Université Jean Monnet

Advanced Algorithms

Project 2017-2018

Implementation of Algorithms For Computing Edit Distance

Author:
Karthik Bhaskar
Allwyn Joseph
Evhenii Zotkin
Tu My Doan

Supervisors:
Amaury Habrard
Leo Gautheron

December 12, 2017

Contents

1 Introduction

The objective of the project is to implement the algorithms for computing the Edit Distance(ED) and to provide an experimental study of their running time and the quality of the solution given.

The following are the algorithms implemented for computing the Edit Distance,

- The classic algorithm based on dynamic programming
- The version combining dynamic programming, divide and conquer approach
- The pure recursive version
- A branch-and-bound version of the recursive approach
- An approximated version of the classical dynamic programming approach where one fills only a stripe of size k around the diagonal of the matrix.
- An approximated version based on a greedy approach

All the algorithms are implemented using Python programming language.

The algorithms are first evaluated with an artificial data and later it has been evaluated with the protein database, which outputs the distance and the possible alignment corresponding to the Edit Distance problem.

In the following sections of the report, a rigorous experimental study about each implemented algorithms is explained.

2 Overview of Edit Distance Problem and Similar Applications

2.1 Edit Distance Problem

The Edit Distance is a way of quantifying how dissimilar two strings are to one another by counting the minimum number of operations required to transform one string into the other.

The operations used for transformation are:

- Insertion
- Deletion
- Substitution

Given two strings a and b on the alphabet Σ, the edit distance d(a, b) is the minimum-weight series of edit operations that transforms a into b.

Insertion of a single symbol. If a = uv, then inserting the symbol x produces uxv. This can also be denoted as $\epsilon \rightarrow$ x using ϵ to denote the empty string.

Deletion of a single symbol changes uxv to uv (x $\rightarrow \epsilon$).

Substitution of a single symbol x for a symbol y $\neq$ x changes uxv to uyv (x $\rightarrow$ y).

Example:

The edit distance between the transformation of string "**INTENTION**" to the string "**EXECUTION**" is 5.

1. INTENTION $\rightarrow$ NTENTION (**Deletion** of I)
2. NTENTION $\rightarrow$ ETENTION (**Substitution** of N $\rightarrow$ E)
3. ETENTION $\rightarrow$ EXENTION (**Substitution** of T $\rightarrow$ X)
4. EXENTION $\rightarrow$ EXECNTION (**Insertion** of C)

5. EXECNTION → EXECUTION (**Substitution** of N → U)

A set of operations can be used to generate the possible alignment of the strings.

2.2 How To Find The Minimum Edit Distance

The above algorithms are used to search for a path (sequence of edits) from the initial string to the goal string in order to minimize the cost with the following elements,

- **Initial state**: The string before transformation
- **Operations**: The set of operations (Insertion, Deletion and Substitution) used to transform one string to other
- **Goal state**: The required transformed string
- **Path cost**: It is the minimum number of edits where the cost of each operation is 1.

2.3 Applications of Edit Distance

Edit Distance finds its application in different domains such as,

- **Bioinformatics**: Edit Distance can be used to quantify the similarity of DNA sequences, which can be viewed as strings of the letters A, C, G and T.
- **Natural Language Processing**:
  - **Spelling Correction**: Where automatic spelling correction can determine candidate corrections for a misspelled word by selecting words from a dictionary that have a low distance to the word in question.
  - **Named Entity Extraction and Entity Co-reference**
  - **Evaluation of Machine Translation and Speech Recognition**

3 Overview of The Algorithms Used, Their Substructure and Complexity

3.1 Dynamic Programming

3.1.1 Overview

First we start by defining rules for the basic cases. If $|a| = 0$ or $|b| = 0$ then edit distance is $|b|$ and $|a|$ respectively. In case of $|a| = |b| = 0$, edit distance is 0. Otherwise, let $D[i, j]$ be the edit distance for two strings at indices i and j. We define the cost function as follows:

$$D[i,j] = \min \begin{cases} D[i-1, j-1] + \delta(i,j) \\ D[i-1, j] + 1 \\ D[i, j-1] + 1. \end{cases} \tag{1}$$

where gap penalty δ is defined as follows:

$$\delta = \begin{cases} 0 & if\ a[i-1] \ = \ b[j-1] \\ 1 & otherwise \end{cases} \tag{2}$$

We essentially are going to initialize matrix of shape $(|a| + 1,\ |b| + 1)$ where $+1$ is for edit distance between empty string and respective prefix. Since the edit distance between the empty string and some prefix $a[0,\ j]$ or $b[i, 0]$ is the length of the prefix, we are going to fill the first row and column of the matrix with values from 0 to length of the respective string. Then we are going to iteratively populate the matrix starting at cell $D[1, 1]$ and using the cost function defined previously. Therefore, once matrix is computed, we can retrieve the edit distance from the bottom right cell. Then, we can print the optimal alignment by retracing our steps. Since at every step of our algorithm for matrix computation we compare the value of 3 cells around the current, we can use the same technique to move backward recursively. Thus, we can retrace the optimal alignment

3.1.2 Computational Complexity

Since we use 2 for loops which pass over entire length of the string a and b, the time complexity is $O(nm)$, where $n = |a|$ and $m = |b|$, thus

approximately quadratic in the length of the input. The space complexity of this algorithm is $O(nm)$ since we store the prefix matrix of size nm.

The time complexity of the alignment function is $O(n+m)$ since at each step we move either to the cell either above current, to the left of current, or diagonally, which gives a maximum of m+n visited cells. The space complexity of the alignment function is linear since only 3 strings of size $\max(n, m)$ are stored, thus $O(\max(m, n)$

3.2 Version combining dynamic programming, divide and conquer approach

3.2.1 Overview

Using the properties of Dynamic Programming solution described in the previous section we can notice that at each step of the algorithm only 2 rows of the matrix are used. Indeed, to compute the edit distance between two substrings we need only to look for the values $D[i-1, j-1]$, $D[i, j-1]$, $D[i-1, j]$ of the prefix matrix. Therefore, we can compute edit distance using linear space. Since at each step of the algorithm only matrix of size $(2, n)$ is stored in memory (technically the original strings and additional variables are stored in memory as well, but we consider them negligible and will treat them as constant during further analysis).

Moreover, edit distance of 2 strings does not change if we reverse them. Using this property, we can compute the prefix matrix in reverse order, starting at $D[i, j]$ (where $i = |a|$ and $j = |b|$) and moving to $D[0, 0]$. This matrix can be computed in linear space as well. while gaining in terms of space complexity we cannot retrace the optimal alignment of two strings.

