PROTEIN FOLDING IN THE 3D HP MODEL

5. August 2016
Lars Christensen - 20115025

Supervisor:
Christian N. S. Pedersen

Datalogisk Institut
Aarhus Universitet
Abstract

The focus of this thesis is to present and implement different algorithms for the protein folding problem in 3D hp model. The prediction of a protein’s structure is an important problem in computational biology. The protein folding problem has been researched in many years and is a NP-hard problem, where different methods have been made over time, to solve the problem. Some of these methods are Ant Colony Optimization (ACO) and the 3/8-algorithm, which are introduced in this project and in addition to it, a new algorithm is also presented. The new algorithm is based on the basic principles of the 1/4-approximation algorithm for 2D HP model. It is adapted for the 3D HP model.

We present an implementation of ACO, 3/8-algorithm and a new algorithm, and the comparison of the quality between the three algorithms. We show that with random generated protein sequences, the new algorithm performs overall better than ACO and the 3/8-algorithm. We also shows how important it is to find the correct parameters for ACO.

The energy outputs from the algorithms for the protein folding problem is still far from the optimal solution. So there are still good prospects to improve the three algorithms, through a more complex and realistic model of protein structure.
# Contents

Abstract i

1 Introduction 2
   1.1 Proteins and their structure .................................. 2
   1.2 HP Protein folding .............................................. 4
   1.3 Methods ..................................................... 5

2 3/8-algorithm - Simple Approximation Algorithm 6
   2.1 Subroutine 1 .................................................. 8
   2.2 Simple Approximation Algorithm ............................ 9
   2.3 Construction and composition of superblocks .............. 11
   2.4 Energy calculation .......................................... 13

3 Ant Colony Optimization 15
   3.1 Initializing the pheromone matrix ............................ 16
   3.2 Construction Phase: Pheromone and heuristic values .... 16
   3.3 Local search ................................................ 18
   3.4 Updating the pheromone trails ................................ 19

4 1/4-Simple3D 20
   4.1 Construction of the blocks and connections ............... 24
   4.2 Major blocks ................................................. 25
   4.3 Connection blocks .......................................... 26
   4.4 Energy calculation .......................................... 28

5 Experiments & Results 30
   5.1 Choosing the right parameters for Ant Colony Optimization 30
   5.2 ACO, Simple Approximation Algorithm & 1/4-Simple3D .... 37

6 Discussion of the results 41
   6.1 Comparison of the three algorithm .......................... 42

7 Conclusion & Future work 44
   7.1 Conclusion .................................................. 44
   7.2 Future work ................................................. 44

Primary Bibliography 45
Chapter 1

Introduction

The goal for this thesis is to introduce and present implementations of algorithms for the folding of proteins in a 3D hydrophobic polar model[1] and then compare the quality of the algorithms with each other. The requirements for the algorithms is that it shall run in linear or polynomial time while getting a satisfied result, which can be compared with other algorithms.

We will start with an introduction to proteins, their structure and how they fold, followed by the way we choose to represent them in a 2D and 3D HP model. Other people’s version of the implementation of a 3D HP model will be presented. It will contain how they have made them and how we have chosen to implement them. Afterward we will present our own version of an implementation, where an explanation of the implementation and how we did it will be included.

Then there will be a section containing the experiments of the comparison between the quality of 1/4-Simple3D and the others. In the same section there is also included an experiment about choosing the correct parameters for the algorithms followed by the results from the experiments. Finally a summary of the everything will be made with a discussion describing the results from the experiments, followed by a conclusion and future work.

1.1 Proteins and their structure

Proteins is one of the three major macromolecules that are essential for life, alongside DNA and RNA. Proteins are composed by polypeptide chains, which generally consist of an arbitrary amount of 20 different kinds of amino-acids, that are covalently bonded through peptide bonds. In certain organisms the protein can include two more amino-acids. There are total 22 standard amino-acids, where 9 of them are called essential amino-acids because the human body cannot synthesize them from other compounds and therefore need to obtain it through food. The non-essential amino-acids are those the human body can synthesize. The 20 amino-acids that we generally can find in a protein are:
<table>
<thead>
<tr>
<th>Amino-acid</th>
<th>Essential</th>
<th>Hydropathy index</th>
</tr>
</thead>
<tbody>
<tr>
<td>A - Alanine</td>
<td>Yes</td>
<td>1.8</td>
</tr>
<tr>
<td>R - Arginine</td>
<td>Yes*</td>
<td>-4.5</td>
</tr>
<tr>
<td>N - Asparagine</td>
<td>Yes</td>
<td>-3.5</td>
</tr>
<tr>
<td>D - Aspartic acid</td>
<td>Yes</td>
<td>-3.5</td>
</tr>
<tr>
<td>C - Cysteine</td>
<td>Yes*</td>
<td>2.5</td>
</tr>
<tr>
<td>E - Glutamic acid</td>
<td>Yes</td>
<td>-3.5</td>
</tr>
<tr>
<td>Q - Glutamine</td>
<td>Yes*</td>
<td>-3.5</td>
</tr>
<tr>
<td>G - Glycine</td>
<td>Yes</td>
<td>-0.4</td>
</tr>
<tr>
<td>H - Histidine</td>
<td>No</td>
<td>-3.2</td>
</tr>
<tr>
<td>I - Isoleusine</td>
<td>No</td>
<td>4.5</td>
</tr>
<tr>
<td>L - Leucine</td>
<td>No</td>
<td>3.8</td>
</tr>
<tr>
<td>K - Lysine</td>
<td>No</td>
<td>-3.9</td>
</tr>
<tr>
<td>M - Methionine</td>
<td>No</td>
<td>1.9</td>
</tr>
<tr>
<td>F - Phenylalanine</td>
<td>No</td>
<td>2.8</td>
</tr>
<tr>
<td>P - Proline</td>
<td>Yes*</td>
<td>-1.6</td>
</tr>
<tr>
<td>S - Serine</td>
<td>Yes*</td>
<td>-0.8</td>
</tr>
<tr>
<td>T - Threonine</td>
<td>No</td>
<td>-0.7</td>
</tr>
<tr>
<td>W - Tryptophan</td>
<td>No</td>
<td>-0.9</td>
</tr>
<tr>
<td>Y - Tyrosine</td>
<td>Yes*</td>
<td>-1.3</td>
</tr>
<tr>
<td>V - Valine</td>
<td>No</td>
<td>4.1</td>
</tr>
</tbody>
</table>

Table 1.1: (*) Essential only in certain cases[6][8]

The hydropathy index are values that define the relative hydrophobicity of amino-acid residues. The more positive value the more hydrophobic is the amino-acid. And likewise the more negative value the more hydrophilic is the amino-acid. Through this project we will work with the symbols H and P, where H is the amino-acids that are hydrophobic and P is the amino-acids that are hydrophilic.

Protein structure is outlined on four different levels. These are: the primary structure, the secondary structure, the tertiary structure and the quaternary structure.

The primary structure describes the order of the amino-acids, which is typically represented by the amino-acid’s 1-letter symbol (which is shown in the table above) or 3-letter symbol. The secondary structure is the pattern of how the polypeptide chain can be folded. This is also called the structural elements. They consist of hydrogen bonds of the protein, such as alpha helices and beta sheets. The tertiary structure shows how the structural elements is located relative to each other and the quaternary structure is when several polypeptide chains are put together to form one large protein.
1.2 HP Protein folding

When proteins fold, they will fold from inactive to active state which is from the left hand side of Figure 1 to the right hand side. The active state is also called the native state, which is assumed to be the conformation with the lowest free energy. The energy is measured by assigning a negative weight to the interactions between the H which bonds. Predicting the way in which various proteins fold can be fundamental in developing treatments of diseases such as Alzeihmer’s and Cystic Fibrosis.[5]

There are different ways to represent a protein folding and the one that we will be working with is the simple folding model as written earlier. The simple folding model, is a way to represent the amino-acids types as binary representations, in the form of H and P or 0 and 1, also called the HP model. With this we can reduce the problem of the protein folding in 3D to a binary problem which we call a protein folding in a 3D HP model. The HP method is one of the ways to simplifying the problem, so it is easier to work with. Even though we disregard many of the details of protein folding, then it still remains as a NP-hard problem[4]. There have been constructed several methods for solving the HP protein folding problem and in this project, we will look into three different ways to solve it.

An algorithm that solves the HP protein folding problem typically gives a conformation as an output. This conformation is a string consisting of symbols that either can be the directional path or the relative path in which the proteins fold. If we look at a 3D plain, a direction in the directional path can either be west (W), north (N), east (E), south (S), up (U) or down (D) compared to which way you move in the 3D plain. The relative path can either be left (L), right (R), straight (S), up (U) or down (D), which is chosen relative to the direction you came from. Both the directional path and the relative path is illustrated in Figure 2
1.3 Methods

There have been a lot of research in the area of protein folding and there is still much to be done for finding the optimal solution in polynomial time or faster. One can easily find the optimal solution for a protein structure in exponential time, with for example brute forcing. But we will not do that in this project since it is resource-intensive and the execution of the algorithm will not be done running in our lifetime, for longer strings of proteins.

We will present algorithms which can solve the 3D HP protein folding problem in polynomial and linear time. These are the 3/8-algorithm, the Ant Colony Optimization and an algorithm we have made called 1/4-Simple3D. The code for the three algorithms, which have been made for this project can be found in the link in the references[3]
Chapter 2

3/8-algorithm - Simple Approximation Algorithm

The first algorithm is the 3/8-algorithm made by Hart and Istrail [7]. They have two versions: the Simple Approximation Algorithm and the Iterative Approximation Algorithm. We have chosen to implement the Simple Approximation Algorithm, which is the one we will present in this project. The reason for why we have chosen the Simple Approximation Algorithm is because the Iterative Approximation Algorithm is a little more complex, while the performance guarantee remains the same.

