Project
Advanced Algorithmics and Programming

MLDM - M1 2020-2021

Pascalle Banaszek, Luiza Dzhidzhavadze, Pablo Sanchez, Richard Serrano

Contents

1 Introduction

The goal of the project was to create and test different implementations of algorithms on two strings among of many different types and sizes to compute the edit distance and output the alignment. The team implemented, analyzed, and compared seven algorithms using custom test sets to compare algorithmic time and space complexity. For the test cases and experiments, we created different alphabet sizes, lengths of strings, and different methods of creating strings. We analyzed the algorithm implementations and the quality of the results and the solutions. The team also attempted two clustering methods on protein sequences using edit distance and letter frequency to see if there was any correlation between the groupings.

2 Project Retrospective

The overall planning was relatively on time, with only few road blocks. The working schedule was designed so that our milestones were planned early to leave room at the end for going over schedule. We felt that going over schedule was inevitable on certain aspects because it was difficult to plan for unknowns. Fortunately, there was some spare time to complete some clustering designs, although results were inconclusive. An example of an unexpected roadblock was that we initially planned 6 days for pseudo code and 6 days for writing the actual algorithm. In actuality, due to some necessary modifications for the algorithms that were only found after their implementation, we ended up needing to retroactively update our pseudo codes. We overestimated the time needed to compute a theoretical complexity for each algorithm, which allowed us more time to work on writing and implementing the algorithms. Although our project ran over time, due to the fact that we scheduled completion ahead of time, we all ended finishing the total project (and more than expected) by the actual deadline. Teamwork, division of tasks, and proper planning allowed our team to be successful and exploratory in the completion of the project.

3 Group Description and Partition

The whole project was divided between four group members. The individual project contributions of each member are presented below:

1. Pascalle Banaszek: 20 %
    - Dynamic programming with single matrix algorithm
    - Cluster method with alphabet frequency
    - Project planning and timeline
    - Other contributions
2. Pablo Sanchez: 40 %
    - Branch and bound algorithm
    - Clustering on edit distance
    - Test set generation and experimental complexity tracking with custom profiler
    - Other contributions
3. Luiza Dzhidzhavadze: 20 %
    - Pure Recursion Algorithm
    - Other contributions
4. Richard Serrano: 20 %
    - Divide and conquer algorithm
    - Greedy algorithm
    - Dynamic programming with multiply matrices algorithm
    - Other contributions

4 Experimental Setup Conditions

The string and test set generation involved two main controlled conditions for the different experiments. The first condition was size of the alphabet used to generate strings and the second condition was how the two strings were generated. String length was also a control factor in evaluating the effectiveness of each of the different algorithms. In order to track the time complexity of each algorithm, a custom-coded profiler was created to track the length of time each algorithm took to run.

Experimental Conditions:

- String Generation

  Same string: Both strings being the same.

  Random string: Both strings being generated randomly.

  At most 20%: First string is generated randomly, while the second one gets generated copying the first one with a 20% chance of changing each character.

- Alphabet Lengths ; Length 2 (binary) ; Length 8 (octdigits) ; Length 10 (digits) ; Length 16 (hexdigits) ; Length 26 (ascii lowercase letters) ; Length 52 (ascii lowercase and uppercase letters)

Two test sets were generated for each pair of conditions. The first contains ten strings for each length from one to ten which was processed by every algorithm. The second contains ten strings for each length from five to two hundred with steps of five, run for the algorithms that have polynomial complexity.

5 Algorithms

5.1 Dynamic Programming Algorithm

See pseudo code in section 10.1.1 for Dynamic Programming with a Single Matrix and in section 10.1.2 for Dynamic Programming with Multiple Matrices.

5.1.1 Specific Features

Two different approaches were taken with the dynamic programming algorithm, in order to analyze the effects on complexity. Each method was straightforward to implement, succeeded in computing edit distance for both small and large strings (regardless of alphabet size), and was efficient on both time and space complexity.

- The first dynamic programming method used multiple matrices to compute the edit distance (it will be called the multiple matrix solution moving forward for simplicity). It began by creating a top-down matrix and a bottom-up matrix, then deduced the optimal path with the sum of these matrices. It is an approach very similar to what was done in the course with the Longest Common Subsequence.
- The second method, the single matrix solution, used only one top-down matrix to determine edit distance followed by a backtracking method to determine the optimal path. This is a more creative method as it was discovered outside of the course, and only uses one matrix to compute both edit distance as well as optimal alignment, saving time and storage.

5.1.2 Complexity

Theoretical Complexity
Theoretically, a dynamic programming algorithm for edit distance would have $O(n^2)$ for strings of lengths n and m assuming $n > m$. This approach has been shown to be the fastest method, while other methods with the same theoretical complexity were slower, but only by a constant.

- Theoretically it is easy to prove the upper bound of time complexity, but difficult to deduce the constant multiplier between algorithms without running experiments.
- The space complexity of both dynamic approaches is easy to deduce theoretically using the size of both strings to compute the size of each matrix. The only difference is that the first approach uses 3 matrices of the same size, where the backtracking approach only uses 1. This would imply a space complexity of $O(n^2)$.

Experimental Time Complexity

- Both methods were theorized to have a time complexity of $O(n^2)$. (Assuming m and n are the lengths of the respective strings with $n > m$). Both implementations showed incredibly similar experimental results fitting the shape of $O(n^2)$. You can note the backtracking method is faster than the other method by a constant between 2.8 and 3.

Experimental Space Complexity

- The backtracking method only uses one matrix to compute both edit distance and alignment, while the other method uses 3 matrices. This is simple to deduce that the backtracking algorithm has an experimental space complexity of $O(m * n)$ (with m and n being the lengths of each string), while the other algorithm has an experimental space complexity of $O(3 * m * n)$.

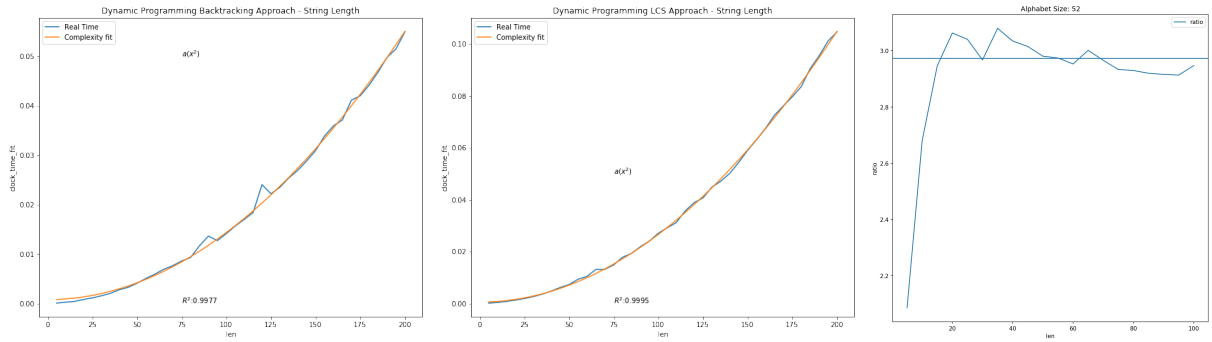


Figure 1: Time complexity for dynamic programming algorithms. The first figure on the left is with single matrix, the second with multiple matrix, and the third shows the constant multiplier of their time comparison.

5.1.3 Quality of Solution

Both approaches to the dynamic programming algorithm always produce the optimal solution of the output alignment and edit distance. The same is true on all strings of varying sizes and alphabets, as well as on the protein data set. The only difference is that the backtracking method has optimal storage compared to the other approach. In fact, this algorithm was used as the benchmark for the other algorithms when comparing optimal solution because by construction dynamic programming algorithms provide the optimal solutions.

5.2 Greedy Algorithm

In few words, the greedy approach is meant to give a good approximation of the optimal solutions in a reasonable amount of time by making locally optimal choices.

5.2.1 Specific Features

Here, the greedy approach is composed of two parts : (see pseudo-code in section 10.1.3 for more details.)

- The first part is to align the strings X and Y (renamed as *X1* and *Y1*) by adding "-" before and/or after each strings, such that they both have the same length.

