Assignment 3 - Report

Armela Ligori
aligori18@epoka.edu.al
Epoka University - Theory of Computation (CEN 350)

Abstract

This assignment illustrates the utility of applying Levenshtein distance Algorithm in detecting the similarities between several text files. Since edit distance metrics are widely used for DNA comparison, the dataset used will comprise of 10 random DNA sequences. The edit distance between each two files will be calculated and will be used to express the similarity in percentage between those files.

Keywords – Edit Distance, Levenshtein Algorithm, Similarity, Deletion, Insertion, Substitution

1 Introduction

Edit Distance In many applications nowadays such as plagiarism detection, spellchecking, DNA comparison, genome analysis for disease diagnosis, speech recognition, linguistic distance etc., it is necessary to determine the similarity between two strings. A widely-used notion of string similarity is the edit distance (the inverse similarity). Named after the Soviet mathematician Vladimir Levenshtein, the Levenshtein distance, also known as edit distance, is a string metric for measuring the difference between two sequences [1]. It is the minimum number of single edits like insertions, deletions, and substitutions required to transform one string into the other [2].The lower the number of operations needed, the more similar are the two strings that are being compared.

2 Dataset

In this experiment a set of 10 DNA sequences of length 100 nucleotide will be considered. The sequences are randomly generated from this website, and stored into different text files, namely DNA0.txt, DNA1.txt ... DNA9.txt inside the directory *files/*.

2.1 Levenshtein Algorithm

Formal Definition
The Levenshtein distance between two strings a, b is given by $lev_{a,b}(\mid a \mid, \mid b \mid)$[3] where:

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

Figure 1: **Recursive Formal Definition**

- $lev_{a,b}$ (i, j) is the distance between string prefixes – the first i characters of a and the first j characters of b .
- The first part of this formula denotes the number of insertion or deletion steps to transform prefix into an empty string or vice versa.
- The second block (min) is a recursive expression in which:
  - The first line represents deletion.
  - The second one represents insertion.
  - The last line represents substitutions.

In classical Levenshtein distance, every operation has a unit cost. Transforming string s into t:

- Copy Operation $\rightarrow$ Cost 0
- Delete Operation $\rightarrow$ Cost 1
- Insert Operation $\rightarrow$ Cost 1
- Substitute Operation $\rightarrow$ Cost 1

Example The edit distance between 'Intention' and 'Execution' is 5. This is the minimum number of cost = 1 operations needed to transform string 1 into string 2.

Str 1	I	N	T	E	*	N	T	I	O	N
Str 2	*	E	X	E	C	U	T	I	O	N
Edit op.	D	S	S	C	I	S	C	C	C	C
Cost	1	2	3	3	4	5	5	5	5	5

Figure 2: **distance('Intention','Execution')**

Naive Recursive Approach based on the definition [4]
Process all characters one by one starting from either left or right side on both strings. Consider m: length of str1 and n: length of str2. There are two possibilities for every pair of character being traversed:

- If last characters of two strings are same, ignore last characters and get count for remaining strings: Recur for lengths m-1 and n-1.
- If last characters are not the same, consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.
  - Insert: Recur for m and n-1
  - Remove: Recur for m-1 and n

– Replace: Recur for m-1 and n-1

Complexity The Naive Recursive approach has an exponential time complexity. The worst case happens when the two strings have no matching characters. In this case, the complexity can become as big as O(3^m).

Dynamic Programming Approach
A more efficient way to calculate Levenshtein Distance is by using the Dynamic Programming Approach:

1. A matrix of size $(m+1) \times (n+1)$ is constructed. This matrix will contain the value $lev_{m,n}(i,j)$ at the position i, j.
2. The first row and the first column of this matrix are known by definition as having the values in ranges 0..m and 0..n, respectively. [3]
3. The approach is to start from upper left corner and move to the lower right corner.
4. Moving horizontally implies insertion (cost = 1), vertically implies deletion (cost = 1), and diagonally implies substitution if the two chars in the row and column doesn't match (cost = 1) or a copy operation if the chars match (cost = 0). Each cell minimizes the cost locally.
5. The number in the lower right corner is the Levenshtein distance required. [5]

		H	O	R	S	E
	0	1	2	3	4	5
R	1	1	2	2	3	4
O	2	2	1	2	3	4
S	3	3	2	2	2	3

Figure 3: **Levenshtein Table** – distance('HORSE','ROS')

Complexity Matrix-filling dynamic programming algorithm is O(m×n) time and space, where m: length of the first string, n: length of the second string. This matrix-filling yields the edit distance. Backtracing to find the optimal alignment/edit transcript is O(m + n) time. In comparison to the Naive Recursive implementation, it needs no discussion that the Dynamic Programming approach O(m×n) is way better than O(3^m). Hence, for this experiment, the Dynamic Programming Approach is going to be considered and implemented.