The figures which we use to explain the algorithm through this section is taken from the paper from Hart and Istrail. The conformation that the algorithm generates, consist of total six superblocks where four of them are the major superblocks and two of them are connecting superblocks. It starts by splitting the string up in two superblocks and the splitting the two superblocks further up into of total six superblocks.

A superblock $B_i$ is a sequence consisting of blocks and block separators, which is made up as follows:

$$B_i = b_iz_i...z_{i_{n-1}}b_{i_n}$$

Where $b$ is a block and $z$ is a block separator. So to construct the superblocks we need analyze the protein string first.

A string for the HP-model, consist of the symbols $h$ and $p$. We will through the review of this algorithm work with the numbers of 0 and 1 instead, where 0 represents $p$ and 1 represents $h$.

A string $S$ can be divided into a sequence of blocks, which is illustrated on Figure 3, where a block $b_i$ the form $b_i = 1$, or $b_i = 1Z_{i_j}1...Z_{i_n}1$. $n \geq 1$ and $Z_{i_j}$ is a odd length sequence of 0s for $j \leq n$. This means that each block always starts and ends with a 1 and between those 1s there is an odd length of 0s.
Figure 3. A string divided into blocks, which are separated by an even number of zeroes.

The blocks are separated by a block separator $z_i$, that separates two consecutive blocks, where the $|z_i| \geq 0$ and $|z_i|$ is even for $i = 1, \ldots, n - 1$.

The 1s in a block $b_i$ can bond with the 1s from its neighbor blocks $b_{i-1}$ and $b_{i+1}$. This is true because the distance between our blocks is an even number and since the blocks start and end with 1s, then there is an even number of 0s between the 1s from $b_i$ and its neighbours. This is illustrated in a 2D plain between block $b_4$ and $b_5$ (from Figure 3) in Figure 4.

Figure 4. Shows how the 1s from a block bond with the 1s from another block.

The way we determine when a 1 bonds with another 1 in the algorithm, is when they are placed next to each other in the 3D plain. Therefore the 1s from a block $b_i$ can only bond with the 1s from a block $b_{i+j}$, where $j$ is an odd number. With this we can divide the blocks into two categories, namely the x-blocks and the y-blocks. To determine if a block $b_i$ is either a x-blocks or a y-blocks, it follows that either $x_i = b_{2i}$ and $y_i = b_{2i-1}$ or the other way around that $y_i = b_{2i}$ and $x_i = b_{2i-1}$.

Since it is possible for all the 1s in the x-blocks to bond with all the 1s in the y-blocks, we can use this to create our superblocks, which is necessary for constructing an conformation.

The string from Figure 3 can be represented as $S = z_0b_1z_1b_2z_2b_3z_3b_4z_4b_5z_5$. Let $N(y_i)$ be the number of 1s that is in the block $b_{2i-1}$, and $N(x_i)$ be the number of 1s in the block $b_{2i}$ for $0 < i < n$, where $n$ is the number of blocks, then it follows that $Z = [1, 0, 0, 0, 4, 0]$, $N(x) = (1, 3)$, and $N(y) = (3, 1, 4)$. This means that $|N(x)| = 2$ and $|N(y)| = 3$, where we can define

$$X = \sum_{i=1}^{|N(x)|} N(x_i) \quad \text{and} \quad Y = \sum_{i=1}^{|N(y)|} N(y_i)$$
so that \( X = 4 \) and \( Y = 8 \). \( X \) and \( Y \) is used to choose between the two possible block labelings by requiring that \( X \leq Y \) or the other way around. This means that the size of \( X \) and \( Y \) determines, whenever the first block in our string belongs to the \( x \)-block or to the \( y \)-block.

### 2.1 Subroutine 1

The labeling of the \( x \)- and \( y \)-blocks is the first step for the Subroutine 1, as shown in Figure 5. To determine \( S \), we weaves the lists of integers together from \( N(x) \) and \( N(y) \). So if we have the lists from before where \( N(x) = (1, 3) \), and \( N(y) = (3, 1, 4) \), then \( S = [3, 1, 1, 3, 4] \)

| 1. Given a protein instance decomposed into \( x \)- and \( y \)-blocks, compute \( N_x(x) \) and \( N_y(y) \). Let \( S \) represent this alternating sequence of integers.  
2. Find a balanced cut that partitions \( S \) into \( S_xS_y \) or \( S_yS_x \) such that \( \min[N_y(S_y), N_x(S_x)] \) is maximized.  
3. Let \( B' = S_x \) and \( B'' = S_y \). |

![Figure 5. A outline for the steps for Subroutine 1.](image)

We can now create the two first superblocks from our sequence of integers \( S \). The way we determine which part of the string that belongs to which of the superblocks, we need to find a balanced cut that partitions our sequence of integers \( S \) into \( S_xS_y \) or \( S_yS_x \) such that \( \min[N_y(S_y), N_x(S_x)] \) is maximized. \( N_y(S_y) \) is the number of 1s from the \( y \)-blocks in the string \( S_y \) and \( N_x(S_x) \) is the number of 1s from the \( x \)-blocks in the string \( S_x \).

Consider a sequence of positive integers, \( S_a = a_1, ..., a_k \) and let

\[
|S_a| = \sum_{i=1}^{k} a_i
\]

A cut for \( S_a \) is given by \((u, v)\) where

\[
u = \sum_{i=1}^{p} a_i \quad \text{and} \quad v = \sum_{i=p+1}^{k} a_i
\]

When \( u = v \), then the cut \((u, v)\) is called an equal cut.

As an example we have a sequence of nonnegative integers that is given by \( S = a_1, b_1, a_2, b_2, ..., a_k, b_k \), where \( a_i > 0 \), \( 1 \leq i \leq k \), \( b_j > 0 \), \( 1 \leq j \leq k - 1 \) and \( b_k \geq 0 \). We can define the a-sequence and the b-sequence of \( S \) as follows:

\[
S_a = a_1, ..., a_k  \\
S_b = b_1, ..., b_k
\]

A cut of \( S \) is a partition of \( S \) into two subsequences \( S = S' S'' \). If the cut is between \( a_i b_i \) then \( S' = a_1 b_1 ... a_i \) and \( S'' = b_i a_{i+1} ... a_k b_k \). If the cut is between \( b_i a_{i+1} \) then \( S' = a_1 b_1 ... b_i \) and \( S'' = a_{i+1} ... a_k b_k \).
A balanced cut for $S$ is a cut $S = S' S''$, where

$$|S'_a| \geq (|S_a| + 1)/2 \text{ and } |S'_b| \geq |S_b|/2$$

or

$$|S'_b| \geq (|S_b| + 1)/2 \text{ and } |S'_a| \geq |S_a|/2$$

2.2 Simple Approximation Algorithm

The two superblocks we have created from the balanced cut is the first step of the Simple Approximation Algorithm. The steps for the algorithm is shown in Figure 6. By labeling the blocks of a string $S$ and applying Subroutine 1, we get our two superblocks.

1. Label the blocks of $s$ with $y_1$ and $x_1$ such that $X \leq Y$ and if $X = Y$ then $T_x(s) \geq T_y(s)$. Apply Subroutine 1 to select a folding point that breaks $s$ into superblocks $B'$ and $B''$.
2. If $\min(N_y(B'), N_x(B'')) < 6$, then apply Algorithm 3. Otherwise, continue.
3. Break $B'$ into three superblocks $B'_1, B'_2$, and $B'_3$, where $B'_1$ is adjacent to the folding point, $N_y(B'_1) = \min(N_y(B'), N_x(B''))/2$, $N_x(B'_1) = 2$ and $N_y(B'_2) = N_y(B') - N_y(B'_1) - 2$. Break $B''$ into three superblocks $B''_1, B''_2$, and $B''_3$, where $B''_1$ is adjacent to the folding point, $N_y(B''_1) = \min(N_y(B'), N_x(B''))/2$, $N_x(B''_1) = 2$, and $N_y(B''_2) = N_y(B'') - N_y(B''_1) - 2$.
4. Construct a stepped $x$-superblock structure for $B'_1$ and a stepped $y$-superblock structure for $B'_2$. Fold about the folding point to place $B'_1$ and $B'_2$ face-to-face in the $x$-$y$ plane. Figure 15a illustrates the placement of $B'_1$ and $B'_2$, along with the configuration of $B''_1$ and $B''_2$, which connect the $x$- and $y$-superblocks respectively.
5. Construct a stepped $x$-superblock structure for $B''_1$ and a stepped $y$-superblock structure for $B''_2$. $B''_1$ is placed immediately behind $B'_1$ in the $y$-$z$ plane, and $B''_2$ is placed immediately behind $B'_2$ in the $y$-$z$ plane. Figure 15b illustrates the placement of $B''_1$ and $B''_2$.

Figure 6. An outline for Simple Approximation Algorithm.

In step two, we use the algorithm Beta, if the hydrophobic interactions are fewer than 6. The algorithm Beta, which is shown in Figure 7, is an improved 2D approximation algorithm made by Hart and Istrail. We have chosen not to explain or implement this algorithm in this project, since the focus lies on the protein folding in 3D. Therefore we always assume that the $\min(N_y(B'), N_x(B'')) \geq 6$. Figure 7 shows an outline of the Beta algorithm, but a more detailed explanation of the algorithm can be found in their paper[7].
Figure 7. Algorithm Beta, an improved 2D approximation algorithm.