Using this two methods we can describe the Divide and Conquer algorithm to the Edit Distance problem. This algorithm was first proposed by Dan Hirschberg for the Longest Common Subsequence(LCS) problem. With a slight modification of the cost function, this algorithm can be applied to the Edit Distance problem. The intuition behind this algorithm is following: first, we divide our matrix in half horizontally. Then, we compute the edit distance for 2 substrings using forward space efficient and backward space-efficient algorithms. We then use 2 matching rows and find the index minimizing the edit distance. We then split the matrix in two vertically, and repeat the procedure recursively until we reach the trivial case, therefore both computing the edit distance and retrieving the optimal alignment.

3.2.2 Computational Complexity

At first pass of the algorithm we are going to process the entire matrix, thus the first pass requires $O(n, m)$ time. Then, each next step we are going to process matrices of dimensions $(n, m/2)$ and $(n/2, m/2)$, therefore, asymptotically algorithm will use (2nm) time to finish and therefore bounded by $O(nm)$. Since at each pass of the algorithm we use exactly 4 matrices of size (2,m), the space complexity is bounded by $O(n, m)$

3.3 Pure recursive version

3.3.1 Overview

First, we define an optimal substructure of the problem. Let E be the edit distance for two strings A and B of length $|A| = n$ and $|B| = m$ respectively. Then $E(A, B)$ contains edit distance for $E'(A[n], B[m])$. We prove this by contradiction:

Suppose that $E(A, B)$ an optimal edit distance does not contain $E'(A[n-1], B[m-1])$. Then if we append to characters $A[n]$ *and* $B[m]$ and plug it to E' we must obtain $E'(A, B)\ != \ E(A, B)$. Since $E(A, B)$ is an optimal edit distance, a contradiction.

From that, we can build a reference cost function for the Edit Distance problem:

$$E[n, m] = \min \begin{cases} E[n-1, m-1] + \delta(i, j) \\ E[n-1, m] + 1 \\ E[n, m-1] + 1. \end{cases} \tag{3}$$

where gap penalty δ is defined as follows:

$$\delta = \begin{cases} 0 & if\ a[i-1]\ =\ b[j-1] \\ 1 & otherwise \end{cases} \tag{4}$$

We also add a condition for the empty strings:

$$E[n, m] = |m|\ if\ |n| = 0; \quad E[n, m] = |n|\ if\ |m| = 0 \tag{5}$$

In the implementation, we almost literally use the formulas provided above, but instead of returning only the edit distance at the bottom case

(when either n or $m = 0$) we also return an operation which must be performed at this step (insertion or deletion). Also, we this way at each step of the recursion we keep track which option was chosen in $min()$ function in order to retrieve the string alignment. Each option corresponds to either insertion, deletion or substitution. Once algorithm found the edit distance we use align() function to follow the instruction we got with edit distance to produce an optimal alignment of two strings.

3.3.2 Computational Complexity

Since each execution of the algorithm we will make 3 recursive calls to the function, we are going to obtain a ternary tree of depth at most $\min(n, m)$, therefore this algorithm have a time complexity bounded by $O(3^{\min(n,m)})$.

3.4 Branch and bound version of the recursive approach

3.4.1 Overview

Branch and bound is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as mathematical optimization.

A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root.

The algorithm explores branches of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm

The goal of a branch-and-bound algorithm is to find a value of each node x that maximizes or minimizes the value of a real-valued function f(x), called an objective function, among some set S of admissible, or candidate solutions. The set S is called the search space, or feasible region.

To compute the objective function we calculate the sum of g(x) as the cost

and h as the heuristic (i.e. f(x)=g(x)+h), where g(x) definition depends on the problem considered A branch and bound algorithm operates according to two principles:

- It recursively splits the search space into smaller spaces, then minimizing f(x) on these smaller spaces; the splitting is called branching.
- Branching alone would amount to brute-force enumeration of candidate solutions and testing them all. To improve on the performance of brute-force search, a Branch and Bound algorithm keeps track of bounds on the minimum that it is trying to find, and uses these bounds to "prune" the search space, eliminating candidate solutions that it can prove will not contain an optimal solution.

Turning these principles into a concrete algorithm for a specific optimization problem requires some kind of data structure that represents sets of candidate solutions. Such a representation is called an instance of the problem. Denote the set of candidate solutions of an instance I by S_I. The instance representation has to come with three operations:

- branch(I) produces two or more instances that each represent a subset of S_I.
- bound(I) computes a lower bound on the value of any candidate solution in the space represented by I, that is, bound(I) $\leq$ f(x) for all x in S_I.
- solution(I) determines whether I represents a single candidate solution.

Using these operations, a branch and bound algorithm performs a top-down recursive search through the tree of instances formed by the branch operation. Upon visiting an instance I, it checks whether bound(I) is greater than the upper bound for some other instance that it already visited; if so, It may be safely discarded from the search and the recursion stops. This pruning step is usually implemented by maintaining a global variable that records the minimum upper bound seen among all instances examined so far.

3.4.2 Applications

Branch and bound approach is used for a number of NP-hard problems

- Edit distance problem
- Longest common subsequence problem
- Integer programming
- Traveling salesman problem (TSP)
- 0/1 Knapsack problem
- Feature selection in machine learning

3.4.3 Working Illustration

The Branch and bound version of the recursive approach to find the edit distance between two strings s1 and s2 sets a bound on the recursive approach to control the branching in the whole search space to find the optimal solution. The length of s1 is considered as m and length of s2 as n.