2.2 DP Levenshtein Implementation

This python implementation of Levenshtein Edit Distance is based on this website [6]. This implementation runs in O(n×m) time and requires O(n×m) space. View on Repl.it

Listing 1: Levenshtein Distance in Python

```python
def LevenshteinDistance(seq1, seq2):

    size_x = len(seq1) + 1
    size_y = len(seq2) + 1

    DP = [[0 for i in range(size_x)] for j in range(size_y)]

    for i in range(size_x):
        DP[i][0] = i

    for j in range(size_y):
        DP[0][j] = j

    for i in range(1, size_x):
        for j in range(1, size_y):
            if seq1[i-1] == seq2[j-1]:
                substitutionCost = 0
            else:
                substitutionCost = 1

            DP[i][j] = min(DP[i-1][j] + 1, DP[i][j-1] + 1, DP[i-1][j-1] + substitutionCost)

    return DP[size_x - 1][size_y - 1]
```

There are also other implementations that use less space, since to fill a row in DP table we require only one row - the upper row. However, I considered the classic straightforward DP approach, in order to have the complete levenshtein table stored in the memory.

2.3 PseudoCode

```
m = length of s1;
n = length of s2;
declare $D_L$[0..m,0..n];
for $i = 0$ to $m$ do
    $D_L$[i,0]= i;
end
for $j = 0$ to $n$ do
    $D_L$[0,j]= j;
end
for $i = 0$ to $m$ do
    for $j = 0$ to $n$ do
        if s1[i] = s2[j then
            cost = 0;
        else
            cost = 1;
        end
        $D_L$[i,j] = minimum($D_L$[i-1,j]+1,$D_L$[i,j-1]+1,
          $D_L$[i-1,j-1] + cost);
    end
end
```

Algorithm 1: FizzBuzz

3 Experiment and Results

Source Code can be found on here.

To test the Levenshtein algorithm, I compared 10 random DNA sequences with each other. The sequences as mentioned beforehand were randomly generated and stored in the files/ subdirectory.

I wanted to read each file once, hence with a for loop I iterated through each file and read the DNA sequence inside of it and appended it into a list of DNA-s.

```python
files = os.listdir('files')
dna =[]

for filename in files:
    with open('files/' + filename, 'r') as infile:
        dna.append(infile.read())
```

Since each file needs to be compared with each other file, 10×10 edit distances need to be computed, I decided to store the results into a 10×10 2D array. Then I started iterating through each file and comparing it with any other file by using the **LevenshteinDistance** function. This function will return the minimum number of required operations needed to transform one string into the other, so the edit distance. The similarity between the files can be found with the formula: (length - editDistance)/length *100%

$$Similarity = \frac{length \;-\; editDistance}{length} \times 100\%$$

Since length of the DNA = 100 this can be reduced to:

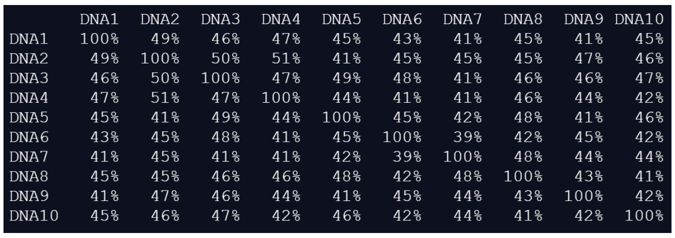
$$Similarity = (length - editDistance)\%$$

```
results = [['' for i in range(n)] for j in range(n)]
for i in range(n):
  for j in range(n):
    if i != j:
      if results[j][i] != '':
        results[i][j] = results[j][i]
      else:
        similarity = (100 − LevenshteinDistance(dna[i], dna[j]))
        results[i][j] = str(similarity) + '%'
    else:
      results[i][j] = '100%'
```

When i == j, a DNA sequence is being compared to itself. In this case, there is no need to calculate the edit distance. It is 0 and the similarity 100%. Also, half of the table duplicates finding similarities between two files. To remove redundant calculations, we can check if we have already calculated the table entry that is symmetric according to the left diagonal.

