Report TDm 2

Algorithms for sequence processing: Optimal alignment and plagiarism detection software

Diego REYNA REYES & Ryan VAN GREUNEN

INF-4202B - Applied Algorithms: Strings and Images

January 15th, 2023

Abstract

During this report we'll give and explain our answers to the questions found in this website. The questions found there will be solved using the knowledge gained in the INF-4202B course, specifically, the knowledge related to string processing.

Objectives

1. Understanding new problems related to string processing.
2. Adapting known algorithms to new problems.
3. Creating algorithms of the desired time complexity.
4. Applying proposed algorithms using programming languages such as C.

Questions

1. **Propose an algorithm in O (| x | × | y |) to calculate the score of an optimal alignment between x and y (indications: we will try to reduce ourselves to a known problem).**

After analysing the problem, we realised we needed to build our x' and our y' stretches one element at the time. Each iteration we can perform one of the 3 following operations:

1. Ins(x[i]) → Inserts $x[i]$ at the end of the x' stretch, and inserts the special character "␣" at the end of the y' stretch.
2. Ins(y[j]) → Inserts the special character "␣" at the end of the x' stretch, and inserts $y[j]$ at the end of the y' stretch.
3. Ins(x[i],y[j]) → Inserts $x[i]$ at the end of the x' stretch, and inserts $y[j]$ at the end of the y' stretch.

Since we have 3 basic operations, we can easily adapt our Edit Distance algorithm to this problem, we'll just have to adapt the cost function we need to minimise. This function is, for two stretches of the same size:

$$d(x', y') = \sum_{i=0}^{i=|x'|} \sigma(x'[i], y'[i])$$

$$\text{where}$$

$$\sigma(a,b) = \begin{cases} 0, & \text{if } a == b \\ 1, & \text{otherwise} \end{cases}$$

For every iteration we can get the cost added at the iteration if we assign a cost to our operations. The cost assigned to each of our basic operations are:

1. Ins(x[i]) = Ins(y[j]) = 1; since the comparison between a word belonging in the alphabet and the special character will always give the same result: they're different
2. Ins(x[i], y[j]) = 1, if x[i]≠y[j];
3. Ins(x[i], y[j]) = 0, otherwise

Since our objective is to obtain the optimal score for every stretch between x_i and y_j, we find 3 possible solutions to find the optimal score between the strings:

1. Apply Ins(x[i], y[j]) to the optimal solution between x_{i-1} and y_{j-1}
2. Apply Ins(x[i]) to the optimal solution between x_{i-1} and y_j
3. Apply Ins(y[j]) to the optimal solution between x_i and y_{j-1}

Translating these three operations to function we get:

$$d(x_i, y_i) = min \begin{cases} d(x_{i-1}, y_{j-i}) + Ins(x[i], y[j]) \\ d(x_{i-1}, y_j) + Ins(x[i]) \\ d(x_i, y_{j-1}) + Ins(y[j]) \end{cases}$$

Using this equation we can modify the Edit Distance dynamic algorithm we've already solved:

Algorithm: Optimal Alignment

Input:

x → A string of length n

y → A string of length m

Output:

T → A table containing the optimal scores between all possible alignments between x_i and y_j

Code:

```
1  T[0][0] = 0
2  for i = 1 to n
3      T[i][0] = T[i-1][0] + Ins(x[i-1])
4  for j = 1 to m
5      T[0][j] = T[0][j-1] + Ins(y[j-1])
6  for i = 1 to n
7      for j = 1 to m
8          T[i][j] = min(T[i-1][j] + Ins(x[i-1]),
9                        T[i][j-1] + Ins(y[j-1]),
10                       T[i-1][j-1] + Ins(x[i-1],y[j-1]))
11 return T
```

Analysing the time complexity of this algorithm we can find that:

- Line 3 is of time complexity O(|x|)
- Line 5 is of time complexity O(|y|)
- Line 8, 9 and 10 are of time complexity O(|x| × |y|)

Given this complexities, we can conclude the time complexity of the algorithm is O(|x| × |y|)

2. **We consider the matrix T of size ($|x|$+1) × ($|y|$+1) such that $T[i][j]$ is the score of an optimal alignment between x_i and y_j, where x_i and y_j denote the prefixes of x and y of length i and j respectively.**

   Given the matrix T (obtained for example with the algorithm of question 2) and the sequences x and y, propose a program diagram returning an optimal alignment (x', y') of x and y.

