Aarhus University

Algorithms for Exact and Approximated Protein Structure Prediction in the 3D HP Model

Masters Thesis

Martin Bjerrum Henriksen

supervised by
Christian Noergaard Storm

March 6, 2019

Abstract

Predicting protein structures from their nucleic acid sequence is a long term goal pursued in bioinformatics. The benefits of achieving this goal are potentially enormous. The HP model, introduced by Lau and Dill, is a very simple abstraction of the problem that has been extensively studied. [9] Predicting structures within this model amounts to maximizing a measure called *score*. Nayak, Sinclair and Zwick showed that this is an NP-complete problem. [11] In this paper I review two structure prediction algorithms.

First, I give an alternative explanation, called S38, of an approximation algorithm by Hart and Istrail, 1995. [5] It provides solutions with at least $\frac{3}{8}$ of the optimal score. My explanation is intended to be more accessible and more directly translatable into fast and simple code. I also analyze the algorithms running time and provide a proof of its approximation ratio, which I was not able to find in other papers. I also prove that the method can be extended to approximate the HP problem in D dimensions with approximation ratio $\frac{2D-3}{4D-4}$. I then test and review S38, and provide full pseudocode for its implementation.

Secondly, I implement the *threading algorithm* which is part of a score maximizing method by Backofen and Will called CPSP. [2] I give an introduction to Constraint Programming which is central to the algorithm, and explain how it can be used to implement CPSP. Here, I also discuss subtleties of implementation that were left out of the original papers. I then test its running time against a naive search algorithm.

In conclusion, the approximation algorithm simply does not make realistic folds at all. It can still be useful for introducing students to the `HP`-problem, and illustrating a problem with bipartite lattices. This is what my S38 explanation tries to support.

With CPSP, it is restrictively time consuming to fold long sequences. It still appears to be a more promising method than S38 due to finding solutions that are optimal within the model, as well as not being specific to the cubic lattice. If applied to the right models, it might increasingly begin to predict structures that resemble real folds.

Contents

Section 1

Introduction

The microscopic processes of life are almost universally governed by chains of amino acids known as proteins. Our antibodies and the keratin in our fingernails are made of protein. They are responsible for DNA replication, chemical messaging via hormones, transporting oxygen and nutrients, and enzymes that catalyze thousands of vital functions of the body.

The function of a protein depends entirely on the physical conformation of its constituent amino acids. In their natural environment, proteins will automatically assume the conformation that carries out their function. We call this conformation the *native state* of the protein.

The *Folding Funnel Hypothesis* states that the native state of a protein is the conformation that minimizes its free energy. It is also recognized that the native states of protein are determined entirely by their amino acid sequences. It is therefore an interesting problem to predict the conformation with minimal free energy, given a sequence of amino acids. This is known as *Protein Structure Prediction*. The prospect of PSP is to specifically design proteins with desirable functions, which could have widespread utility in medicine, agriculture, energy production and other exciting areas.

To perform PSP, we make mathematical models of the proteins, their environment, and how they fold in their environment. The algorithms studied in this project are concerned with the *HP model with cubic lattice*. The *HP model* is the model of proteins and how they fold, while the *cubic lattice* models the environment they fold in. We introduce the models in Section 1.1.

One of the challenges of protein structure prediction is the sheer complexity of the problem. PSP is NP-complete in the HP model [11], even though it is one of the most simplifying models. This means it is strongly conjectured that a globally optimal solution can only be found by enumeration of an exponential number of solutions.

Given the hardness of the problem, we are forced to choose between two types of algorithms that

each have an undesirable property: One is *exact algorithms*, which enumerate the solution space with exponential running times. This can be restrictive in terms of what problem instances can feasibly be solved. The other type is *approximating algorithms*, that make compromises w.r.t. the quality of the solution, but run in polynomial time. Approximation algorithms prove a lower bound on the solution relative to the optimal, and the ratio between them is called the *approximation ratio*.

This project is concerned with two algorithms [5, 2] for PSP in the HP model, one being exact and the other, approximating. It contributes by:

- Providing more detailed and accessible explanations of the algorithms, including prerequisite programming techniques.
- Formally analyzing the approximation algorithm and proving its guarantees. The analysis and its proofs are results of this project.
- Implementing the algorithms and running experiments to reproduce the results.
- Giving detailed explanations of how both algorithms can be implemented in code, with examples of challenges that can appear, and how to solve them.

The algorithms are covered in Section 2 and 3. At the end of each section we discuss the merits, problems and prospects of the algorithms. Section 4 covers the experiments and their results. A summary and conclusion is given in Section 5.

1.1 The HP model

This section introduces the HP model for protein structure prediction.

We know that the free energy of a conformation to a relevant degree is determined by the intermolecular electric forces that allows polar molecules (such as water) to mix with each other, while separating them from non-polar, or *hydrophobic*, ones. In proteins, hydrophobic amino acids are observed to clump together to minimize contact with water, forming what is known as the proteins *hydrophobic core*. This is what gives rise to the **H**ydrophobic/**P**olar, or HP model, as introduced by Lau and Dill, 1995 [9].

The HP model is a type of *lattice model*, meaning that a proteins tertiary structure is modeled as a set of 0-size points on a mathematical graph, called a lattice. The vertices of the lattice can be occupied by at most one amino acid, and amino acids that are adjacent in the chain must also be neighbors in the lattice. The type of lattice used with the HP model is a parameter that can be changed independently of the rest of the model. In this project we will use the cubic lattice, which is the most commonly studied lattice for PSP. The cubic lattice is depicted in Figure 1.1

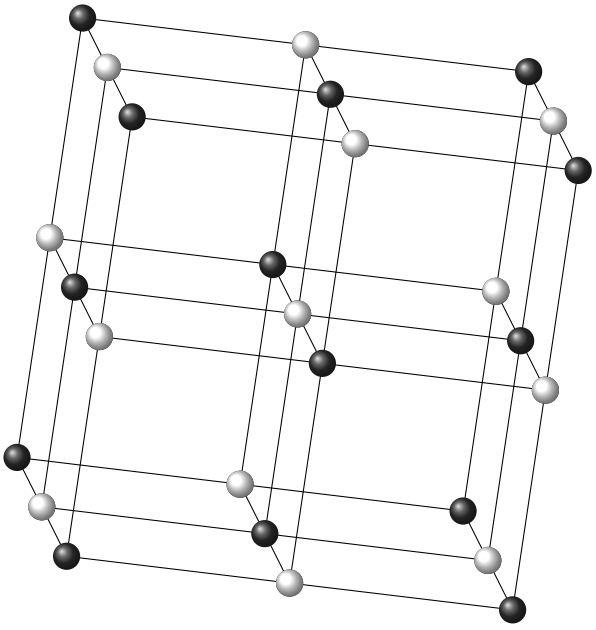


Figure 1.1: The cubic lattice. Each vertex is connected to six other vertices. Monomers are placed on the vertices of the lattice. Two monomers form a *contact* if their vertices are connected. The lattice can be 2-colored in such a way that no vertices of the same color are connected. This means that monomers can only form connections if they have opposite parities in the sequence.

When a conformation is represented on the lattice, we model its free energy based on the compactness of the hydrophobic core. Each amino acid is classified as either *hydrophobic* or *polar*. we then assign to that structure an energy of $-$`HHContacts` where `HHContacts` is the number of "contacts" between hydrophobic amino acids. We say that monomers form "contacts" if they are not successive in the sequence, but their vertices are connected in the lattice. Figure 1.2 shows a conformation on a cubic lattice with a free energy of -11. In Figure 1.2 and throughout the report, H-monomers are represented by orange while P-monomers are represented by transparent blue, since they are water soluble.

Structure prediction in the HP model amounts to finding folds that minimize the free energy, or equivalently, maximizes the number of `HH`-connections, also called *score*.

We will represent conformations as a string of absolute directional symbols: U, D, L, R, F, B.

Parity of the Cubic Lattice An important property of the cubic lattice is that it can be "checkerboard-colored", so that colors of adjacent nodes are heterogeneous, see Figure 1.1. This means that nodes of equal parity in a sequence can not be neighbors when placed in the lattice. We can derive an upper bound on the score from the fact that "even" H can only be paired with "odd" H and vice versa. Let H_{even} is the number of *even*-indexed H-monomers in the sequence,

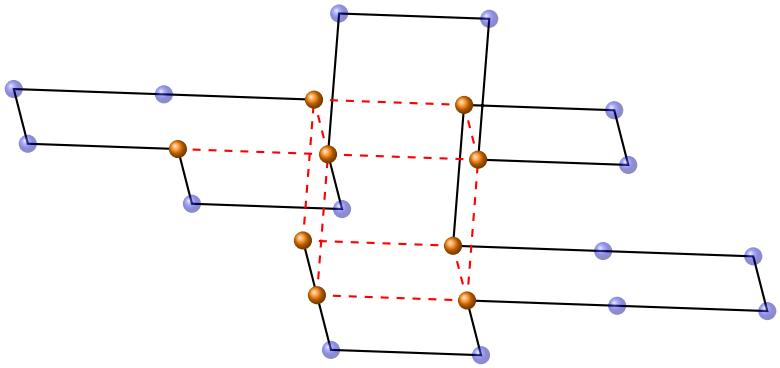


Figure 1.2: An optimal conformation with a score of 11, Found using CPSP. Orange represents H-monomers, blue represents P-monomers, and HH-connections are marked with dashed red lines. The sequence in the example is `HHPPHPPPPHHPPHPPHPPHPPPH` and the fold is `BBRFRRFLLURBLULDBLFLFRR`.

and H_{odd} is the number of odd-indexed H. We then get the bound in (1.1).

$$2 \cdot (D-1) \cdot \min(H_{even}, H_{odd}) + 2 \tag{1.1}$$

This is because that number of contacts saturates all `H` of the scarcest parity, so that no more HH-contacts are possible. There are $\min(H_{even}, H_{odd})$ of the scarcest `H`. Each of these can make $2(D-1)$ connections, because they have $2 \cdot D$ neighbors in the lattice, of which 2 must be be occupied by amino acids that are adjacent i the sequence. Figure 1.1 provides an example in three dimensions: Each vertex has six neighbors in the lattice, meaning that the H-monomers can form up to 4 connections each. The $\mathbf{+2}$ term comes from the fact that the first and last element of the sequence has only one adjacent monomer. Given numbers D and $\min(H_{even}, H_{odd})$, it is easy construct sequences that reach this limit. The bound is therefore tight.

Section 2

S38

The S38 (Simple three-eights approximation) algorithm produces protein conformations that are guaranteed to be within a ratio of the optimal score that converges to three eights. The guarantee is ensured by relating a lower bound for the algorithms conformations, to an upper bound for the optimal.

The basis for the algorithm was introduced, and its approximation ratio proved, by Hart and Istrail, 1995. [5] In a previous project [6], I implemented the approximation algorithm based on their paper. In that project, I found that my learning and implementation process was rather slow, and the resulting program was also very slow due to having implemented structures from the paper that were only useful as thinking tools.

I then concluded in [6] that a more straightforward explanation of the method could be could be easier to understand and implement. S38 is an attempt to provide this straightforward explanation. The hope is that someone who is new to the subject might make a better first implementation than I did, after reading it. The plots on page 44 show how my own first implementation was very inefficient.

In Section 2.1, I provide my explanation of the algorithm. In Section 2.2, I analyze the algorithm and prove the $\frac{3}{8}$-approximation ratio and $\mathcal{O}(n)$ running time. In Section 2.3 I discuss the algorithm and prove that the algorithm extends to D dimensions with approximation ratio $\frac{2D-3}{4D-4}$. The score and running time of the algorithm is tested in Section 4.1.

2.1 Overview

At its heart, the S38 algorithm simply assembles pairs of `H`-monomers, and stacks them on top of each other to produce contacts between them.

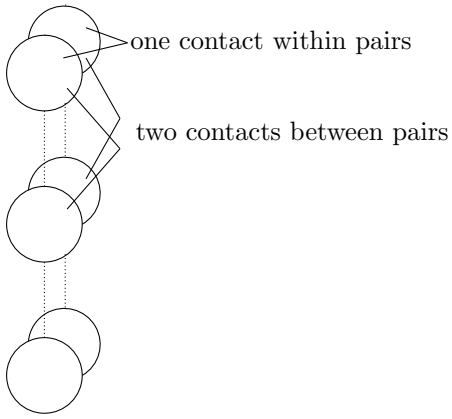


Figure 2.1: Stacked pairs yield three contacts per pair: One in the pair itself and two with the pair they were stacked on. The bottom pair yields only one contact.