  For example, let's say that $X = H\ E\ L\ L\ O$ and $Y = H\ O\ L\ A$. Then possible alignments could be :

$$\begin{array}{lccccccc} X1 & = & H & E & L & L & O & - \\ Y1 & = & - & - & H & O & L & A \end{array} \quad OR \quad \begin{array}{lccccccc} X1 & = & H & E & L & L & O & \\ Y1 & = & H & O & L & A & - & \end{array} \quad etc.$$

- The second part is to go from left to right of both strings and compare them, recording the operations to do to go from *X1* to *Y1* and incrementing the Edit_Distance by 1 each to the operation is different then *"keep"*.

Repeat these two steps for each possible alignment and return the best alignment/Edit_Distance.

5.2.2 Complexity

Theoretical Complexity

The first step will, in a first place, add a maximum of $k\times$"$-$" after or before the strings - k being equal to $\min(len(X) - len(Y), \lceil \frac{len(Y)}{2} \rceil)$ considering the $len(Y) \leq len(Y)$.
After, it will add "$-$" before and after the shortest string, keeping it's length equal to the one of the longest string. This is $len(X) - len(Y)$ possibilities (considering $len(Y) \leq len(X)$).

Thereby, let's write $n := \max(len(X), len(Y))$ the complexity of the step one is $O(n)$.

The step two will count the differences between the two strings, which length will always be lower then $2 \times n$. So the complexity of this step is $O(n)$

And we get the **time complexity** of the whole algorithm be $O(n^2)$.

The **space complexity**, by construction, is $O(2n)$.

Experimental Complexity and Run-Time We can see here that the theoretical complexity of the greedy approach is $O(n^2)$ (see orange line in figure 2 for the best fit with function $f : x \to ax^2+b$ with a and b real numbers).

The experimental complexity in anyway is very good since even for 200 length strings the algorithm is able to output an edit distance and an alignment in less than 2 ms.

We have to remember that the greedy approach doesn't output an optimal solution. Thus if the edit distance has to be optimal it can't be used.

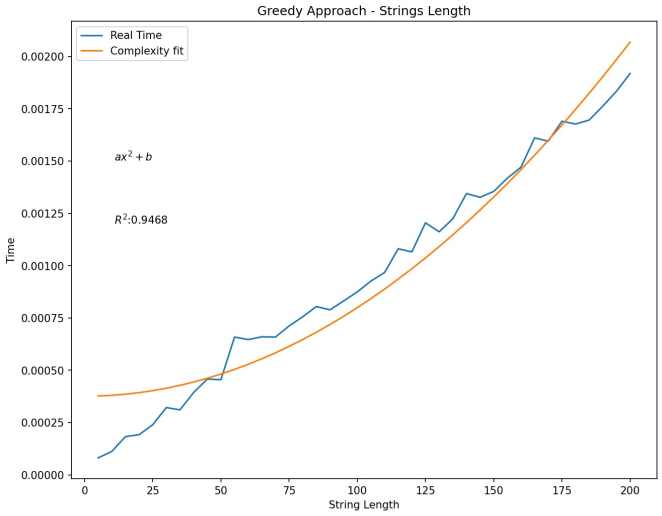


Figure 2: Greedy algorithm - run-time and comparison with theoretical complexity

5.2.3 Quality of Solution

For any strings, the approximation made by the greedy approach is the fastest one but for some strings it's not the optimal one.

• **2 strings with at most 20% of difference** (*See figure 3*)

As we can see on the graph below, when the 2 strings have at most 20% difference, the greedy approach returns the optimal edit distance (and thus alignment).

We have to be critical toward our results because the way the strings are generated has a big influence on the results driven here.

The generation of the strings is made as follow :

- The first strings is randomly generated.
- The second strings is generated by taking one by one each element of the first one with a probability of 20% of changing each letter.

Then, the greedy approach surely outputs the solution where there is no changes made to the strings (the strings are just align as originally with substitution of the changed letters in the second string).

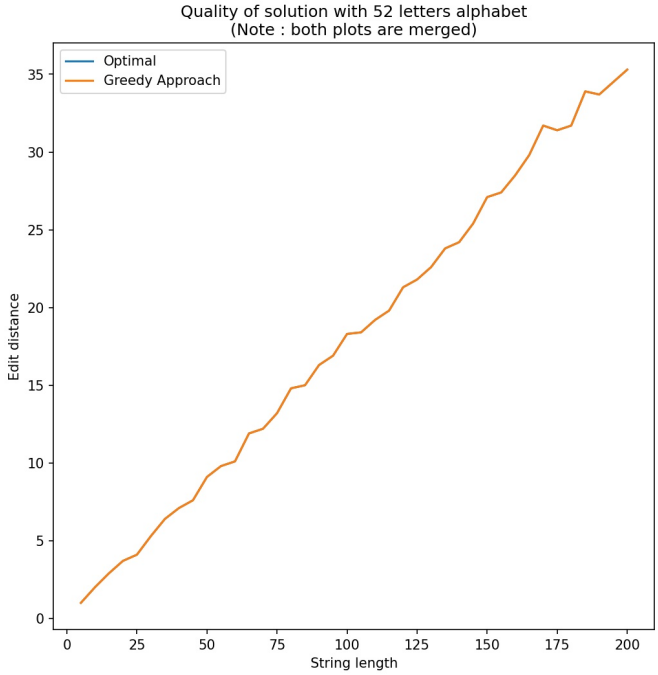


Figure 3: Greedy algorithm - strings different by at most 20% - Quality of solution

• **Randomly generated strings** (*See figure 4*)

Here, with randomly generated strings, we can see that the greedy algorithm differs from the optimal solution.

Also, we may add that using the Protein data base we get about 15% of the edit distances returned that are not optimal results.

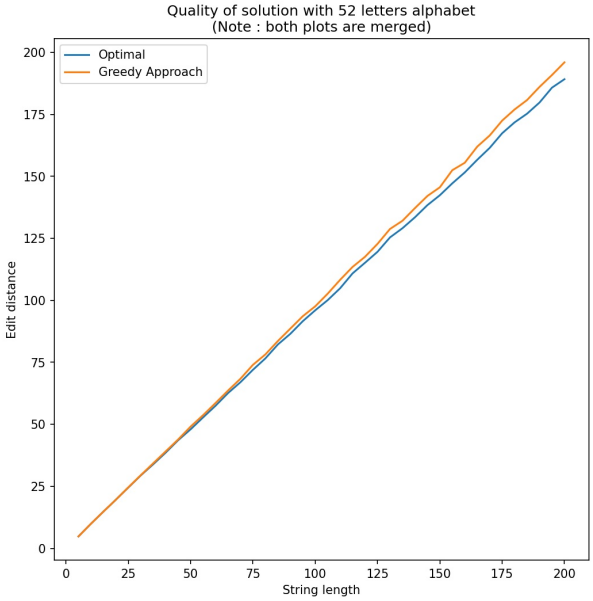


Figure 4: Greedy algorithm - strings of randoms letters - quality of solution

5.3 Divide and Conquer Algorithm (in linear space)

5.3.1 Specific features of the algorithm

This algorithms uses a space efficient version of dynamic programming which records only a locally optimal Edit_Distance. Then, divides the strings into smaller problems and starts again – this is the divide and conquer part. The coordinated (in the basis of the two strings) corresponded to each Edit_Distance found is kept into an array P.

As the dynamic part is space efficient the only not constant storage is P, which is of size $O(m+n)$ (with $m := len(X)$ and $n := len(Y)$, X and Y being the two strings inputted in the algorithm).

See pseudo-code in section 10.1.4.

The dynamic part was the easiest to implement because the rule is close to the one saw for Longest Common Sequence algorithm and it was possible to re-use the code from dynamic programming algorithm.

The whole algorithm was globally hard to implement because the divide and conquer part is recursive on sub-strings. The algorithm has to keep track of the absolute coordinate with respect to the original strings.

After running the divide and conquer part of the algorithm we won't have an alignment but only a "path" from which we have to deduce the alignment, which is the third and last part of the algorithm.

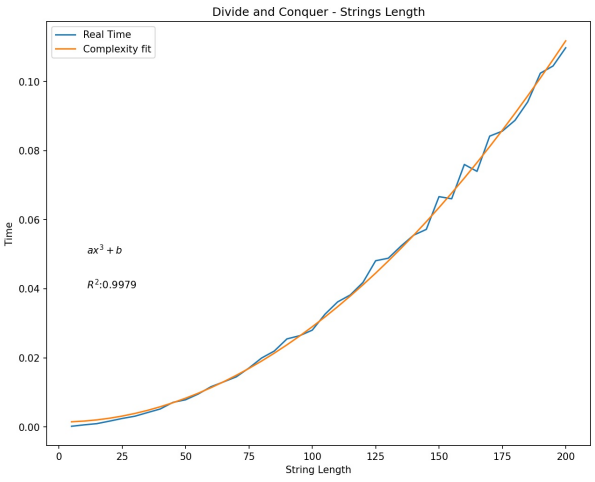


Figure 5: Divide and Conquer - run-time and comparison with theoretical complexity

5.3.2 Complexity

Theoretical Complexity

The theoretical complexity of this algorithms is the same as the one of the dynamic programming approach. That's to say the **time complexity** is $O(n^3)$ – we assume that $n \geq m$.