At the moment we have two superblocks, where all the x-blocks in one of the superblocks can bond with all the y-blocks in the other superblock. We want to split the superblocks further up into three superblocks each, because it is required with the way that Hart and Istrail has chosen to construct the two superblocks relative to each other.

Therefore at step three in the Simple Approximation Algorithm for 3D, we need to break the two superblocks, received from Subroutine 1, into total of six superblocks. $B'$ becomes $B'_1$, $B'_2$ and $B'_3$, and $B''$ becomes $B''_1$, $B''_2$ and $B''_3$. For $B'_1$, $B'_2$ and $B'_3$, $N_y(B'_1) = (\min(N_y(B'), N_x(B'')) - 2)/2$, $N_y(B'_2) = 2$ and $N_y(B'_3) = N_y(B') - N_y(B'_1) - 2$.

For $B''_1$, $B''_2$ and $B''_3$, $N_x(B''_1) = (\min(N_y(B'), N_x(B'')) - 2)/2$, $N_x(B''_2) = 2$ and $N_x(B''_3) = N_x(B'') - N_x(B''_1) - 2$. The superblocks $B'_2$ and $B''_2$ works as a connection between the other superblocks, which will be explained in more details in the next section.
2.3 Construction and composition of superblocks

The six superblocks can now be used to construct the path, which either can be the relative path or the directional path. This is up to the developer. \( B'_{1}, B''_{1}, B'_{2}, B''_{2}, B'_{3}, B''_{3} \) are constructed in a very similar way. \( B'_{1}, B''_{1}, B'_{2}, B''_{2} \) are shown in Figure 8(a) and \( B'_{3}, B''_{3} \) in Figure 8(b). The black dots represents the 1s from \( N(B) \) that are on a fixed position relative to each other. All the elements between the 1s in one superblock has to be folded in the same way as a branch, compared to the skeleton of the stepped superblock structure. The longer the sequence is between two fixed 1s, the longer does the branch expand in the positive or negative \( x \) direction for \( B'_{1}, B''_{1}, B'_{3}, B''_{3} \). The superblocks \( B'_{2} \) and \( B''_{2} \), as mentioned earlier, are the blocks that connects \( B'_{1} \) with \( B'_{3} \) and \( B''_{1} \) with \( B''_{3} \). The final result where all the superblocks are connected together is shown in Figure 8(c). \( B'_{2} \) and \( B''_{2} \) always have two fixed positioned of 1s and like the other four superblocks, the elements around these 1s are constructed in a unique way. Let us say that \( B'_{2} = z_{1}x_{1}z_{2}x_{2}z_{3} \) where \( x_{1} \) and \( x_{2} \) are the fixed 1s and \( z_{1}, z_{2} \) and \( z_{3} \) are the elements around the fixed 1s, then \( z_{1} \) are constructed along the negative \( x \)-axis, \( z_{2} \) along the negative \( y \)-axis and \( z_{3} \) along the negative \( z \)-axis.

Likewise for \( B''_{2} = z_{1}x_{1}z_{2}x_{2}z_{3} \), \( z_{1} \) are constructed along the negative \( y \)-axis, \( z_{2} \) along the positive \( z \)-axis and \( z_{3} \) along the positive \( x \)-axis. Figure 8(d) shows the relative locations of the fixed hydrophobic residues in the protein instance, with lines between the topological neighbors.
Figure 8. This is an example of a run with the Simple Approximation Algorithm to a sequence of the length 111. (a) shows us the configuration of the stepped x- and y-superblocks $B_1'$ and $B_1''$ and the connecting superblocks $B_2'$ and $B_2''$. (b) shows the configuration of the x- and y-superblocks $B_3'$ and $B_3''$. (c) is the final conformation, and (d) is the hydrophobics that bonds with each and another in the final conformation, with lines between the topological neighbors.
2.4 Energy calculation

The way that we determine the quality of our algorithm, is by finding a lower bound and an upper bound for the energy output an algorithm gives. Usually the lower bound is the one we use to determine the quality of an algorithm, because the upper bound is the same for every algorithm, unless an algorithm proves otherwise. The way we measure the energy is, as mentioned in the introduction, by assigning a negative weight to the interactions between the H which bonds. So every time we have one interaction between the H which bonds, we add -1 to the total energy output for our algorithm.

For the upper bound, let $E(S)$ be the energy of the final conformation for a sequence $S$. Then the performance of the algorithm can be bounded as follows:

**Lemma 2.1.** Let $X' = X/2$. If $X' \geq 6$ then $E(S) \leq -3X' + 5$

**proof:** The algorithm selects a folding point using Subroutine 1. The folding point that have been selected splits the protein instance into two superblocks whose superblock structures have at least $X/2$ hydrophobic residues. The algorithm divides these two superblocks into several superblocks that are placed adjacent. The energy from the final conformation can be found by looking at the interactions between the blocks $B_1'$ and $B_2''$, $B_1'$ and $B_3''$, $B_1''$ and $B_2'$, and between $B_3'$ and $B_3''$, along with the interactions for $B_2'$ and $B_2''$. Since $N_x(B_1') + N_x(B_3'') \geq X' - 2$ and $N_y(B_1') + N_y(B_3'') \geq X' - 2$, there are $X' - 2$ interactions between $B_1'$ and $B_2''$, and $B_3'$ and $B_3''$ along the x-dimension. We know that $N_x(B_3'') = N_y(B_1') = (X' - 2)/2$ and because of the stepped block structure, each hydrophobic residue on the face of the stepped block structures for $B_1'$ and $B_2''$ is adjacent along the z-dimension to a hydrophobic residue on the face of the stepped block structure for $B_3'$ and $B_3''$. Therefore there are $2(X' - 2)/2$ interactions along the z-dimension. At the same time we know that $\min(N_x(B_3''), N_y(B_3'')) \geq X' - (X' - 2)/2$, so there are at least $2(X' - 3 - (X' - 2)/2)$ interactions along the y-dimension. The -2 is because that the two first hydrophobic residues, do not have an interaction in the y-dimension. It follows that

\[
E(S) \leq -4 - X' + 2 - 2(X' - 2)/2 - 2(X' - 3 - (X' - 2)/2) + 1
\]

which can be reduced to

\[
E(S) - 3X' + 5
\]

This is a lower bound for the Simple Approximation Algorithm.

Since we work in a 3D space, each of our hydrophobic residues can at maximum have interaction with four other hydrophobic residues and five others, if it is either the first or last character in the string. This means that

\[
OPT(S) = -4X - 2
\]
and therefore the quality of the Simple Approximation Algorithm is

\[ \frac{E(S)}{OPT(S)} = \frac{-3(X/2) + 5}{-4X - 2} \approx \frac{3}{8} \]

The quality gets closer to 3/8 the more our \( X \) increases.
Chapter 3

Ant Colony Optimization

The next algorithm is Ant Colony Optimization (ACO), which can solve the 3D HP protein folding problem in polynomial time. The algorithm mimics the behavior of a colony of ants, which are trying to find the shortest path between the nest and a food source. It does that by breaking the problem down into a shortest path problem, like the traveling salesman problem, and creates a colony of artificial ants, which are used to search for the shortest path. The version of the ACO algorithm that we have based on, can be found in the paper written by D.Chi, M. Till and A. Zomaya[5] and from the article written by Shmygelska and Hood[9]. The algorithm consist of three steps where in one of the steps, there are three substeps. Figure 9 shows the steps.

1. Initialize pheromone trails
2. While optimal solution not found do
   (a) Construct candidate solutions
   (b) Perform local search
   (c) Update pheromone trails
3. Return best found solution

Figure 9. The three steps for Ant Colony Optimization, where step two is where the complexity lies.

We know that biological ants form and maintain a path primarily through a trail of pheromone. The ants deposit a certain amount of pheromone when they travel around and probabilistically chooses the direction richest with pheromone. Because the shortest path between the food and the nest is reached faster, then the pheromone will also be stronger, which results in another ant to more likely travel that way.

The artificial ants from this algorithm has the same behaviour as the biological ants, so therefore the ants traverse a graph in order to construct a higher
quality solution. In the end of each iteration the quality of all the candidate solutions are evaluated based on a heuristic function. The best solution are then used to update the pheromone matrix.

3.1 Initializing the pheromone matrix

The first step of the algorithm is about setting everything up for the ants. We need to initialize a pheromone matrix which keeps tracks of the pheromone values, for all the directions at a given index in our protein sequence. To begin with we initialize the whole matrix with 1s, and for each iteration in step two we update the pheromone matrix by adding the pheromone values from the best candidate to a specific direction. The values which we add are depended on the quality of the ant, and therefore are proportional to the objective function value of a given solution.

3.2 Construction Phase: Pheromone and heuristic values

Step two (a) is the construction phase where we construct different candidate solutions, and compare these with each other, to determine the best candidate. The number of ants that are used is up to the user. As the number of ants increases, so does the time consumption and the quality of the algorithm. To determine how many ants to use, to get a good quality compared to the running time, will be discussed in section 5.

Each ant first randomly determines a starting point within the given protein sequence. From that point the protein sequence is folded in both directions, adding one h or p symbol at a time. The path in which the conformation is extended in each construction step, are determined using a pheromone value $\tau_{i,d}'$ and a heuristic value $\eta_{i,d}'$, where $i$ is the sequence position and $d \in W,N,E,S,U,D$ which is the direction of the folding at position $i$ to $i+1$ in a 3D plain. In this algorithm we have chosen to work with directional path instead of relative path, because it will be easier to explain and illustrate how the paths are constructed.

