Levenshtein distance

Constantinescu Delia-Georgiana

Politehnica University of Bucharest

Abstract. This paper aims to assert the most efficient way of discovering all words within a dictionary situated at a Levenshtein distance no greater than K from a given word.

1 Introduction

1.1 Description

Levenshtein distance, also known as edit distance, is a measure of the similarity between two strings. It is defined as the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another. The Levenshtein distance is named after the Russian scientist Vladimir Levenshtein, who developed the algorithm in 1965.

A problem that arises is how to discover all words in a dictionary within a K edit distance from a given word. This appears in the context of most search engines, which tolerate spelling mistakes and still present content relevant to the user's intended search. This process is also known as "fuzzy search", a technique that allows a search algorithm to return relevant results even when the user's query does not exactly match the desired information.

1.2 Applications

The Levenshtein distance has many applications, including spell checking, natural language processing, and DNA analysis. It is commonly used to determine the difference between two strings and can be useful for finding the closest matching string in a database.

In linguistics, the Levenshtein distance is used as a metric to quantify the linguistic distance, or how different two languages are from one another. It is related to mutual intelligibility: the higher the linguistic distance, the lower the mutual intelligibility, and the lower the linguistic distance, the higher the mutual intelligibility. It can also be used in spell correction or computational biology to align sequences of nucleotides.

1.3 Algorithms

There are 2 main algorithms used for finding words within a given edit distance from a given string that will be analyzed in this paper:

- dynamic programming (both iterative and memoized recursive)
- trie data structure

Since these are deterministic algorithms, they will be compared based on complexity and efficiency rather than correctness.

2 Solutions

In order to calculate the Leveshtein distance between 2 given strings, we have to process all characters one by one starting from either from left or right sides of both strings. Assuming that the first string has a length of L_1 and the second has a length of L_2, we can intuitively follow these steps:

1. Last characters are equal: ignore them and compare strings of decremented lengths
2. Last characters differ: consider all 3 operations (insertion, deletions, replacements) and recursively compute minimum cost for all three operations and take minimum of three values
   (a) Insert: recur for L_1 and L_2 - 1
   (b) Remove: recur for L_1 - 1 and L_2
   (c) Replace: recur for L_1 - 1 and L_2 - 1

Algorithm 1 Naive recursive

```
1: procedure LEVESHTEIN(string str1, string str2, int L1, int L2)
2:
3:     if L1 == 0 then return L2
4:
5:     if L2 == 0 then return L1
6:
7:     if str1[L1−1] == str2[L2−1] then return Leveshtein(str1, str2, L1−1, L2−1)
8:
9:     return 1 + min(Leveshtein(str1, str2, L1, L2 − 1),
10:    Leveshtein(str1, str2, L1 − 1, L2),
11:    Leveshtein(str1, str2, L1 − 1, L2 − 1))
```

The space complexity of the above solution is $O(1)$, as no auxiliary space is needed. However, the time complexity is exponential. In the worst case, we may end up doing $O(3^L)$ operations, if none of the characters of two strings match.

2.1 Dynamic programming: iterative

Using the previous algorithm, many subproblems were solved multiple times, thus making many useless and resource-consuming operations. This problem qualifies as a dynamic programming problem (Overlapping Subproblems property). The dynamic programming solution involves constructing a temporary matrix that stores results of subproblems.

Algorithm 2 Dynamic programming iterative

```
1: procedure LEVESHTEIN(string str1, string str2, int L_1, int L_2)
2:     i ← 0
3:     while i <= L_1 do
4:         j ← 0
5:         while j <= L_2 do
6:             if i == 0 then
7:                 distancematrix[i][j] ← j
8:             else if j == 0 then
9:                 distancematrix[i][j] ← i
10:            else if str1[i − 1] == str2[j − 1] then
11:                distancematrix[i][j] ← distancematrix[i − 1][j − 1]
12:            else
13:                distancematrix[i][j] ← 1 + min(distancematrix[i][j − 1],
14:                distancematrix[i − 1][j],
15:                distancematrix[i − 1][j − 1])
16:            j ← j + 1
17:        i ← i + 1
       return distancematrix[L_1][L_2]
```

The method first creates a two-dimensional array to store the results of previously calculated subproblems, each cell representing the minimum number of operations required to transform the first i characters of the first string into the first j characters of the second one. Then, it fills the array in a bottom-up manner by iterating over the lengths of the two strings.