Note : as said earlier, there is a third part necessary to turn the "path" (recorded during the divide and conquer processing) into an actual alignment. This part is computed in $O(n+m)$ as it consist in going through the "path" P once.

Experimental Complexity and Run-Time

The empirical complexity (see blue line) is very close to

the theoretical complexity (see orange line).

5.3.3 Quality of Solution

The solution with divide and conquer is always optimal.

5.4 Pure Recursion Algorithm

5.4.1 Specific Features

The algorithm checks if last characters of two strings are the same. If it is the case, last characters are ignored and recursion is called for strings with remaining characters of lengths n-1 and m-1. Otherwise, considering all three operations on the last character of the first string, the algorithm recursively compute minimum cost for all three possible operations and take the minimum (*See pseudo-code in section 10.1.5*):

1. REPLACING : recursion call for n-1 and m-1
2. DELETION : recursion call for n-1 and m
3. INSERTION : recursion call for n and m-1

5.4.2 Complexity

Theoretical Complexity
In the case, when one of two strings is empty, theoretical time complexity is defined as :

- ***T(0,n) = n*** where m is a length of the first string.
- ***T(m,0) = m***, where m is a length of the first string.

In the case, when both strings are not empty, theoretical complexity is defined as:

- ***T(n,m) = T(n-1,m-1) + T(n-1,m) + T(n,m-1) + 1.***

The longest path will have the length m+n, so an upper bound of time complexity is $\boldsymbol{O}(3^{(n+m)})$.

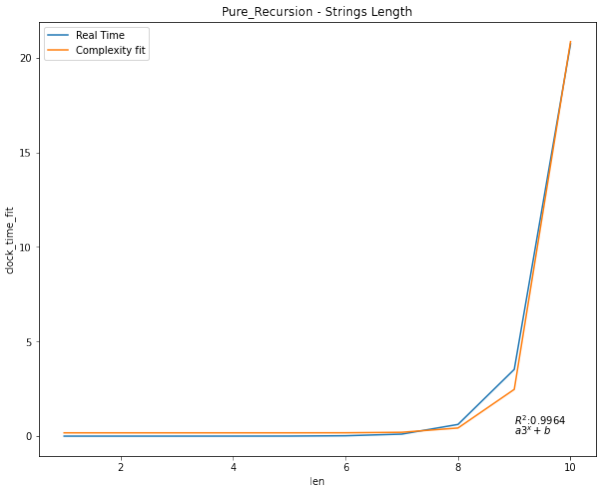


Figure 6: Pure Recursion - run-time and comparison with theoretical complexity

Experimental Complexity and Run-Time
Experiments on 3 different types of strings (same strings, 20 % of difference strings and random strings) showed that the run time grows exponentially as the length of strings increases *(See Figures 17, Figure 18, Figure 19 in the Appendix Section)*.

The theoretical complexity was compared to the experimental one in the most difficult type of experiment: computing Edit Distance of a pair of random strings with an alphabet of 52 letters and length at most 12 characters *(See Figure 6)*. The experimental time complexity well matches with the graph of theoretical complexity, exponentially increasing in terms of length of strings.

5.4.3 Quality of Solution

Pure Recursion approach is not an optimal solution for Edit Distance problem. This algorithm has an Overlapping subprolems property, since it solves same subproblems several times. That is why computing Edit Distance for strings longer than 12 characters is not efficient.

5.5 Branch and Bound Algorithm

5.5.1 Specific Features

Two different approaches where used, giving two different branch and bound algorithms. This allows us to compare.

- The first one estimated the maximum possible edits, giving an upper bound on the cost of the set of solutions in a branch. The main benefit in this approach is cutting branches that are worse than the current best solution.
- The second counts common letters between both strings and penalizes deleting or replacing them over skipping them. This approach benefits by encouraging using common letter for matches, to reduce the final cost.

The pseudo-code explaining the algorithm along with the different heuristics can be found in section 10.1.6.

5.5.2 Complexity

Theoretical Complexity
The theoretical time complexity of the branch and bound is the same as the pure recursion implementation. This being $O(3^{n+m})$, where n and m are the length of the strings to compare. In terms of upper bound it remains the same as pure recursion, but due to the branch cutting it has lower run times that recursion.

Experimental Complexity and Run-Time

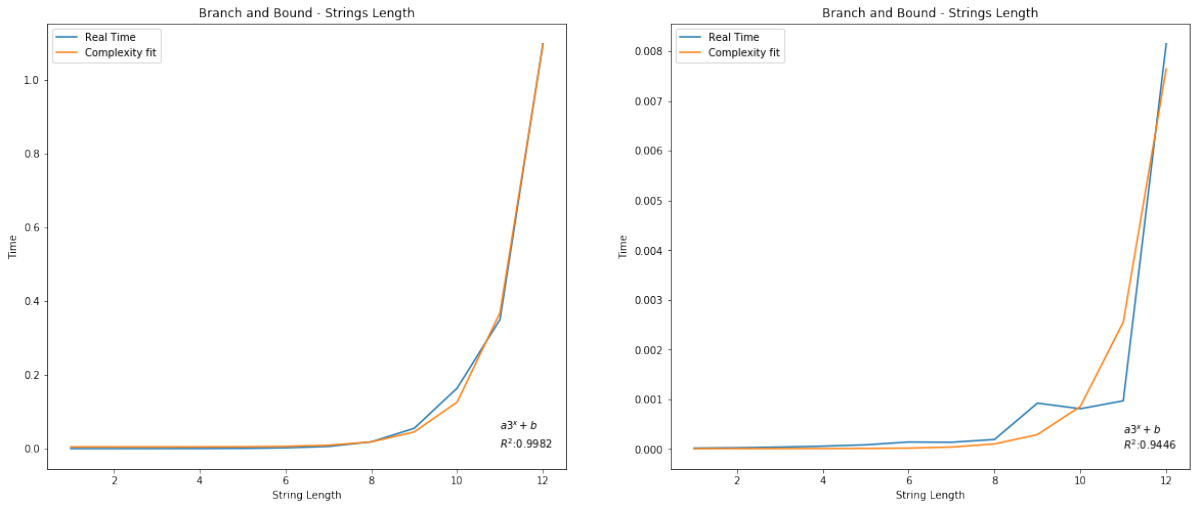


Figure 7: Left : Complexity with String Length heuristic
Right : Complexity with Frequency Count heuristic.

To determine the experimental complexity, the theoretical complexity was compared to the most difficult scenario in the experiments. This being a pair of random strings with an alphabet of 52 letters, ranging from length 1 to 12.

In both implementations of branch and bound, the experimental time complexity matches very well with the theoretical one, having an exponential growth with respect of the length of the strings.

5.5.3 Quality of Solution

Branch and bound does not guaranteed the optimal solution, however in both implementations the obtained solutions where the optimal one, or very similar. With an eleven length strings the Letter Frequency implementation was not able to obtain the optimal solution, but it deferred by less than 10%. Due to the exponential nature of the algorithm, it was not run with large strings (more than 12 characters). For the rest of the strings, the optimal solution was obtained by both implementations.

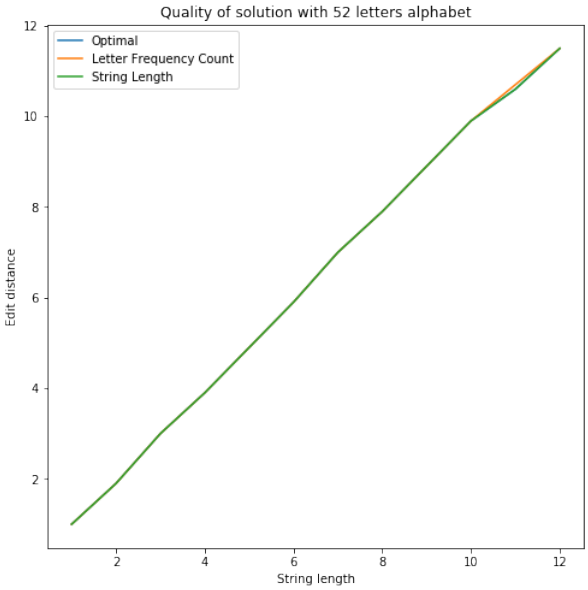


Figure 8: Comparison of obtained solution with the Optimal.