The heuristic values helps with the construction process to yield a better result, in terms of the maximum number of hydrophobic interactions. In the implemented algorithm the heuristic value is defined as $\eta' = h_{i+1,d} + 1$, where $h_{i+1,d}$ is the number of new hydrophobic interaction achieved by placing $s_{i+1}$ in direction $d$ from to $s_i$ and $s_{i+1}$, when folding forward. If the next character in the string is p, i.e. $s_{i+1} = p$, then the amino-acid cannot contribute to any hydrophobic interactions and therefore $h_{i+1,d} = 0$, for all the cases of $h_{i+1,d}$. Beside that, if we are not at the end of the string, the new hydrophobic interactions can only be at most 4, where at the end of the protein string there can be at most 5 hydrophobic interactions. So the $h_{i+1,d}$ values can be found by checking the seventeen neighbours for possible hydrophobic interactions, as illustrated in Figure 10.
Figure 10. The figure shows the seventeen neighbours (the red crosses), that has to be checked for possible hydrophobic interaction.

When folding from $i$ to $i-1$ we use $\tau'_{i,d}$ with the opposite direction, e.g. $\tau'_{i,W} = \tau'_{i,E}$, $\tau'_{i,E} = \tau'_{i,W}$, $\tau'_{i,N} = \tau'_{i,S}$, $\tau'_{i,S} = \tau'_{i,N}$, $\tau'_{i,U} = \tau'_{i,D}$ and $\tau'_{i,D} = \tau'_{i,U}$ and the same goes for the pheromone values $\eta'$. The way we then construct the conformation is to extend the folding from position $i$ to $i+1$ by placing $s_{i+1}$ to the right of $s_i$ and from position $i$ to $i-1$ by placing $s_{i-1}$ to the left of $s_i$. When we extend a partial conformation $s_k...s_i$ to $s_{i+1}$, the probabilities for the directions is determined by

$$p_{i,d} = \frac{[\tau_{i,d}]^\alpha [\eta_{i,d}]^\beta}{\sum_{e \in W,N,E,S,U,D} [\tau_{i,e}]^\alpha [\eta_{i,e}]^\beta}$$

and likewise for $s_i...s_m$ to $s_{i-1}$, it is handled with the opposite directions. $\alpha$ and $\beta$ is a parameter that is set by the user, which controls the influence of the pheromone and heuristic values. Where $\alpha \geq 0$ and $\beta \geq 1$.

While each ant extend their conformation, we can risk that the ants find themselves in a position where they cannot find any possible direction to go. It might have trapped itself, since we do not allow any point in our 3D plain to be visited twice, so therefore we use backtracking if no direction is found, else we select the direction with highest probability and move on to the next amino-acid. If there are more than one direction which have the highest probability, then it would be best to choose a random between those directions, as will be experimented with in section 5.
3.3 Local search

Like many other ACO algorithms, this algorithm uses local search to make the quality of the algorithm better. The way we have chosen is based on a long range mutation move which is made to try to avoid infeasible conformations. An outline of the local search procedure is shown in Figure 11.

\begin{verbatim}
While (termination condition not satisfied) do
    i = (Random index in our protein)
    c' = longRangeMove(c, i)
    if E(c') \leq E(c) then
        c = c'
    end
end
return(c)
\end{verbatim}

Figure 11. The local search method that try to make a better conformation by calling the long range move method and then comparing the energy between the old conformation and the new conformation.

We start by choosing a random position in our conformation, and from there we make a long range move and compare the energy of the new conformation with the energy of the regular conformation. If the energy is higher we return the new conformation else we return to the regular. The termination condition in this algorithm is the number of times it shall make the long range move on the current best conformation and is a parameter that the user can determine by themselves.

Long range move is a method which takes a conformation and a position, from the conformation, as the parameters. The direction of the residue which is selected, is randomly mutated and then adapts the rest of the chain by probabilistically changing the directions starting from the index given as the parameter. So for each residue in the rest of the chain, with a probability \( p \), where \( 0 \leq p \leq 1 \), its current direction is unchanged, if it is still feasible. Otherwise a new direction is chosen, where the probability for each direction \( d \) is proportional to the corresponding heuristic value. Figure 12 illustrates the probability of a direction to change.
Figure 12. Shows the probability of a residue that will change direction. $\hat{p}$ is the probability of the old direction and $P[d_i := \hat{d}]$ is the probability of choosing as the relative direction $d_i$ at the sequence position $i$.

While performing the long range move, we can risk as in the construction phase, that we end up a place where there are no feasible direction. So therefore we need to be able to perform backtracking. If the backtracking reach the residue which we started with, then we return the original conformation, since it is not possible to make a different conformation from the current chosen position.

### 3.4 Updating the pheromone trails

The last thing to do in each iteration is to update the pheromone trails. When all the ants have given a conformation, then the trails in the pheromone matrix are updated by the following

$$\tau_{i,d} = (1 - p)\tau_{i,d} + \frac{E(c)}{E^*}$$

$p$ is the pheromone persistence which determines how much the pheromone evaporates in each iteration of the algorithm. This value is determined by the user. The selected best ants updates the pheromone values by adding their quality, $E(c)/E^*$, to the respective place in the pheromone matrix, depending on which direction the ants took at the given position in our protein sequence. $E(c)$ is the energy of our conformation and $E^*$ is the known minimal energy for the given protein. This minimal energy can be found by for example using the 3/8-algorithm or another algorithm. If the value is unknown an approximation can be calculated by counting the number of $h$ residues in the sequence. This ensure us that the conformation which have a lesser quality compared to other, contributes lower amounts of pheromone to our pheromone matrix.
Chapter 4

1/4-Simple3D

The last algorithm is one we have made ourselves which is based on the 1/4-approximation algorithm for 2D HP model. Just like the 3/8-algorithm, we splits the string up in blocks. That which makes this algorithm different is the way we choose to split the string up in blocks and the construction of the branches. Instead of only one time delegating even H blocks and odd H blocks into two major blocks, we first make two major blocks which then again will be made into two major blocks each, by delegating the even H and odd H a second time.

The conformation that the algorithm generates, consist of four major blocks and three connections blocks. The four major blocks forms a rectangle, with a hydrophobic (H) at every second position in the 'skeleton' of the rectangle as shown in Figure 13.
Figure 13, The major blocks consist of black circles and white squares, forming a rectangle, where the connection blocks consist of the black squares, connecting the major blocks with each other. Each H (the black circles) bond with the H from its topological neighbors.

The algorithm is formally defined in Figure 14, which roughly shows the approach of the construction of the algorithm. Each step of the algorithm is linear, which means that the algorithm’s running time is in linear time. Basically the algorithm consist of two major steps, namely the partitioning of the string and the construction of the blocks and the connections.

1. Count occurrences of evens and odds H, so that, $N_e(S) = \{k \mid k \text{ is the index of an even H in the sequence } S\}$ and $N_o(S) = \{k \mid k \text{ is the index of an odd H in the sequence } S\}$, of a given string $S$.
2. Match the evens from the left with the odds from the right and match the odds from the left with the evens from the right and pick the largest list $M$, such that $M = (L_1, R_m), (L_1, R_{m-n}), ..., (L_n, R^{m-n})$, where $0 \leq n \leq m$ and $L_j \subset R_u$ for $j = 0, ..., n$, $u = m, m-1, ..., m-j$.
3. Split $M$ into $M'$ and $M''$, so that $|M''| + 2 > |M'| \geq M'$.
4. Repeat step 2 and 3 for $M'$ and $M''$
5. Construct $M'_L$, $M'_R$, $M''_L$, $M''_R$ and the three connection blocks, based on the scenarios from Figure 16.

Figure 14 an outline of 1/4-Simple3D.
We want to start with counting the occurrences of the even and odd Hs in the string, because we can use it to determine the minimum number of Hs that we at least can bond together. This number is used to split the string up in the two major blocks, which we can use to construct a conformation.

So to begin with we start by counting the occurrences of the even and odds H in the string. This gives us the lists \( N_e \) and \( N_o \), where

\[
N_e(S) = \{ k \mid k \text{ is the index of an even H in the sequence } S \}
\]

\[
N_o(S) = \{ k \mid k \text{ is the index of an odd H in the sequence } S \}
\]

For example, if a sequence \( S = hpuchenppphh \), then \( N_e(S) = [0, 4, 6, 10] \) and \( N_o(S) = [3, 9] \). We will use the two lists to analyze different possibilities of splits and compare those splits with each other before constructing the conformation.

Let us consider a list \( N_x = x_1, \ldots, x_n \). A split of \( N_x \) consist of the pair \((u, v)\), where \( u = x_1, \ldots, x_i \), \( v = x_i+1, \ldots, x_n \) for \( 0 < i < n \).

Before the split is used we want to find out which of the Even-Odd or Odd-Even sets that is largest. An Even-Odd set is where we matches the even Hs from the left with the odd Hs from the right. This is illustrated in Figure 15.