As seen in Figure 2.1, stacking pairs yields 3 contacts for each pair added to the stack, beyond the first. The core concept of the algorithm is that we can create at least $\frac{1}{8} \cdot \text{OPT}$ pairs, where OPT is the optimal number of contacts. We then stack them like in Figure 2.1, to yield at least $\frac{3}{8}$OPT contacts. The final configuration is reached in three steps:

1. Selecting what `H` will form pairs
2. Joining the pairs together
3. Stacking the joined pairs on each other

I will now explain each of the three parts in turn.

First, we choose which `H` should pair with each other. This process is rather simple.

Second, we create a configuration for the sequence called Structure 1, that joins the pairs together in a chain, as illustrated in Figure 2.2. In this configuration, one side of the chain consists of odd H, and the other consists of even H. Each side of the chain is referred to as a *strand*.

Third, we stack the pairs in Structure 1 on top of each other by folding the strands, giving a shape akin to the one in Figure 2.3. Like in Figure 2.1, there is three contacts per pair except for some pairs in the bottom and sides. We call this Structure 2, and it is the final configuration. We can think of Structure 2 as a "meta-structure" being applied on top of Structure 1.

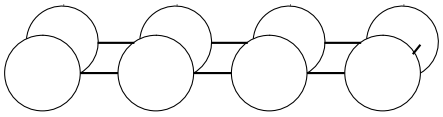


Figure 2.2: Structure 1 is a juxtaposition of two chains of H-monomers, called "strands". Each strand has one half of every H-pair. The number of contacts in Structure 1 is therefore the number of pairs.

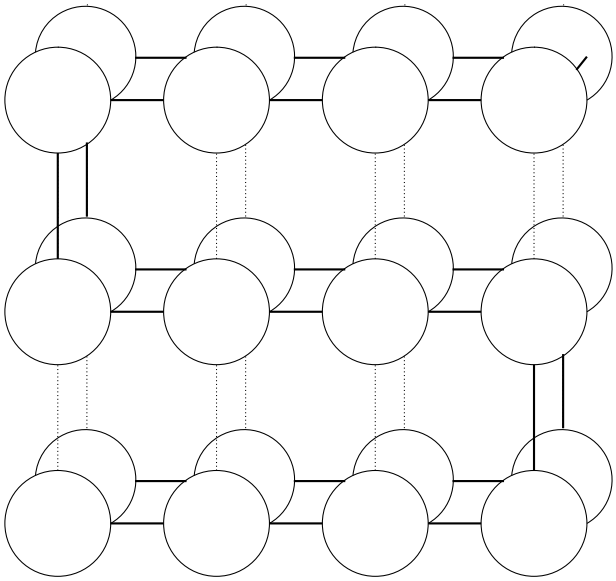


Figure 2.3: Simplified illustration of Structure 2. Like in Figure 2.1, we have three HH-contacts per pair, except for some pairs in the bottom and the sides. Structure 1 can be folded into Structure 2.

2.1.1 Selecting the pairs

The pairs are chosen simply by matching every odd H from one end of the sequence with every even H from the other, until the paired regions overlap. This can be done in two ways, with evens in the first part and odds in the second, or with odds in the first part and evens in the second. We call the former type an "even-odd" pairing, and the second type, and "odd-even" pairing. The best of the two pairings necessarily has $\frac{1}{2} \cdot \min(H_{even}, H_{odd})$ pairs. This is proven in the analysis section on page 14.

2.1.2 Structure 1

To make our structure, the paired H must be adjacently placed in the lattice. We achieve this, we simply join the two halves of the sequence, see Figure 2.4.

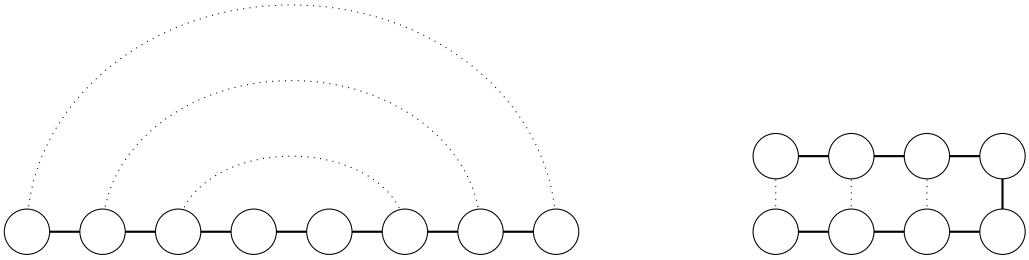


Figure 2.4: Joining the pairs of the sequence. Paired H are indicated with dotted lines.

In practice, the paired `H` do not align themselves because they are interspersed with `P`-monomers and unpaired `H`. Since we are not interested in these, we filter them out with the technique called looping.

Consider the example in Figure 2.5 where orange represents H-monomers, and blue represents P-monomers. The paired H are connected to their partners, and the rest of the sequence has been folded together and moved out of the way. If the distance between paired `H` in the same strand is uneven, then they are separated by $2 \cdot l + 1$ elements for some $l \geq 0$. These $2l + 1$ elements can be placed in a loop of height l, and moved out the way, as the figure shows. Since the paired `H` of a strand must have the same parity, the distance is always uneven, so we can always apply this looping technique.

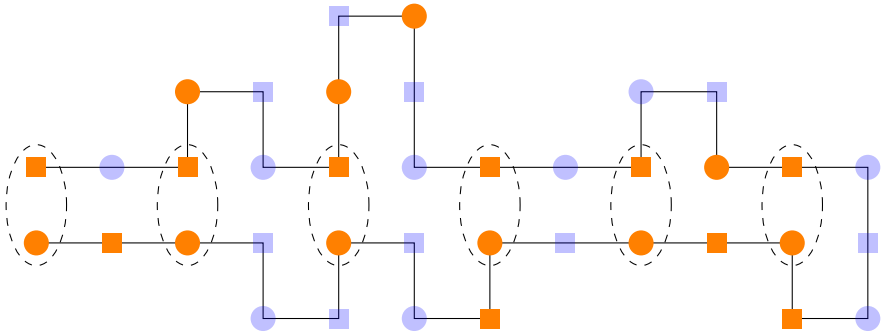


Figure 2.5: The creation of Structure 1. The paired H's are placed in the strands with uniform distance between each pair. Other elements are moved out of the way with looping. Orange elements denote `H`-monomers while blue denotes `P`. The shapes indicate the parity of the monomers.

Structure 1 thus consists of two strands of opposite parities, where `H`'s in one strand are paired with `H`'s of the other. Note that Structure 1 already generates at least one `HH`-connection per pair. It is therefore a $\frac{1}{8}$-approximation on its own. It is also a $\frac{1}{4}$-approximation to the 2D HP problem. Recall from section 1.1 that the upper bound generally is given by $2 \cdot (D - 1) \cdot \min(H_{even}, H_{odd}) + 2$.

The "horizontal" length of structure 1 depends not on the sequence length, but on the number of paired `H`, which in turn is given by $\min(H_{even}, H_{odd})$. Since Structure 2 is only concerned with paired `H`, it is therefore more natural to think of this measure as the length of the input for Structure 2.

2.1.3 Structure 2

The last part of the method is to stack the pairs we have made to produce something akin to Figure 2.3. We can think of each layer in Structure 2 as an instance of Structure 1. By bending one layer back on itself a number of times, we can fold it into several layers.

However, since paired `H` of the same strand cannot make contacts, the strands must "swap sides" in each consecutive layer.

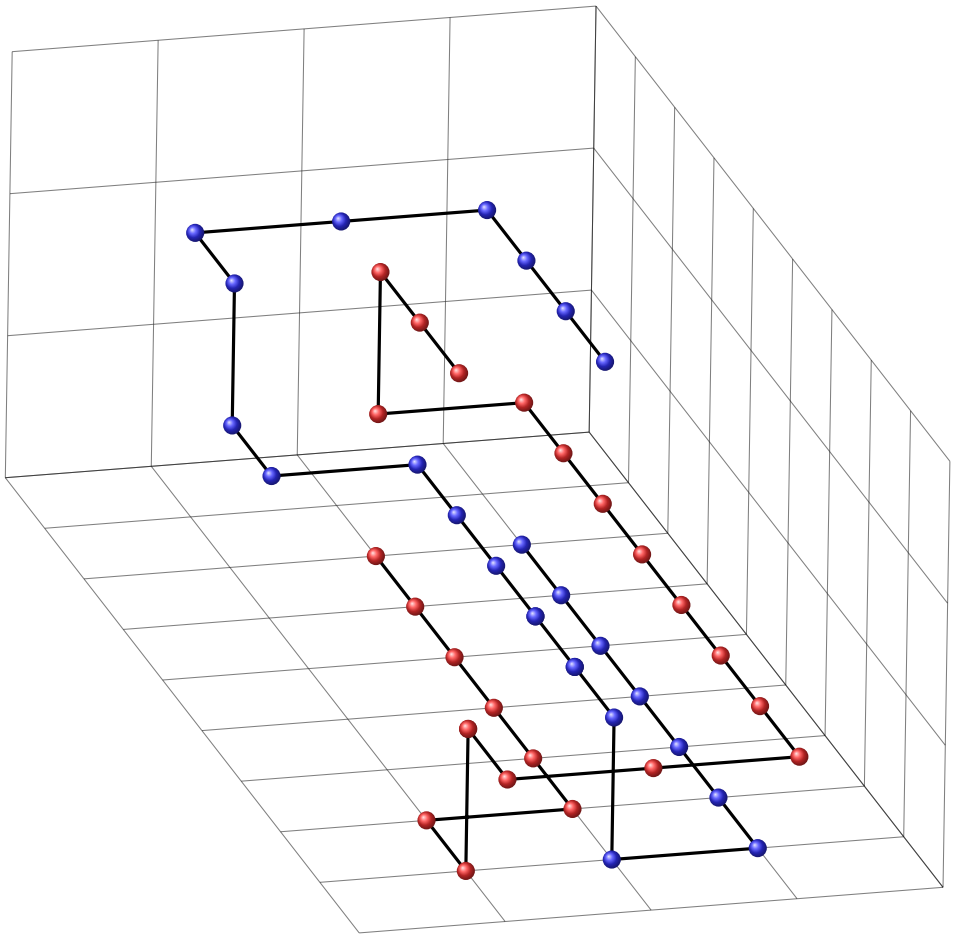


Figure 2.6: Layer switching in S38. The strands swap positions in each consecutive layer, using the illustrated pattern.

Let k be the desired length of each layer. We then keep the first k-length segment of Structure 1, forming the bottom layer. After the k'th pair, we break up the structure and form the configuration seen in Figure 2.6. We call this procedure a "layer switch". After switching layers, the strands

of structure 1 have swapped sides and are connected to the previous layer in the third dimension. We can then repeat this process of layer switching to produce the layered structure that we call Structure 2. If p is the length of Structure 1, Structure 2 will have $\frac{p}{k}$ layers, each of length k.

Since we must break up the first structure during a layer switch, some `HH`-connections may be disrupted. The disrupted connections in a single layer switch is at most some constant number.

The HP-sequence is now in a configuration where we are guaranteed 3 `HH`-connections for each of the $\frac{1}{2} \cdot \min(H_{even}, H_{odd})$ pairs, barring the connections lost in the layer switches, and the bottom layer. Since $\mathcal{O}(k)$ connections are lost in the bottom layer, while $\mathcal{O}(\frac{p}{k})$ are lost in layer switches, this yields a total score loss of

$$L = \mathcal{O}(k + \frac{p}{k})$$

To minimize this loss is to minimize $\max(k, \frac{p}{k})$. We will therefore chose $k = \sqrt{p}$ to "even out" the two terms, giving $L = \mathcal{O}(\sqrt{p})$. This will be used to prove convergence of the approximation ratio in the upcoming analysis section.

2.2 Analysis

In this section, I prove that S38 is indeed within $\frac{3}{8}$ of optimal. This is not a new result by any means. Indeed Hart and Istrail and many others know this to be the case. [5] I was, however, not able to find arguments for the bounds and their convergence to $\frac{3}{8}$ in the limit. It may have been considered trivial enough to exclude from articles with limited space. I therefore provide my own proof in the following.

First we will see that the number of pairs found in the pairing step is more than $\frac{1}{2} \min(H_{even}, H_{odd})$. Then we will show a lower bound for the score, and prove that it converges to $\frac{3}{8}$ of the upper bound derived from the parity of the lattice. Lastly it is argued that the algorithm runs in linear time.