5.5.4 Comparing Heuristics

Both approaches reduce the amount of recursive calls compared to the theoretical bound, but using just string length gives a behavior very similar to the theoretical maximum amount of calls. In contrast, Counting letter frequency reduces considerably the amount of recursive calls but still remains exponential with respect of the strings length.

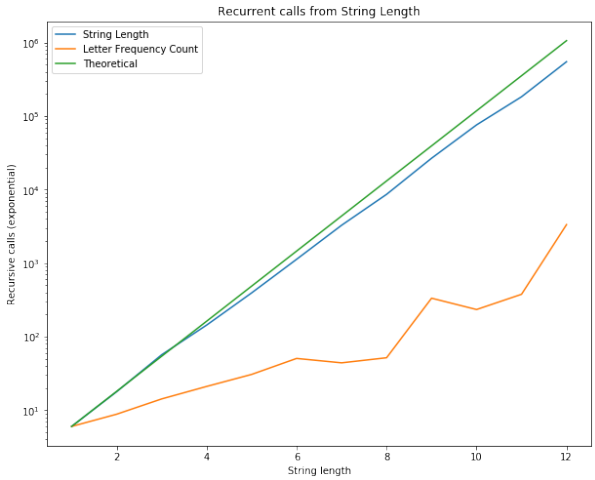


Figure 9: Count of recursive calls in exponential scale.

6 Experimental Comparisons

Overall Observations of experimental results between algorithms.

6.1 Experimental Comparisons for Strings up to 10 Letters

Experimental comparisons for strings with length less than 10 showed that the Greedy approach seems to be the fastest (showing the run-time looking to be constant) ; even though the difference is not as obvious as it will get later. Pure Recursion showed the worst results with its graph increasing much faster than the others. Of the two Branch and Bound approaches, the Frequency Count implementation was faster than the String Length implementation due to branch cutting which resulted in saved time. Divide and Conquer and the Dynamic Programming approaches showed nearly the same run time which were the second most efficient.

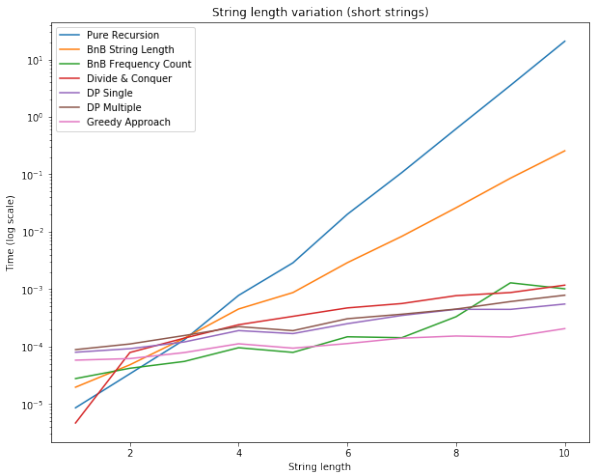


Figure 10: Comparisons for strings up to 10 characters

6.2 Experimental Comparisons for Strings Greater than 10

For strings with length greater than 10 characters, only the Divide and Conquer, Greedy and Dynamic programming approaches were used. This is because the run-time of the Pure Recursive and Branch and Bound approaches increased at an exponential rate and were not successful with strings greater than length 10.

The Greedy algorithm presented excellent results of polynomial time (denoted by $c_0 \times n^2 + b_0$), while the run-time of the other algorithms grew at a faster rate. If we denote the run-time of the other algorithms as $c_i * n^2 + b_i$ (each having a different c_i). We see here that for $i > 0$ the c_i are of the same order of magnitude whereas they are bigger then c_0 *(more formally* $\forall i > 0,\ c_0 << c_i$*)*.

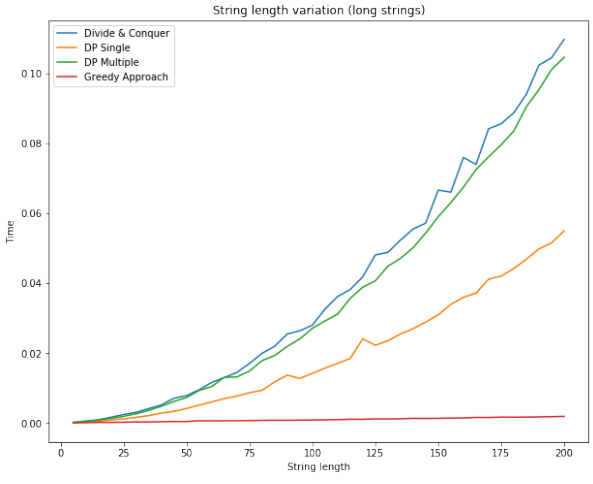


Figure 11: Comparison strings which length is greater than 10 characters

6.3 Experimental Comparisons for Alphabet Variation

Experimental comparisons on different alphabet variation showed that Greedy, Dynamic Programming and Divide and Conquer algorithms have constant run time, with the fastest results belonging to the Greedy approach. The Branch and Bound frequency count approach would be the only implementation that benefits from a larger alphabet; as common letters are less common with a larger alphabet, branches can be cut faster when focusing on counting common letters between strings. That explains why at one point the run time of the algorithm starts to decrease (*see figure 12*). The worst result was presented by Pure Recursion. These experiments showed that the run time of most implemented algorithms does not depend on variation of the alphabet used to create strings.

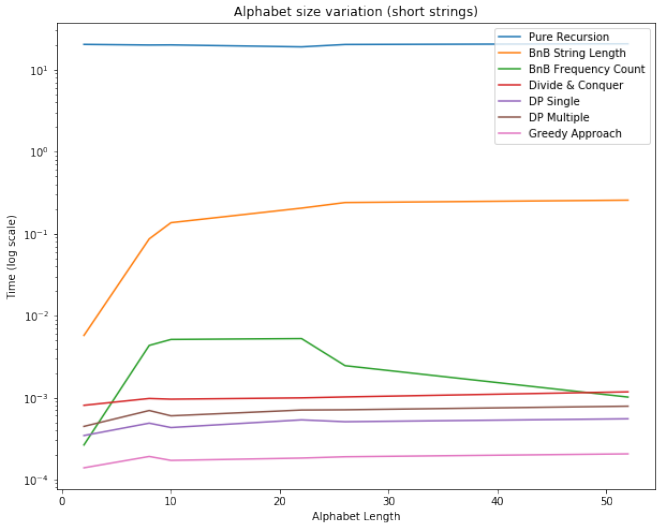


Figure 12: Comparison of behaviour varying the length of the alphabet.

6.4 Experimental Comparisons for String Type variation

Figure 13 presents experimental results of algorithms on different types of string generation: same strings, strings with at most 20% of difference, and randomly generated strings. The run time does not depend on the type of strings for the Divide and Conquer, Pure Recursion, Greedy and Dynamic Programming with multiple matrices approaches. The fastest solution of computing Edit Distance of random strings belong to Greedy approach. Branch and Bound Frequency count algorithm works faster on random strings but slower on the same strings type to compare with Branch and Bound string length approach.