This version uses the bellow formula to control branching

$$f(x) = g(x) + h \tag{6}$$

where f(x) is the objective function of the node x and g(x) is the cost incurred to reach that node and it is initially 0 and h is the heuristic and it is calculated by

$$h = absolute(m - n) \tag{7}$$

Initially the bound is chosen to be worst which is

$$bound = maximum(m, n) \tag{8}$$

The algorithm checks the value of f(x) with the bound before proceeding with the recursion of 3 operations to explore the search tree, the algorithm explore the search space for optimal solution only if (f(x) $<$ bound), after reaching the first solution the algorithm checks if f(x) less than the worst bound chosen at the first and if the f(x) is lesser than the worst bound then it updates the bound with the best solution as the bound and continues to explore the search space for f(x) to be less than the bound and after the completion of 3 recursive call for each branch the algorithm takes the minimum of f(x) of the 3 recursive calls made and outputs the edit distance of

the two strings. By keeping track of the minimum f(x) the related operations are performed for the string alignment by the align function, which is either (nothing,substitution,deletion,insertion) The bellow recursive operations are performed for each branch in the algorithm by calling the function editdistBB(s1,s2,g)

$$\begin{aligned} &editdistBB(s1(m-1), s2(n-1), g+\delta) \\ &editdistBB(s1(m-1), s2(n), g+1) \\ &editdistBB(s1(m), s2(n-1), g+1) \end{aligned} \tag{9}$$

where δ is defined as follows:

$$\delta = \begin{cases} 0 & if\ s1[m-1] \ = \ s2[n-1] \\ 1 & otherwise \end{cases} \tag{10}$$

When we reach empty strings we return the value of f(x) with either of the operation which is insertion if s1 is 0 or deletion if length of s2 is 0, by updating the bound if f(x) is less than the previous bound.

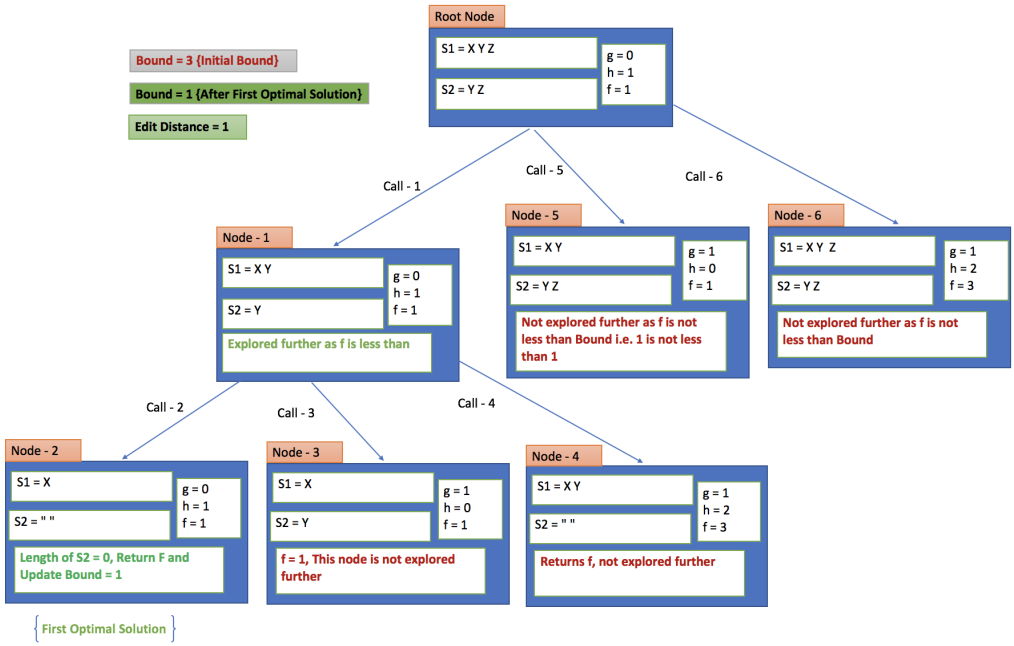


Figure 1: Branch and bound example to find edit distance

The above is the example to find the edit distance using branch and bound.

In this example the initial worst bound was 3 and it was updated to 1 after the first optimal solution was obtained and other nodes are not explored because f value is either greater than bound (which is $f > 1$) or it is equal to the bound ($f = 1$).

Edit Distance = 1 is calculated by minimum of f value i.e. minimum of f value of Node-2,Node-3,Node-4 is returned to Node-1 which is 1 and minimum of f value of Node-1,Node-5,Node-6 is returned to Root Node which is 1, therefore the edit distance is 1.

Operations performed here is [Deletion,Nothing,Nothing] because, At Node-2 the operation returned is Deletion as length of S2=0, At Node-1 the operation returned is Nothing as the character Y and Y matches, At Root Node the operation is Nothing as the character Z and Z matches.

3.4.4 Computational Complexity

In the worst case, the branch and bound algorithm must explore every node in the state space. And on each node 3 recursive operations are performed, which becomes a ternary tree of depth at most min(m,n) thus the time complexity will be $O(3^{\min(m,n)})$ and storing each node of the search space, then the space complexity will be $O(3^{\min(m,n)})$. Therefore the computational complexity of branch and bound algorithm to compute edit distance is exponential.

3.5 K-Strip Dynamic Approach

3.5.1 Overview

The dynamic case in section 3.1 witnessed the cost analysis of all possible combinations of a given input of any two strings. While this approach ensures optimal edit distance, exploring every one of their combinations is not necessary to arrive at the same. This is quite evident as in the majority of the cases the optimal path to the least cost solution does not stray too far from the main diagonal of the cost matrix. To avoid unnecessary computations and still arrive at the optimal solution, the K strip dynamic approach is used. This approach is an approximation of the dynamic implementation of the problem of edit distance. The only difference between the two is that, within the K strip approach a strip length of only 2k+1 along the main diagonal of the cost matrix is explored. This would drastically help reduce the number of computations, especially when dealing with long strings.

Like in the above cases we start by defining the rules for the cases one will encounter during the iterations process if $|a| = 0$ or $|b| = 0$ then edit distance is $|b|$ and $|a|$ respectively. Otherwise, let $D[i, j]$ be the edit distance for two strings at indices i and j. The define the cost function as follows:

$$D[i,j] = \min \begin{cases} D[i-1, j-1] + \delta(i,j) \\ D[i-1, j] + 1 \\ D[i, j-1] + 1. \end{cases} \tag{11}$$

where gap penalty δ is defined as follows:

$$\delta = \begin{cases} 0 & if\ a[i-1] = b[j-1] \\ 1 & otherwise \end{cases} \tag{12}$$

With the cost function defined we move on to defining the matrix of dimension ($|a|+1$, $|b|+1$). An infinity matrix is first defined onto which the K-strip algorithm is applied, populating the main diagonal of the cost matrix based on a k value. To understand better the working of the K-strip algorithm let's consider the example below.

Given two input strings a = "asdf" and b = "sdsd" below are the cost matrices returned by the dynamic and K-strip algorithm - assuming that a is to be converted to b . The cells in blue signify the main diagonal of the matrices, and the cells in the green illustrate the strip along the diagonal. In case 1, the dynamic approach conducts a thorough search for every combination of the two strings to arrive at an optimal cost of 3. Case 2 sees a similar implementation, but only along the main diagonal since k = 0. This results in an optimal cost of 5, which is evidently higher than 3. In case 3, k is assigned a value of 1, which increases the search space of the algorithm and an optimal cost of 3 is attained.

