Advanced Algorithms Project
Report

THE EDIT DISTANCE PROBLEM

Master Machine Learning Data Mining

Submitted by

Group 2	Names of Students
n° 11	FERNANDEZ Natalia
n° 11	KESKIN Mustafa
n° 11	TRAN Tuan-Anh
n° 11	UPENDRA Nisal

Under the guidance of
Leo GAUTHERON

University Jean Monnet
ST Etienne, France

December 2017

Contents

List of Figures

Chapter 1

Dynamic Programming Approach

1.1 Classic Approach (Alignment with Trace-Back Matrix)

The classic approach involves processing all characters one by one staring from either from left or right sides of both strings. If we traverse from right corner, there are two possibilities for every pair of character being traversed.

If m is the length of str1 (first string) and n is the length of str2 (second string).

- If last characters of two strings are the same, there is nothing to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
- If last characters are not same, we consider all operations on str1, 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
  - Remove: Recur for m-1 and n

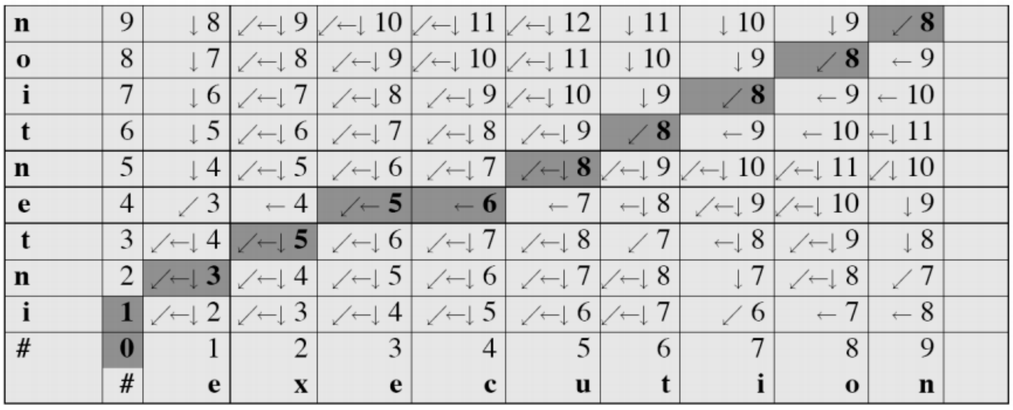
The time complexity is 0(nm) to find the numerical value of edit distance. Through this method it is possible to get the edit distance matrix for the

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

Figure 1.1: Edit Distance Formula from Wiki

two strings. To get the steps in the calculation, we implement a traceback matrix, with same dimensions as the edit distance matrix. This traceback matrix will keep note of all the operations done in each step to calculate the edit distance. By keeping track of the movements, it is possible to trace the steps required in the calculation.

The following image shows the functionality of the traceback matrix, as it keeps the moves for each step:

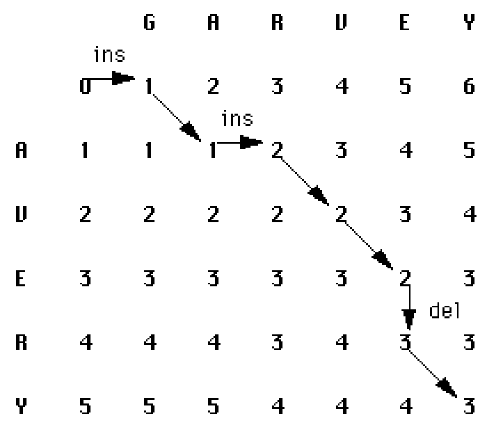


n	9	↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙←↓ 12	↓ 11	↓ 10	↓ 9	↙ **8**	
o	8	↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↙←↓ 11	↓ 10	↓ 9	↙ **8**	← 9	
i	7	↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9	↙ **8**	← 9	← 10	
t	6	↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙←↓ 9	↙ **8**	← 9	← 10	←↓ 11	
n	5	↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ **8**	↙←↓ 9	↙←↓ 10	↙←↓ 11	↙↓ 10	
e	4	↙ 3	← 4	↙← **5**	← **6**	← 7	←↓ 8	↙←↓ 9	↙←↓ 10	↓ 9	
t	3	↙←↓ 4	↙←↓ **5**	↙←↓ 6	↙←↓ 7	↙←↓ 8	↙ 7	←↓ 8	↙←↓ 9	↓ 8	
n	2	↙←↓ **3**	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙←↓ 8	↓ 7	↙←↓ 8	↙ 7	
i	**1**	↙←↓ 2	↙←↓ 3	↙←↓ 4	↙←↓ 5	↙←↓ 6	↙←↓ 7	↙ 6	← 7	← 8	
#	**0**	1	2	3	4	5	6	7	8	9	
	#	**e**	**x**	**e**	**c**	**u**	**t**	**i**	**o**	**n**	