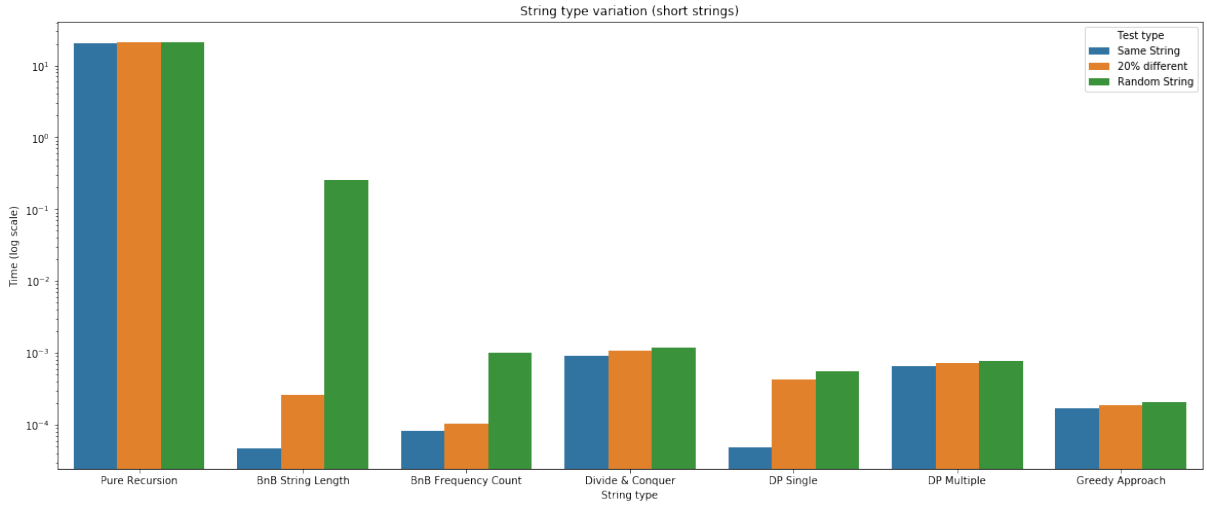


Figure 13: Comparison of behaviour varying the pair of string type.

6.5 Comparing Dynamic Programming Approach with a Greedy Algorithm

- Time Complexity: The greedy approach has the same theoretical time complexity of $O(n^2)$ as the dynamic programming approach, but it has a much faster experimental run-time.
- Space Complexity: The greedy approach is more space efficient, it is $O(n+m)$ whereas the dynamic approach is $O(n*m)$ (with n and m being the lengths of the strings)
- Solution Quality: Dynamic programming approaches provide an optimal solution by construction. Although the Greedy approach is faster, it does not provide an optimal solution.

6.6 Comparing Dynamic Programming Approach with Divide and Conquer

- Time Complexity: The Dynamic Programming and Divide and Conquer approaches have similar time complexities, except that the Divide and Conquer approach is larger by a constant of about 3x (see Figure 14).
- Space Complexity: However, Divide and Conquer has a better space complexity than the Dynamic Programming approach. Divide and conquer has linear space complexity while dynamic programming is polynomial.
- Solution Quality: Both approaches provide an optimal solution. So the trade off lies between time and space. Since the time complexity differs only by a constant, if time is the priority one would choose the Dynamic approach, but if space is a priority it would be advised to use the Divide and Conquer approach.

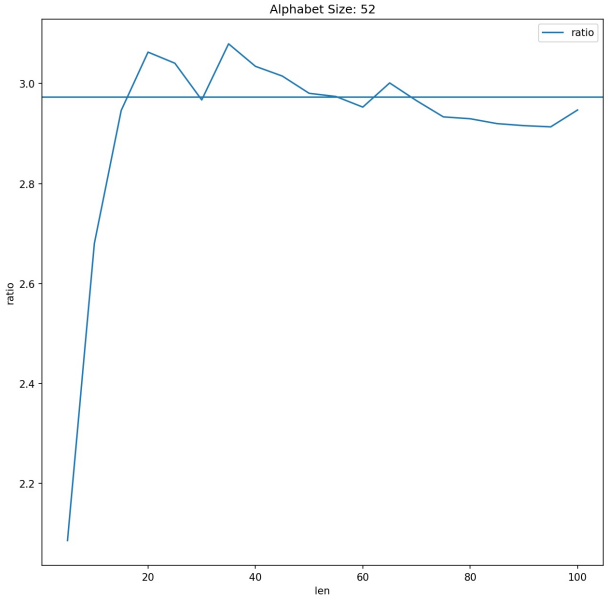


Figure 14: The time complexity comparison between the Dynamic Programming and Divide and Conquer approaches showing a constant multiplier of about 3x.

7 Clustering Results

7.1 Clustering of Protein Names on Edit Distance

Conditions for the Experiments

- For this approach the proteins where modeled as an undirected weighted graph, where the weight of every vertex corresponded to the edit distance between the two proteins. While running the edit distance with the entire base would have been ideal, this would have required to process upwards of 42 million edit distances on large strings which was not possible.
- A sample of the protein database of two percent of the database was used to test the approach. This resulted in 125 proteins analyzed, which required 15500 edit distances to be calculated. This took a little under 13 minutes to calculate.
- Initially the created graph would have every node connected, and in order to create the distinct node groups vertices larger than a defined value would be cut. Different runs were made varying the value of this cut in order to find the optimal. Then, every subgraph of the original would be a different protein group.

Experimental Results

Edit distance Cut	Generated Graphs	Nodes used
5	2	4
25	10	23
50	13	36
70	19	54
75	19	59
80	19	63
85	19	67
100	9	85
150	3	113
200	2	121

Table 1: Groups and proteins taken into account varying the cut value.

On the table there are some of the values tested to cut the graph and their resulted groups, along with the amount of used nodes. The biggest amount of groups was generated with 70 to 85, with 85 taking into account the largest amount of nodes. On the appendix there's the visual representation of some of this graphs.A visualization of the generated graphs can be found in figure 16.

Unfortunately most of the groups generated by the this approach consisted on pairs of proteins, with only three big groups including multiple proteins. While the proteins seem to be related among each other in their groups, having just pairs is not really useful. The approach may benefit from using a smarter way of cutting the vertices, instead of just using the same value for every branch.

7.2 Cluster Method with Alphabet Frequency

Conditions for the Experiments

Upload the final clustering algorithm code when finished.

- This approach to clustering based on the Alphabet Frequency was a more creative method to see if protein sequences could be clustered on other attributes besides edit distance. Next was to see if this attribute contributed to edit distance. The idea behind this experiment was to count the frequency of each letter in the alphabet for each string and attempt to cluster on the occurrences of different letters present in the sequences. The goal was to see if these frequencies had any impact on protein grouping names or groupings based off of edit distances.
- The thought process arose from analyzing methods to compare and cluster DNA sequences, because those rely heavily on frequency of specific letters within their alphabet.
- The approach used a K-means clustering algorithm across a random sample of 2.5% of the protein database.
- Once the proteins were clustered based off of their letter frequencies, the edit distance was plotted within each of the groups of proteins to see if the frequency of letters had an impact on the edit distances between the sequences.

Experimental Results

- A k of 3 or 4 was found to be the number of clusters, so the experiment used both k values to interpret results.
- In k=3 clusters, there was no noticeable difference in the edit distances between most of the groups. However, in the last group it seemed as if the majority of sequence pairs had larger edit distances, suggesting that the contents of the sequences were further apart in this specific group. It was interesting to get these results so the analysis continued with the next possible optimal grouping.
- Using k=4 clusters, there were a bit more interesting results, but unfortunately nothing very conclusive. There still shows one group to have a much larger distance than the rest, potentially indicating that the sequences in this group can be split into two unrelated groups. Unfortunately, this analysis does not provide substantial conclusive results, but ti was still fun to approach the problem in a different way.

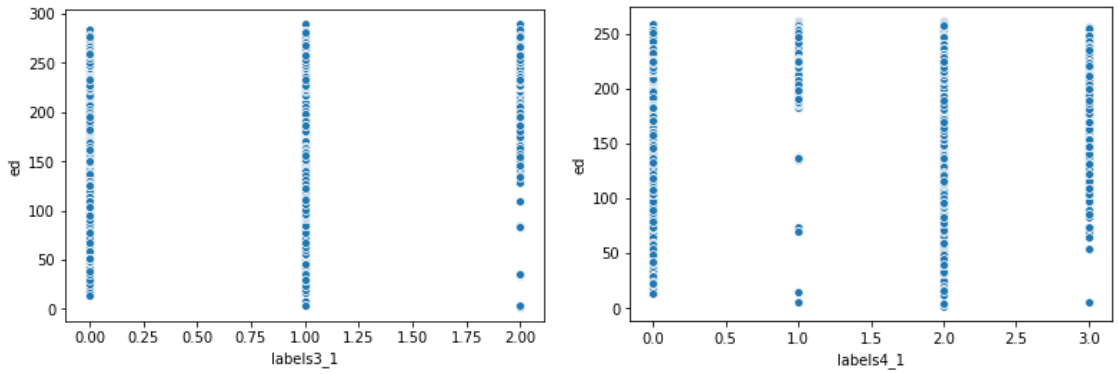


Figure 15: Comparing edit distances between clusters found using k-means. The figure on the left represents the results found with 3 clusters, and the figure on the right represents 4 clusters.