Hence, selecting the right k value would help achieve optimal cost as in the dynamic case, with lesser number of computations.

Case 1 : Dynamic

		a	s	d	f
	0	1	2	3	4
s	1	1	1	2	3
d	2	2	2	1	2
s	3	3	2	2	2
d	4	4	3	2	3

Case 2.1 : K-strip with k = 0

		a	s	d	f
	0	Inf	Inf	Inf	Inf
s	Inf	1	Inf	Inf	Inf
d	Inf	Inf	2	Inf	Inf
s	Inf	Inf	Inf	3	Inf
d	Inf	Inf	Inf	Inf	4

Case 2.2 : K_strip with k = 1

		a	s	d	f
	0	1	Inf	Inf	Inf
s	1	1	2	Inf	Inf
d	Inf	2	2	1	Inf
s	Inf	Inf	2	2	2
d	Inf	Inf	Inf	2	3

3.5.2 Defining the main diagonal

Defining the main diagonal for non square matrices is not as straight forwards as with square matrices. Assuming that the string a is to be converted to b and the cost matrix D is defined by D[i,j] we have two cases to deal with,

Case 1 is when $|a|$ (a = fhj) is greater than $|b|$ (b = "frjl"). The diagonal here is defined by using the formula below

$$i =| ((j * m)/n) | \tag{13}$$

The index 'i' is assigned the absolute value of the index times m (length of the shorter string - $|a|$) divided by n (length of longer string - $|b|$). Index 'i' would also serve as the rows along which a strip of the cost matrix will be explored when the value of k ¿ 0. Case 1.1 illustrates this with k = 1. We are able to observe how one cell is explored above and below each diagonal element.

Case 2 simply inverses the scenario and here length of a= "fhj" is smaller than length of b = "frjl". The diagonal here is defined by using the formula below:

$$j =| ((i * m)/n) | \tag{14}$$

Here index 'j' serves as the columns along with a strip of the cost matrix will be explored when the value of k ¿ 0. Case 2.1 illustrates with k = 1.

Case 1 : length of string 1 > length of string 2

		f	r	j	l
	0	1	Inf	Inf	Inf
f	Inf	Inf	2	Inf	Inf
h	Inf	Inf	Inf	3	Inf
j	Inf	Inf	Inf	Inf	4

Case 2 : length of string 2 > length of string 1

		f	h	J
	0	Inf	Inf	Inf
f	1	Inf	Inf	Inf
r	Inf	2	Inf	Inf
j	Inf	Inf	3	Inf
l	Inf	Inf	Inf	4

Case 1.1 : k =1

		f	r	j	l
	0	1	2	Inf	Inf
f	1	0	1	2	Inf
h	Inf	Inf	1	2	3
j	Inf	Inf	Inf	1	2

Case 2.1 : k =1

		f	h	J
	0	1	Inf	Inf
f	1	0	Inf	Inf
r	2	1	1	Inf
j	Inf	2	2	1
l	Inf	Inf	3	2

3.5.3 Computational Complexity

The space complexity of this algorithm is O(mn) similar to the dynamic approach. The time complexity though is given by O(k* min(m,n)). As seen in the cases above, whenever a given string was longer the k-strip was formed along the rows or columns the longer sting was defined on.

3.5.4 Linear Model to find optimal K

Finding the right k value would require us to run the dynamic approach and obtain the optimal cost of edit distance. Once we have this, iterations are performed with the K-strip algorithm for different k values in an effort to arrive at the optimal cost. Though this method would help find the value of k required to obtain the optimal edit distance, the time complexity for the same would be similar to dynamic approach's. To make use of the clear advantage that the K-strip algorithm holds over the dynamic approach we used a linear model to predict k values, given the length of the longer string. To our surprise the linear model made predictions that were quite accurate, nearing 99% as seen in section 4.1.3.

Given the model's high accuracy and linear time complexity the K-strip algorithm is later even used to compute the edit distance for 2000 protein strings in the protein database

3.6 Greedy approach

3.6.1 Overview

The goal is to convert string s1 into string s2 such that s1 is identical to s2 at the end of the algorithm based on greedy approach.

The idea of this greedy approach is to compare character from the current index of string s1 to same index of string s2 and the next two indexes then making greedy decision which will be based on:

- The length of two strings.
- The number of available index for checking in string s2 when comparing with string s1

3.6.2 Working, Illustration and complexities

To understand how the approach works, let's consider examples below:

Example 1: First string is shorter than the second string

s1 = "ACELR"

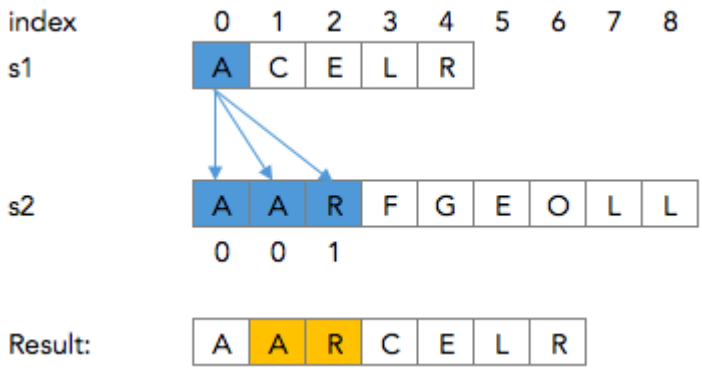
s2 = ”AARFGEOLL”

Legend for the illustration:

A	Checking position	d: number of insertion
A	Checked position	0: identical character
A	New insertion	1: different character
A	Substitute	i: iteration
A	Deleted	c: cost
A	No change	

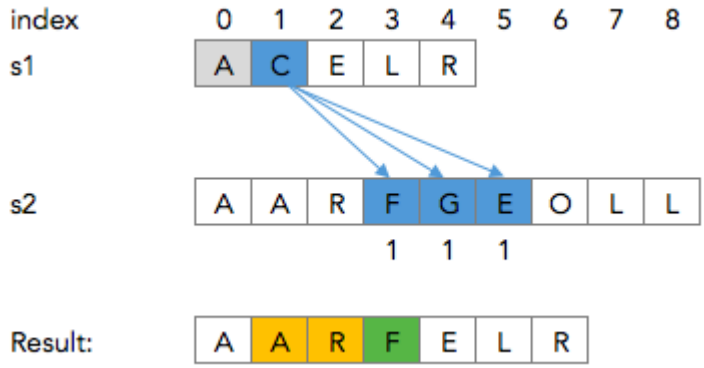
The algorithm will check each character in string s1 with characters in string s2. This will go through a loop.