Figure 1.2: Edit Distance matrix with all operations

1.2 Alignments in Dynamic Approach Without Traceback Matrix

The dynamic programming approach without using a traceback matrix to the edit distance problem is faster since it uses the same edit distance matrix used for calculating the edit distance to calculate the steps required for the conversion. This lowers the time complexity of the algorithm.



$$\begin{array}{c|ccccccc}
 & & G & A & R & V & E & Y \\
\hline
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
A & 1 & 1 & 1 & 2 & 3 & 4 & 5 \\
V & 2 & 2 & 2 & 2 & 2 & 3 & 4 \\
E & 3 & 3 & 3 & 3 & 3 & 2 & 3 \\
R & 4 & 4 & 4 & 3 & 4 & 3 & 3 \\
Y & 5 & 5 & 5 & 4 & 4 & 4 & 3
\end{array}$$

Figure 1.3: Finding Path from ED matrix

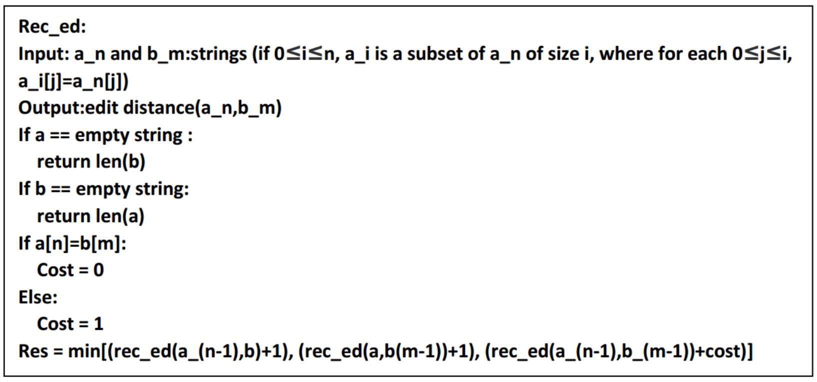
The calculation of edit distance and alignment is exactly similar as the method with the traceback matrix. When calculating the steps for the conversion, the algorithm starts at ED[n,m] where ED is the Edit distance matrix and n,m are the lengths of the two strings being compared. Afterwards, the algorithm calculates the conversion using the following rules:

- If the corresponding characters at the strings in the upper left diagonal position of the matrix are similar, then there is no change. Move to the upper left diagonal position and mark not changed in the steps.
- If they are not similar, get the minimum value from the corresponding cells. If it is up, then mark as remove, left as insert, diagonal as replace. Move to the marked position and update the steps with the move and index.

For example, if the change at ED[2,3] is to go to the left, since ED[2,2] is the lowest value, add a step as Inserted character A[1], where A is the string to be achieved and A[n-1] is the character of that string to be replaced.

Chapter 2

Recursive Approach



```
Rec_ed:
Input: a_n and b_m:strings (if 0≤i≤n, a_i is a subset of a_n of size i, where for each 0≤j≤i,
a_i[j]=a_n[j])
Output:edit distance(a_n,b_m)
If a == empty string :
   return len(b)
If b == empty string:
   return len(a)
If a[n]=b[m]:
   Cost = 0
Else:
   Cost = 1
Res = min[(rec_ed(a_(n-1),b)+1), (rec_ed(a,b(m-1))+1), (rec_ed(a_(n-1),b_(m-1))+cost)]
```

Figure 2.1: Pseudo code Recursive Algorithm

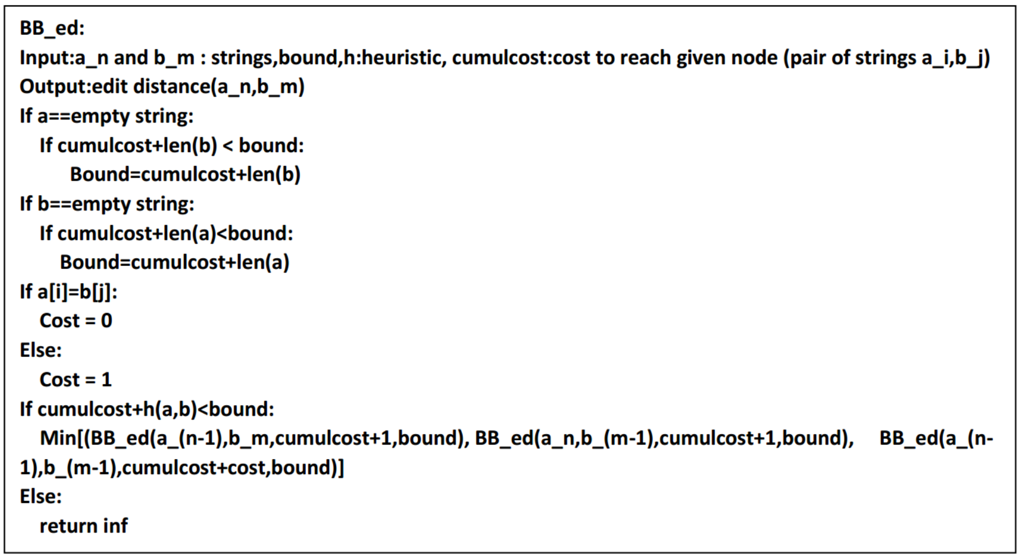
The recursive algorithm compute edit distance by applying the operations in the definition by making recursive calls: we will compute the minimum of edit distance of the 3 substrings obtained by removing last character for the first string, the second string and by removing from both, at each step and adding one for each except for the one which is removing the last character of the two strings when they are equals. To obtain the edit distance of those 3 substrings, we repeat the same recursively. The recursive call stops when one of the string is empty. The computation can be seen with a tree (that we browse in depth first according to the way we did the algorithm) where one node is a pair of strings (root node is (a,b) if the initial problem is looking for ed(a,b)), and for a given node that is not a leaf, there are three successors that corresponds to the pair of strings when we remove the last character of first string, the second string, and finally both, and each operations represented by branch that costs 1 excepted when we remove from both and they have common last character.