   Assess the computational complexity of the proposed algorithm; if this one is not linear, propose another algorithm of linear complexity.

Before starting to design our algorithm we have to consider certain limitations imposed on us by our problem.

Firstly, since x'_i needs to be the same as x_i when we delete all the special characters, there's no point in evaluating any of the results for x_k for any $k > i$ and for any y_h for any $h > j$. This means that for point (i,j) we need to find the optimal route to (0,0).

The value we're going to use to find the optimal route is the optimal score stored in Table T. The next position we're going to travel, will be decided by the lowest score of all the valid neighbours of our point (i,j). Given the limits we set on the last paragraph, we find that all valid neighbours for point (i,j) are:

1. (i-1,j)
2. (i,j-1)
3. (i-1,j-1)

We need to consider the cases where more than one of the neighbours have the smallest value. In this cases we'll apply the next rules:

1. If there are 2 equal neighbours, and this neighbours are the (i-1,j-1) and any of the other 3 neighbours, we'll move to the (i-1,j-1), if and only if we're in the case where x[i] == y[j]
2. In the unusual case where all the neighbours have the same value, we'll move to the position (i-1,j-1)

3. In the yet to be seen, on testing, case where the smallest values are in the positions (i-1,j) and (i,j-1), we'll move to whichever case reduces the biggest value between i and j.

Having defined these rules, we can create our reconstruction algorithm:

Algorithm: RECONSTRUCTION ALGORITHM

Input:

x → A string of length n

y → A string of length m

T → A (n×m) table containing the optimal cost of the alignments x_i and y_j

(i,j) → Integer pair indicating the size of the prefix of x and y we want to find the optimal alignment of

Output:

x' → A stretch of x_i that yields the optimal alignment with y'_j

y' → A stretch of y_j that yields the optimal alignment with x'_i

Code:

```
x_prime = ""
y_prime = ""
k = i
h = j
while k >= 0 and h >= 0
    min_val = min(T[k-1][h-1],T[k-1][h], T[k-1][h])
    if min_val == T[k-1][h-1]
        if min_val == T[k][h-1] and min_val == T[k-1][h]
            Ins(x[k],y[h])
        else if min_val == T[k-1][h]
            if x[k] == y[h]
                Ins(x[k],y[h])
            else
                Ins(x[k])
        else if min_val == T[k][h-1]
            if x[k] == y[h]
                Ins(x[k],y[h])
            else
                Ins(y[h])
        else
            Ins(x[k],y[h])
    if min_val == T[k-1][h]
        if min_val == T[k][h-1]
            if k > h
                Ins(x[k])
            else
                Ins(y[h])
        else
            Ins(x[k])
    else
        Ins(y[h])
return x_prime.reverse(), y_prime.reverse()
```

As we can see, our algorithm uses multiple if statements. The complexity of these statements will be considered constant since they compare integers with integers or characters with other characters.

In line 6 we use the min function. Considering a brute force algorithm to search for the min, this functions complexity is linear O(k), with k being the number of characters inside the array, but given the context of our implementation, we see this function will have to work on an array for 3 elements every time, meaning that the time complexity of this line is O(3) meaning it's constant.

The Ins function time complexity is constant since it only adds 2 elements to a string. Which can be seen as a dynamic array, whose insert_back amortised complexity is O(1). Inside this function the value of h and k can change as shown next:

Case	Value of h	Value of k
Ins(x[k])	k = k - 1	h = h
Ins(y[h])	k = k	h = h - 1
Ins(x[k],y[h])	k = k - 1	h = h - 1

Given the previous table, we can obtain the most number of times our line 5 will be run. If we decrease the value of k, without modifying the value of h(doing k times the Ins(x[k]) function), and then doing the same with h (doing h times the Ins(y[h]) function) we'll find that line 5 will be executed n + m times. There's no point in using the Ins(x[k],y[h]) function to obtain the worst case scenario since the effect of this function is equivalent to running Ins(x[h]) and Ins(y[k]).