- First iteration: i = 0



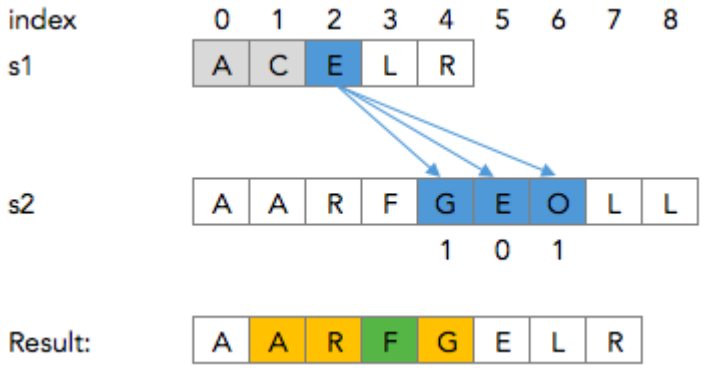
In this case, we check if length of s1 is shorter than s2 or not. If yes, we will prioritize insertion to make s1 longer. Since the length gap between s1 and s2 is greater than 2, we can insert more than 1 character. In this case, value s2[1] and s2[2] will be inserted after value s1[0]. Now, we have 3 common characters in new string and s2. Number of insertion is 2 (or d=2). Increase cost by 2 (c=2).

- Second iteration: i = 1



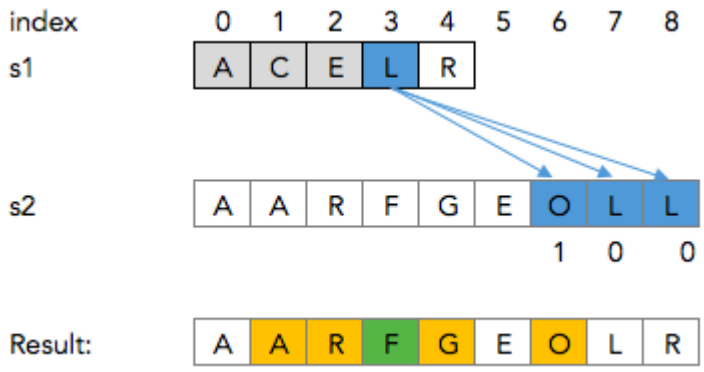
We check character s1[i] (or s[1]) with characters in s2[i+d] , s2[i+d+1], and s2[i+d+2] (or s2[3], s2[4], s2[5]). In this case we don't have common character so we substitute character at s2[3] for s1[1]. Increase cost by 1 (c=3).

- Third iteration: i = 2



We check character s1[i] (or s[2]) with characters in s2[i+d] , s2[i+d+1], and s2[i+d+2] (or s2[4], s2[5], s2[6]). In this case we have 1 common character and length gap between the 2 strings after insertion in previous steps is still greater than 0 so we insert character at s2[4] before s1[2]. Increase number of insertion by 1 (or d=3). Increase cost by 1 (c=4).

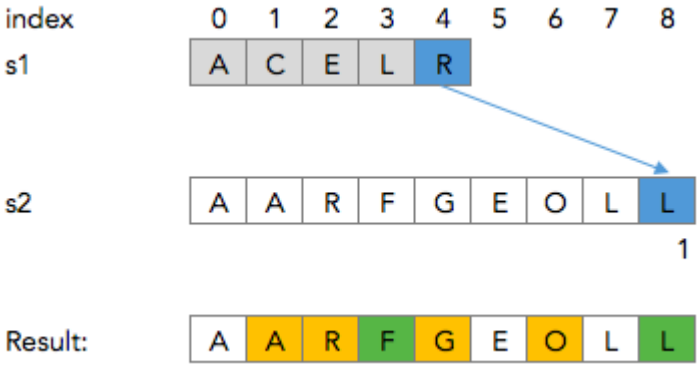
- Fourth iteration: i = 3



We check character s1[i] (or s[3]) with characters in s2[i+d] , s2[i+d+1], and s2[i+d+2] (or s2[6], s2[7], s2[8]). In this case we have 2 common characters and length gap between the 2 strings after insertion in previous steps is only 1 so we insert character at s2[6] before s1[3]. Increase number of insertion by 1 (or d=4). Increase cost by 1 (c=5).

Note: Normally, if the length gap is 2 or greater, we will insert 2 characters at position s2[6] and s2[7] before s1[3].

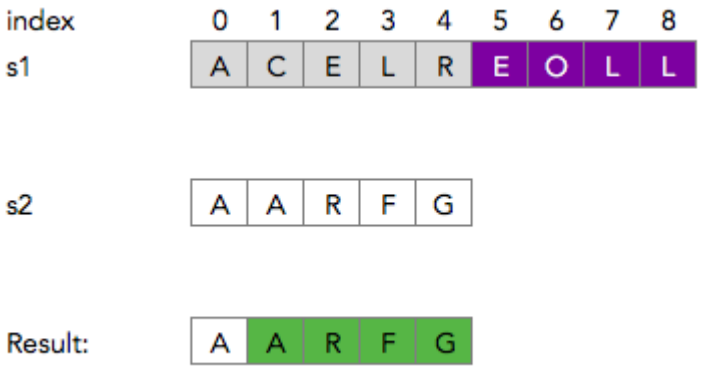
- Last iteration: i = 4



We check character s1[i] (or s[4]) with characters in s2[i+d] (or s2[8]) because there is no more index to check further. In this case we have no common characters and length gap between the 2 strings after insertion in previous steps is zero so we can only substitute character at s2[8] for s1[4]. Cost is 6 now. This is the last iteration.

Example 2: First string is longer than the second string

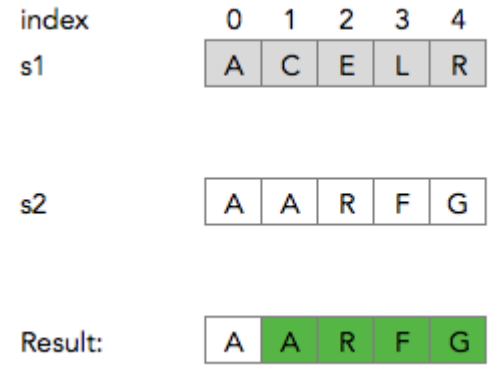
s1 = "ACELREOLL"
s2 = "AARFG"



In this case, at the end of iteration (i=4). The rest of s1 will be deleted. Total cost will be 8 (4 for substitute and 4 for deletion).