The algorithm first initializes the first row and first column of the matrix. The first row represents the case where the first string is empty, so the minimum number of operations is simply to insert all of the characters of the second string into the first. Same goes for the first column.

The algorithm then iterates through the rest of the two-dimensional array and fills out each cell based on the three possible operations described above. If the last characters of the substrings being considered are the same, then no operation is needed and the value in the current cell is simply the value of the cell diagonally above and to the left (which represents the minimum number of operations required to transform the substrings without the last characters). If

the last characters are different, then the algorithm considers all three operations and chooses the one that results in the minimum value.

This method can calculate the Leveshtein distance in a maximum time of O(L_1xL_2). However, it also requires O(L_1xL_2) additional space for the two-dimensional array, which is not suitable if the length of strings is greater than 2000.

2.2 Dynamic programming: memoized recursive

This method uses memoization in order to improve the time complexity of the recursive algorithm, as well as implementing the dynamic programming two-dimensional array for storing previous results.

Algorithm 3 Dynamic programming memoized recursive

```
1: procedure LEVESHTEIN(string str1, string str2, int L1, int L2)
2:
3:     if L1 == 0 then return L2
4:
5:     if L2 == 0 then return L1
6:
7:     if distancematrix[L1][L2] != -1 then return distancematrix[L1][L2]
8:
9:     if str1[L1 − 1] == str2[L2 − 1] then
10:        distancematrix[L1][L2] = Leveshtein(str1, str2, L1 − 1, L2 − 1)
11:        return distancematrix[L1][L2]
12:    else
13:        insert ← Leveshtein(str1, str2, L1, L2 − 1
14:        delete ← Leveshtein(str1, str2, L1 − 1, L2)
15:        replace ← Leveshtein(str1, str2, L1 − 1, L2 − 1)
16:        distancematrix[L1][L2] = min(insert, delete, replace)
17:        return distancematrix[L1][L2]
```

This method is very similiar to the previous one, except fot the fact that it makes use of the recursive stack in order to consider all possibilities.

In terms of complexity, there is not any significant improvement. The time complexity maintains itself at O(L_1xL_2), while the space complexity worsens. The recursive stack space added to that occupied by the matrix gives this algorithm a space complexity of O(L_1xL_2) + O(L_1+L_2).

2.3 Trie data structure

Rather than iterating through a dictionary and comparing each word directly with the given word, a better solution would be to store the dictionary using

a Trie data structure. When trying to find the closest matching words in a whole dictionary, possibly with many thousands of words, the straightforward way becomes costly and inefficient.

For each word, we have to fill in an L_1xL_2 table. An upper bound for the runtime is O(NxL^2), where N is the number of words in the dictionary, and L is the maximum word length.

How the algorithm works is, when looking at two words and composing the two-dimensional array that actually calculates the Levenshtein distance between them, we can move on to the next word lexicographically in the dictionary and slightly modify the array, rather than recomposing it from scratch. This is why a trie is a great way of storing a dictionary, because we can process the words in order, so we never need to repeat a row for the same prefix of letters. With a trie, all shared prefixes in the dictionary are collaped into a single path, so they can be processed in the best order for building up the Levenshtein tables one row at a time.

This method has the same time complexity of O(L_1xL_2), but has a much better space complexity for dictionaries: O(alphabet size * average key length * N) where N is the number of words in the trie.

3 Performance evaluation

The evaluation criteria consist of the time and space complexity of the algorithms applied to a set of tests. Since all the solutions for calculating the Levenshtein distance grant a correct result, there is no need to verify the correctness of the algorithms. However, the time and space complexities still need to be asserted. With this in mind, the algorithm which finds all words from the given dictionary within a K edit distance from the given word will be considered more efficient, as long as the memory it uses is not exponentially larger than that used by other algorithms.

3.1 Test input format

The test files consist of 2 numbers on the first line, N and K.

$$1 \leq N \leq 10^5 \tag{1}$$

$$1 \leq L, K \leq 10^3 \tag{2}$$

For the evaluation of the practical performance I have compiled a Java program that runs the algorithms for multiple input text files and generates output files consisting of the answers, al well as keeps track of the time taken by the algorithm to run. This method allows us to compare the respective runtimes and assert which algorithm is the most efficient one.