- Conclusion : Although this process was a creative and interesting approach to determine if frequency of letters in a protein sequence contributed to edit distance between sequences, it was perhaps too high-level to provide conclusive results. Or perhaps, this means that protein sequences do not have the same dependency on letters and alphabets as DNA sequences.

8 Conclusion

- Experimental results on different alphabet sizes strings showed that run time of all implemented algorithms barely depends on the variance of the alphabet. The only exception being with the Branch and Bound algorithm.
- Experimental results on 3 different string types showed that increasing the percentage of difference between strings causes the increase of run time only for both types of Branch and Bound approaches.
- Time Complexity: The algorithm which had the best run-time was the Greedy approach, although it did not output optimal alignment. The fastest method with optimal alignment is the Dynamic Programming approach (with the single matrix option running faster than the one with multiple matrices by a multiple of about 3x). For the non-optimal recursive options, branch and bound worked better than pure recursion for strings under 12, but both options are not great solutions compared to other methods.
- Space Complexity: The Divide and Conquer algorithm succeeded in having the best space complexity (linear), but wasn't the most efficient on time. The best approach when combining time and space would be again the Dynamic Programming approach with polynomial time and space complexity.
- Solution Quality: If time is a more important constraint, but producing the optimal result is not, it could be suggested to use the Greedy approach. Where space complexity and an optimal solution are the most important constraints, then it would be best to use the Divide and Conquer approach. The best option overall would be the Dynamic Programming approach as it produces optimal results, in polynomial time, with polynomial space complexity.

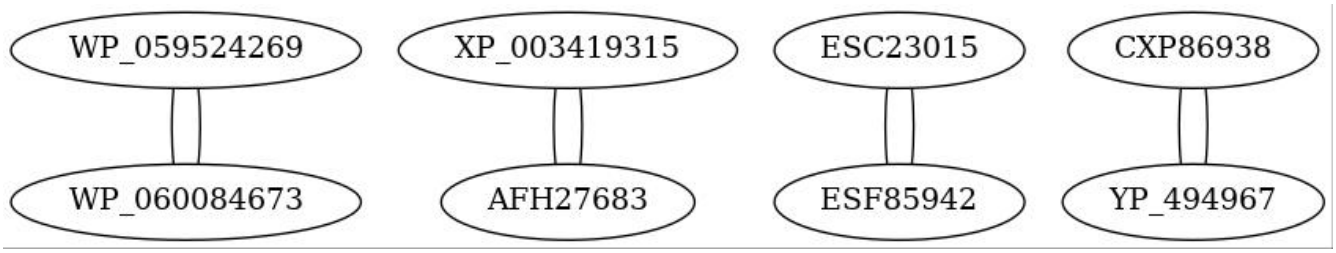
  Since increasing the space in a real setting is often less costly (only needing to add more RAM) than increasing the time efficiency (by needing better CPU or a longer wait time) the Dynamic Programming approach would probably be the best approach overall.
- Clustering: There were two attempts to cluster the sequences in the protein database. The first attempt grouped all sequences on edit distances between pairs to find associated groupings of sequence names. It was interesting to see how the group names appeared in different clusters, but there were no conclusive results. The other attempt clustered sequences by the frequency of occurrence of each letter in the alphabet, and then compared the edit distances within each cluster. The results found here were that some sequences in one grouping seem to have very large edit distances, where the expectation was to find small edit distances between similar groupings. Results did not align with expectations for the experiment, so this approach was also inconclusive.

9 Appendix

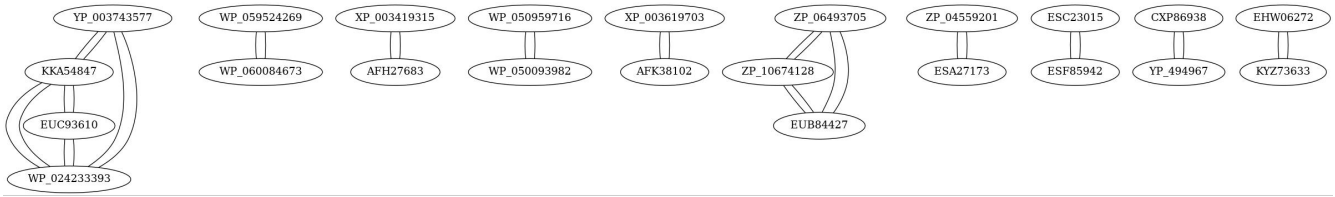
9.1 External links

- Our python code is available on this GitHub Repository.
- Our project planning and timeline is available in Google Sheets.

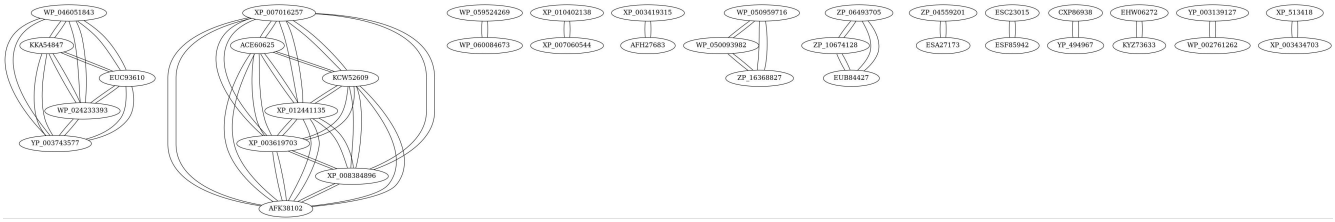
10 Graphs



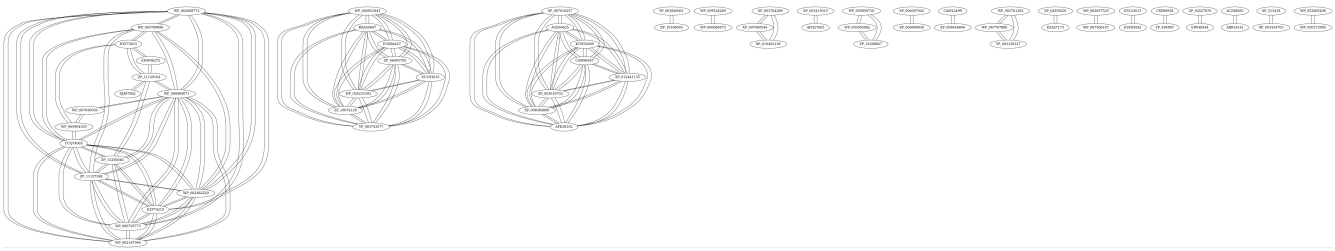
threshold = 10



threshold = 25



threshold = 50



threshold = 85

Figure 16: Clustering attempt based on the edit distance between protein strings
To have a better view of these images and see more results – with different threshold – please visit our GitHub Repository/protein_clustering_graphs

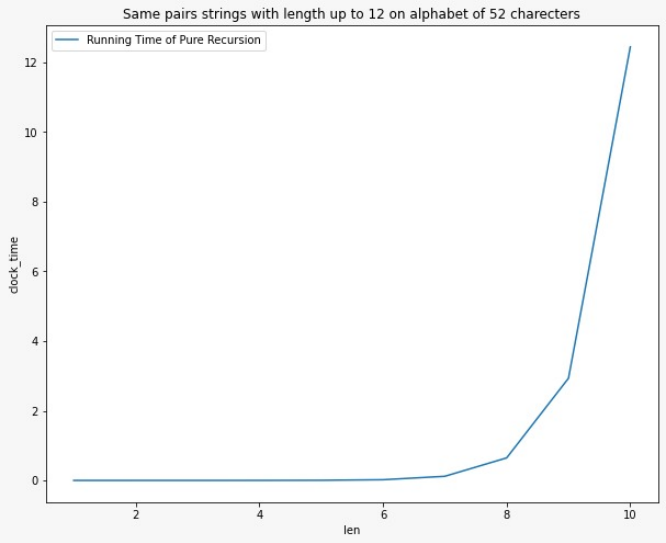


Figure 17: Pure Recursion - same pair, string length = 20 , alphabet size = 52

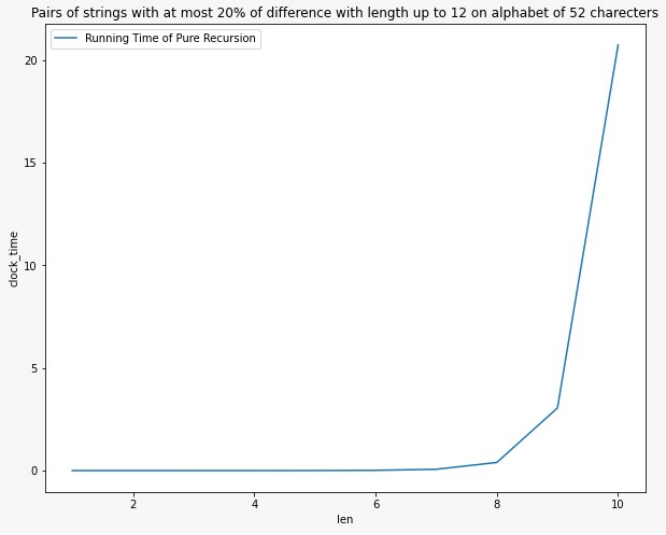


Figure 18: Pure Recursion - 20 % difference pair, string length = 12 , alphabet size = 52