To display the results in a table format, I am using **pandas** library.

```
headers=[filename for filename in files]
pandas.set_option('display.max_columns', None)
pandas.set_option('display.width', None)
dataframe = pandas.DataFrame(results, headers, headers)
```



```
        DNA1  DNA2  DNA3  DNA4  DNA5  DNA6  DNA7  DNA8  DNA9 DNA10
DNA1    100%   49%   46%   47%   45%   43%   41%   45%   41%   45%
DNA2     49%  100%   50%   51%   41%   45%   45%   45%   47%   46%
DNA3     46%   50%  100%   47%   49%   48%   41%   46%   46%   47%
DNA4     47%   51%   47%  100%   44%   41%   41%   46%   44%   42%
DNA5     45%   41%   49%   44%  100%   45%   42%   48%   41%   46%
DNA6     43%   45%   48%   41%   45%  100%   39%   42%   45%   42%
DNA7     41%   45%   41%   41%   42%   39%  100%   48%   44%   44%
DNA8     45%   45%   46%   46%   48%   42%   48%  100%   43%   41%
DNA9     41%   47%   46%   44%   41%   45%   44%   43%  100%   42%
DNA10    45%   46%   47%   42%   46%   42%   44%   41%   42%  100%
```

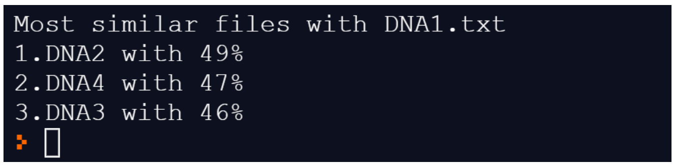
Figure 4: **Results**

	DNA1	DNA2	DNA3	DNA4	DNA5	DNA6	DNA7	DNA8	DNA9	DNA10
DNA1	100%	49%	46%	47%	45%	43%	41%	45%	41%	45%
DNA2	49%	100%	50%	51%	41%	45%	45%	45%	47%	46%
DNA3	46%	50%	100%	47%	49%	48%	41%	46%	46%	47%
DNA4	47%	51%	47%	100%	44%	41%	41%	46%	44%	42%
DNA5	45%	41%	49%	44%	100%	45%	42%	48%	41%	46%
DNA6	43%	45%	48%	41%	45%	100%	39%	42%	45%	42%
DNA7	41%	45%	41%	41%	42%	39%	100%	48%	44%	44%
DNA8	45%	45%	46%	46%	48%	42%	48%	100%	43%	41%
DNA9	41%	47%	46%	44%	41%	45%	44%	43%	100%	42%
DNA10	45%	46%	47%	42%	46%	42%	44%	41%	42%	100%

To find which files are most similar to file DNA1.txt, I sorted the entries at that file's specific row using heap-sort. To perform heap-sort, for each (file, similarity) tuple I created a node and pushed it into the maxHeap maintaining the heap invariant: the node with the highest similarity is the root node. After constructing the max-heap, I removed the first root (comparison to self with value 100) and popped the next 3 elements from the heap.

```
class Node:
  def __init__(self,file, similarity):
    self.file = file
    self.similarity = int(similarity[:−1])

  def __lt__(self, other):
    return self.similarity > other.similarity

def calculateTop3(row):
  maxHeap = []
  for i in range(len(results[row])):
    heapq.heappush(maxHeap, Node(files[i],results[row][i]))

  print('\nFrom the above table three most similar files with the ↩
      first file are:')
  heapq.heappop(maxHeap) # Remove comparison with self ↩
      (100%)
  for i in range(3):
    node = heapq.heappop(maxHeap)
    print('{}.{} with {}%'.format(i+1,node.file,node.similarity))

calculateTop3(0)
```

Top 3 files that are most similar to DNA1.txt are:



```
Most similar files with DNA1.txt
1.DNA2 with 49%
2.DNA4 with 47%
3.DNA3 with 46%
```

Figure 5: **Top 3 most similar files to DNA1.txt**

4 Conclusion

Levenshtein Distance is one of the most famous string metrics for measuring the difference between two sequences. As explained in this assignment, there are different approaches in implementing this edit distance. The naive recursive approach, is extremely inefficient, it requires O(3^n) time. Meanwhile, the Dynamic Programming approach that was considered in this experiment performs really well in O(nm) time and space. However, multiple files were compared with eachother and this resulted in an increase of time required to finish the comparisons.

References

[1] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” in *Soviet physics doklady*, vol. 10, pp. 707–710, Soviet Union, 1966.

[2] E. S. Ristad and P. N. Yianilos, “Learning string-edit distance,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 522–532, 1998.

[3] S. Grashchenko, “Levenshtein distance computation.” `https://www.baeldung.com/cs/levenshtein-distance-computation`. Accessed: 2021-2-9.

[4] GeeksForGeeks, “Edit distance.” `https://www.geeksforgeeks.org/edit-distance-dp-5/`. Accessed: 2021-2-9.

[5] Cuelogic, “The levenshtein algorithm.” `https://www.cuelogic.com/blog/the-levenshtein-algorithm`. Accessed: 2021-2-9.

[6] F. Hofmann, “Levenshtein distance and text similarity in python.” `https://stackabuse.com/levenshtein-distance-and-text-similarity-in-python/`. Accessed: 2021-2-9.