![Figure 15](image_url)

Figure 15 Shows the matches of even Hs and odd Hs for the Even-Odd matches

The string position of the even Hs has to be lower than the string position of the odd Hs and you can only match an even H once with an odd H. It is also required that you shall be able to connect the matches with a line without any overlapping, as shown in Figure 15. An Odd-Even sets is where we matches the odd Hs from the left with the even Hs from the right. So for the matches with Even-Odd we have the pairs

\[
M_{eo} = (L^0, R^m), (L^1, R^{m-1}), \ldots, (L^n, R^{m-n})
\]

Where for \( j = 0, \ldots, n \), \( L^j = N_e[j] \) and for \( u = m, m-1, \ldots, m-j \), \( R^u = N_o[u] \) and it is required that \( 0 \leq n \leq m \), where \( m = |N_o| \) and \( L^j < R^u \). The matches with Odd-Even, we have the pairs

\[
M_{oe} = (L^0, R^n), (L^1, R^{n-1}), \ldots, (L^m, R^{n-m})
\]

Where for \( u = 0, \ldots, m \), \( L^u = N_o[u] \) and for \( j = n, n-1, \ldots, n-u \), \( R^j = N_e[j] \) and it is required that \( 0 \leq m \leq n \), where \( n = |N_e| \) and \( L^u < R^j \).

Since we have chosen to work with four major blocks, we want to have at least two matches, so it is possible to assign elements to all the four blocks. If the length of the matches is below two, it is better to perform another algorithm.
e.g. 1/4-approximation algorithm for 2D protein folding, because the sequences cannot be split into four major blocks, without it will yield a result which will be the same or worse.

When the two first major blocks are found, we want to split them up into two major blocks each by applying the exact same method on the substring they represent, as we did with the whole string before. We do it because we want total of four major blocks, that will be used to as the skeleton of the rectangle shown in Figure 13.

We make total of three splits, the first one is splitting the set of the matches \( M \) into two equal length sets \( M' \) and \( M'' \), or so \( |M'| = |M''| + 1 \). With \( M' \) and \( M'' \), we do the exact same matching with even and odds as we did with \( M \), so we get a total of four blocks. Each of the four blocks can either be represented as an even block or as an odd block and its neighbour blocks, which are two of the three other blocks that \( M' \) and \( M'' \) was split into, represents the opposite. This gives us total of four different normal combinations and two extras as a special case, where in the special cases, it is better to make only one split between the evens and odds, instead of three splits. We still make a split to get the four blocks, but it will be in the evens on the one side and in the odds in the other side. The cases are illustrated in Figure 16.

![Figure 16](image)

Figure 16. (a), (b), (c) and (d) shows us the normal combinations of the division between the four blocks. (e) and (f) is a special case where we only make one split between the matches of even and odd, and then a split in the even part and a split in the odd part, so there still will be four major blocks.
When we split the string for the first time, there can be a scenario where one of the sides does not contain many evens or odds, compared to its opposite. In that case it is best not to split the two subsections further up between evens and odds, since this will yield a result with less matches than the matches we can get with one split, which is approximate around $|M'|/2$ or $|M''|/2$.

For example if we have $|M'| = 5$, we can delegate three matches to $M'_L$ and two matches to $M'_R$, where if we matched the even-odd and odd-even for $M'$ and then did a split, we might get $|M'_L| = 0$ and $|M'_R| = 3$.

The structure of the four major blocks located relative to each other and the connection between the major blocks is illustrated in Figure 17. It shows that an even block has two odd blocks as neighbours and likewise an odd block has two even blocks as neighbours. There is three connection blocks that connects all the four major blocks together, which is constructed differently depending on which scenario we get from the split. These constructions will be explained in the subsection 4.1.

Each of the four blocks contains a list of indices which tells us who will be their topological neighbors. The elemental of the first index in each list from the four blocks, forms a matching pair, so does the next index in the four blocks and so on.

4.1 Construction of the blocks and connections

Now when we have our matches, which tells us which H that should be bonded in our four major blocks, the only thing left is to construct the path. All our blocks combined after the splits consist of $S = S_1C_1S_2C_2S_3C_3S_4$, which is also in that order that we construct our path. Figure 17 also shows us which way the path go, beside how the four major blocks and three connection blocks are located relative to each other.
Figure 17. An example of an conformation made by 1/4-Simple3D. It shows the four major blocks $S_1, S_2, S_3$ and $S_4$, consisting of the black circles and white squares, and the three connection blocks $C_1$, $C_2$ and $C_3$, consisting of the black squares. It also shows how the blocks are constructed relative to each other, starting from $S_1$ and ending at $S_4$.

4.2 Major blocks

There are four major blocks where each have a direction they get constructed in the 3D plain. Two of them are in the northern part and two of them are in the southern part, where all four either are constructed along the positive $z$ axis or along the negative $z$ axis. When we construct the paths for the major blocks, we want every second position in the skeleton of the rectangle, that the major blocks forms, to be the index from our matches. This allows us to place the Hs from the indices beside each other in the 3D plain.

For example if $S_1 = [0, 4, 6]$, we want index 0, 4 and 6 of our string to be on every second position for the first major block as illustrated in Figure 17. If the distance between the indices are more than 2, then we have to make a branch. The distances between our indices from our matches, in a major block, is always an even number and is at least 2.

Depending on if the major block is the northern or southern part, we construct the path differently. For the southern part we construct the branches along the positive $y$ axis and for the northern part we construct the branches along the negative $y$ axis, which is shown in Figure 18. This means that the branches in the even and odd block, from the southern part, has a possibility
to bond some H, if it happens to be a branch at the same place. Likewise there is also a possibility for that in the even and odd block from the northern part.

Figure 18. It shows how the branches of the northern and southern part is constructed. Notice that the path of the branches is constructed differently compared to each other depending on if the major block is constructed along the positive z-axis or the negative z-axis.

### 4.3 Connection blocks

There are three connection blocks which is constructed differently depending on which scenario we get from Figure 16. There exist three different variations. These variations are shown in Figure 19.
Figure 19. An example of how the connection blocks can be constructed. (a) is a normal case connecting two major blocks of different parity. (b) has to be at least the length of 4 and is connecting two major blocks of the same parity. (c) same as (b) but has to be at least the length of 6.

(a) always have a length which is odd, where (b) and (c) always have a length which is even. For (a) the connection, connect an even block with an odd block and the case of (b) and (c), the connection connect two blocks of the same parity. The length of case (b) has to be at least 4 and the length of (c) has to be at least 6. The reason for that is because the occurrence for (c) can only happen if there is an occurrence of (b) in the same side of the rectangle and since there has to be a crossover between those two cases, then one of them must be at least 6 to be able to reach to the other block.

It can happen that the matches we get from the partitioning of the string, only have the length of 2 between the index of our matches, which results in that our connection block $|C| \leq 4$. If that happens we need to make a reduction in our matches. Let us say we have $S_1 = [0, 4, 6]$ and $S_2 = [8, 10, 12]$, then the connection block $C_1$ consist of those paths we can make between the last element in $S_1$ and the first element in $S_2$, which in this case $|C_1| = 2$. To make $|C_1| \geq 4$, we start by removing the last element in $S_1$ from the matches. In the case where we need $|C_1| \geq 6$ and removing the element from $S_1$ was not enough, we then remove the first element in $S_2$. This ensures us that $|C_1| \geq 6$. It also reduces our matches in the rectangle and therefore also the lower bound of the quality of this algorithm, which is explain in the next section.

When we construct our connection blocks, there exist three possible scenarios of how they need to be constructed relative to each other. The first possible scenario (a) and (b) from Figure 16, we have our three connections blocks $C_1$, $C_2$ and $C_3$. All three connection blocks is constructed after (a) from Figure 19. The second possible scenario (c) and (d) from Figure 16, $C_1$ and $C_3$ is constructed after (a) from Figure 19 and $C_2$ after (b). For the last possible scenario (e) and (f) from Figure 16, $C_1$ is structured after (a), $C_2$ after either (b) or (c) and $C_3$ after the other one compared to $C_2$. 
4.4 Energy calculation

To determine the quality of our algorithm, we need to find a lower bound and an upper bound. To calculate an upper bound, we look at how big our matching is. If we work with a sequence $S$, then for our first split there exists a $S_L$ and $S_R$, such that $S = S_LS_R$, where either 1) or 2) hold:

1) $|\text{Even}(S_L)| \geq 1/2|\text{Even}(S)|$ and $|\text{Odd}(S_R)| \geq 1/2|\text{Odd}(S)|$
2) $|\text{Odd}(S_L)| \geq 1/2|\text{Odd}(S)|$ and $|\text{Even}(S_R)| \geq 1/2|\text{Even}(S)|$

Since the construction plain we are working with is 3D, we know that the neighbours an H maximum can bond with is 4, except for the first and last character in the string which can bond with 5 parities. Therefore it follows that

**Lemma 4.1.** $OPT(S) \leq 4\min(|\text{Even}(S)|, |\text{Odd}(S)|) + 2$

**proof:** Since an even H only can bind with an odd H, we know that the minimum occurrences of H even and odd times four is an upper bound for the optimal solution.

To calculate a lower bound for the algorithm, we look at the elements in our major blocks. Let $S$ be a sequence, we split $S$ up in the four major and three connection blocks, so that $S = S_1C_1S_2C_2S_3C_3S_4$.

As shown in section 4.1, we can see that each match in $(S_1, S_2)$ and $(S_3, S_4)$, is bonded together, as long as there are still more elements to bond with. There need to be at least two of $S_1, S_2, S_3$ or $S_4$, that contains elements, then it is possible to make a bond between some H. For each time that there is still more elements in all four $S$, we can bond the Hs on their indices, so we get an energy bond of at least -4. This is because of how the construction the rectangle makes the H interacts, as was shown in Figure 13.

If three of the four $S$ contains elements, then the energy bond will be at least -2. and for two out of four, the energy bond will be at least -1, which can happen if we need to remove some of the elements, to expand the connection blocks.