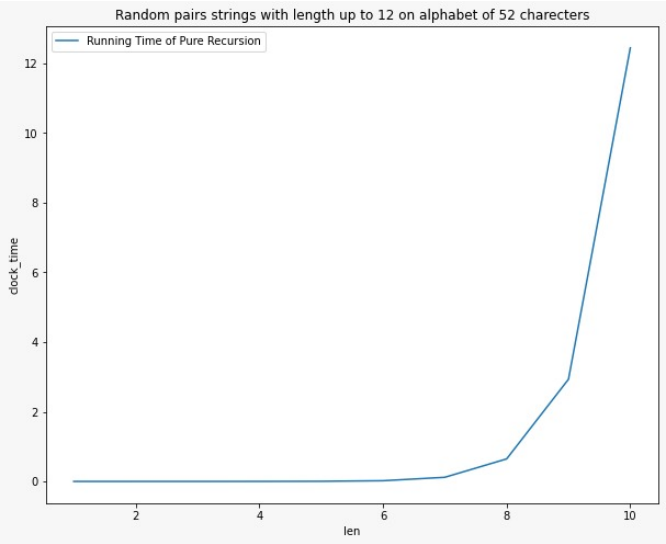


Figure 19: Pure Recursion - random pair, string length = 12 , alphabet size = 52

10.1 Pseudo-Code

10.1.1 Dynamic programming with a single matrix

Algorithm 1: Dynamic_programming_single_matrix

Input: S1 , S2

```
/* Parameters :                                                              */
/*    s1,s2 :  strings.                                                      */
/* Return :                                                                 */
/*    ED, alignment                                                         */
```

begin

  n ←len(s1) ; m ←len(s2)

  /* Use one matrix to compute edit distance and backtrack through at the end to get alignment */

  define dist_mat : array with first row 0 to m and first column 0 to n

  define alignment : empty array

  /* Edge case for same strings */

  if *(s1 == s2)* **then**

    return {*"ed" : 0 , "alignment" : "match" for all letter* };

  else

    for *j ←1 to m+1* **do**

      for *i ←1 to n+1* **do**

        if *s1[i-1] == s2[j-1]* **then**

          dist_mat[j,i] = dist_mat[j-1 , i-1]// "keep" this letter

        else

          /* Custom set of rules */

          $$\text{dist\_mat[j,i]} = 1+\min\begin{cases}\text{dist\_mat[j,i-1]} \text{// "remove" letter}\\ \text{dist\_mat[j-1,i]} \text{// "add" letter}\\ \text{dist\_mat[j-1,i-1]} \text{// "substitute" letter}\end{cases}$$

  alignment ←retrieve alignment by backtracking dist_mat;

  return {*"ed" : dist_mat[-1,-1] , "alignment" : alignment* };

10.1.2 Dynamic programming with multiple matrices

Algorithm 2: Dynamic_Programming_Multiple_Matrices_Approach_Edit_Distance

Input: X, Y

```
/* Parameters :                                                    */
/*    X , Y : strings                                             */
/* Return :                                                       */
/*    ed :  integer, optimal edit distance between X and Y.       */
/*    alignment :  array of instructions, to go from Y to X.      */
/*                                                                */
```

1 R ←ed_dynamic_mat(X,Y)// top-left to bottom right
2 L ←backward_ed_dynamic_mat(X,Y)// bottom right to top-left
3 S ←R+L;
4 ed ←S[0,0];
5 alignment ←extract optimal "path" from S.;

```
/* The optimal path is one of the possible path which goes from top-left to bottom-right cell of
   S, using only the cells with minimum number (the edit distance).                              */
```

6 **return** {ed : ed , alignment : alignment}

10.1.3 Greedy Algorithm

Algorithm 3: Greedy_Approach_Edit_Distance

Input: X, Y

```
/* Parameters :                                                          */
/*    X , Y : strings, we assume here that len(X) <= len(Y).             */
/* Return :                                                              */
/*    ed :  integer, optimal edit distance between X and Y.              */
/*    alignment :  array of instructions, to go from Y to X.             */
/*                                                                       */
```

n ←len(X);
m ←len(Y);
ed ←∞;
for *all possible combinaison* $0 \leq j \leq i \leq min(n\text{-}m, \lceil m/2 \rceil)$ *s.t.* $i = j + (n\text{-}m)$ **do**
    str1 ←i*"-" + X// e.g. X= H E L L O , str1 = - H E L L O
    str2 ←Y + j*"-"// e.g. Y= H O L A,   str2 = H O L A - -

    OR   // When all the possibilities are explored this way do it the other way.

    str1 ←X + i*"-"// e.g. X= H E L L O , str1 = H E L L O - -
    str2 ←j*"-" + Y// e.g. Y= H O L A,   str2 = - - - H O L A

    ed_tmp,alignment_t ←greedy_ed(str1,str2);
    if *ed ¿ ed_tmp* **then**
        ed ←ed_tmp;
        alignment ←alignment_tmp;
for $0 \leq i \leq n\text{-}m$ **do**
    str1 ←X // e.g. X= B O N J O U R , str1 = B O N J O U R
    str2 ←(n-m-i)*"-" + Y + (i)*"-" // e.g. Y= H O L A,   str2 = - H O L A - -
    ed_tmp,alignment_t ←greedy_ed(str1,str2);
    if *ed ¿ ed_tmp* **then**
        ed ←ed_tmp;
        alignment ←alignment_tmp;
return {ed : ed , alignment : alignment}

Algorithm 4: greedy_ed

```
Input: str1 , str2
/* Check from left to right, 1 by 1 the letters of str1 and str2 to compare them.   */
/* Parameters :                                                                     */
/*    str1,str2 :  srings of same length.                                           */
/* Return :                                                                         */
/*    not optimal edit_distance and alignment, computed without modifying str1 or str2. */
ed ←0;
alignment ←[];
for i in range(len(str1)) do
    if str1[i] == str2[i] then
        alignment.append(["skip" , str1[i]]);
    else if str1[i] == "-" then
        alignment.append(["del" , str2[i]]);
    else if str2[i] == "-" then
        alignment.append(["add" , str1[i]]);
    else
        // str1[i] != str2[i]
        alignment.append(["sub" , str1[i]]);
    return {ed : ed , alignment : alignment}
```

10.1.4 Divide and Conquer Algorithm

Algorithm 5: Space_Efficient_Dynamic_Programming_Edit_Distance

Input: X , Y

```
/* Parameter :                                                                    */
/*    x,y :  strings.                                                             */
/* Return :                                                                       */
/*    l[1,:]  :  the last line of the array created by the dynamic programming approach.  */
```

begin

```
    /* Blank letter added to X and Y corresponds to the uppest row and leftest column of the array
       ℓ.                                                                        */
    X ←concat(∅,X);
    Y ←concat(∅,X);
    n ←len(X);
    m ←len(Y);
    Create Array ℓ of size [0..1 , 0..n];
    Initialize ℓ :
    /* Initializing ℓ with uppest row equal to 0,1,...,n-1 and leftest column equal to 0,1,...,m-1.
       As ℓ is composed of two rows (for space efficiency), the edge case corresponding to the
       leftest column must be handelded in the main loop.                        */
        ℓ[0 , i] = i for i∈[0..n];
        ℓ[1 , 0] = 1;
    /* Algorithm's main loop.                                                     */
    for i ←1 to m do
        for j ←1 to n do
            /* Edge case :  the i-nth leftest element must be equal to i          */
            if X[j] == Y[i] then
                ℓ[1 , j] ←ℓ[0 , j-1];
            else
                                          | ℓ[0 , j]
                ℓ[1 ,j] ←1 + min          | ℓ[0 , j-1]
                                          | ℓ[1 , j-1]
        /* Make uppest row equal to the lowest before starting again.             */
        ℓ[0 , :] ←ℓ[1 , :]
    return ℓ[1 , :]
```