Pros: Easy to implement.

Cons: Bad time complexity.

Chapter 3

Branch and Bound



```
BB_ed:
Input:a_n and b_m : strings,bound,h:heuristic, cumulcost:cost to reach given node (pair of strings a_i,b_j)
Output:edit distance(a_n,b_m)
If a==empty string:
  If cumulcost+len(b) < bound:
      Bound=cumulcost+len(b)
If b==empty string:
  If cumulcost+len(a)<bound:
     Bound=cumulcost+len(a)
If a[i]=b[j]:
  Cost = 0
Else:
  Cost = 1
If cumulcost+h(a,b)<bound:
  Min[(BB_ed(a_(n-1),b_m,cumulcost+1,bound), BB_ed(a_n,b_(m-1),cumulcost+1,bound),    BB_ed(a_(n-
1),b_(m-1),cumulcost+cost,bound)]
Else:
  return inf
```

Figure 3.1: Pseudo code Branch and Bound Algorithm

Unlike the recursive algorithm where we were computing the ed of each successor nodes, the branch and bound algorithm consists in finding an upper bound that will tell us when it is useless to compute ed of a node (and so his successors). The upper bound means that the edit distance of the initial problem (a,b) which is root node is at most the upper bound. Note that we can use max(len(a),len(b)) as upper bound at initialization since the edit distance is upper bounded by this. We update the upper bound each time we have a complete path (reach a leaf) by the cost of this path. We define an evaluation function f(n)=g*(n)+h(n) where g* is the cost to reach the actual node from root, and h a lower bound that means it remains a cost of at least g(n) to reach leaf (+ length of not empty string) from n. It is then useless to compute the edit distance of a node that have a value of f greater than the upper bound since if we do this, the computation will return a final value that is greater than the cost of the current best path, whereas we are searching for the path that costs the minimum. We did the branch and bound algorithm in two ways where the one of them is done the same of the first but where we use a pile to store the visited nodes.

Pros: Slightly more efficient than the recursive algorithm.

Cons: Still have an exponential complexity.

Beside, we also success to create an other version of Branch and Bound using the same strategy but instead of basing on recursive approach, we use a loop while and a LIFO queue containing all nodes/leaves.

Chapter 4

Divide and Conquer

We can reduce the space complexity of the ED algorithm from O(mn) to O(m) by only storing one row of the memorization table.

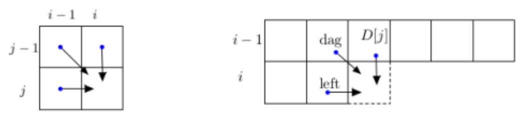


Figure 4.1: Illustation linear space methode

We know that in Dynamic Programming method, the value ED[i,j] depends only on value of 3 cells around. Therefore, by saving row i -1 of the table ED, the element on left (left) and diagonal element of the cell ED[i,j]. We can reduce the memory down to O(m).

```
LINEAR_SPACE_EDITDISTANCE(A[1..n],B[1..m]):
    for j←1 to m
        ED[j]←j
    for i←1 to n
        left ←i
        dag ←i−1
        for j←1 to m
            curr ←min{ED[j]+1,left +1,dag +I[A[i]≠B[j]]}
            left ← curr
            dag ←ED[j]
            ED[j]← curr
    return ED[m]
```

Figure 4.2: Pseudo-code linear space algorithm

Unfortunately, this technique is useful only if we are interested in the value of ED, but not in the optimal edit sequence. Because we cannot trace back to find an optimal sequence by using this technique.

Thats why we have to consider Hirshberg method to solve this problem. By using divide and conquer strategy, this method allows us to compute the optimal sequence in O(mn) time but only using O(m) space.