Example 3: Two strings have the same length

s1 = "ACELR"
s2 = "AARFG"



In this case, the length of 2 strings is the same, we can't insert more characters but have to substitute every different one until the end of the string.

Here the cost is 4 for all substitution.

The time complexity of this approach is linear O(n) with n is the length of the shortest string between s1 and s2. However, the cost of this approach is not optimal; we can have estimated cost only.

4 Class Alignment

When the instance of Dynamic Programming Alignment is created, __init__ method is called, which takes 2 strings to align as the parameters. On initialization, we define the following instance variables:

A -string- Instance variable for the first string B -np.array- Instance variable for the second string pref_matrix - an empty numpy array of shape

$$(|A| + 1, |B| + 1)$$

where +1 is for the cost of aligning the empty string. After the creation, we assign to the first row and first column of pref_matrix the values returned by np.arange() function. np.arange enumerates all integer values in given range, here,

$$(0, |A| \, or \, |B|)$$

i.e. edit distance for the empty string. edit_distance int Instance variable for the edit distance, at the moment of initialization, is None. got_matrix bool Instance variable to signalize if prefix matrix was computed. At the moment of initialization is False.

row1,row2,row3 -string- Instance variables which contains optimal alignment of 2 strings + operational row in the middle. At the moment of initialization is empty

4.1 Instance Methods

$$compute\_matrix(self)$$

This method takes an instance of the class as a parameter and computes the prefix edit distance matrix between two string using classic dynamic programming algorithm. Once the matrix is computed, function assigns the

value of the bottom right cell of the matrix to the edit distance variable. get_matrix is assigned True. Prefix matrix is returned.

print_alignment(self,i=None,j=None) This method computes the optimal alignment between two strings. When the method called explicitly, it is called without parameters and default parameters None are assigned to indexes i and j. Once called, method checks if i and j are not None. If they are None, it means that it is the first iteration of the algorithm and we must start from the bottom right cell of the matrix, therefore indices

$$i = |A|, j = |B|$$

are assigned. Following, it checks if the prefix matrix had been computed, if not it calls compute_matrix(self) function. Then, algorithm back-trace the path which led to edit distance and record the corresponding operations into the instance variables row1,row2,row3. Every time function choose an operation to perform, it moves up,left-right, or left by calling itself recursively. We reach the bottom of recursion when i and j are both equal 0, which mean that we reached the top left corner of the matrix. The optimal alignment is printed as the last step.

get_edit_distance(self,print_distance=False) First, it checks is prefix matrix is computed, if not is compute it. Then, if print_distance is true, it prints the edit distance. Otherwise, it returns edit_distance as a value.

4.2 Optimal Alignment Style

We use the following style for the optimal string alignment (which appears to be commonly used in this domain in academia):

- if two characters are equal, we write the following:

$$\begin{array}{c} a \\ | \\ a \end{array}$$

  where pipe symbol indicates the equality of the string and absence of the operation.

- when two strings are not equal we choose to substitute one string into another we write following:

$$a$$

$$b$$

  where empty space signifies substitution.

- when we choose to either insert of delete string, we use the following notation

$$- \qquad c$$

$$b \qquad -$$

  which means that either insertion performed be made in the top(bottom) string or deletion must be performed in the bottom(top) string.

5 Evaluation of Algorithms On Artificial Data

5.1 Experimental study of running time of the algorithms

To evaluate the performance of our algorithms, we split them into two group: Exponential-time and Polynomial-Time. Greedy, K-Strip, Divide and Conquer, Dynamic Programming have polynomial running time, pure Recursive and Branch and bound have exponential running time. Then, we also have to group of algorithms regarding accuracy, exact and approximate. K-strip and Greedy are approximate; all others are exact. For the approximate algorithms, we provide an accuracy study. The experimental protocol for each study will be specified in each respective section. We perform all the tests on Intel Core i7 2.3 GHz machine running Mac OS X.

5.1.1 Polynomial time algorithms

We evaluate the running time in the following setting: for each string length from 1 to 50 we generate 20 random string pairs and measure execution

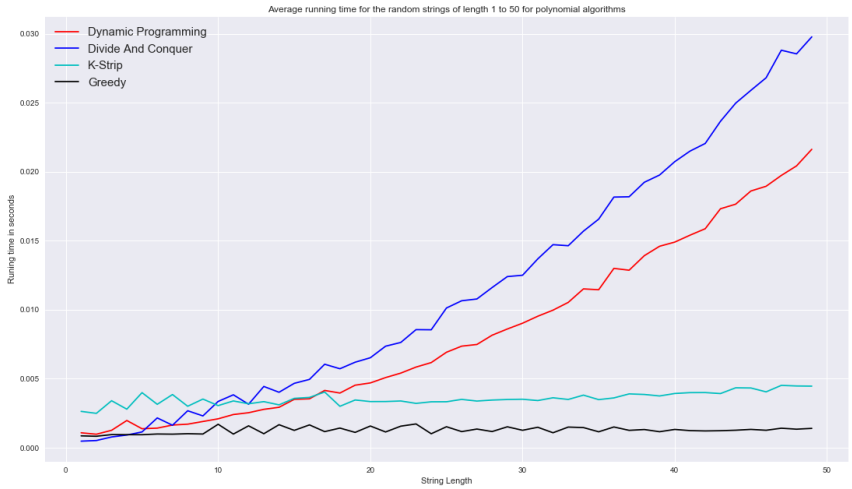


Figure 2: Average running time of random strings of length 1 to 50 for polynomial algorithms

time for both computing edit distance and printing optimal alignment. We then compute the mean running time for 20 strings and save it. The resulting running times are presented in the graph below: As one may observe, Greedy and K-Strip algorithms having the smallest, almost constant running time. This performance is achieved mainly because of their inexact nature, so it comes trade-off regarding accuracy of edit distance. The accuracy is discussed in further sections. Also, we can observe that for string length up to 10 K-Strip performance is the worst among others, mainly due to fixed K, which fits better for longer strings. Further, we observe the expected performance of dynamic programming algorithms. The classic version is faster than Divide and Conquer, since as we showed earlier, the Divide and Conquer version on average computes the same matrix as a classical algorithm but twice, while gaining regarding space efficiency.