2.2.1 Pairings

Let the number of pairings for an odd-even matching be denoted by p_{oe} and let the number of pairings for an even-odd matching be denoted by p_{eo}. The number of pairings used by the algorithm is then $\max(p_{eo}, p_{oe})$. We will now prove inequality (2.1).

$$\max(p_{eo}, p_{oe}) \geq \frac{1}{2} \cdot \min(H_{even}, H_{odd}) \tag{2.1}$$

Consider an even-odd pairing of a sequence S, and assume $H_{even} \leq H_{odd}$. Then by definition there are $\min(H_{even}, H_{odd})$ even H in S. The even-odd pairing is a partition of S into S_1 and S_2 so that $S = S_1 \oplus S_2$, and (2.2) is satisfied.

$$p_{eo} = \max_{S_1, S_2}(\min(H_{even}^{S_1}, H_{odd}^{S_2})) \tag{2.2}$$

Furthermore, we select S_1 and S_2 so that $p_{eo} = H_{even}^{S_1} = H_{odd}^{S_2}$ is true. This is possible since any unpaired H must have the same parity, and be part of a contiguous subsequence that can be included in either strand. Including evens in S_2 and odds in S_1 means only paired H will be counted in $H_{even}^{S_1}$ and $H_{odd}^{S_2}$.

If $p_{eo} \geq \frac{1}{2}\min(H_{even}, H_{odd})$ then (2.1) holds, so assume $p_{eo} < \frac{1}{2}\min(H_{even}, H_{odd})$. This leads to the following derivations.

$$\begin{aligned}
H_{even}^{S_2} &= H_{even} - H_{even}^{S_1} \\
&= \min(H_{even}, H_{odd}) - p_{eo} \\
&> \min(H_{even}, H_{odd}) - \frac{1}{2}\min(H_{even}, H_{odd}) \\
&= \frac{1}{2}\min(H_{even}, H_{odd}) \\
H_{odd}^{S_1} &= H_{odd} - H_{odd}^{S_2} \\
&\geq H_{even} - H_{even}^{S_1} && \text{Using } H_{odd} \geq H_{even} \text{ and } H_{odd}^{S_1} = H_{even}^{S_2} \\
&> \frac{1}{2}\min(H_{even}, H_{odd}) && \text{Using same derivation as for } H_{even}^{S_2} \\
\min(H_{odd}^{S_1}, H_{even}^{S_2}) &> \frac{1}{2}\min(H_{even}, H_{odd}) && (*)
\end{aligned}$$

Finally by using (2.2) and (*) we get

$$p_{oe} = \max_{S_1,S_2}(\min(H_{odd}^{S_1}, H_{even}^{S_2})) \geq \min(H_{odd}^{S_1}, H_{even}^{S_2}) > \frac{1}{2}\min(H_{even}, H_{odd})$$

Proving that (2.1) is true. The mirror argument is used when $H_{odd} < H_{even}$.

2.2.2 Convergence of Bounds

It is now argued that

$$\frac{LB}{UB} \to \frac{3}{8} \text{ when } \min(H_{even}, H_{odd}) \to \infty$$

Where UB is an upper bound for the HH-connections possible for a given sequence and LB is the lower bound for the HH-connections produced using S38.

Structure 2 produces 3 HH-connections for each of p H-pairs, minus the loss term L. we have already shown that we can ensure $p \geq \frac{1}{2}\min(H_{even}, H_{odd})$ and $L = \mathcal{O}(\sqrt{p})$. This gives us the lower bound

$$LB = 3p - R \cdot \sqrt{p} = 3 \cdot \frac{1}{2}\min(H_{even}, H_{odd}) - R \cdot \sqrt{\frac{1}{2}\min(H_{even}, H_{odd})}$$

For some constant proportionality factor R.

I have earlier claimed [6] about the structure that $R \approx 5.5$ on average, though the size of R has no bearing on the convergence in the limit as we will now see.

A tight upper bound of $UB = 4 \cdot \min(H_{even}, H_{odd}) + 2$ was found in Section 1.1.

Let n denote $\frac{1}{2}\min(H_{even}, H_{odd})$. Now the relation between the upper and lower bound as n goes to infinity is described by the series in (2.3).

$$\begin{aligned} \lim_{n\to\infty}\left(\frac{LB}{UB}\right) &= \lim_{n\to\infty}\left(\frac{3n - R\sqrt{n}}{8n+2}\right) \\ &= \lim_{n\to\infty}\left(\frac{3\sqrt{n} - R}{8\sqrt{n}}\right) \\ &= \lim_{n\to\infty}\left(\frac{3}{8} - \frac{R}{8\sqrt{n}}\right) \\ &= \frac{3}{8} \end{aligned} \tag{2.3}$$

2.2.3 Running time

This section argues that the running time of S38 is $\mathcal{O}(n)$. This asymptotic time cannot be improved, since the length of valid output is also $\mathcal{O}(n)$.

The S38 algorithm is described in detail with pseudocode in Methods 1, 2 and 3 on pages 20-22. The entry point of the algorithm is Method 1, which executes the Method 2 and 3, reverses strings of length $\mathcal{O}(n)$, and substitutes characters in those strings as well. Reversing and substitution takes linear time. It is now argued that the same is true for Method 2 and 3. I will base my arguments on the pseudocode.

Method 2 consists of constant time statements and a loop structure in lines 20-35. The loop structure has inner loops nested inside an outer loop, and the code manipulates i and j to find pairings.

Consider the termination function

$$T = j - i$$

All `while` statements are escaped and we jump straight to line 35 if $T = 0$ is the case. The argument now is that $T = 0$ will occur within a linear number of steps:

The loops at 21-25 and 26-30 both take constant time per iteration, and each iteration decreases T. This means a linear number of executions of inner loops will result in $T = 0$. To spend more than a linear number of steps in the outer loop, we must not enter any of the inner loops. But in

that case, the *outer* loop will run in constant time. Since the outer loop decreases T as well, $T = 0$ is ensured in linear time.

Lastly, Method 3 runs in linear time as well. Lines 8, 20 and 29 are `while` statements that increase the variables i or g. The loops scan the string for the next paired "H" and end when it is found. Since $p > 0$ is checked before each of these loops, the loops are only entered if there is another paired H to find and therefore will end in linear time. Since g is a temporary counting variable that is eventually added to i, increasing g is equivalent to increasing i. Since p is initially the number of pairings, and we decrease p for each paired "H" found, and we check every index of the relevant parity, p must be 0 when the end of the string has been reached, that is, when $i = \texttt{seq.length} - 1$. So because i is never decreased and each inner loop increases i (or g), $p = 0$ will happen in linear time, at which point the outer loop is exited.

This is consistent with the test results in Section 4, cf. Figure 4.1.

2.3 Discussion

My personal view is that the previous sections provide a clearer illustration of the method than [5]. Some or all of Section 1.1, 2.1 and 2.2 can be used as complimentary reading, e.g. by students. Also, the pseudocode pp. 20-22 is very detailed and serves both as a low-level type of explanation, and as a template for implementation.

In an earlier project [6], I implemented the 2D and 3D versions of the algorithm, based on [5]. The "2D version" is equivalent to "Structure 1" in my explanation. These were unnecessarily slow, and my subsequent re-implementations were simpler and faster. I compare their running times in Section 4.

2.3.1 Criticism

Any useful application of a linear time algorithm for approximating the 3D HP problem is not apparent. While protein structure prediction is an important problem to learn about, The approximations do not give us reason to believe that they look anything like actual protein conformations. It is quite obvious that not every protein looks like Structure 2. It also feels like a 'cheat' that the upper bound used to argue the approximation ratio is dependent on the lattice being bipartite. This property of the lattice is an artifact of the models coarseness, and does not represent a property found in real proteins, yet it is exploited to the fullest in the analysis of the algorithm. In this sense, the algorithm intends only to approximate the formal 3D HP problem, not to resemble proteins of the kind that we were trying to model in the first place.

2.3.2 Extension to 4D and Above

In Section 2.1.2 we saw that Structure 1 was a $\frac{1}{4}$ approximation to the "2D HP problem". Here, the "2D HP problem" is shorthand for "protein structure prediction in the HP model with the *square* lattice". In general, the "nD HP problem" is the HP model using a lattice where each vertex has n coordinates, and two vertices are connected exactly if all their coordinates are equal, except for one coordinate which has distance 1. Other parts of the model remain unchanged.

We have seen how the $\frac{1}{4}$-approximation to the 2D HP problem can be used to make a $\frac{3}{8}$-approximation to the 3D HP problem. Similarly, the 3D HP approximation can be used to make a $\frac{5}{12}$-approximation to the 4D HP problem, and generally, a $\frac{2 \cdot n-3}{4 \cdot n-4}$ approximation to the "nD HP problem". Of course the protein analogy goes completely out the window for these problems, but other fun and mind bending problems of math have historically been studied before an application was known, and these "nD HP" problems are indeed fun and mind bending.

4D Method Extending the method to higher dimensions is done by re-applying the technique that builds a Structure 2 from Structure 1. The same technique can be used to build Structure 2 into what we can call "Structure 3" and in general it can build Structure n from Structure $n-1$. I will now explain how the structure can be created, and how the score loss is calculated for 4D, and then in general.

Consider a 4-dimensional lattice with two new directions called "in" and "out". The connectivity of this lattice is 8, so from Equation (1.1), p. 8, we get the tight upper bound of

$$6 \cdot \min(H_{odd}, H_{even}) + 2$$

The number of paired H is not dependent on the lattice model, so we reuse the argument showing it is $\frac{1}{2} \min(H_{odd}, H_{even})$.

Now, the 3D-algorithm is already a $\frac{3}{12}$-approximation to this problem, though with an unexploited 4th dimension to to spare. To achieve the $\frac{5}{12}$-approximation, each matched "H" should be connected to 5 (rather than 3) other "H" except for a loss function that contributes 0 to the ratio in the limit. To achieve this, we layer Structure 2 in the fourth dimension by applying the meta structure again, folding it into Structure 3.

To fold the structure in the new 4th direction we can reuse the layer switching patterns losing only a constant number of HH-connections per layer. The strands will wrap around each other the 'third' dimension, i.e. the one in which layers of Structure 2 are stacked. Since we switch in/out-layers at the *ends* of Structure 2, we know there unoccupied space in the third dimension to perform the wrapping.

We now select $k = \sqrt[3]{p}$. Each layer of Structure 3 is then an instance of Structure 2 with $\sqrt[3]{p}$ as both its layer length and layer count. Each layer of Structure 3 therefore has $\sqrt[3]{p}^2$ paired `H` in it. A total of $\frac{p}{\sqrt[3]{p}^2} = \sqrt[3]{p}$ layers can be created this way, placing each paired `H` in the layers of Structure 3 (instances of Structure 2) next to two additional `H`, save for the loss at the edges of Structure 3.

This extension generally leads to a hierarchical structure where each "nD *layer*" consists of one entire "$(n-1)$D" *structure*. We can always switch between layers in dimension n by wrapping in dimension $n-1$. We still only accumulate a constant amount of score loss per switch.

Score Loss Adding a dimension increases layer switches and end layers k-fold, which means the overall score loss increases. The S38 score loss of $L \propto \sqrt{p}$ that was argued in section 2 was based on end layer loss proportional to k and layer switch loss proportional to $\frac{p}{k}$. As $k = \sqrt{p}$, they both came out to $\sqrt{p}$. The 4D case is similar, with $k = \sqrt[3]{p}$. We now have k of these 3D structures stacked "inside" of each other, each one containing $\frac{p}{k} = \sqrt[3]{p}^2$.

The *layer switch loss* from Section 2.1 of each 3D structure therefore comes out as $\frac{\sqrt[3]{p}^2}{k} = \sqrt[3]{p}$, while their *end layer losses* become $k = \sqrt[3]{p}$. This means total loss suffered in each 3D structure is proportional to $k = \sqrt[3]{p}$. Since we have stacked k of those, the total loss in the 4D structure is proportional to k^2. In the 4D case we therefore get $L \propto (\sqrt[3]{p})^2$. Generally, we can nest k structures in a higher-dimension structure, to get k times more layers. We can therefore generally select $k = \sqrt[D-1]{p}$ to get score loss $L \propto \sqrt[D-1]{p}^{D-2}$

Convergence The limit from (2.3), page 16, is still applicable with the increased score loss of higher dimensions. For 4D:

$$\lim_{n\to\infty}\left(\frac{LB}{UB}\right) = \lim_{n\to\infty}\left(\frac{5n - R(\sqrt[3]{n}^2)}{12n+2}\right) = \lim_{n\to\infty}\left(\frac{5\sqrt[3]{n} - R}{12\sqrt[3]{n}}\right) = \lim_{n\to\infty}\left(\frac{5}{12} - \frac{R}{12\sqrt[3]{n}}\right) = \frac{5}{12}$$

And since $\lim_{n\to\infty} \sqrt[D]{n} = \infty$ for any D, the convergence argument can be used for any D as well.