Algorithm 6: Backward_Space_Efficient_Dynamic_Programming_Edit_Distance

Input: X , Y

```
/* Parameter :                                                              */
/*    x,y :  strings.                                                       */
/* Return :                                                                 */
/*    r[0,:]  :  the first line of the array created by the backward dynamic programming approach.
                                                                            */
```

begin

```
   /* Blank letter added to X and Y corresponds to the lowest row and rightest column of the array
      r.                                                                    */
```

X ←concat(∅,X);

Y ←concat(∅,X);

n ←len(X);

m ←len(Y);

Create Array r of size [0..1 , 0..n];

Initialize r :

```
   /* Initializing r with lowest row equal to 0,1,...,n-1 and rightest column equal to
      0,1,...,m-1.  As r is composed of two rows (for space efficiency), the edge case
      corresponding to the rightest column must be handelded in the main loop.  */
```

r[1 , i] = n-i for i∈[1..n+1];

r[0 , n-1] = 1;

```
/* Algorithm's main loop.                                                   */
```

for *i ←m-2 to -1 step -1* **do**

  for *j ←n-1 to -1 step -1* **do**

```
    /* Edge case :  the (m-i)-nth rightest element must be equal to (m-i-1)  */
```

    if *X[j] == Y[i]* **then**

      r[0 , j] ←r[1 , j+1];

    else

$$r[1 ,j] \leftarrow 1 + \min \begin{vmatrix} r[1 , j] \\ r[1 , j+1] \\ r[0 , j+1] \end{vmatrix}$$

```
  /* Make lowest row equal to the uppest before starting again.             */
```

  r[1 , :] ←r[0 , :]

return r[0 , :]

Algorithm 7: Divide_and_Conquer_Edit_Distance

Input: X , Y

```
/* Parameter :                                                                    */
/*    x,y :  strings, the two words we work with.                                 */
/* Return :                                                                       */
/*    {'p' :  p, 'ed' :  ed} dictionnary :                                        */
/*    • p :  the collection of points of interest to compute the alignement between x and y.  */
/*    • ed :  the edit distance between x and y.                                  */
begin
    n ←len(X);
    m ←len(Y);
    if n ¡ 1 or m ¡ 2 then
        /* Edge cases, if (len(X) == 0,1) or (len(Y) == 0,1)                      */
        if m == 1 and n ¿ 1 then
            l ←Space_Efficient_Dynamic_Programming_Edit_Distance(X,Y);
            s ←l + [n,n-1,...,0];
            mini ←leftest index minimizing s;
            p = [(mini , m)];
        else if n == 1 and m ¿ 0 then
            /* if X is in Y then there are m-1 operations to do to go from Y to X, else there are m
               changes to do.                                                     */
            ed = m-1 if x in y else m;
            p = [(X , Y[0])];
        else
            Handle case X empty or Y empty;
        return { 'p' : p , 'ed' : ed };
    else
        ℓ ←Space_Efficient_Dynamic_Programming_Edit_Distance(X,Y[:⌊m/2⌋]);
        r ←Backward_Space_Efficient_Dynamic_Programming_Edit_Distance(X,Y[⌊m/2⌋:]);
        s ←ℓ+ r;
        mini ←leftest index minimizing s;
        ed = s[mini];
        p = [(mini , ⌊m/2⌋)];

            Concatenate(p , Divide_and_Conquer_Edit_Distance(X[:mini] , Y[:⌊m/2⌋])['p']) ;
         // top-left
            Concatenate(p , Divide_and_Conquer_Edit_Distance(X[mini:] , Y[⌊m/2⌋:])['p']) ;
         // bottom-right
        /* concatenate :  w.r.t.  original X and Y coordinates and such that the final p is still
           sorted*.                                                               */
        return { 'p' : p , 'ed' : ed };
        /* Post-treat p to get alignment from the points of interest.             */
        /* * :  here                                                              */
```

10.1.5 Pure Recursion

Algorithm 8: Pure_Recursion_Edit_Distance

```
   Input: S1 , S2
   /* Parameters :                                                    */
   /*    s1,s2 :  strings.                                           */
   /*    operations :  list.                                         */
   /* Return :                                                       */
   /*    ED, operations                                              */
1  begin
       /* if s1 is empty,we insert all characters of s2 in s1        */
2      if len[s1] == 0 then
3          return ED ←ED+len(s1), operations ←operations + [['insert'+x] for x in s2 ]
       /* if s2 is empty,we insert all characters of s1 s2           */
4      if len[s2] == 0 then
5          return ED ←ED+len(s2), operations ←operations + [['insert'+x] for x in s1 ]
       /* If last characters of both strings are the same,we set k=0 because we ignore them and we
          compute Edit Distance for these strings without last characters */
6      if (s1[-1]== s2[-1]) then
7          k ←0;
8          w1 = RecursiveED (s1[:-1],s2[:-1],operations+['skip':s1[-1]],ED+k)
       /* if last characters are different, we set k=1 and we consider all three operations on last
          character of s1, compute cost for all three operations     */
9      else
10         k ←1
11         w1=RecursiveED(s1[:-1],s2[:-1],operations+['replace':s1[-1]],ED+k)
12     w2=RecursiveED(s1[:-1],s2,operations+['delete':s1[-1]],ED+1)
13     w3=RecursiveED(s1,s2[:-1],operations+['insert':s2[-1]],ED+1)
       /* compare all costs and choose the minimal one of them       */
14     return min(w1,w2,w3)
```

10.1.6 Branch and Bound

Algorithm 9: Edit_Distance-Branch_and_Bound

Input: X,Y, cost, bound
Output: edit_distance

begin
  /* Initialize variables */
  n ←len(X) m ←len(Y)
  /* Manage edge cases */
  if $n == 0$ *and* $m == 0$ **then**
    return 0
  if $n == 0$ **then**
    return m
  if $m == 0$ **then**
    return n
  /* Calculate BnB weights */
  $\text{weight}_{\text{substitution}} = calculate\_weigth\_substitution() + cost$
  $\text{weight}_{\text{deletion}} = calculate\_weigth\_deletion() + cost$
  $\text{weight}_{\text{insertion}} = calculate\_weigth\_insertion() + cost$
  /* Explore branches */
  if $weight_{substitution}$ ¡= $bound$ **then**
    $cost_{substitution}$ = Calculate ℓ[X-1 , Y-1] $bound = min(bound, cost_{substitution})$
  if $weight_{deletion}$ ¡= $bound$ **then**
    $cost_{deletion}$ = Calculate ℓ[X , Y-1] $bound = min(bound, cost_{deletion})$
  if $weight_{insertion}$ ¡= $bound$ **then**
    $cost_{insertion}$ = Calculate ℓ[X-1 , Y] $bound = min(bound, cost_{insertion})$
  /* Return minimum of the explored branches */
  $\text{edit\_distance} = \min \begin{vmatrix} cost_{substitution} \\ cost_{deletion} \\ cost_{insertion} \end{vmatrix}$

Algorithm 10: Length_Heuristic-Branch_and_Bound

Input: X,Y
Output: weight

```
begin
    n ←len(X) m ←len(Y)
    weight_substitution = abs(m − n)
    weight_deletion = abs(m − n − 1)
    weight_insertion = abs(n − m − 1)
```

Algorithm 11: Letter_Frequency_Heuristic-Branch_and_Bound

Input: X,Y
Output: weight

```
begin
    n ←len(X) m ←len(Y)
    /* Create dictionary with common letter count */
    letter_frequency = common_letters(X,Y)
    /* Variables used to calculate weights */
    common_count = sum(letter_frequency)
    if X[0] in letter_frequency then
        X_Look_Ahead = 1
    else
        X_Look_Ahead = 0
    if Y[0] in letter_frequency then
        Y_Look_Ahead = 1
    else
        Y_Look_Ahead = 0
    if X[0] == Y[0] and (Y[0] in letter_frequency or X[0] in letter_frequency) then
        Common_Ahead = 1
    else
        Common_Ahead = 0
    weight_substitution = max(m, n) − common_count + Common_Ahead
    weight_deletion = max(m, n) − common_count + Y_Look_Ahead
    weight_insertion = max(m, n) − common_count + X_Look_Ahead
```