The idea is that there exist an interger h such that an optimal editing sequence that transforms A[1..n] to B[1..m] can be split into 2 smaller sequences, one transforming A[1..n/2] into B[1..h], the other transforming A[n/2+1..n] into B[h+1..n]. There are 2 methods to find the position h of string B:

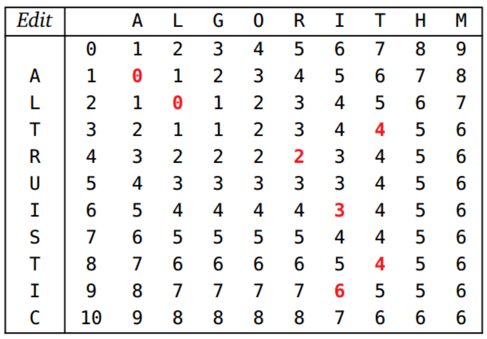
4.1 Method 1

The idea is to compute a matrix Hirshberg at the same time with the matrix Edit Distance using dynamic programming. The matrix Hirshberg is defined in the following formula:

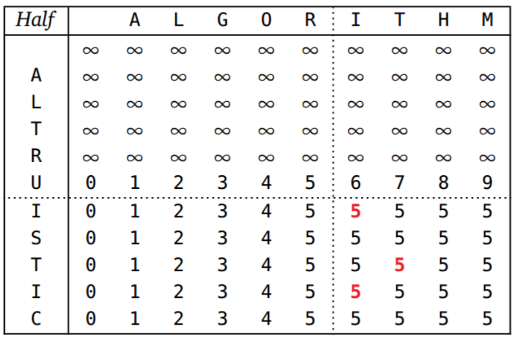
$$Half(i,j) = \begin{cases} \infty & \text{if } i < m/2 \\ j & \text{if } i = m/2 \\ Half(i-1,j) & \text{if } i > m/2 \text{ and } Edit(i,j) = Edit(i-1,j)+1 \\ Half(i,j-1) & \text{if } i > m/2 \text{ and } Edit(i,j) = Edit(i,j-1)+1 \\ Half(i-1,j-1) & \text{otherwise} \end{cases}$$

Figure 4.3: Hirshberg Formula to calcul matrix H

Ex: String A = ALTRUISTIC, string B = ALGORITHM, so the position h of string B is H[n,m] = 5. That means we can transform ALTRU to ALGOR and ISTIC to ITHM.
We can easily modify our earlier algorithm so that it computes H(n, m) at the same time as the edit distance ED(n, m), all in O (nm) time, using only O (m) space.



Edit		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6



Half		A	L	G	O	R	I	T	H	M
	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
A	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
L	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
T	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
R	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
U	0	1	2	3	4	5	6	7	8	9
I	0	1	2	3	4	5	5	5	5	5
S	0	1	2	3	4	5	5	5	5	5
T	0	1	2	3	4	5	5	5	5	5
I	0	1	2	3	4	5	5	5	5	5
C	0	1	2	3	4	5	5	5	5	5

Figure 4.4: Edis distance matrix and Hirshbergh matrix

4.2 Method 2

We compute a matrix ED using dynamic programming of string A[1..n/2] and B[1..m] with a forward (top-down) procedure. Then we compute another matrix ED of A[n/2..n] and B[1..m] with a backward (bottom-up) procedure.

Adding corresponding elements of the last row of the first matrix and the first row of the second one. By choosing a minimum, so we can receive the position h of string B.
Finally, to compute the optimal editing sequence that transforms A into B,

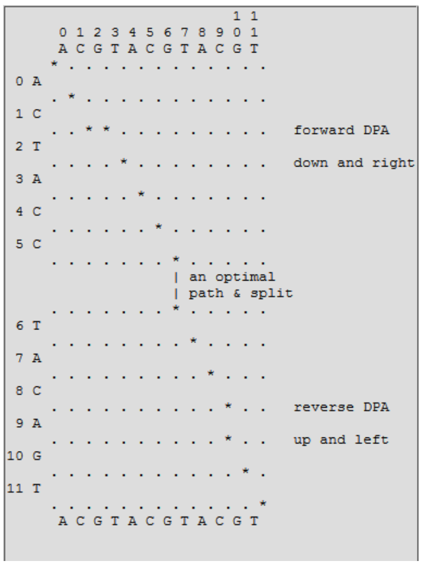


Figure 4.5: foward and backward matrix

we recursively compute the optimal sequences transforming A[1.. n/2] into

B [1.. H(n,m)] and transforming A[n/2 + 1.. n] into B [H(n, m) + 1.. m]. The recursion stop when the length of one string lower than 2, in this case we can use the dynamic programming algorithm to compute the edit distance and also find optimal sequences.

Pros: reducing memory down to O(m).

Cons: not really useful in case of short strings A and B.

Chapter 5

Stripe K Approach

In this approximated version, we just focus to k cells around the diagonal of the matrix ED in dynamic programming method. The goal is to reduce the time bound for the solution from O(nm) to O(kn). This approach is to compute the edit distance using dynamic programming but fill in only an O(kn) size portion of the full table.