Finally, on line 32 we obtain the reverse of our x_prime and y_prime string, but this operation can be done in linear time, given that our string can be seen as dynamic arrays. Hence forth, the complexity of this line is O(2×(n+m)) if we consider the worst case scenario for our strings.

Taking this into account, we can find that our algorithm's complexity is O(n + m), making it linear.

4. The method proposed in the previous questions works correctly to align a text comprising a single line or a single paragraph. We now propose to align and match the different lines/paragraphs of a text. It is therefore a question of trying to match the lines (and not the characters) of a first text with those of a second. Propose a method, a resolution algorithm and an implementation for this new feature (indications: we will try to bring back a known problem, of which we already have an implementation).

To match paragraphs we're going to use the optimal alignment score, in fact, this algorithm is an extension of the algorithm proposed in questions 1 and 2. We'll use a table similar to the one constructed on question 1. This table will contain the score of the alignment when we align based on paragraphs.

Given this definition we can perform 3 basic operations every iterations:

1. Ins($x_{txt}[k]$) $\rightarrow$ Inserts paragraph $x_{txt}[k]$ at the end of the x'_{txt} stretch, and inserts a paragraph filled with $|x_{txt}[k]|$ special characters "␣" at the end of the y'_{txt} stretch.
2. Ins($y_{txt}[h]$) $\rightarrow$ Inserts a paragraph filled with $|y_{txt}[h]|$ special characters "␣" at the end of the x'_{txt} stretch, and inserts paragraph $y_{txt}[h]$ at the end of the y'_{txt} stretch.
3. Ins($x_{txt}[k]$,$y_{txt}[h]$) $\rightarrow$ Inserts the optimal stretch of paragraph $x\_txt[k]$ at the end of the x'_{txt} stretch, and inserts the optimal stretch of paragraph $y_{txt}[h]$ at the end of the y'_{txt} stretch.

Given this is a similar algorithm to the one shown in question 1, we can use the same basic structure, we only need to modify the score function, which is, for two stretches, both containing the k number of paragraphs and the same amount of characters on each paragraph:

$$s(x'_{txt}, y'_{txt}) = \sum_{i=0}^{i=k} d(x'_{txt}[i], y'_{txt}[i])$$

where

$$d(x', y') = \sum_{i=0}^{i=|x'|} \sigma(x'[i], y'[i])$$

where

$$\sigma(a, b) = \begin{cases} 0, & \text{if } a == b \\ 1, & \text{otherwise} \end{cases}$$

To apply a similar algorithm, we need to define a cost to our 3 basic functions:

1. Ins($x_{txt}[k]$) = $|x_{txt}[k]|$, since we're adding a paragraph of length $|x_{txt}[k]|$ filled with special characters. The score of alignment between this empty paragraph and any paragraph belonging to the alphabet will always be the length of the paragraphs.
2. Ins($y_{txt}[k]$) = $|y_{txt}[k]|$, since we're adding a paragraph of length $|y_{txt}[k]|$ filled with special characters. The score of alignment between this empty paragraph and any paragraph belonging to the alphabet will always be the length of the paragraphs.
3. Ins($x_{txt}[k]$,$y_{txt}[h]$) = d($x_{txt}[k]$,$y_{txt}[h]$), given that our best result is obtaining the optimal alignment between both paragraphs

Since our objective is to find the optimal alignment between every possible k and h first number of paragraphs of our texts, we find that to calculate the best solutions for texts $x_{txt}[k]$ and $y_{txt}[h]$ we can do the following options:

1. Apply Ins($x_{txt}[k]$,$y_{txt}[h]$) = d($x_{txt}[k]$,$y_{txt}[h]$) to the optimal solution between $x_{txt}[k - 1]$ and $y_{txt}[h - 1]$
2. Apply Ins($x_{txt}[k]$) to the optimal solution between $x_{txt}[k - 1]$ and $y_{txt}[h]$
3. Apply Ins($y_{txt}[k]$) to the optimal solution between $x_{txt}[k]$ and $y_{txt}[h - 1]$