We have now seen that the all parts of the method and the approximation arguments can be generalized to D dimensions, which means the algorithm works in any dimension! Generally we get the bound:

$$\begin{aligned}
\lim_{n\to\infty}\left(\frac{LB}{UB}\right) &= \lim_{n\to\infty}\left(\frac{(2D-3)n - R(\sqrt[D-1]{n}^{D-2})}{(4D-4)n+2}\right) \\
&= \lim_{n\to\infty}\left(\frac{(2D-3)\sqrt[D-1]{n} - R}{(4D-4)\sqrt[D-1]{n}}\right) \\
&= \lim_{n\to\infty}\left(\frac{2D-3}{4D-4} - \frac{R}{(4D-4)\sqrt[D-1]{n}}\right) \\
&= \frac{2D-3}{4D-4}
\end{aligned}$$

And the approximation ratio approaches $\frac{1}{2}$ as $D \longrightarrow \infty$.

Method 1 S38

Input:
`string` seq ▷ seq $\in \{H, P\}^*$

$p, H_1, H_2 \leftarrow$ `Pairing`(seq)
$S_1 \leftarrow reverse(seq[0 : H_1])$
$S_2 \leftarrow seq[H_1 + 1 : H_2 - 1]$
$S_3 \leftarrow seq[H_2 : seq.length - 1]$

$F_1 \leftarrow$ `FoldStrand`$(S_1, p, 1)$
$F_2 \leftarrow$ "B" $\cdot (H_2 - H_1) +$ "D" $+$ "F" $\cdot (H_2 - H_1)$
$F_3 \leftarrow$ `FoldStrand`$(S_3, p, 0)$
Reverse F_1 and replace symbols "F"$\longleftrightarrow$"B" and "L"$\longleftrightarrow$"R" but leaving "U" and "D"

return $F_1 + F_2 + F_3$

Method 2 Pairing

Returns the number of H-pairs and the indexes of the innermost pair

Input:
`string` seq ▷ seq $\in \{H, P\}^*$

```
    i_even ← seq[0]
    i_odd ← seq[1]

    if seq.size mod 2 = 0 then
        j_even = seq.size − 2
 5:     j_odd = seq.size − 1
    else
        j_even = seq.size − 1
        j_odd = seq.size − 2

10: with i_even, j_odd, p_eo as i, j, p
        Run lines 20-35
    with i_odd, j_even, p_oe as i, j, p
        Run lines 20-35

15: if p_eo > p_oe then
        return p_eo, i_even, j_odd
    else
        return p_oe, i_odd, j_even

20: while i < j do
        while seq[i] = P do                          ▷ find next i or stop
            if i + 3 < j then
                i ← i + 2
            else
25:             goto END_LABEL                       ▷ Done
        while seq[j]="P" do                          ▷ find next j or stop
            if i + 3 < j then
                j ← j − 2
            else
30:             goto END_LABEL                       ▷ Done
        if j − i > 1 then                            ▷ Only count pair if they are not adjacent in seq
            p ← p + 1
        i ← i + 2
        j ← j − 2
35: END_LABEL
```

Method 3 FoldStrand

Returns a fold description for an HP-strand

Input:
`string` seq ▷ $seq \in \{H, P\}^*$
`int` p ▷ Number of pairs found in Method 2
`flag` w ▷ Indicates which layer switching method to use

inc, dec, forw, back ← "U", "D", "R", "L"
$j \leftarrow 0$ $k \leftarrow \sqrt{p}$ $\leftarrow 0$ $i_{prev} \leftarrow 0$ $fold \leftarrow \lambda$

$p \leftarrow p - 1$
while $p > 0$ **do**
  if $j < k$ **then** ▷ We switch layers when $j = k$
    while $seq[i] =$ "P" **do**
      $i \leftarrow i + 2$
    $h_l \leftarrow \frac{i - i_{prev}}{2} - 1$ ▷ h_l is height of the loop to be made
    $fold \leftarrow fold + (h_l \cdot \text{inc}) + \text{forw} + (h_l \cdot \text{dec}) + \text{forw}$
    $p \leftarrow p - 1$
    $i_{prev} \leftarrow i$
  else
    if w **then** ▷ 'Wrapping' layer switch
      $p \leftarrow p - \sum_{l=1}^{3} x_l$ where $x_l \begin{cases} 1, & \text{if } seq[i_{prev} + 2l] = "H" \\ 0, & \text{otherwise} \end{cases}$
      $g \leftarrow 0$
      if $p > 0$ **then**
        $p \leftarrow p - 1$
        while $seq[i_{prev} + 8 + 2 \cdot g] =$ "P" **do**
          $g \leftarrow g + 1$
      $fold \leftarrow fold + forw + dec + dec + back + "F" + g \cdot dec + back + g \cdot inc + inc + back$
      $i, i_{prev} \leftarrow i_{prev} + 8 + 2 \cdot g$
    else ▷ 'Non-wrapping' layer switch
      $p \leftarrow p - 1$ **if** $seq[i_{prev} + 2] =$ "H"
      $g \leftarrow 0$
      if $p > 0$ **then**
        $p \leftarrow p - 1$
        while $seq[i_{prev} + 4 + 2 \cdot g] =$ "P" **do**
          $g \leftarrow g + 1$
      $fold \leftarrow fold + F + dec + g \cdot dec + back + g \cdot inc + back$
      $i_{prev} \leftarrow i \leftarrow i_{prev} + 4 + 2 \cdot g$
    fi
    $forw \longleftrightarrow back$ $inc \longleftrightarrow dec$ ▷ after switching, change directions and reset j and w
    $j \leftarrow 0$
    flip w
  fi

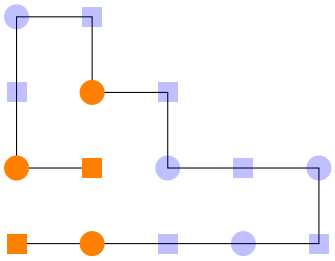
Section 3

CPSP

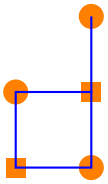
3.1 Overview

CPSP (for Constraint Programming based Structure Prediction) is a search algorithm that finds optimal solutions to the HP problem. It uses the concepts of *hydrophobic cores* and *constraint programming* to narrow down the solution space, which is enumerated in exponential time. The algorithm does not improve upon the worst case asymptotic running time compared to a naive search algorithm.

The *hydrophobic core* of a protein conformation is just the set of `H` monomers. An example in 2D is given in Figure 3.1. The *contact number* of a core, is the number of neighboring `H`-pairs.



(a) An optimal 2D conformation.



(b) The hydrophobic core of the conformation in (a). The blue lines indicate contacts. The contact number is 5.

Figure 3.1: Figure (b) shows the hydrophobic core of the conformation in (a).

The CPSP algorithm is based on the following observations:

1. The score of a conformation in the HP-model depends only on the number of contacts in its hydrophobic core.

2. The contact number of an optimal fold with n H-monomers is usually very close to the maximal number of contacts with n elements in general.
3. We can therefore restrict the search space to only include conformations that have (almost) contact-optimizing hydrophobic cores.

One of the merits of the algorithm is that most of its computations depends only on the number of H-monomers rather than the exact sequence. As a consequence, most calculations can be performed in advance, and their results stored for use at runtime. The algorithm thus has a preparation step and an execution step.

Core Construction is the preparation step. It means computing what we call the "core database". we do this by determining all contact-optimal cores for a given number of elements, and once all solutions are found, the second most optimal cores, and so on. Since an optimal conformation of a sequence s has an almost contact maximizing hydrophobic core, it is realistic to have such a core precomputed with very high probability.

Threading is the execution step. Threading sequence s to a core c means folding s so that its H-monomers are in c. The algorithm starts by threading the sequence on the optimal cores, and then works its way downwards. For each core we either output a valid threading or prove that none exists. Once a threading is successful, we know that it is an optimal conformation, because every configuration that has more contacts has already been proven to be incompatible with the sequence.

Section 3.2 introduces Constraint Programming, a central concept employed by CPSP. Section 3.3 explains how the threading algorithm was implemented. Section 3.4 covers the prerequisite step of core construction. I cover some implementation considerations in detail, but have not implemented the algorithm. Section 3.5 contains some general discussion topics about CPSP. Finally, the running time of CPSP is tested in Section 4

3.2 Constraint Programming

This section covers the basics of the *Constraint Programming* paradigm, which is used in CPSP to solve both the threading problem and the core construction problem.

Constraint Programming is a form of declarative programming, meaning one does not control the actions to be taken, but rather specify properties of the solution to be found. In constraint programming, the desired solution to a problem is stated as a set of relations between variables, called constraints.

Solving a CSP amounts to satisfying the constraints, i.e. assigning to each variable a value within its permissible domain such that all constraints are upheld.

Solutions are found by enumerating the search space while restricting it more and more, based on

logic derived from the constraints during the search process. Being a search method, it is often used for problems that are computationally hard. It is generally suitable for problems that involve assigning values to variables based on rules. These problems are often "puzzle"-like in nature. Examples include scheduling, sudoku, n-queens, bin packing, and of course, structure prediction in the HP model.

Commonly in constraint programming, the user specifies a CSP in a tool that lets a general purpose solver find a solution. If the problem calls for it, she can also implement a specialized solver that is optimized for the given problem. My CPSP implementation was done with a general purpose solver, while Mann and Backofen programmed their own solver that was optimized for threading. In the next chapter we shall see how a CSP solver can use *branching* and *constraint propagation* to solve CSP's.

3.2.1 Constraint Propagation

Constraint propagation is a technique that aims to reduce the search space by logically applying the constraints of the CSP. It does this be enforcing *arc consistency* between *domains*.

The *domain* X of a variable x is the set of values that the variable can take. An *arc* from domain X to Y is a directed edge that represents a relation between the domains as defined by constraints.

Let c be a constraint on variables x and y with domains respectively X and Y. The directed arc $c(X, Y)$ is *consistent* if

$$\forall v \in X \quad \exists w \in Y : x = v \text{ and } y = w \text{ satisfies } c$$

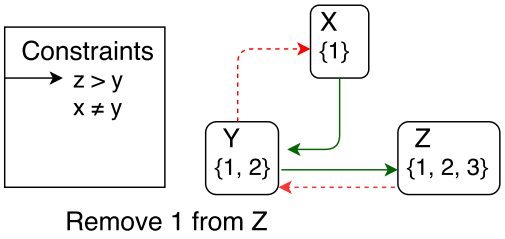
That is, arcs from X to Y are said to be consistent if for all remaining values in X, there is a value in Y such that the use of both values is valid with respect to the constraint. A set of domains is *arc consistent* if all arcs between all domains in the set are consistent.

If an arc (X, Y) is inconsistent, we can enforce consistency by deleting values of X that do not meet the condition. In Figure 3.2a we see two arcs between domains Y and Z representing the constraint $z > y$. The (Y, Z) arc is consistent because for each element in $Y = \{1, 2\}$ there is an element in $Z = 1, 2, 3$ that satisfies $z > y$. The (Z, Y) arc is inconsistent because if $z = 1$ then there is no valid assignment to y. Since $z = 1$ is therefore not allowed in our solution, we can remove it from the domain so that the arc becomes consistent as showed in Figure 3.2b.

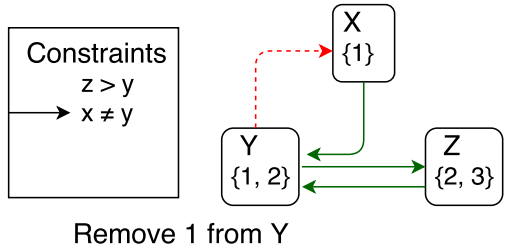
Before searching a solution space, we exclude every domain value that we can, based on the constraints. Figure 3.2 illustrates how. After arriving at Figure 3.2b with the method just described, we move on to enforce the constraint $x \neq y$ leading to 3.2c. Here we see that after removing a value in Y, the (Z,Y) arc that was previously consistent is now inconsistent, and thus have to be enforced again, leading to Figure 3.2d. We repeat this process until all arcs have been verified as consistent, without any domain being changed. When all arcs are consistent, we say that the system is arc consistent, as seen in 3.2d. This process is what we call constraint propagation. In this case, the constraint propagation lead to the solution $x = 1$, $y = 2$, $z = 3$. If a solution is not found after

constraint propagation, we *branch*, which is the subject of Section 3.2.2

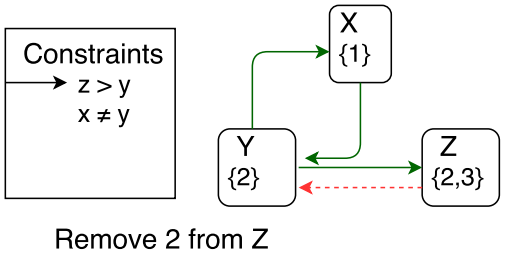
Note that relations that are naturally stated as a single constraint can entail several arcs. Domains can have any number of arcs between them. If a domain is empty after enforcing an arc, then the constraints are not satisfiable.