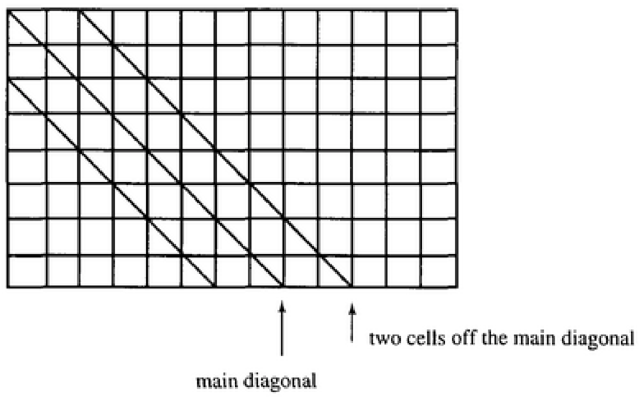


Figure 5.1: Illustration Stripe K matrix

In order to reach the final cell ED[n,m], we can increase the value k from 1 to m and compute the matrix until getting the solution. However, we can also get a good value of k from the beginning with the following condition: $\|n - m\| <= k$.

Pros: Time complexity is reduced to O(kn).

Cons: This version is useful when two strings A and B are believed to be fairly similar. Otherwise, the solution may not be optimal.

Chapter 6

Greedy Approach

6.1 Greedy Normal Approach

The greedy algorithm is the simplest one. It is a very fast algorithm (linear time) but in return, it gives only an approximation of the solution. It consists in setting the edit distance equal to 0 and then simply check directly if the last characters of the two strings are equal or not (and put 1 to the value of edit distance if the characters are different) and doing the same on the substring obtained by removing last characters of both, until the smaller string is empty (and we add to the value of edit distance the numbers of characters that remains on the bigger string at this final step).

Pros: Fast algorithm (linear time).

Cons: It returns only an approximation of the solution.

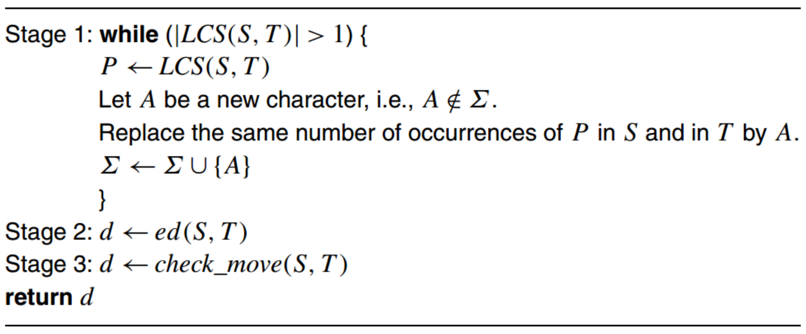
Listing 6.1: Pseudo-code Greedy Algorithm

```
Greedy_ed(a,b):
Input:a and b:strings
Output:edit distance (a,b)
value=0
if len(a)>len(b):
    for i in range (len(b)):
        if (a[-1-i])!=(b[-1-i]):
            value <- value+1
    value <- value+(len(a)-len(b))
else:
    for i in range (len(a)):
        if (a[-1-i])!=(b[-1-i]):
            value <- value+1
    value <- value+(len(b)-len(a))
return value
```

6.2 Optimized Dynamic Approach with Greedy Behavior

This approach is based on the publication Edit distance with move operations by Dana Shapira and James A. Storer. (`http://www.sciencedirect.com/science/article/pii/S157086670600030X`). However the algorithm they propose has move operations, which is the ability to swap characters in the string, which is out of the scope of this exercise. As such, we removed the move operations from the algorithms and modified it to be an improvised greedy approach.

The basic approach of this algorithm is to go through the strings and find the Longest Common Substrings (LCS) repeatedly and replace them with a different character. After this conversion, the dynamic programming approach is applied as before. The advantage of this approach is that for string conversions where there is a lot of common substrings, this can decrease time complexity.



```
Stage 1: while ($|LCS(S,T)| > 1$) {
            $P \leftarrow LCS(S,T)$
            Let $A$ be a new character, i.e., $A \notin \Sigma$.
            Replace the same number of occurrences of $P$ in $S$ and in $T$ by $A$.
            $\Sigma \leftarrow \Sigma \cup \{A\}$
            }
Stage 2: $d \leftarrow ed(S,T)$
Stage 3: $d \leftarrow check\_move(S,T)$
return $d$
```

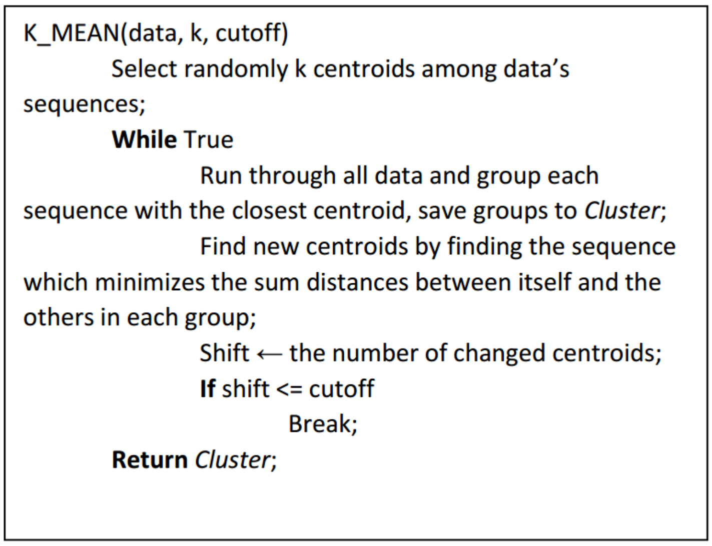
Figure 6.1: Pseudo code Greedy behavior

As per the pseudo code shown in the above image, we have omitted step 3 which checks whether the edit distance can be reduced by move operations instead of the regular, insertion, deletion and substitution steps.

