory. A Trie saves, even if it's not necessary, 26 children for each node, which is very cost consuming for the memory use.

2.2 Levenshtein distance

To implement the spell-checker with Levenshtein distance, I used two different methods for storing the dictionary:

– into an array
– in a Trie

Both methods use a matrix of size (m, n), where m is the length of the first word and n is the length of the second one. The first line and column of the matrix are initialized with consecutive values, between 0 and the length of the word. For each operation (insertion, deletion or substitution), we consider the cost equal to 1. The partial cost is found on the diagonal and it can be 0, if the letters are the same, or 1 if they are different. After computing the sum of the values on the diagonal, the distance will be found in the lower left cell. The complexity for this algorithm is O(M * N), where M is the length of the first word, N is the length of the second one, because in this implementation,the whole matrix must be computed.

Levenshtein distance with an array

This solution compares the computed cost for every word in the dictionary with a word given as an argument and, if the distance is lower than a maximum global cost (usually, a cost 2 is sufficient for relevant suggestions), accept the word as a suggestion to correct the word given as an argument and display it and the distance.

```
std::vector<std::pair<std::string, std::size_t>> results;
  for (auto word : words)
  {
    cost = spellchecker::levenshtein(target, word);

    if (cost <= max_cost)
    {
      results.emplace_back(word, cost);
    }
  }

  return results;
```

The complexity of this implementation is, in the worst case scenario, O(N*M^2), where N is the number of words in the dictionary, and M is the maximum length of the words. This complexity results from the assumption that it is necessary to compute the distance between words with the same length, this case being the worst one.

The main disadvantage of this implementation is the run time. On a 150-word dictionary, one run can take up to 7 minutes.

Levenshtein distance with a trie

The difference between these two implementations is that, instead of checking every word to see if it's possible match like above, here I use a depth search and compute the cost, letter by letter, as I go lower in the Trie.

```
std::vector<TrieDictionary::WordCost> results;
for (auto child : trie_.root_->children)
{
  if (!child)
    continue;
  auto letter = child->chr;
  search_recursive(
      trie_.root_->children[Trie::charToIndex(letter)], letter, seq,
          curr_row, results, max_cost);
}
return results;
```

This optimization is very easily seen in the complexity: instead of O(N*M^2) like in the previous case, here it is O(N*M), where N is the number of node in the Trie and M is the maximum length of the words.

Obviously, the complexity represents a big advantage. But if the dictionary is very big, the Trie will take up a lot of memory, this being a big disadvantage of the implementation. To optimize this, instead of a Trie, a "Minimal Acyclic Finite State Automaton" (MA-FSA) can be used.

3 Evaluation

For the evaluation of these implementations, I created unit tests with Google Test to check for the correctness of it, and benchmarks to test the running time and to compare the 2 implementations of the spell checker. All tests were made on a virtual machine with the following specifications: 2.3 GHz Intel Xeon® E5-2686 v4, 2vCPU, 8GiB RAM.

3.1 Trie

To check the correctness of the Trie implementation, I created 4 unit tests that verify the creation of the Trie, insertion, search and deletion of words in it.

TEST(Trie, Construct)

In this test,I create an empty Trie and check if his size is 0. Then, I create a tree with 2 elements and check the size again to see if it is 2 as expected.

TEST(Trie, HasWord)

To verify the correctness of the search of a word in the Trie, I created a dictionary with 2 words, and then I called the *has_word* function with these as argument, looking for the result "TRUE". Also, I called the same function with a word that was not in the dictionary, expecting "FALSE".

TEST(Trie, Insert)

Because I made sure that the *has_word* function is correct, I could use it to test the insertion of the words. After the creation of an empty Trie and the insertion of 2 words in it, I called the search function on them, expecting "TRUE".

TEST(Trie, Remove)
For a two-word dictionary, after the removal of one of them, I check the size of the Trie to make sure it was modified by the erase and also search the deleted word to make sure it's no loger there. In case I try to remove a word that does not exist in the Trie, the assertion will return "FALSE".