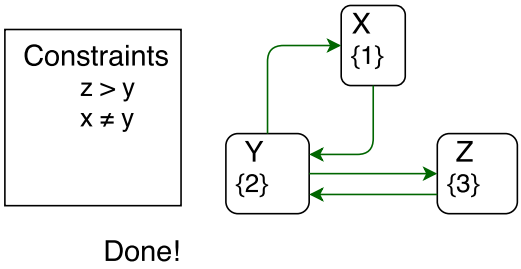
(a) (Z,Y) inconsistent: Remove 1 from Z to enforce $z > y$.



(b) Since (Z,Y) and (Y,Z) are consistent, we move on to enforce the next constraint. Remove 1 from Y to satisfy $x \neq y$.



(c) After Y was reduced, the (Z,Y) arc is inconsistent again, and must be re-enforced.



(d) We repeat this until all arcs pass consistency check. The system is arc consistent when all arcs are consistent.

Figure 3.2: Algorithm for arc consistency.

Running Time We now argue the running time for the arc consistency algorithm. Let E be the number of arcs, V the number of variables, and D the size of the largest variable domain. Checking an arc (A,B) for consistency takes $\mathcal{O}(D^2)$ time, since for each value in A we must search all of B to find a value that satisfies the arc.

We check $\mathcal{O}(E)$ arcs in time $\mathcal{O}(ED^2)$ before potentially discovering an inconsistency and starting over. We start over at most $\mathcal{O}(VD)$ times, since we remove at least one value from a domain each time. The algorithm therefore runs in time $\mathcal{O}(VD \cdot ED^2) = \mathcal{O}(EVD^3)$.

3.2.2 Branching and Search

Often in cases with many variables, arc consistency does not directly result in a solution. Therefore, once arc consistency is reached without emptying a domain, we need to search the remaining solution

space. We can assign a tentative value to a variable, and then reduce the remaining branches with constraint propagation. We can always assign a tentative value since the domains are non-empty if the branch is still being considered. This leads to a iterative process of branching and constraint propagation that attempts to assign values to each variable, solving the CSP. If the current branch is discovered to be invalid, we roll back the last tentative assignment, and try again with a different value. This kind of searching is exemplified in Figure 3.3.

Figure 3.3 also demonstrates that it matters how we branch. A rule of thumb is that it is often preferable to pick the variable with the *smallest* domain, and assign to it the value that reduces the other domains the *least*. Doing this results in the smaller search tree shown in Figure 3.4.

3.2.3 The MiniZinc language

MiniZinc is a language used to model problems as a decision or optimization constraint programming problem. MiniZinc and the associated constraint modeling language FlatZinc resulted from deliberate efforts to create a standardized model representation that could be used with many different solvers. FlatZinc was designed to be supported by CSP solvers with only small efforts by their developers, to encourage widespread adoption, while MiniZinc was meant to be a high-level language in which the end user expresses their model. MiniZinc then translates the model into FlatZinc, after which a variety of solvers can be run.

Any MiniZinc model must include the 3 mandatory items

- One or more `variable` items
- A `solve` item
- An `output` item

A `variable` item represents an unknown. It belongs to one of the types {`int, float, string, boolean`} Each variable must have a finite domain of legal values associated with it. The goal of the solver is to find valid assignments to every variable from their respective domains.

A `solve` item specifies the type of solution to be found. A solution is either satisfying or optimizing, and in the latter case, an optimization function must be included in the solve item.

An `output` item is specification of the text to print in case a solution is found. This is a convenience measure since any solution could be given in the form of every variable assignment, though models can have large quantities of variables of which most are not interesting to view.

A model may also include:

- Values

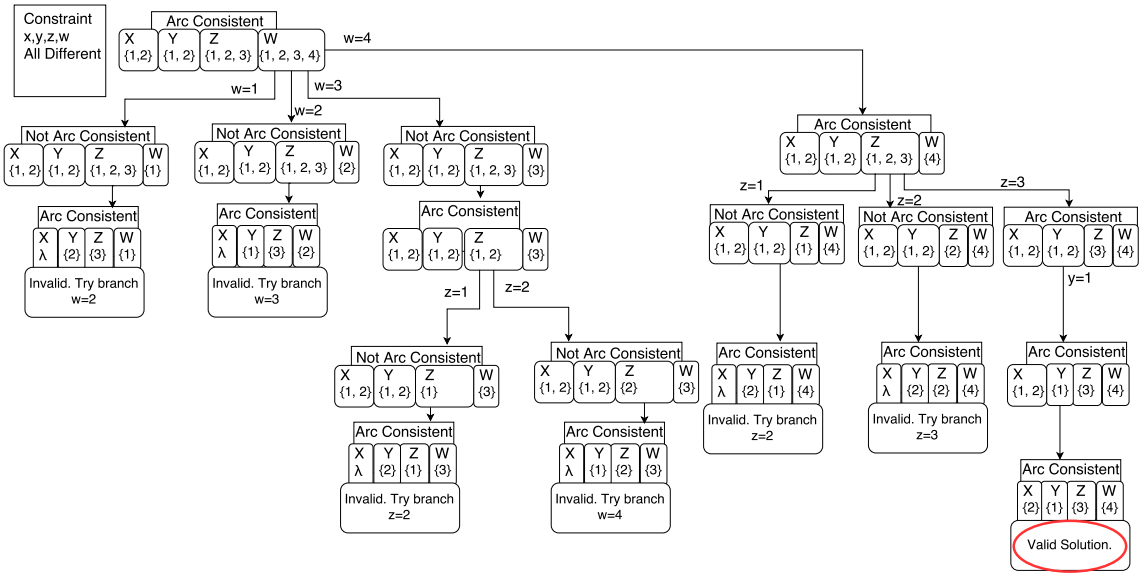


Figure 3.3: The tree for a depth-first search. The branches are chosen in left to right order. The search repeatedly assigns a tentative value to a variable, enforces arc consistency, and then rolls back the last tentative value if the current branch is found to be invalid, i.e. if one of the domains is empty. The constraint in this example is that all variables must be different. This particular tree results from branching in the order w, x, y, x. As a rule of thumb, starting with w and z is bad strategy because their domains W and Z are larger than X and Y.

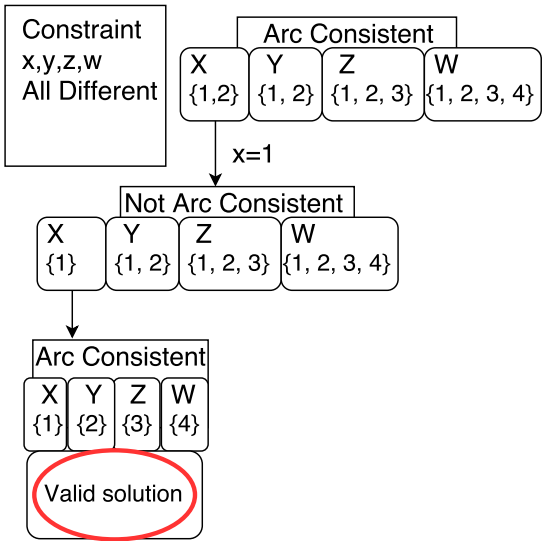


Figure 3.4: A tree that searches the same space as in Figure 3.3, but branches with x first. This choice is better, and reduces the search time.

- Constraints
- Annotations

`Values` represent the knowns of the model. They provide a way to save expressions to symbols for later use. They can have the same types as variables.

`Constraints` describe relations between variables and/or values. A valid variable assignment must satisfy all constraints. Though not technically required, any problem that calls for the use of constraint modeling, will need to have constraints.

`Annotations` are meant to express search preferences on the modelers behalf for how the solver should branch and propagate.

Lastly, the language provides some items that can improve the syntax of a model. These include `Predicates`, that encode a constraint-like statement into a function on values and variables that returns a boolean; composite types, such as `sets` and `tuples` of values and variables; and `include` items, to make use of libraries of pre-defined constraints and predicates.

MiniZinc and the related toolchain is an example of the inversion-of-control principle. Modeling problems in these terms is enough to solve a CSP if handed to any generic solver, but one does not control how the solution is found. The language allows for gradual building of complex models. Complicated relations are given shorthands through the use of `predicate`, and a model can be distributed across any number of files can that are included with `include`, supporting a modular model design.

3.3 Threading

The CPSP algorithm proves the optimality of a solution by trying them in order from best to worst. For each core in order, we must either find a valid threading with the given sequence, or prove that none exists.

A sequence of coordinates r, is a valid threading of sequence s to a core c if the following is true.

- No two coordinates in r are the same (the fold is self avoiding)
- Successive coordinates in r are neighbors (the fold is a walk)
- If $s_i = \texttt{H}$ then r_i must be in c (the fold threads the core)

By formulating these requirements as constraints, the threading problem is conveniently modeled as a CSP. The solution to the CSP is a sequence of coordinates that meets the requirements. Section 3.3.1 contains more details on how to model the threading problem in MiniZinc.

3.3.1 MiniZinc Threading Model

The threading CSP constructed for this project uses the MiniZinc language described in section 3.2.3. I here give a brief description of how the threading problem was modeled as a CSP.

Coordinate Representation The base type `int` is the obvious choice for representing coordinates but there are several ways to encode the representation. The first idea may be to model coordinates as three integers, one for each coordinate dimension.

In my CSP, a scheme encoding each coordinate set as a single `int` was preferred. Among other things, it allowed the use of the library function `AllDifferent` which provided a significant speedup over equivalent 'homemade' constraints. To do this we choose a value d greater than any coordinate value we will need to use. An obvious upper bound for d is the length of the sequence. We then encode a set of 3D coordinates as a integer x so that $0 \leq x \leq d^3 - 1$. The coordinate set is then given by the d-ary representation of x. For example, if $d = 8$, then coordinates $(2, 1, 2)$ would be represented by the base-10 number 138, since 138 in base 8 is 212.

Bound for d-value The base number d represents the range of values that each coordinate can assume, and it also dictates the size of the solution space since we must search for each coordinate in the range $[0, d^3 - 1]$. We can think of d as the length, height and width of a cube that the fold must fit inside. If d is too small, the optimal fold may not "fit" in the cube and we will get a suboptimal solution. If it is too large, we may not be able to search the space at all.

Whether the fold "fits in the cube" (which is just a geometric way of thinking about whether the optimal coordinates can be represented with the given d), also depends on where we position the hydrophobic core in the cube.

An upper bound for the value of d can be extrapolated from the fact that for any valid threading, the sequence s simply must allow for each `H` to be placed in the core. This means that the maximal distance that any segment of s can 'stray' away from the core is

$$m_c = \lfloor \frac{g}{2} \rfloor$$

where g is the longest interval of `P` in s. An illustration in 2D is given in Figure 3.5. We can think of m_c as the "margin" of the core, since the "cube" is ensured to contain the fold if the core has distance m_c to each "cube wall". If w is the maximal length of any side of the core, we can choose

$$d = w + 2m_c$$

to hold the core and its margins. We then "place" the core by offsetting each of its elements by m_c in all three directions. This ensures that we have at least the required margin on each side.

Before thinking of this bound for d, I was making heuristic guesses for the d-values that were not accurate. I tried to choose d-values that were a little larger than necessary, to ensure that the

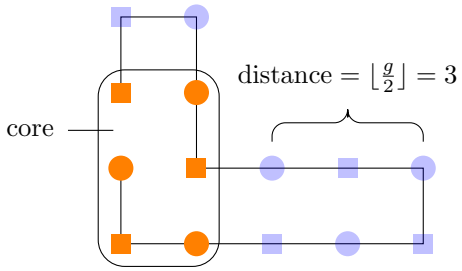


Figure 3.5: When all `H` are in the core, no part of the sequence can be further than $\lfloor \frac{g}{2} \rfloor = 3$ from the core where g is the longest segment of `P`-monomers.

output was correct, but occasionally I would get an error when the guess was not conservative enough. When I introduced this simple method, it removed the errors, while also producing lower d-values on average, meaning smaller search spaces, and a faster running time.

Constraints The three main constraints of my CSP is explained here, first by intuition and then by formal logic.

The *self-avoiding* constraint requires simply that no two amino acids occupy the same vertex. This means all coordinates must be different.

The *core* constraint requires the H-subset of the sequence s to be in the hydrophobic core c.