Chapter 7

Protein Database

The method selected to cluster the protein data set is the k metoids. The iterative greedy algorithm is based in two steps. Before the first step, 21 examples of protein was chosen to be the centroid of each group, this selection was done randomly. At the first step, each protein was assigned in one of the cluster according with the smallest edit distance between the element and each centroid. After that, the second step that consist in update the centroids by chosen the protein which minimize the edit distance between the elements of the cluster and the new centroid .The two steps was repeated until no change was observed in the clusterings.



```
K_MEAN(data, k, cutoff)
    Select randomly k centroids among data's sequences;
    While True
        Run through all data and group each sequence with the closest centroid, save groups to Cluster;
        Find new centroids by finding the sequence which minimizes the sum distances between itself and the others in each group;
        Shift ← the number of changed centroids;
        If shift <= cutoff
            Break;
    Return Cluster;
```

Figure 7.1: pseudo code Kmean algorithm

In a first moment the K-means was applying using the dynamic classic algorithm and the stripe k to compute the edit distance. Both algorithms were applied in order to make a comparation.

To evaluate the speed and the efficiency of the two different approach in clustering, the K-means algorithm was applied in two different samples with 20% of the protein data set. The time spent and the percentage of accuracy is presented in the following worksheet :

Algorithm	**Time(s)**	**Percentage**
Dynamic Classic	14514.07	51.86%
Stripe K	3572.88	52.85%

As can be seen the Stripe K was much faster than the Dynamic Classic Algorithm. While the computation of the first spent more the four hours, the clustering using stripe K was running in only one hour. You can also see that, in spite of the stripe K is not optimal; the percentage of accuracy is very similar in both results.

The following graphics show the accuracy by class and cluster. Following figures show the percentage of correct and wrong classification, as can you can verify, some class was 100% correctly classified, but others are 100% misclassified. The main difference between the both methods involves the clustering of the class 2 and 8 which the results are opposite in the samples.

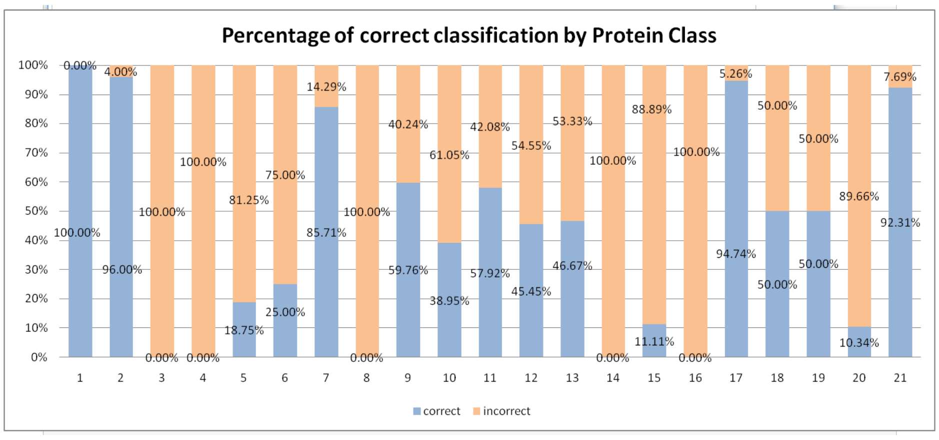


Figure 7.2: Percentage of correct classification using the classic dynamic approach(20% data)

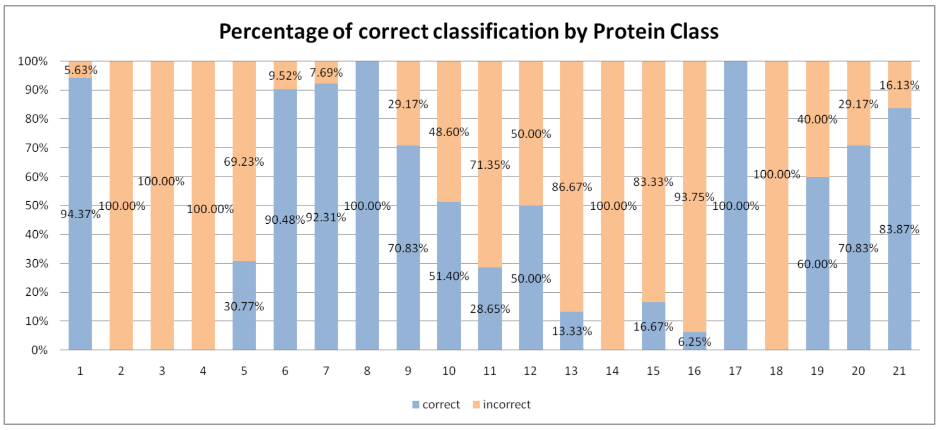


Figure 7.3: Percentage of correct classification using the stripe K(20% data)