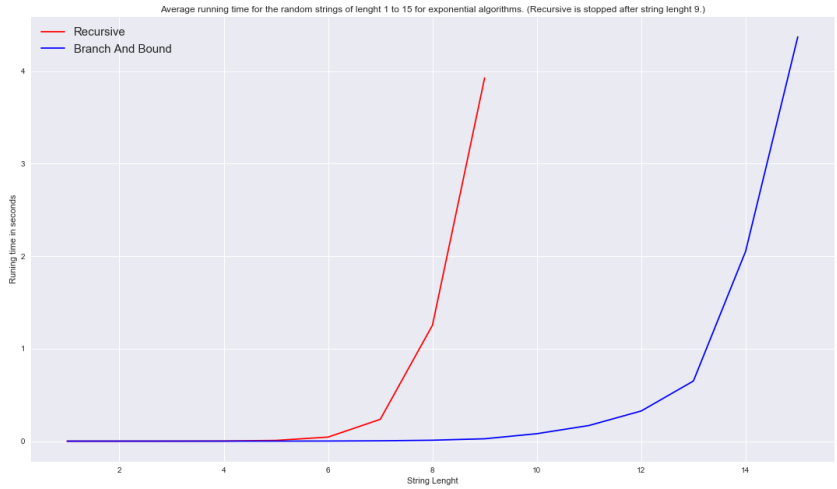


Figure 3: Average running time of random strings of length 1 to 15 for exponential algorithms

5.1.2 Exponential time algorithms

For the exponential time algorithms we use a similar setting as for polynomial, but we use maximum string length of 15 for the branch and bound algorithm and 9 for a recursive algorithm. These values were found empirically and allow for evaluation of the performance of the algorithms in a reasonable time. We see that for strings length up to 12 branch and bound performs efficiently, mainly due to heuristic used. After length 12 the heuristic fails and we observe the exponential performance similar to a recursive algorithm.

5.1.3 Accuracy study for approximate algorithms

We define the accuracy of the approximate algorithm as follows:

$$\frac{\textit{Edit distance given by exact algorithm}}{\textit{Edit distance given by approximate algorithm}}$$

Therefore the accuracy is 1 when algorithms output is an exact edit dis-

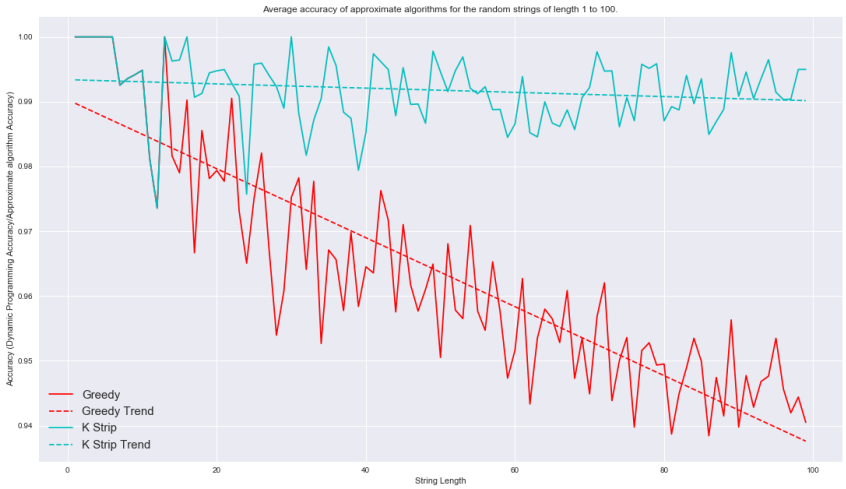


Figure 4: Average accuracy of approximate algorithms for the random strings of length 1 to 100

tance, and less than 1 to the degree of approximation. We have the similar test protocol as in previous sections; we measure average performance of 20 strings for each string length from 1 to 100. As we can observe on the graph, the accuracy for both algorithms is very uneven (we include linear trend for visual convenience), it can be explained by the fact that both algorithm use heuristic of some kind and the one is highly dependent on the particular input. Nevertheless, K-strip algorithm is accurate enough for most of the applications with the low of 0.975. Not surprisingly, we serve the downwards trend for both models when string length increases.

6 Evaluation of Algorithms On Protein Database

6.1 Protein Evaluation

From the protein database a list of 6222 amino acids sequences was exported for alignment evaluation and label prediction based on the edit distances between each sequence and every other in the list.

The idea was to create an edit distance matrix - a matrix with the edit distances between every possible pairwise combination of protein sequences - using the dynamic approach, as an optimal solution is always guaranteed with it. But given the size of the dataset and the polynomial time complexity of the algorithm, creating the edit distance matrix was taxing and time-consuming.
To overcome this, the K-strip algorithm was used in combination with a linear model to predict k values - as discussed in section 3.5.4. Using this approach the computation time was drastically reduced, however, due to time constraints only 4000 out of the 6222 protein sequences were evaluated.

6.2 Dataset and Model Creation

The dataset was created by first calculating the edit distance matrix and then appending to it the labels of each of the protein strings. Resulting in a total of 4000 rows and 4001 columns. The edit distances between the protein sequences were assumed to be features and the label the dependent variable.

Since the problem at hand - classification of protein strings based on their labels - was a classification type problem, we thought it is best to train a random forest classifier over it. Being an ensemble type classifier, it is less prone to overfilling, and studies have attested to its performance on classification problems. The dataset was trained with up to 1000 tress under a ten-fold cross-validation setting.