The *walk* constraint requires subsequent amino acids to also be connected in the lattice. Lattice elements are connected if their three coordinates differ by exactly one, zero and zero. When represented as a single number with some base d, the coordinates should have difference of 1, d or d^2. In the two former cases, we also need to rule out the possibility that a difference of 1 or d was obtained by changing a higher level coordinate and compensating with the other coordinates. For example if $d = 6$, then 33 and 39 have difference d but are not connected, as they correspond to coordinates $(0, 5, 3)$ and $(1, 0, 3)$.

Now let n be the length of s, and let $X = x_1 \dots x_n$ be variables over the domain $[0, d^3 - 1]$.

The *self-avoiding* constraints are given by

$$\forall x_i, x_{j \neq i} \in X : x_i \neq x_j$$

the *core* constraints are given by

$$\bigwedge_{i=1}^{n} s_i = \texttt{'H'} \longrightarrow x_i \in c$$

And, letting $\div$ stand for integer division without remainder, the *walk* constraints are given by

$$\forall x_i \in X/\{x_n\} : Connected(x_i, x_{i+1})$$

Where $Connected(x_i, x_j)$ is the logic predicate

$$\begin{array}{llllll} & |x_i - x_j| = d^2 & & & & \\ \vee & |x_i - x_j| = d & \wedge & x_i \div d^2 = x_j \div d^2 & & \\ \vee & |x_i - x_j| = 1 & \wedge & x_i \div d^2 = x_j \div d^2 & \wedge & x_i \div d = x_j \div d \end{array}$$

In words the constraints say "No two coordinates in the solution can be the same", "If the i'th amino acid is hydrophobic, the i'th coordinates must be a member of c" and "All coordinates of the solution, save for the last one, should be adjacent to the one that comes after". Satisfying these constraints yields a solution to the threading problem.

We can accept the first threading that is found, since no threading on a core c is superior to another. This is because the score of a conformation is completely given by the number of contacts in c. If the contact number of c is $\mathcal{C}$ and the number of "HH"-occurrences in the sequence, including overlap, is m then we get

$$\text{score} = \mathcal{C} - m$$

And since m is fixed, we maximize the score purely by maximizing the contact number of the threaded core c.

For example, the core from Figure 3.1, page 23 has contact number 5, and the number of "HH"-occurrences in the sequence is 2, so the conformation has a score of $5 - 2 = 3$.

Since the score is given from the sequence and the core that was threaded, the CPSP implementation does not need a separate program that counts it.

3.3.2 Implementation Efficiency

I will round off the threading section with a discussion on efficiency and techniques described in the Backofen and Will paper. [2] The reason for using CPSP over a naive search algorithm, is that it will run faster in practice, which makes efficient implementation of the threading algorithm important. Experience with constraint programming and relevant tools is rather pivotal in this regard. I do not believe any threading implementation in MiniZinc will run faster than Backofen and Wills algorithm, which uses a lower-level Constraint Programming toolkit called GECODE.

Skipping Scheme One important implementation detail for the cubic lattice version of CPSP was not mentioned in the Backofen and Will paper [2], namely that one does not even need to solve the threading problem for most cores.

As an example, imagine threading a sequence with 8 H-monomers on a 2x2x2 core. This need not even be attempted unless the parity of the H-monomers are such that 4 are even and 4 are odd, to match the elements in the core. Otherwise, we know beforehand that the H-monomers in the sequence can not assume the cores configuration, due to the parity property of the cubic lattice mentioned in Section 1.1. The CSP's core constraints would therefore not be satisfiable.

Let S be a sequence with n H-monomers, let C be a core of size n, let S_{even} be the number of H in S that have even indexes, and let C_{even} be the number of coordinates (x, y, z) in C that have "even" coordinates. We say a coordinate is "even" if $x + y + z \mod 2 = 0$. If C is threadable by S, then statement (3.1) is true.

$$S_{even} = C_{even} \quad \vee \quad n - S_{even} = C_{even} \tag{3.1}$$

For problems of relevant sizes, the parities most likely do not match, which means we can use this to skip past most cores without threading, in a matter of milliseconds. Often when do this, the first attempted threading is successful. This is made even better by the fact that successful threadings usually take less time to perform than failed ones. Some sequences were folded hundreds of times faster once I found this shortcut. Table 4.2, p. 50 shows how many cores were skipped for each test string.

Backofen and Will likely omitted this scheme from their paper because it is relevant specifically to the *cubic lattice*, rather than to the algorithm that is the subject of their paper.

Branching Factor A very important aspect is how much the constraints limit the search space on each propagation. Backofen and Will mention issues of slow propagation, [2, p. 25] and describes a technique used to better the problem. This means that their threading algorithm likely searches a much smaller domain than mine. Their technique involves replacing the "walk" constraints with ones that implement "3-avoiding walks". "3-avoiding" means that any length 3 segment of the sequence is self avoiding. This is logically redundant since they also have constraints that enforce self-avoidance of the entire sequence, but reportedly provides stronger propagation when coupled with the self avoiding constraints. I was not able get better propagations using that technique.

This was only one of many time saving techniques mentioned. The paper also discusses avoiding symmetric solutions with a CP technique that they wrote a different paper about [3], as well as a way to limit the usage of the expensive `AllDifferent` constraint by using specialized versions of it in some circumstances. I did not try to use these techniques. They likely deployed many more subtle techniques than they could mention in their paper.

The propagation issue however, is different and more important than speedup by a constant factor, because the payoff from restricting the search space more efficiently is *exponential* in the problem size. I will now explain what I mean by that.

First, recall that the skipping scheme mentioned above was a big time saver in the sense that some folding times were reduced to less than one percent. The foldable lengths, however, were only increased by less than a handful. When lengths were increased, the algorithm could not even thread

a single core, so the improvement did not do much in that regard. Generally, in an exponential setting, a speedup by any factor only goes a short way in terms of problem size.

But propagation improvements are different. If we reduce the search space more efficiently on every propagation, then the number of new branches at each level of the search tree, known as the "branching factor", becomes smaller. This means that instead of reducing the search time by some factor, it is reduced by a factor *on every subsequent branching*. If $X > Y$, and we reduce the branching factor of tree from X to Y then the number of leaves in the new tree is $(\frac{Y}{X})^k$ times that of the old tree, where k is the height of the trees. Since $\frac{Y}{X} < 1$ we know that

$$(\frac{Y}{X})^k \longrightarrow 0 \text{ for } k \longrightarrow \infty$$

Therefore if the problem size increases enough, the benefit of branching factor reduction, no matter how small, outweighs the benefit of a constant factor speedup, no matter how big. Reduction of branching factor is not just better in a mathematical sense, but also much more effective in practice. My CPSP algorithm will not realistically fold sequences in the hundreds unless the branching factor is reduced. Further work for this algorithm would therefore start by learning more about the practice of constraint programming and research on how the propagation might be made more effective, as this type of improvement is potentially enormous.

The next chapter describes the ideas of the *core construction* algorithm that computes the cores used for threading.

3.4 Core Construction

The prerequisite for threading is a database of cores. A core is a connected three dimensional set of points on the lattice with many contacts between the points. Threading a sequence requires knowing every contact-optimizing core of the same size as the number of H-monomers in the sequence, as well as the second most contact-optimizing, third most, and so on. The more cores you have, the more likely it is that the threading will succeed before trying all of them. A constraint based approach to Core Construction was developed by Backofen and Will [2] and successfully used to construct the core databases utilized by CPSP. It uses Constraint Programming, Dynamic Programming, and a theoretical construction they call *Frame Sequences*. This section gives an overview of their method, and also includes details that they omitted, especially about their dynamic programming algorithm. These include

- The base step and recursion step of the dynamic programming algorithm
- How to fill the dynamic programming table in time $\mathcal{O}(n^5)$ time
- What table entries can safely be omitted
- The order in which to compute the entries

- How to backtrack the results.

The core construction algorithm was not implemented as a part of this project.

Like threading, the actual core construction is implemented as a CSP to which the solution is a set of 3D coordinates, this time representing a hydrophobic core, rather than a protein conformation. A solution to the CSP is a hydrophobic core of n elements, that has at least some minimum number of contacts. Unlike the threading problem, we are interested in finding *every* solution, symmetries excluded, meaning the entire search space must be enumerated. This is a feasible computation due to efficient bounding of the search space through the notion of *frame sequences*. This section explains how the bounding is achieved. I will not go into details about how the CSP is formulated.

Section 3.4.1 introduces the notion of *frame sequences*. In section 3.4.2 we define an upper bound of the number of contacts for a core with a given frame sequence. In Section 3.4.3 we describe a dynamic programming algorithm for finding every frame sequence with upper bounds greater than a given number, and how the frame sequences are used to bound the search space for the CSP. The last step, which is not covered here, is searching the bounded space with CP to find the cores.

3.4.1 Frame Sequences

Like brain slices, a three dimensional core can be decomposed in several ways into a series of two dimensional slices called *layers*. Each layer has a minimal surrounding rectangle, called a *frame*. A frame i represented as a triplet of integers (n, a, b) where n is the number of H-monomers in the layer, while a and b are the width an height of the frame. A frame sequence for a hydrophobic core is then the sequence of frames that correspond to the sequence of layers that make up the core. Figure 3.6 shows one of the possible frame sequences of the hydrophobic core in Figure 1.2, p. 8.

3.4.2 An upper bound based on Frame Sequences

Here we define an upper bound for the number of contacts possible in a core, given the cores frame sequence. We will do this by separately defining bounds on the number of *in-layer contacts* and *cross-layer contacts*, and combining them to get the total upper bound. *In-layer contacts* are contacts between H-monomers in a single layer, while *cross-layer contacts* are contacts between H-monomers in adjacent layers. For instance, the layers in Figure 3.6 has 5 in-layer contacts in the first layer, 4 in-layer contacts in second layer, and up to 4 cross-layer contacts between them.

In-layer Bound To define an upper bound for the in-layer contacts we use the notion of *surface*. Formally, two lattice points v, w form a *surface pair* of a core c if $v \in c \wedge w \notin c \wedge dist(v, w) = 1$, and we say that the *surface of* c is the number of surface pairs. Similarly, the surface of a single layer l of c is the number of pairs v, w so that $v, w \in l$ and $v \in c$ and $w \notin c$ and $dist(v, w) = 1$. If $w \in c$ then v, w is a *contact pair*. See Figure 3.6.

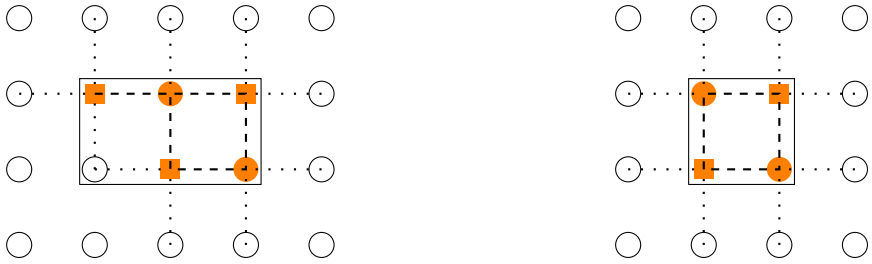


Figure 3.6: A layer deomposition of the core in Figure 1.2, and the corresponding frame sequence. Dashed lines indicate contact pairs while dotted lines indicate surface pairs

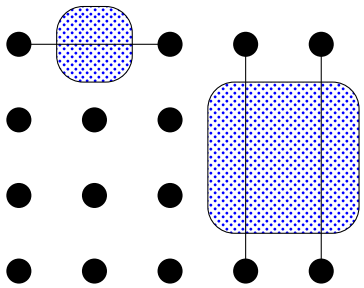


Figure 3.7: Example of a core layer with two cavities. Each point p in the cavities are intersected by a line that has core elements on either side of p, and is parallel to an axis.

Now observe that for an H-monomer $v \in c$ of a layer, all 4 of its neighbors within the layer must form either a surface pair or a contact pair with v. The layer surface and in-layer contacts are therefore related by equation (3.2).

$$2 \cdot \texttt{Contacts} + \texttt{Surface} = 4 \cdot n \tag{3.2}$$

Let $u_i = (a_i, b_i, n_i)$ be the frame of the layer. A lower bound for the surface of the layer is then given by $2 \cdot (a_i + b_i)$, cf. Figure 3.6. This bound is tight when the layer has no cavities, i.e. no "holes" in the structure.

Formally, we can say that a cavity is a connected set of points not in the core, so that for each point p, there is a line parallel to one of the axes that intersects p as well as one or more H-monomers on both sides of p. See Figure 3.7.