So we can represent the energy of the conformation as $E(S)$, where $S$ is the sequence generated by the algorithm, then:

$E_1 = \min(|S_1|, |S_2|, |S_3|, |S_4|)$

$E_2 = \min(|S_1|, |S_2|, |S_3|, |S_4|) - E_1$, pick the second min, if $E_2 \leq 0$, then $E_2 = 0$

$E_3 = \min(|S_1|, |S_2|, |S_3|, |S_4|) - E_1 - E_2$, pick the third min, if $E_3 \leq 0$, then $E_3 = 0$

This will give us a lower bound for our $E(S)$, where
\[ E(S) \geq 4E_1 + 2E_2 + E_3 \quad (4.1) \]

The case where we use three splits for even and odd matches, both \( E_2 \) and \( E_1 \) can maximum be 1 and only one of them at a time. When we make a split, we split in the middle every time, which means that for every split of a given sequence \( S \), where \( S = S_L S_R \), then \(|S_L| - |S_R| = 0\) or \(|S_L| - |S_R| = 1\). This holds even after a reduction is caused by the connection, because as mentioned earlier we never remove an element from the same list twice without removing an element from the others too. Therefore the lower bound can roughly be determined as \( 4E_1 \).

Because we split three times, one length of a block is \( 1/4 \min(|Even(S)|, |Odd(S)|) \). Given equation 4.1, we know that each match in our block gives approximate 4 energy which implies that:

\[ E(S) \geq \min(|Even(S)|, |Odd(S)|) \]

Given Lemma 4.1, then

\[ E(S) \geq 1/4 \OPT(S) \]

Which means that this algorithm’s quality of a given sequence \( S \) is:

\[ 1/4 \OPT(S) \leq E(S) \leq \OPT(S) \]
Chapter 5

Experiments & Results

This section is about the comparison of the different algorithms, where we will look at how well they perform compared to each other. We will also look specific at ACO and make some experiments between the choice of the parameters that the user can choose. We will compare the algorithm with the others when some satisfying parameters are found, within a reasonable running time, since our ACO algorithm is running in polynomial time. The protein sequence that we have chosen to make experiments with, is a random generated protein string with the length of 350\[2\]. The protein string is converted to \( h \) and \( p \) based on if they are hydrophobic or hydrophilic as shown for the individual amino-acid in the table from the introduction section.

5.1 Choosing the right parameters for Ant Colony Optimization

As mentioned in the ACO section, the algorithm has different variables that the user can determine by themselves. Those are:

- **Iterations** : The number of iterations
- **Ants** : The number of ants to run with
- **Heu, Phe** : The importance of the heuristic and pheromone values
- **Scans** : The number of scans of local search
- **Mutation** : The probability of a given residue mutates in the long range move
- **Decay** : The percentages the pheromone that shall decay in each iteration

In the first experiments we will see how much importance each parameter has, where only one of the variables changes and the others are stationary. The ants starts at a random position and the probability of a given residue will mutates, in the long range move, is random. This makes the energy output from the algorithm non deterministic. Therefore to determine how well the algorithm performs, We choose to run the algorithm ten times with the same sequence and parameters, and then take the average of the results. The stationary parameters we choose to begin with is **Iterations** = 10, **Ants** = 20, **Heu** = 1,
$\textit{Phe} = 1$, $\textit{Scans} = 10$, $\textit{Mutation} = 50$ and $\textit{Decay} = 0.25$.

In the construction phase of the algorithm, where we calculates the probabilities of the directions, we can risk that more than one direction is the best probability. If this is so, we have two options. The first option is to just take the first direction in the list that has the best probability, and the second option is to choose a random direction between the directions that has the best probability. To determine which option to take, we make a comparison of the energy between these two options. The parameter we test it with is $\textit{Iterations}$, which is the number of times we will let our ants try to get a better conformation, with an updated pheromone matrix for each iteration.

Figure 20 shows us the results from the algorithm starting from 1 iteration to 15 iterations with option one and Figure 21 shows us the results with option two from iteration 1 to 15.

**Figure 20.** The results for option one, where the iteration parameter is the only one which gets changed.
As we can see there are a big difference in the quality between the two options. So therefore we have chosen to use option two instead of option one for the future experiments, to get the best possible quality.

The energy output for each different amount of iterations, does not change the quality of the algorithm much. This could be due to some of the other parameters have too much influence, or that the parameters we run with have too little influence. It can also be because we already run the algorithm with the same parameters ten times and then take the average, which could mean that these ten "iterations" is enough for the algorithm to approximate give the same result every time.

So we want to see if the stationary parameters is a good choice to determine the good parameters for our algorithm. The way we have done that, is by incrementing the parameters so it will tell us if the quality of the algorithm changes much, as the parameters changes. The table below shows us the results of the experiment. Each column represent the parameter we change and each row is the value of our $n$, which is used to determine the value of the parameter.
As we can see the quality of our algorithm does not change much as we change the parameters individually, and so it is hard from this point to choose some good parameters. The only part for some of the parameters, where we see a little difference, is when the parameters are close to zero. Knowing this we will choose some lower values of the stationary parameters, to see if we get a bigger difference in the quality when changing the parameters individually. We will also see if the values are different compared to the other stationary parameters. Our new stationary parameters will then be $\text{Iterations} = 3$, $\text{Ants} = 5$, $\text{Heu} = 1$, $\text{Phe} = 1$, $\text{Scans} = 3$, $\text{Mutation} = 20$ and $\text{Decay} = 0.25$. The table in Figure 23 below, shows us the results with the new parameters.

```
<table>
<thead>
<tr>
<th>n</th>
<th>Iterations (n)</th>
<th>Ants (n)</th>
<th>Heuristic (n/10)</th>
<th>Pheromone (n/10 + 1)</th>
<th>Scans (n)</th>
<th>Mutation (n * 10)</th>
<th>Decay (n/10 - 0.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-76.7</td>
<td>-66.0</td>
<td>-79.7</td>
<td>-75.8</td>
<td>-74.0</td>
<td>-75.9</td>
<td>-75.5</td>
</tr>
<tr>
<td>2</td>
<td>-75.7</td>
<td>-65.9</td>
<td>-77.1</td>
<td>-80.3</td>
<td>-75.5</td>
<td>-75.4</td>
<td>-77.0</td>
</tr>
<tr>
<td>3</td>
<td>-75.9</td>
<td>-70.8</td>
<td>-78.0</td>
<td>-78.6</td>
<td>-78.3</td>
<td>-75.1</td>
<td>-79.3</td>
</tr>
<tr>
<td>4</td>
<td>-78.6</td>
<td>-70.7</td>
<td>-76.2</td>
<td>-75.8</td>
<td>-78.6</td>
<td>-78.4</td>
<td>-74.4</td>
</tr>
<tr>
<td>5</td>
<td>-77.3</td>
<td>-72.1</td>
<td>-75.9</td>
<td>-76.7</td>
<td>-76.6</td>
<td>-75.9</td>
<td>-77.9</td>
</tr>
<tr>
<td>6</td>
<td>-78.0</td>
<td>-74.2</td>
<td>-77.2</td>
<td>-77.3</td>
<td>-77.4</td>
<td>-75.0</td>
<td>-78.2</td>
</tr>
<tr>
<td>7</td>
<td>-74.6</td>
<td>-74.2</td>
<td>-76.8</td>
<td>-75.7</td>
<td>-74.0</td>
<td>-75.5</td>
<td>-75.8</td>
</tr>
<tr>
<td>8</td>
<td>-76.7</td>
<td>-71.4</td>
<td>-77.6</td>
<td>-75.5</td>
<td>-76.0</td>
<td>-76.9</td>
<td>-79.4</td>
</tr>
<tr>
<td>9</td>
<td>-76.7</td>
<td>-72.8</td>
<td>-76.7</td>
<td>-74.1</td>
<td>-76.3</td>
<td>-78.0</td>
<td>-76.2</td>
</tr>
<tr>
<td>10</td>
<td>-77.1</td>
<td>-73.8</td>
<td>-74.7</td>
<td>-76.5</td>
<td>-77.2</td>
<td>-73.6</td>
<td>-77.9</td>
</tr>
</tbody>
</table>
```
Figure 23. The results of the output from ACO with low stationary parameters, where we only increment one of the parameters at a time. Each parameters is affected differently as \( n \) increases, which is indicated in the parentheses under the names.

Again here the energy output from the different runs does not change much compared to each other. On the other hand we can see that the overall performance is better with the higher value parameters compared to the lower value parameters.

Now we want to see if the parameters we run with have too little influence, so we want to do a some runs where the intervals between the individually parameters is high. And then we choose some new stationary parameters out of the results which can guide us further into choosing some good parameters for ACO. We will not increase the parameters for the \textit{Iterations} and the \textit{Scans} too much, because the local search part has a worst case scenario running time on \( O(n^k) \), where \( k = |\text{possible directions}| \) and the iterations also increases the amount of total scans we do. This means that the time it will take the algorithm to be finished will not be inside the scope of the time given for this project, as the parameters increases. Therefore we will keep \textit{Iterations} = 3 and \textit{Scans} = 3, for now. \textit{Mutation} can at maximum be 100, which means that there is a limit to how much we can increase this. The same goes for \textit{Decay}, which can at maximum be 1. Since they did not have a noticeable impact on the quality, we choose \textit{Mutation} and \textit{Decay} to be the values we have worked with so far as the stationary parameters. This means that \( \textit{Mutation} = 50 \) and \( \textit{Decay} = 0.25 \), which are the only two parameters that we will not change.