3.2 Evaluation

The generated tests consist of various dictionary densities, ranging from 20 to 400 words. I chose words of lengths ranging from 1 to 50 letters, in order to stress test the algorithm, as well as setting various values for K, particularly from 1 to 20.

The software used for generating the tests were some online random word and string generators, such as `https://randomwordgenerator.com` for words with maximum lengths of 10 letters, and `http://www.unit-conversion.info/texttools/random-string-generator` for randomized strings with more than 10 characters.

The time complexity assertion was made with the help of *java.time* library, which contains the *Instant.now*() and *Duration.between*() functions, which can determine time with a precision of nanoseconds.

Here are the results of the tests:

Test	Input file name	Iterative time	Recursive time
1	test1.in	0 s 1 ms 586100 ns	0 s 2 ms 514400 ns
2	test2.in	0 s 1 ms 401900 ns	0 s 0 ms 499900 ns
3	test3.in	0 s 0 ms 497600 ns	0 s 0 ms 665100 ns
4	test4.in	0 s 0 ms 499600 ns	0 s 1 ms 9900 ns
5	test5.in	0 s 0 ms 685700 ns	0 s 0 ms 499600 ns
6	test6.in	0 s 0 ms 499700 ns	0 s 0 ms 571800 ns
7	test7.in	0 s 0 ms 305500 ns	0 s 1 ms 66000 ns
8	test8.in	0 s 0 ms 499200 ns	0 s 1 ms 154800 ns
9	test9.in	0 s 0 ms 927600 ns	0 s 0 ms 799000 ns
10	test10.in	0 s 0 ms 489200 ns	0 s 0 ms 501300 ns
11	test11.in	0 s 23 ms 698700 ns	0 s 46 ms 996900 ns
12	test12.in	0 s 2 ms 202400 ns	0 s 2 ms 400500 ns
13	test13.in	0 s 9 ms 590300 ns	0 s 6 ms 236800 ns
14	test14.in	0 s 7 ms 659300 ns	0 s 15 ms 688100 ns
15	test15.in	0 s 2 ms 684700 ns	0 s 11 ms 13100 ns
16	test16.in	0 s 6 ms 132100 ns	0 s 21 ms 531700 ns
17	test17.in	0 s 5 ms 286100 ns	0 s 15 ms 639100 ns
18	test18.in	0 s 5 ms 199700 ns	0 s 15 ms 674700 ns
19	test19.in	0 s 0 ms 498800 ns	0 s 2 ms 351800 ns
20	test20.in	0 s 0 ms 499100 ns	0 s 1 ms 377700 ns

4 Conclusions

As expected, the results show that for shorter tests, the memoized recursive algorithm for calculating the Levenshtein distance has a slightly better performance that the iterative one in terms of execution time. However, for denser dictionaries, this becomes untrue. Considering the fact that the recursive algorithm also

occupies more recursive stack space, we cam consider the iterative algorithm as having a better performance.

It can be safely assumed that the most efficient way of storing and iterating through a dictionary word by word is by using a trie data structure. Searching for a key in a trie takes time O(L), where L is the length of the key being searched for, because the search starts at the root of the trie and follows the branches corresponding to the characters in the key, until it reaches the end of the key or a null reference. The space complexity of a trie is O(AxL), where A is the size of the alphabet being used and L is the maximum length of the keys being stored in the trie. Overall, trie data structures have very fast insertion and search times, but they can use a lot of space if the keys are long and the alphabet is large.

References

1. https://web.stanford.edu/class/cs124/lec/med.pdf
2. https://www.geeksforgeeks.org/edit-distance-dp-5/
3. https://en.wikipedia.org/wiki/Levenshtein_distance
4. http://dbgroup.cs.tsinghua.edu.cn/technicalreports/triejoin.pdf
5. https://stackoverflow.com/questions/4868969/implementing-a-simple-trie-for-efficient-levenshtein-distance-calculation-java
6. https://randomwordgenerator.com
7. http://www.unit-conversion.info/texttools/random-string-generator
8. https://www.baeldung.com/cs/string-similarity-edit-distance