This lower bound coupled with equation (3.2) gives us an upper bound on in-layer contacts:

$$\texttt{Contacts}_i \leq 2n_i - (a_i + b_i)$$

Full Bound The upper bound for cross-layer contacts between layers with frames $r_i = (a_i, b_i, n_i)$ and $r_{i+1} = (a_i, b_i, n_i)$ is simply $\min(n_i, n_{i+1})$. Combined with the in-layer bound, this gives a bound on the total number of contacts possible in a core c, given only its frame sequence $(a_1, b_1, n_1), (a_2, b_2, n_2) \dots (a_l, b_l, n_l)$:

$$\texttt{Contacts}(c) \leq \text{in-layer bound} + \text{cross-layer bound} = \sum_{i=1}^{l} \big(2n_i - (a_i + b_i)\big) + \sum_{i+1}^{l-1} \big(\min(n_i, n_{i+1})\big) \tag{3.3}$$

3.4.3 Frame Sequences with upper bound c or more

Recall that the goal is to enumerate every core of size n, with at least c contacts. We will start by finding every frame sequence of n monomers, whose cores *could* have c or more contacts, i.e. the ones were the upper bound is greater than or equal to c. This is done through the use of *dynamic programming*. A dynamic programming algorithm solves a problem instance by using solutions to smaller versions of the problem. Each solution is computed only once and is then stored in memory, allowing the algorithm to recall solution in constant time in all subsequent problems. The storing of results is a trade-off that lowers the running time but increases the storage space. We will now see how to use dynamic programming to find frame sequences with an upper bound of c or more.

For a dynamic programming algorithm to work, we must know three things:

1. How to solve a problem instance using solutions to simpler instances.
2. An ordering of problem instances s.t. every instance only requires solutions to the ones before it.
3. The solutions to a set of simple instances that can be used as a starting point.

These are described in the following, along with some time-saving details, namely:

1. Which table entries we do not need to compute.
2. How to fill the table in $\mathcal{O}(n^5)$ time, where n is the size of the core.

The process is then demonstrated with an example on page 42.

The DP Table Define a 4-dimensional table B, where each entry of the form (n, n_1, a_1, b_1) contains the maximal upper bound for a frame sequence with n total monomers, whose first frame is (n_1, a_1, b_1). This upper bound is the bound for the number of in-layer contacts in the first frame plus, maximizing over all possible second frames, the number of cross-layer contacts between frame

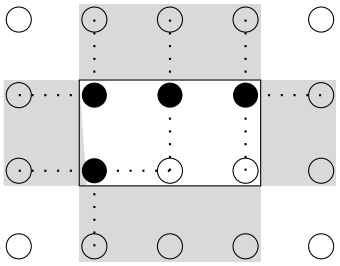


Figure 3.8: A core of size 4 and its minimally surrounding rectangle. The shaded areas are the regions reachable with one additional row or column in the frame. The dotted lines indicate which new points would be connected to the core. Encompassing any connected new point takes at most one additional row or column.

one and two, plus the upper bound for the equivalent problem beginning with frame two. This is calculated as shown in (3.4)

$$\begin{aligned} B(n, n_1, a_1, b_1) &= B_{ILC}(n_1, a_1, b_1) + \max_{n_2, a_2, b_2} (B_{CLC}(n_1, a_1, b_1, n_2, a_2, b_2) + B(n - n_1, n_2, a_2, b_2)) \\ &= 2n - (a_1 + b_1) + \max_{n_2, a_2, b_2} (\min(n_1, n_2) + B(n - n_1, n_2, a_2, b_2)) \end{aligned} \tag{3.4}$$

Skipping Table Entries Each dimension of the table has length $\mathcal{O}(n)$, but we can define some limits for the values that need to be considered, so that we only fill out a fraction of the table. If any frame sequence we can encounter necessarily has some mathematical property, then we do not need to calculate the table entries that violate the property. We will now define five such properties, which are used as rules when filling the table.

Rule 1: n_1 should be less or equal to n since a layer cannot have more H-monomers than its corresponding core.

Rule 2: a_1 and b_1 are both less or equal to n_1 since the rectangle is minimally surrounding

Rule 3: there is no functional difference between values a_1 and b_1, meaning that $\forall a_1, b_1 : B(n, n_1, a_1, b_1) = B(n, n_1, b_1, a_1)$. It is therefore enough to only consider the cases where $a_1 \geq b_1$.

Rule 4: If the layer is connected then $a_1 + b_1 \leq n_1 + 1$. Figure 3.8 illustrates why.

Rule 5: Since the area of a frame is no less than the number of elements contained in it, we require that $a_1 b_1 \geq n_1$.

Be aware that Rule 4 assumes that *all layers are connected*. I believe this is a fair assumption when the cores we seek are nearly contact-optimizing, but have not formalized an argument for it. Note that the core gets included if *any* of its frame sequence decompositions adhere to the assumption.

To pursue strict formality, one may therefore want to discard Rule 4 or analyze the soundness of the assumption.

Any of the rules can be discarded with no asymptotic difference to the running time.

Combining Rule 1-5, we get

$$\begin{aligned} b_1 \leq a_1 \leq n_1 \leq n \\ a_1 + b_1 - n_1 \leq 1 \\ n_1 \leq a_1 b_1 \end{aligned} \tag{3.5}$$

Which significantly reduces the number of table entries to be filled, as well as the number of calculations needed for each entry.

Ordering To compute an entry $B(n, n_1, a_1, b_1)$ we must first fill every entry with core size $n - n_1$. Once entries $B(0, 0, 0, 0) = 0$ and $B(1, 1, 1, 1) = 0$ are provided as a starting step, the table can be filled out in order of increasing n. Equation (3.4) shows how an entry is computed, while (3.5) shows which entries to compute given some n. Figure 3.9, page 42 shows computations for entries $n \leq 5$.

$\mathcal{O}(n^5)$ Shortcut Filling table B requires computing $\mathcal{O}(n^4)$ cells, where each cell requires time $\mathcal{O}(n^3)$ for a total of $\mathcal{O}(n^7)$ if one implements the computation in Equation (3.4). We can, however, reduce the computation time for each entry in B from $\mathcal{O}(n^3)$ to $\mathcal{O}(n)$ by storing another table M of size $\mathcal{O}(n^2)$. In Equation (3.4), we only use a_2 and b_2 for scanning previous entries in B. Instead of maximizing over n_2, a_2, b_2 in we can therefore maximize of just n_2 if we store a table $M(n, n_1)$ so that

$$M(n, n_1) = \max_{a_1, b_1}(B(n, n_1, a_1, b_1))$$

As an example of how to do this, consider the calculations in Figure 3.9, p. 42. After each calculation $B(4, 1, 1, 1) \ldots B(4, 4, 4, 1)$ we would check in constant time whether $B(n, n_1, a_1, b_1) \geq M(n, n_1)$ is true, and set $M(n, n_1) \leftarrow B(n, n_1, a_1, b_1)$ if it is. Then the calculation of $B(5, 1, 1, 1)$ would instead be (3.6).

$$(5, 1, 1, 1) = \max(0 + 1 + M(5, 1), 0 + 1 + M(5, 2), 0 + 1 + M(5, 3), 0 + 1 + M(5, 4)) \tag{3.6}$$

Since the in-layer and cross-layer contact bounds (the 0's and 1's in (3.6)) do not depend on a_2 and b_2, no information is lost in this simplification. The maximization in (3.4) now has only one running variable of size $\mathcal{O}(n)$, rather than three. We can therefore calculate all $\mathcal{O}(n^4)$ entries in time $\mathcal{O}(n^5)$, rather than $\mathcal{O}(n^7)$.

This of course requires the extra space of storing the size $\mathcal{O}(n^2)$ table M, but seeing as we already need to store a size $\mathcal{O}(n^4)$ table, I think one would favor this trade-off unless memory is scarcer than time.

Backtracking To find all frame sequences with upper bound at least C, we backtrack through the table to recover the frames that resulted in the upper bounds.

To do this, scan B in time $\mathcal{O}(n^4)$ to find every entry equal or greater than C. For each of those entries, start the recursion in method 4, p. 40 with an empty frame sequence σ.

Method 4 BacktrackRecursion

Input:
`FrameSequence` σ ▷ List of frames (triplets of integers)
`Integers` n, n_1, a_1, b_1 ▷ $B(n, n_1, a_1, b_1)$ must be non-empty

$\sigma \leftarrow \sigma \oplus (n_1, a_1, b_1)$ ▷ appends a frame to the sequence
if $B(n, n_i, a_i, b_i) = 0$ **then**
  return σ
for each non-empty cell $(n - n_1, n_2, a_2, b_2)$ that satisfies

$$B(n - n_1, n_2, a_2, b_2) + \min(n_1, n_2) + (2 \cdot n_1 - a_1 - b_1) = B(n, n_1, a_1, b_1)$$

do
  `BacktrackRecursion`$(\sigma, n - n_1, n_2, a_2, b_2)$
end for

This takes time $\mathcal{O}(n^3 \cdot m \cdot l)$ where m is the number of frame sequences and l is the length of the longest frame sequence.

Finding the Cores The backtracking algorithm returns *every* frame sequence of the given n whose core has bound at least C. Therefore any core of size n with C or more contacts, necessarily satisfies one of those frames. We can therefore find them by searching the space of cores that satisfies those frame sequences, which is much, much, smaller than the space of all cores of size n.

In practice, this search can be done by encoding the relationship of a core satisfying a frame sequence, as *constraints* in a *Constraint Satisfaction Program*, cf. Section 3.2. The solution to this CSP is a set of coordinates like in the threading problem. The only other constraint that has to be modeled is that the contact number of the coordinate set should be C or more.

A coordinate set that satisfies the latter "contact constraint" meets our criteria for cores. This models the problem completely, and is sufficient given unlimited computing power. The function of the "frame sequence constraint" is that its propagation narrows down the search space so much that it becomes realistic to enumerate it.

3.5 Discussion

Length Restrictions Being a search algorithm, CPSP faces challenges of restrictive folding times as input size grows. This would not matter much if one could feasibly fold sequence of, say length 1000, since that is enough for most actual proteins of interest.

As it stands, the fast and well-propagating threading implementation by Will and Backofen can fold sequences of approximately 200 length. While this is certainly in the right ballpark, it is barely not as much as we would have liked. Brocchieri and Karlin analyzed the length of proteins and found that the median length of proteins in 67 bacteria was about 270, while it was 360 across five eukaryotes, and 375 in humans. [4]

Given the exponential nature of the problem, the 200 figure will not soon be outpaced by Moore's law of exponentially increasing computing power. We would therefore need to have more cunning innovations applied to the problem to find exact folds to most proteins. The conformations would of course only need to be computed once, perhaps as a distributed computing project, and would then be stored in online databases. Even if threading of this length was possible, finding the necessary H-cores is however an even more demanding computation that must first be undertaken.

For now, 200 is enough to fold some proteins, and perhaps CPSP could be used to folds segments of larger proteins to learn about their structure.

Other Models Just by virtue of bringing exact solutions, the utility of CPSP far surpasses that of S38. That said, the inability of cubic 3D model to resemble real folding is a weak point.

CPSP can, however, be extended to lattices other than the cubic lattice, if one can solve the problem of computing the cores for that lattice. Backofen and Will also implemented CPSP for the *Face Centered Cubic Lattice*, which represents a more dense packing of monomers, and also does not have the parity issue.

$$B(0,0,0,0) = 0$$
$$B(1,1,1,1) = 0$$

$$B(2,1,1,1) = \max(0+1+B(1,1,1,1)) = 1$$
$$B(2,2,2,1) = \max(1+0+B(0,0,0,0)) = 1$$

$$B(3,1,1,1) = \max(0+1+B(2,1,1,1),\ 0+1+B(2,2,2,1)) = 2$$
$$B(3,2,2,1) = \max(1+1+B(1,1,1,1)) = 2$$
$$B(3,3,2,2) = \max(2+0+B(0,0,0,0)) = 2$$

$$B(4,1,1,1) = \max(0+1+B(3,1,1,1),\ 0+1+B(3,2,2,1),\ 0+1+B(3,3,2,2)) = 3$$
$$B(4,2,2,1) = \max(1+1+B(2,1,1,1),\ 1+2+B(2,2,2,1)) = 4$$
$$B(4,3,2,2) = \max(2+1+B(1,1,1,1)) = 3$$
$$B(4,3,3,1) = \max(2+1+B(1,1,1,1)) = 3$$
$$B(4,4,2,2) = \max(4+0+B(0,0,0,0)) = 4$$
$$B(4,4,3,2) = \max(3+0+B(0,0,0,0)) = 3$$
$$B(4,4,4,1) = \max(3+0+B(0,0,0,0)) = 3$$

$$B(5,1,1,1) = \max(0+1+B(4,1,1,1),\ 0+1+B(4,2,2,1),\ 0+1+B(4,3,2,2),\ 0+1+B(4,3,3,1),\ 0+1+B(4,4,2,2),\ 0+1+B(4,4,3,2),\ 0+1+B(4,4,4,1)) = 5$$
$$B(5,2,2,1) = \max(1+1+B(3,1,1,1),\ 1+2+B(3,2,2,1),\ 1+2+B(3,3,2,2)) = 5$$
$$B(5,3,2,2) = \max(2+1+B(2,1,1,1),\ 2+2+B(2,2,2,1)) = 5$$
$$B(5,3,3,1) = \max(2+1+B(2,1,1,1),\ 2+2+B(2,2,2,1)) = 5$$
$$B(5,4,2,2) = \max(4+1+B(1,1,1,1)) = 5$$
$$B(5,4,3,2) = \max(3+1+B(1,1,1,1)) = 4$$
$$B(5,4,4,1) = \max(3+1+B(1,1,1,1)) = 4$$
$$B(5,5,3,2) = \max(5+0+B(0,0,0,0)) = 5$$
$$B(5,5,3,3) = \max(4+0+B(0,0,0,0)) = 4$$
$$B(5,5,4,2) = \max(4+0+B(0,0,0,0)) = 4$$
$$B(5,5,5,1) = \max(4+0+B(0,0,0,0)) = 4$$

Figure 3.9: Demonstration of the pattern for computing a Dynamic Programming table to find bounds on contacts of cores. Compute from top to bottom, starting with the left side.