The results of the benchmarks for *insert*, *has_word* and *remove* functions are:

Crt.	**Function**	**Time (ns)**
1	Insert	1716.93
2	Search	1307.96
3	Remove	1849.51

Table 1: Benchmark pentru Trie

The third column in the table represents the medium running time for inserting 300000 words random. Similarly, for the *has_word* function, the time computed is the medium one for searching every word in the dictionary and the time for *remove* is for the removal of all the words in the dictionary. I chose to test everything using a random order so that no result is altered by it.

3.2 Levenshtein distance

The distance computed with the Levenshtein algorithm is tested in **TEST(Levenshtein, Dummy)**. To do that, I compare the results with the expected ones in 2 different cases.

To check both implementations, I created two tests: **TEST(TrieDictionary, SmallDictionary)** and **TEST(TrieDictionary, EnglishDictionary)**.

The first test verifies if the results given by the implementations is the same, using a a small dictionary. The second one does the same, only on a bigger dictionary.

For timing both implementations, two benchmarks were created. Because they are very similar, I will describe only one.

On a given dictionary, a number between 0 and 2 is chosen random. This number represent how many letters will be switched in the words. For each word, I measured how much time it took to get the suggestions to correct the word.

With a Python program, I computed graphics to show the differences between the implementation that uses an array to store the dictionary and the one that uses a Trie.

```
import numpy as np
import math
import matplotlib.pyplot as plt

plt.plot([10, 60, 100, 300], [0.378, 2.207, 3.654, 10.820], 'b-',
    label="Trie")
plt.plot([10, 60, 100, 300], [21.280, 127.800, 213.924, 654.600],
    'r-', label="Linear")
plt.legend(loc='best')
plt.ylabel("Time in s")
plt.xlabel("Number of words")
plt.show()
```

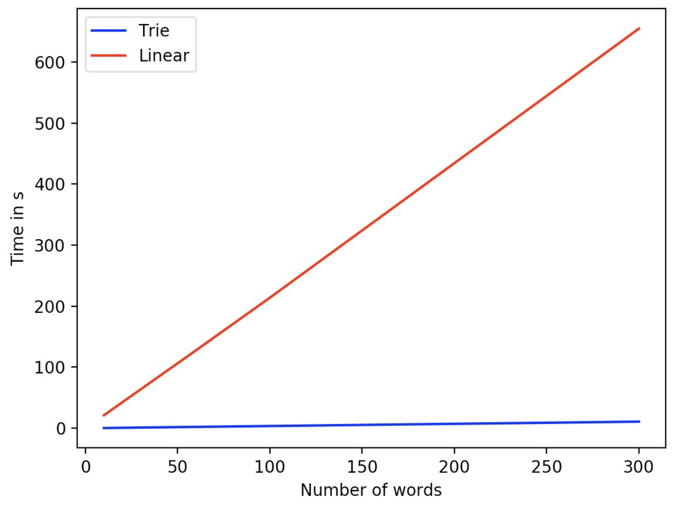

Fig. 1

4 Conclusions

Considering the all presented above, in a real-life situation when I have to implement a spell-checker, I would use the Trie implementation because it is much faster than the other one. But, I should also consider the fact that the Trie takes a lot of memory and this. can be a big issue. A better solution that solves the memory problem, is using a structure called "Minimal Acyclic Finite State Automaton" (MA-FSA) instead of a Trie. This structure removes the duplicated nodes and sequences, reducing a lot of memory use.

References

1. Black, P.E.: "levenshtein distance", dictionary of algorithms and data structures. U.S. National Institute of Standards and Technology (2016)
2. DM, M.: Bioinformatics: Sequence and genome analysis (2nd ed.). Cold Spring Harbor Laboratory Press: Cold Spring Harbor, NY (2004)
3. Sedgewick, R., Wayne, K.: Digit-based sorting and data structures (2016)