Following figures present the scatterplot of the Class by the Cluster which the size of the ball is proportional to the frequency of the classification. The expected configuration of this graph was big balls in the diagonal and small ones otherwise. However, this is not exactly the configuration that we can observe. Apparently the bigger confusion is about clustering the proteins of the classes 1, 2, 9, 10 and 11.

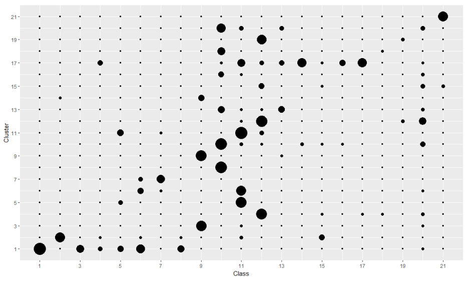


Figure 7.4: Percentage of correct classification using the dynamic approach(20% data)

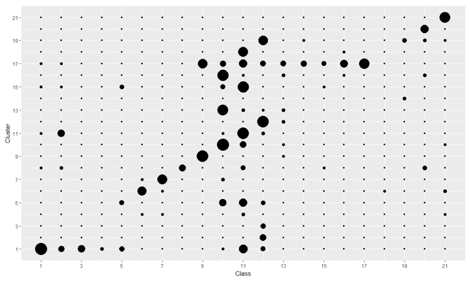


Figure 7.5: Percentage of correct classification using the stripe K approach (20% data)

Finally, as the stripe k had a similar accuracy in a time much smaller, the K-means was applied in 3200 protein data set (51%). The time spent by the computation of the cluster was 784,89, more than 13 hours and the accuracy was 48%, a little smaller than the results observed in the samples with 20% of the data. The accuracy by class is still similar with the previous results as showed by the following graphs:

In parallel with the kmean implementation, others cluster algorithm was studied. The Pro-Clara algorithmic looks very suitable and an attempt to implement has made. This algorithm is based in select the best partition by using the PRO PAM algorithm to find the k centroides in each sample. The Pro-PAM algorithm spent 4985.25 (more than one hour) to find the centroid in a sample with 82 protein, which make its implementation nonviable, and we keep the k-means analysis.

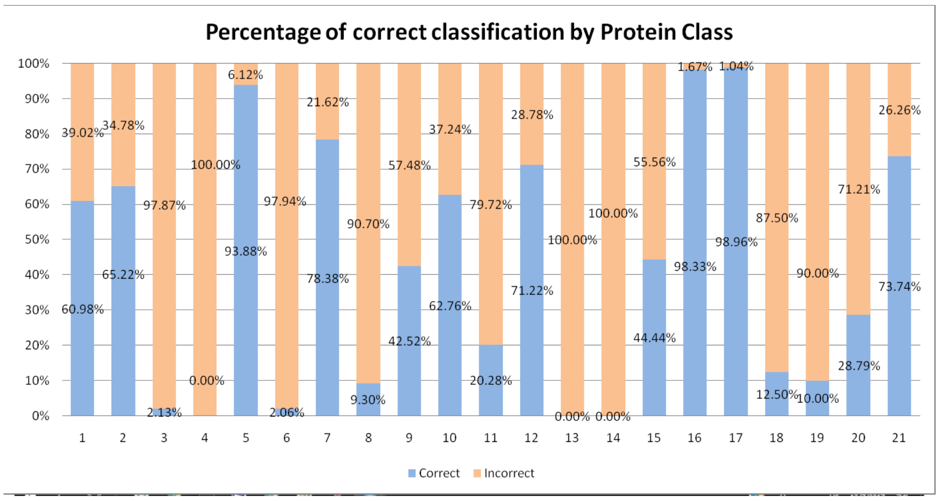


Figure 7.6: Percentage of correct classification using the stripe K approach(51% data)

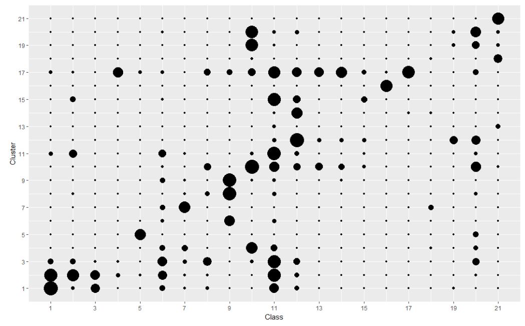


Figure 7.7: Percentage of correct classification using the stripe K(51% data)

As you can see in some class presented very different cluster, like class 16 that showed 100% of misclassification. And the classes 1,2,9, 10 and 11, as before, have apparently more wrong clustering than the others class.