<table>
<thead>
<tr>
<th>n</th>
<th>Iterations (n)</th>
<th>Ants (n)</th>
<th>Heuristic (n/10)</th>
<th>Pheromone (n/10 + 1)</th>
<th>Scans (n)</th>
<th>Mutation (n * 10)</th>
<th>Decay (n/10 - 0.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-75.2</td>
<td>-59.8</td>
<td>-68.7</td>
<td>-73.0</td>
<td>-71.9</td>
<td>-70.9</td>
<td>-70.7</td>
</tr>
<tr>
<td>2</td>
<td>-73.5</td>
<td>-68.1</td>
<td>-69.1</td>
<td>-70.7</td>
<td>-70.9</td>
<td>-71.3</td>
<td>-71.9</td>
</tr>
<tr>
<td>3</td>
<td>-72.0</td>
<td>-68.4</td>
<td>-68.3</td>
<td>-71.4</td>
<td>-71.5</td>
<td>-72.0</td>
<td>-73.1</td>
</tr>
<tr>
<td>4</td>
<td>-69.3</td>
<td>-70.0</td>
<td>-69.8</td>
<td>-68.3</td>
<td>-70.7</td>
<td>-72.4</td>
<td>-67.9</td>
</tr>
<tr>
<td>5</td>
<td>-70.3</td>
<td>-68.3</td>
<td>-75.7</td>
<td>-69.2</td>
<td>-70.6</td>
<td>-71.4</td>
<td>-71.9</td>
</tr>
<tr>
<td>6</td>
<td>-70.1</td>
<td>-72.3</td>
<td>-69.7</td>
<td>-71.5</td>
<td>-68.9</td>
<td>-71.4</td>
<td>-72.0</td>
</tr>
<tr>
<td>7</td>
<td>-72.5</td>
<td>-74.9</td>
<td>-73.8</td>
<td>-70.0</td>
<td>-71.0</td>
<td>-74.0</td>
<td>-71.3</td>
</tr>
<tr>
<td>8</td>
<td>-73.5</td>
<td>-73.3</td>
<td>-69.8</td>
<td>-72.7</td>
<td>-69.0</td>
<td>-71.4</td>
<td>-69.2</td>
</tr>
<tr>
<td>9</td>
<td>-69.6</td>
<td>-73.5</td>
<td>-70.6</td>
<td>-70.1</td>
<td>-71.4</td>
<td>-70.3</td>
<td>-73.7</td>
</tr>
<tr>
<td>10</td>
<td>-72.6</td>
<td>-75.5</td>
<td>-71.5</td>
<td>-72.3</td>
<td>-74.6</td>
<td>-72.2</td>
<td>-72.4</td>
</tr>
</tbody>
</table>
The table in Figure 24 shows us the results where we increment the Iterations, Ants, Heu, Phe and Scans values much for each run. The stationary parameters are still Iterations = 3, Ants = 5, Heu = 1, Phe = 1, Scans = 3, Mutation = 50 and Decay = 0.25, for those parameters we are not making the experiment with.

<table>
<thead>
<tr>
<th>n</th>
<th>Iteration (n * 4 + 10)</th>
<th>Ants (n * 20)</th>
<th>Heuristic (n)</th>
<th>Pheromone (n)</th>
<th>Scans (n*4 + 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-70.2</td>
<td>-75.3</td>
<td>-73.2</td>
<td>-73.5</td>
<td>-69.5</td>
</tr>
<tr>
<td>2</td>
<td>-69.5</td>
<td>-79.1</td>
<td>-70.7</td>
<td>-68.0</td>
<td>-72.9</td>
</tr>
<tr>
<td>3</td>
<td>-71.5</td>
<td>-81.1</td>
<td>-76.1</td>
<td>-71.7</td>
<td>-71.0</td>
</tr>
<tr>
<td>4</td>
<td>-69.6</td>
<td>-82.9</td>
<td>-72.6</td>
<td>-71.0</td>
<td>-74.0</td>
</tr>
<tr>
<td>5</td>
<td>-71.3</td>
<td>-83.1</td>
<td>-70.4</td>
<td>-72.8</td>
<td>-74.5</td>
</tr>
<tr>
<td>6</td>
<td>-71.9</td>
<td>-80.8</td>
<td>-69.9</td>
<td>-73.3</td>
<td>-73.4</td>
</tr>
<tr>
<td>7</td>
<td>-73.3</td>
<td>-82.6</td>
<td>-73.1</td>
<td>-71.1</td>
<td>-71.0</td>
</tr>
<tr>
<td>8</td>
<td>-72.6</td>
<td>-82.7</td>
<td>-76.0</td>
<td>-71.2</td>
<td>-71.7</td>
</tr>
<tr>
<td>9</td>
<td>-69.7</td>
<td>-81.1</td>
<td>-69.8</td>
<td>-71.7</td>
<td>-69.7</td>
</tr>
<tr>
<td>10</td>
<td>-71.3</td>
<td>-84.2</td>
<td>-76.1</td>
<td>-70.1</td>
<td>-70.7</td>
</tr>
</tbody>
</table>

Figure 24. The results of the output from ACO with low stationary parameters, where we increase one of the parameters much for each iteration. Each parameters is affected differently as n increases, which is indicated in the parentheses under the names.

As we can see from the results, then the quality of the algorithm increases as the parameter of the Ants increases, whereas the others does not increase the overall quality of the algorithm.

With the experiments we have ran so far, we can see that the overall quality increases as we increase all the parameters at once, beside Phe, Heu and Decay, which have had the same stationary parameters. Mutation have been changed, but since it has a limit, then we want it to be stationary for the next experiment. This means that the parameters that we mostly are interested in to change is Iterations, Ants and Scans.

Last experiment we need to perform, before we can determine which values our parameters shall be, is to increase Iterations, Ants and Scans at the same time. The stationary parameters this time is Heu = 1, Phe = 1, Mutation = 50 and Decay = 0.25. We choose to increment n with 1 every...
run, where the parameter for $\text{Iterations} = n, \text{Ants} = 10n$ and $\text{Scans} = n/2$ (rounded down). Figure 25 shows us the energy results as $n$ increases.

![Energy Graph]

Figure 25. It shows us the results of the output from ACO, when we increment the parameters $\text{Iteration}$, $\text{Ants}$ and $\text{Scans}$ simultaneously.

As we can see here, then the quality of our algorithm increases to a certain point, where it does not matter if we increases the parameters more. Therefore with the current experiments, we choose our parameters to be the values where the quality of the algorithm is highest. By this we can conclude that the parameters of the Ant Colony Optimization algorithm, that we will use, for the experiments of the comparison with the other algorithms is $\text{Iterations} = 14$, $\text{Ants} = 140$, $\text{Heu} = 1$, $\text{Phe} = 1$, $\text{Scans} = 7$, $\text{Mutation} = 50$ and $\text{Decay} = 0.25$.

So to see how good our parameters is, then we will run them on ten sequences we already know the optimal solution on and compare our results with the optimal solution. The strings and their optimal solution is taken from Shmygelska and Hood[9] and the comparison can be seen in Figure 26 below.
Figure 26. Shows us the ten protein sequences we know the optimal score on and the output that ACO gives.

\[ E^* \] is the optimal solution for the protein sequence. As we can see our ACO gets around 2/3 of the optimal solution, which is satisfied for us.

### 5.2 ACO, Simple Approximation Algorithm & 1/4-Simple3D

The other two algorithms do not have any parameters that the user needs to set. The algorithms are also deterministic, which means that we cannot improve the algorithm’s quality without changing the algorithm itself.

The sequences we choose for the experiments is random generated proteins of the length 500.[2] The results of the algorithms is shown in the table in Figure 27 and the graph in Figure 28 sets out the results so it is easier to compare them with each other.
<table>
<thead>
<tr>
<th>Seq 1</th>
<th>ACO</th>
<th>Simple approximation</th>
<th>1/4-Simple3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq 2</td>
<td>-119</td>
<td>-118</td>
<td>-129</td>
</tr>
<tr>
<td>Seq 3</td>
<td>-126</td>
<td>-126</td>
<td>-130</td>
</tr>
<tr>
<td>Seq 4</td>
<td>-122</td>
<td>-65</td>
<td>-130</td>
</tr>
<tr>
<td>Seq 5</td>
<td>-139</td>
<td>-55</td>
<td>-140</td>
</tr>
<tr>
<td>Seq 6</td>
<td>-94</td>
<td>-127</td>
<td>-113</td>
</tr>
<tr>
<td>Seq 7</td>
<td>-105</td>
<td>-55</td>
<td>-120</td>
</tr>
<tr>
<td>Seq 8</td>
<td>-126</td>
<td>-119</td>
<td>-126</td>
</tr>
<tr>
<td>Seq 9</td>
<td>-117</td>
<td>-143</td>
<td>-114</td>
</tr>
<tr>
<td>Seq 10</td>
<td>-135</td>
<td>-130</td>
<td>-134</td>
</tr>
<tr>
<td>Seq 11</td>
<td>-106</td>
<td>-115</td>
<td>-117</td>
</tr>
<tr>
<td>Seq 12</td>
<td>-100</td>
<td>-57</td>
<td>-119</td>
</tr>
<tr>
<td>Seq 13</td>
<td>-107</td>
<td>-130</td>
<td>-127</td>
</tr>
<tr>
<td>Seq 14</td>
<td>-114</td>
<td>-74</td>
<td>-129</td>
</tr>
<tr>
<td>Seq 15</td>
<td>-116</td>
<td>-58</td>
<td>-126</td>
</tr>
<tr>
<td>Seq 16</td>
<td>-112</td>
<td>-75</td>
<td>-119</td>
</tr>
<tr>
<td>Seq 17</td>
<td>-126</td>
<td>-128</td>
<td>-129</td>
</tr>
<tr>
<td>Seq 18</td>
<td>-123</td>
<td>-81</td>
<td>-132</td>
</tr>
<tr>
<td>Seq 19</td>
<td>-103</td>
<td>-55</td>
<td>-115</td>
</tr>
<tr>
<td>Seq 20</td>
<td>-146</td>
<td>-76</td>
<td>-144</td>
</tr>
<tr>
<td>Seq 21</td>
<td>-122</td>
<td>-135</td>
<td>-142</td>
</tr>
<tr>
<td>Seq 22</td>
<td>-104</td>
<td>-133</td>
<td>-122</td>
</tr>
<tr>
<td>Seq 23</td>
<td>-112</td>
<td>-54</td>
<td>-119</td>
</tr>
<tr>
<td>Seq 24</td>
<td>-106</td>
<td>-106</td>
<td>-130</td>
</tr>
<tr>
<td>Seq 25</td>
<td>-121</td>
<td>-111</td>
<td>-127</td>
</tr>
<tr>
<td>Seq 26</td>
<td>-115</td>
<td>-126</td>
<td>-125</td>
</tr>
<tr>
<td>Seq 27</td>
<td>-129</td>
<td>-73</td>
<td>-139</td>
</tr>
<tr>
<td>Seq 28</td>
<td>-131</td>
<td>-133</td>
<td>-126</td>
</tr>
<tr>
<td>Seq 29</td>
<td>-105</td>
<td>-68</td>
<td>-119</td>
</tr>
<tr>
<td>Seq 30</td>
<td>-131</td>
<td>-67</td>
<td>-127</td>
</tr>
</tbody>
</table>
Figure 27. Shows us the result from ACO, SAA and S3D of the 30 protein sequences with a length of 500 each.