6.3 Results

The results from the random forest classifier have been illustrated using the classification report and an accuracy histogram below. The model at average exhibited an accuracy of 88% with up to 100% accuracy on predicting a few classes. The histogram attests to this by illustrating the difference in frequencies between the actual and predicted classes. (Note:- Class 99 was converted to 22 to visualize the classification better)

Table 1: Classification Report of Random Forest Model

Class	**Precision**	**Recall**	**F1-score**	**Support**
1	0.97	0.93	0.95	83
2	1.00	1.00	1.00	21
3	0.86	1.00	0.92	6
4	0.83	0.71	0.77	7
5	1.00	1.00	1.00	8
6	0.92	0.92	0.92	24
7	0.95	0.83	0.89	24
8	0.90	1.00	0.95	9
9	1.00	0.97	0.98	65
10	0.95	0.99	0.97	158
11	0.85	0.65	0.73	188
12	0.92	0.89	0.90	112
13	1.00	0.75	0.86	12
14	1.00	0.81	0.90	16
15	1.00	1.00	1.00	4
16	1.00	1.00	1.00	14
17	1.00	1.00	1.00	21
18	1.00	0.50	0.67	2
19	1.00	1.00	1.00	3
20	1.00	0.88	0.94	33
21	1.00	1.00	1.00	28
99	0.76	0.89	0.82	362
Avg/Total	0.88	0.88	0.88	1200

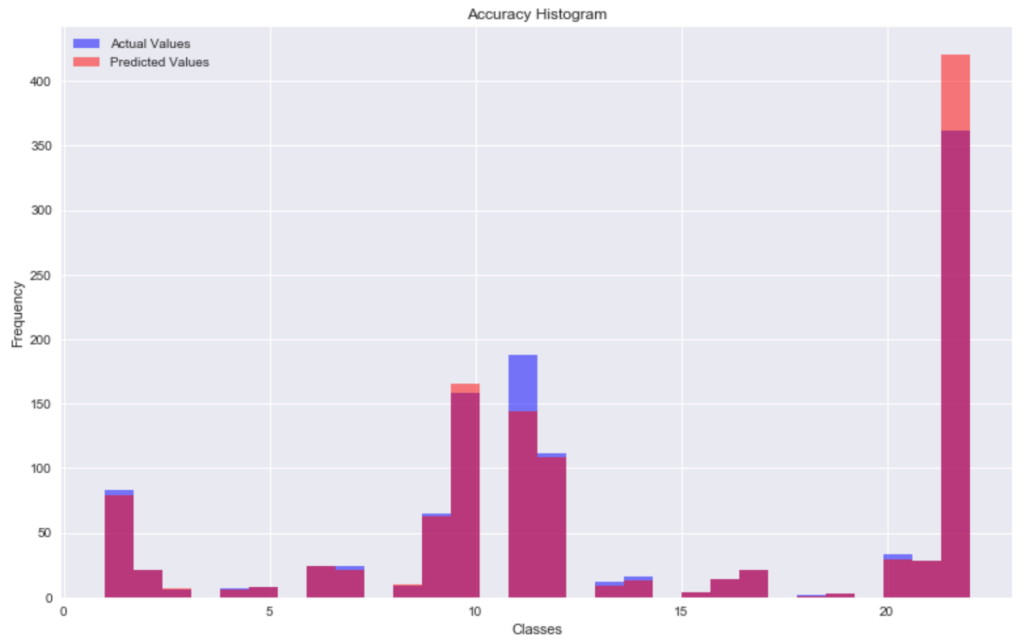


Figure 5: Accuracy Histogram on Predicted Classes vs Actual Classes

7 Planning

The planning realized (shaded in red) for the project came out to be rather different from the initial planning (blue fill) as can be seen from the illustration above (Figure best viewed magnified). This was essentially due to the following difficulties encountered by the group:

- Lack of experience coding in the programming language Python.
- Disparate learning rates for each member of the group.
- Uncertainty regarding the methodology to be followed and implementation of algorithms on the protein database.
- Due to the lack of time GUI could not be implemented (shaded in grey)

Project Name	Edit Distance	Group Name	MLDM
Group Members	Allwyn Joseph Doan Tu My Evhenii Zotkin Karthik Bhaskar		

ADVANCED ALGORITHMS PROJECT PLANNING

			November																									December											
Month			F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T					
Day			10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	1	2	3	4	5	6	7	8	9	10	11	12					
Date																																							
Principal Tasks	**Sub Tasks**	**Contributors**																																						
Algorithm notion, substructure and complexity	Classic version	Evhenii																																						
	Dynamic version	Evhenii																																						
	Pure recursive version	Evhenii																																						
	Branch and Bound version	Karthik																																						
	Approximation of dynamic version	Allwyn																																						
	Approximation of greedy version	My																																						
Implementation and evaluation of algorithm with python on artificial data	Classic version	Evhenii																																						
	Dynamic version	Evhenii																																						
	Pure recursive version	Evhenii																																						
	Branch and Bound version	Karthik																																						
	Approximation of dynamic version	Allwyn																																						
	Approximation of greedy version	My																																						
Implementation and evaluation of algorithm with python on data from protein data base	Classic version	Evhenii																																						
	Dynamic version	Evhenii																																						
	Pure recursive version	Evhenii																																						
	Branch and Bound version	Karthik																																						
	Approximation of dynamic version	Allwyn																																						
	Approximation of greedy version	My																																						
~~GUI implementation~~		~~Allwyn, Evhenii~~																																						
Report	Introduction	Karthik																																						
	Overview of the problem of edit distance and similar applications	Karthik																																						
	Overview of Algorithms used, their substructure and complexity	Allwyn,Evhenii, Karthik, My																																						
	Results after having it tested on the artificial data	Allwyn,Evhenii, Karthik, My																																						
	Results after having it applied on protein data base	Allwyn,Evhenii, Karthik, My																																						
	~~GUI used and overview~~	~~Allwyn, Evhenii~~																																						
	Conclusions	Allwyn																																						
KEY	***Completed***		***Realised planning***																																					
	Planned (to-do)		***Tasks Omitted***																																					

Figure 6: Planning

8 Conclusions

The Advanced Algorithm project was an immense learning experience for the group as a whole. Coming from different backgrounds, both culturally and professionally working together in sync was quite the learning experience.The planning in section 7 illustrates this to some extent. As planned, we were able to stick to our schedule and realize tests with our algorithms on both, artificial and the protein database with much success. Unfortunately, due to time constraints and other commitments we were not able to realize a functional GUI. Nevertheless, we enjoyed the challenge and would welcome another in the coming semesters.

9 References

- **Hirschberg's Algorithm** `http://users.monash.edu/~lloyd/tildeAlgDS/Dynamic/Hirsch/`
  `https://en.wikipedia.org/wiki/Hirschberg%27s_algorithm`
- **Dynamic Programming Algorithm (DPA) for Edit-Distance** `http://users.monash.edu/~lloyd/tildeAlgDS/Dynamic/Edit/`
- **Recent Developments in Linear-Space Alignment Methods** `https://www.csie.ntu.edu.tw/~kmchao/papers/1994_JCB.pdf`
- **Pairwise Sequence Alignments** `http://www.bgjackson.net/bcb597/supplement/alignment.pdf`
- **Edit Distance.** `https://en.wikipedia.org/wiki/Edit_distance`