Chapter 8

Conclusion

We computed the time complexity of each algorithm in to compare which ones were faster than the others in this way: drawing a sample of 200 pairs of strings, where the size of each strings are equals. The size of strings in the pair goes from 1 to 200. Weve computed this sample with each algorithm, except for the ones which have an exponential time complexity that we did apart (the recursive and the branch and bound). The results weve found for the four non-exponential algorithms are given by those graphics:

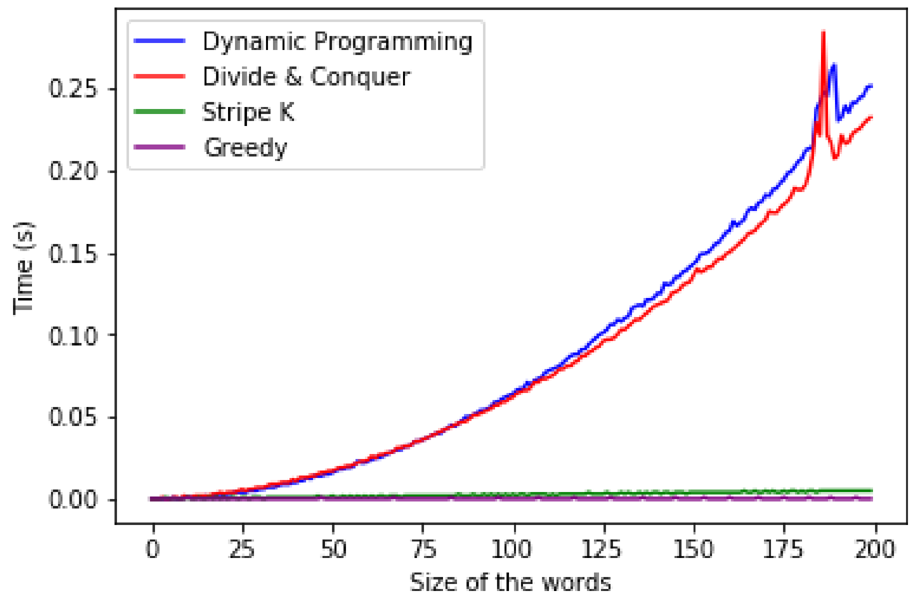


Figure 8.1: Comparison of effectiveness of algorithms)

We can see that the greedy and the stripe K are very fast algorithms, whereas Dynamic Programming and Divide & Conquer are slower. As the greedy gives only an approximation of the solution, we can say that the Stripe K algorithm appears as the best algorithm.

The results of the time complexity for the recursive and the branch and bound are less hopeful, as we would expect:

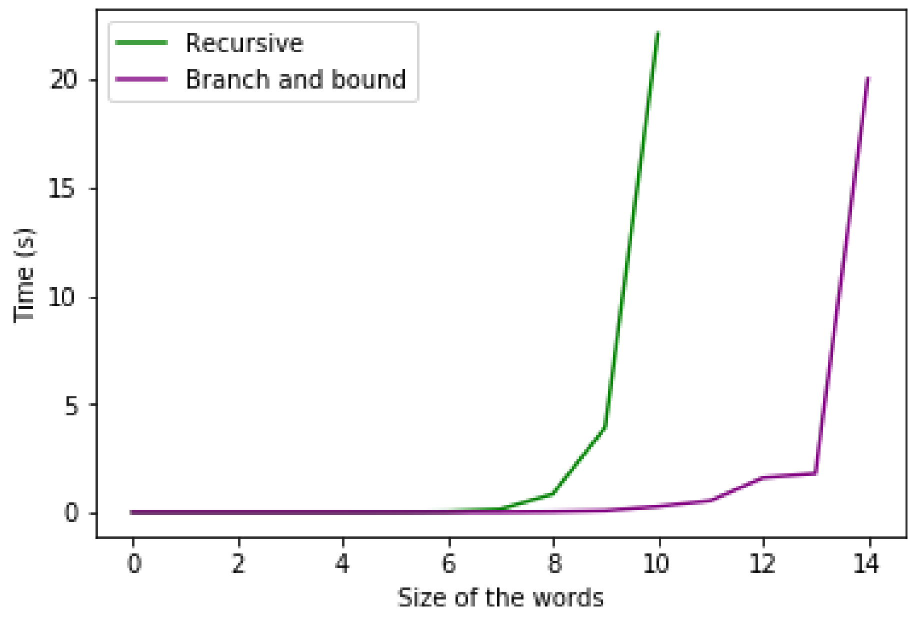


Figure 8.2: Comparison of effectiveness of algorithms)

We can see that the time complexity is too big to use those algorithms in practice for strings of size more than 15, however it works well for the other ones which are smaller than 15. Note that the gain obtained by using branch and bound instead of recursive is relatively poor since it allows strings of size only about 4-5 greater in size before a drastic increase of computation time.

We have also implemented a graphical user interface using Python FLASK and HTML, which show the functionality of these algorithms.