Figure 28. Shows of the same results as from the table before, but with a better overview for comparison.

The results shows that there is not one of the three algorithms that gives best results in all the sequences. The ACO algorithm and 1/4-Simple3D output mostly around the same results, with the exception of a few. The Simple Approximation Algorithm sometimes output around the same results as the other
two and sometimes it gives around half the energy. The results of the experiments will be discussed more about in the next section.
Chapter 6

Discussion of the results

This discussion section contains a more detailed explanation of the experiments chosen to determine the parameters for ACO and the energy comparison of the three algorithms. To begin with the string used for the experiments was made by a random protein string generator. This could have been any homemade string, or a protein string obtained from a protein databank. The reason for that the random protein string generator was chosen, was because it was the most easy and fastest way to generate a lot of test data. And at the same time, it has a low risk of getting a string, which results in the ‘perfect’ data. Perfect data for an algorithm is test data where the algorithm outputs the best possible outcome. This can be a problem if an algorithm only receive perfect data when testing. This causes the test results to be inaccurate.

When we look at the experiments for ACO, then the intentions behind the choices was first to find out whenever any of the parameters for the algorithm had a significant impact on the results compared to the other parameters. The choices of the first stationary parameters is based on common sense and the parameters that Shmygelska and Hoos[9] did choose for their experiments for the 3D HP protein folding. The primary goal was not to find the best parameters for our algorithm, but some parameters which gave satisfying results from our experiments, without the time consumption was too high. If we were to find the perfect parameters, then it would require more time than we would have available in this project. The perfect parameters means that, if the parameters were further changed, then it will only result in a worse output from our algorithm.

With the first stationary parameters as a starting point, we got some results as was shown in Figure 22. These results showed us the output when we increased the parameters individually. This would tell us if any of the parameters had any significant influence, when they got changed. There was also the possibility to change two parameters at a time, since some of the parameters are close related to each other, compared to where in the code they have influence. But the more simple approach were chosen, since we only wanted parameters that would make satisfying results. Satisfied results is in this case where the improvement of the quality does not increase much compared to the running time.
Since the results did not deviate much, the most common sense was to, either try to change two parameters at a time or change the stationary parameters. The stationary parameters was changed to small numbers, to ensure that none of the stationary parameters was dominating too much. Since the deviation of the output was roughly the same, the intervals between the parameters might have been too small, and therefore an experiment was made with a higher interval between the parameters, for those that did not have an upper bound.

Again here Ants was the only parameters which looked like it had a directly impact on the output. From this point a question rose, that if Ants was the only parameters that was affecting the quality of the algorithm? As the way the parameters is affecting the Ants, it would make sense that the requirements of the ants would be less as the Iterations and Scans values increased, since Iterations contributes to how many times the ants may make a new conformation and Scans contributes to how many times we try to find a better conformation based on the conformation an ant has given. So therefore an experiment was made to confirm the influence of Iterations and Scans, which would lead us to the choice of our parameters. Since we just wanted some satisfying parameters and not the perfect ones, it would make sense to pick the ones which would yield the highest output.

At last the quality of our parameters was tested against some protein sequences that we already knew the optimal energy on. As mentioned in the last section then the outputs from ACO was around 2/3 of the optimal energy output for each sequence, which resulted in satisfying parameters for us. With more experimenting we would have achieved a closer output towards the 'perfect' parameters, which would lie with an energy output between the ones we got and the optimal ones. But we chose to stop the experimenting here because we had an idea for how far we might be from an optimal solution, and another reason was that the time consumption was starting to be high compared to what was worthwhile. Maybe the improvement of the parameters would not result in a much better output, and thereby it will then require an improvement in the algorithm instead, for getting closer to the optimal solution. To achieve a better accurate parameters choosing, we could have made more experiments regarding to how the Heu, Phe and Decay values would influence the conformations made from the ants and how important their relation was to each other. Another choice that would improve the accuracy of the parameters, is to run the algorithm more than 10 times with the same parameter and then take the average.

### 6.1 Comparison of the three algorithm

An experiment with the running time between the algorithms could have been a possibility, but it would not matter much since both the Simple Approximation Algorithm (SSA) and 1/4-Simple3D (S3D) runs in linear time and Ant Colony Optimization (ACO) runs in polynomial time. Therefore SSA and S3D will always be a lot faster than ACO, if ACO wants to reach around the same quality as the other two algorithms. The question lies in how long do we want
to wait before it is no longer worthwhile. It all depends on how well optimized ACO is programmed and the parameters which was chosen.

As we can see from the results between the three algorithms then as mentioned earlier, there is not one of the algorithms which perform best in all the sequences. The results for ACO can possible be improved earlier as explained. SAA have a few fluctuation in the results compared to ACO and S3D. This could be due to different reasons.

For the first then SAA and S3D construct the blocks differently. SSA consist of six blocks, let us say $B = B_1B_2B_3B_4B_5B_6$, and is constructed so $B_1, B_2$ and $B_3$ is matched at either even Hs or odd Hs, and $B_4, B_5$ and $B_6$ is matched at the opposite Hs of $B_1, B_2$ and $B_3$. S3D consist of 7 blocks, let us say $B = B_1C_1B_2C_2B_3C_4B_5$, but for some cases $B_1$ and $B_3$ is matched at either even Hs or odd Hs and $B_2$ and $B_4$ is matched at the opposite Hs of $B_1$ and $B_3$. Because of this the results can differ greatly. Sometimes SSA yield a better result and sometimes S3D does.

Another reason for that the S3D yields better results, according to the test results, might be the way the branches of the major blocks is constructed and how they lie compared to each other.

S3D is constructed so the branches from the southern part lies up against each other and the branches form the northern part lies up against each other as was illustrated in Figure 18 in the paper written by Hart and Istrail[7]. They do not specify how the branches should be constructed, but it would make sense to expand the branch to the sides, so it does not collide with the others. This result in a branch to only have one side again a branch from the other side, as can be seen in Figure 8(c). This also means that the one side from a branch, can be against two branches from another superblock, but it require that the other superblock have two branches to be connected with, where in S3D it only require one branch to be connected with.
Chapter 7

Conclusion & Future work

7.1 Conclusion

When we sum it all up, then we reached the goals for this thesis, which was to introduce and present an implementation of an algorithm for the folding of protein in a 3D HP model and then compare the quality of the algorithm with other existing algorithms. With the examination of the algorithms and based on the results, we can conclude that 1/4-Simple3D performs overall better with random generated sequences, even though the lower bound for the Simple Approximation Algorithm is better. On the other side it is hard to conclude a lower bound for the ACO, which makes the choices of the parameters for ACO important, if we want to reach a quality better than the other two algorithms.

7.2 Future work

The development stage of the HP protein folding problem still have some way to go, and there is constantly being worked on improving the qualities of the algorithms. There are many different directions that we can take to improve the quality. We can try to improve the Ant Colony Optimization algorithm, which mostly lies in finding the correct parameters through a long series of experiments and validations. Beside that we can develop a better local search part and make a more complex version, as long as it improve the quality of the algorithm and complies with a maximum running time of polynomial time, if not better.

A higher step of complexity can be taken, by starting to include angles for how much the directions leans compared to their previous direction in the 3D plain. This require that the current way we construct our path, and measure the energy, needs to be changed, because we no longer can have a fix 90 or 180 degree for each direction we make. There could be a project of experiments, that would allow us to determine how much the directions should leans by looking at the values of how much the amino-acids deviates from water, since the amino-acid that deviates from water tends to chunk together. When looking at S3D, then it can be improved by applying SSA stepped construction method to it. This means that S3D will be constructed as a stepped block instead as a
rectangle and therefore will it also yield a better result, because the lower bound gets increased. The division of the blocks should remain the same, which mean that the connections also should remain the same.
Bibliography

[1] Also the term for hp, which we will represent it as for the rest on this project.