If we describe the past steps in a function we find:

$$s(x_{txt}[k], y_{txt}[h]) = min \begin{cases} d(x_{txt}[k-1], y_{txt}[h-1]) + Ins(x_{txt}[k], y_{txt}[h]) \\ d(x_{txt}[k-1], y_{txt}[h]) + Ins(x_{txt}[k]) \\ d(x_{txt}[k], y_{txt}[h-1]) + Ins(y_{txt}[h]) \end{cases}$$

And if we insert this result in the pseudo code of our algorithm:

Algorithm: OPTIMAL PARAGRAPH ALIGNMENT

Input:

x_{txt} → A string containing n paragraphs

y_{txt} → A string containing m paragraphs

Output:

T → A table containing the optimal scores between all possible alignments between $x_{txt}[k]$ and $y_{txt}[h]$

Code:

```
1  T[0][0] = 0
2  for k = 1 to n
3      T[k][0] = T[k−1][0] + Ins(x_txt[k−1])
4  for h = 1 to m
5      T[0][h] = T[0][h−1] + Ins(y_txt[h−1])
6  for k = 1 to n
7      for h = 1 to m
8          T[k][h] = min(T[k−1][h] + Ins(x_txt[k−1]),
9                        T[k][h−1] + Ins(y_txt[h−1]),
10                       T[k−1][h−1] + Ins(x_txt[k−1],y_txt[h−1]))
11 return T
```

To analyse the time complexity of our algorithm, we see that line 8, 9 and 10 are executed O(n× m), while those lines time complexity is given by the Ins($x_{txt}[k]$,$y_{txt}[h]$) function, which has a complexity of O($|x_{txt}[k]| \times |y_{txt}[h]|$), and given this line is executed (n×m) times, we find that adding the time complexity of each time this line executed gives the next equation:

$$\sum_{k=0}^{n} |x_{txt}[k]| \times \sum_{h=0}^{m} |y_{txt}[h]|$$

And since we know that adding the size of all paragraphs of a text is |x| and |y|, we can define the complexity as O(|x|×|y|).

Our reconstruction algorithm is based on the same algorithm as question 2. We simply change the Ins function so it adds the stretch of the corresponding paragraphs instead of just the last character of the string. This algorithm is:

Algorithm: PARAGRAPH RECONSTRUCTION ALGORITHM

Input:

x_{txt} → A string containing n paragraphs

y_{txt} → A string containing m paragraphs

T → A (n×m) table containing the optimal cost of the paragraph alignments x_i and y_j

(i,j) → Integer pair indicating the size of the prefix of x and y we want to find the optimal alignment of

Output:

x' → A stretch of x_i that yields the optimal alignment with y'_j

y' → A stretch of y_j that yields the optimal alignment with x'_i

Code:

```
x_prime = ""
y_prime = ""
k = i
h = j
while k >= 0 and h >= 0
    min_val = min(T[k-1][h-1],T[k-1][h], T[k-1][h])
    if min_val == T[k-1][h-1]
        Ins(x_txt[k],y_txt[h])
    if min_val == T[k-1][h]
        Ins(x_txt[k])
    else
        Ins(y_txt[h])
return x_prime.reverse(), y_prime.reverse()
```

The reasoning that line 5 is executed O(n+m) times still stands, since we're keeping the same restrictions as question 2, but now we take into account that to do our insert function we have to use the reconstruct function we proposed for question 2. This means we're executing a function with O($|x_{txt}[k]| \times |y_{txt}[h]|$) time complexity. But if we consider that when we do Ins(a,b) with one of the string being empty, the time complexity is O(|a|), that the time complexity when both string are different to empty is O(|a| + |b|) and that we need to add every paragraph inside of x_{txt} and y_{txt}, we find that the time complexity is given by the function:

$$\sum_{k=0}^{n} |x_{txt}[k]| + \sum_{h=0}^{m} |y_{txt}[h]|$$

And since we know that adding the size of every paragraph in text adds up to the size of the text, the time complexity of this algorithm is: O(|x|+|y|).