Section 4

Experiments

Here we discuss the experiments that were run on S38 and CPSP, and their results.

The BFSP and S38 algorithms were written in C++ and compiled using `Microsoft C/C++ Optimizing Compiler` version `19.00.24215.1`. The CPSP algorithm was written in python 2.7, and the CSP's were solved with the `MiniZinc G12/FD` solver for finite domain problems.

The programs were executed on an `Intel Core i5-6300HQ` CPU running at clock rate 2.30 GHz.

Plotting and fitting the data was done using `MatPlotLib` library for python.

4.1 S38 Experiments

S38 is tested with respect to correctness and asymptotic running time in this section. I also compare its running time to the algorithms that were implemented in my previous project. [6]

The running times were measured on procedurally generated strings of length $1000, 2000, \ldots 50'000$ to confirm the $\mathcal{O}(n)$ asymptotic time argued in Section 2.2. It was also measured on strings of length $50, 100, \ldots 950, 1000$ which are more realistic protein lengths. The algorithms perform equally well on random sequences as on real HP-sequences, since they do not make use of any properties of real proteins. The plots are shown on page 44. To preserve the naming used in [6], Algorithm 2 and Algorithm 3 refer to implementations of the 2D and 3D method based on [5], and Algorithm 1 refers to a straightforward implementation of the 2D method. The plots support the $\mathcal{O}(n)$ running time.

The score of S38 and Algorithm 3 along with their upper and lower bounds is seen in Figure 4.2. As expected, the scores are within the bounds obtained in the analysis.

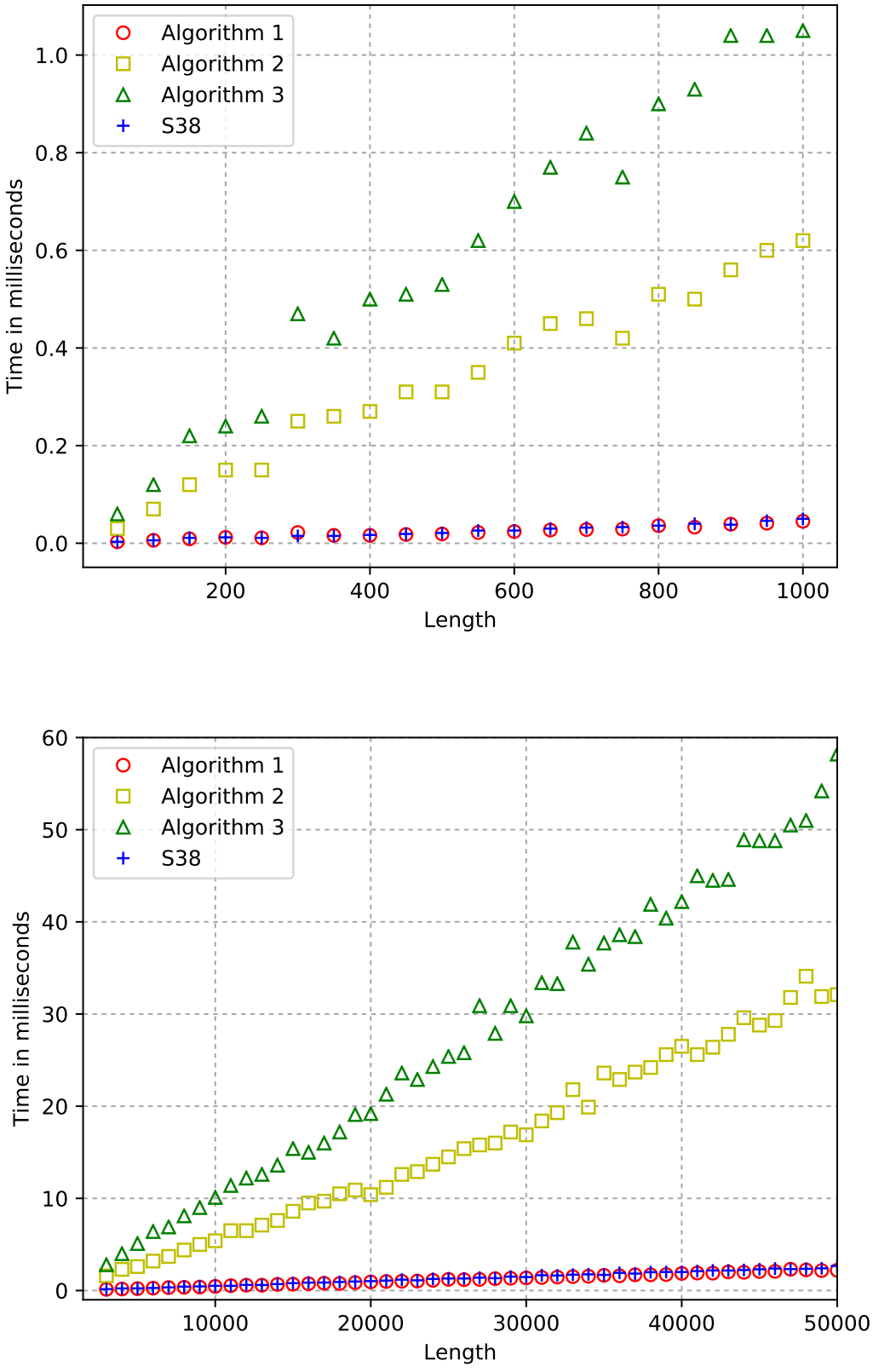


Figure 4.1: Running time comparison of S38 to algorithms in [6] for respectively lengths $50 - 1000$ and $2000 - 50'000$. Algorithm 1 is a simple implementation of the 2D approximation algorithm. Algorithm 2 and 3 are my 2D and 3D implementations based on the Hart and Istrail paper. [5] Algorithm 2 and 3 were inefficient because I inappropriately implemented structures from the paper that were not necessary.

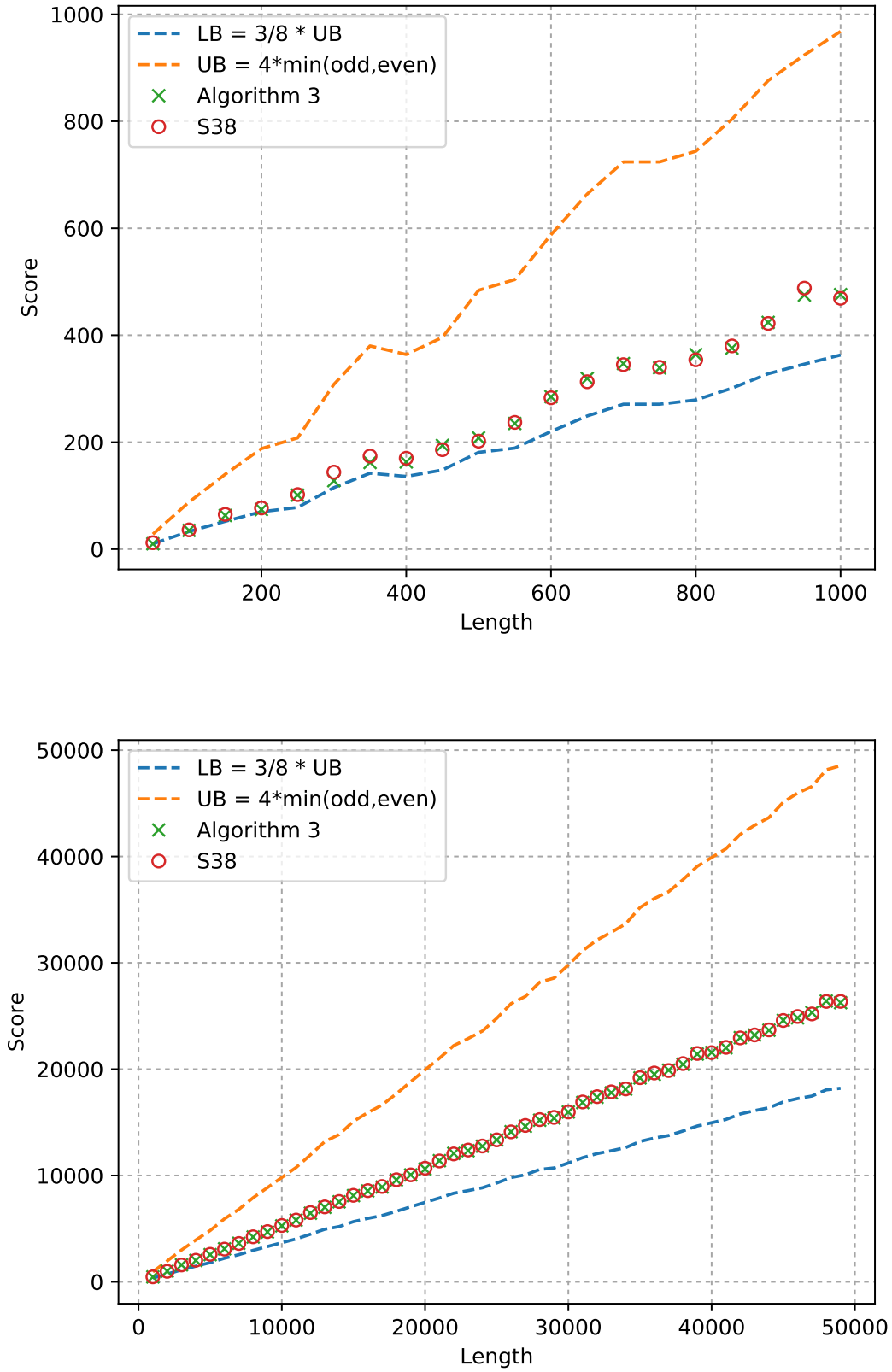


Figure 4.2: Scores of S38 and Algorithm 3 for respectively lengths $50 - 1000$ and $2000 - 50'000$. The scores are within the bounds given by the analysis.

4.2 CPSP Experiments

Since CPSP does not improve upon the asymptotic worst case time, we shall gauge its value, by comparing it to a naive brute force algorithm. For this purpose, a brute force structure prediction algorithm we shall call BFSP is introduced in the next section, and will serve as a benchmark for the running time of CPSP.

4.2.1 Brute Force with BFSP

An easy exact algorithm is to just evaluate the score for each of 6^{n-1} possible fold descriptions of correct length, and pick (one of) the best. But though brute forcing is not meant to be sophisticated, this is however quite unnecessary, as we can make use of an important shortcut that takes only a little effort.

Not all of the 6^{n-1} fold descriptions are legal folds, and some groups of illegal folds are easy to identify. In particular, if a fold has a prefix that is itself an illegal fold, then the entire fold is also illegal. When enumerating the folds we can therefore check the prefixes, and if they are illegal, skip past all folds that share the illegal prefix. As an example, say that we are enumerating fold descriptions of length 10 to find the maximal scoring one. Say we have finished evaluating `"URUFFFFFFF"` and are to evaluate `"URDLLLLLLL"`. Since `"URDL"` is not a self avoiding walk, we skip to `"URDR"` (or whatever our enumeration scheme dictates), avoiding 6^6 score evaluations. Since `"URDR"` is self avoiding, we "branch down" another character, to `"URDRL"`. This fold is also illegal, meaning we can skip to "URDRR", avoiding another branch of 6^5 evaluations, and so on.

The effectiveness of this adjustment scales up with input length. Any fold coordinate beyond the first has at least one illegal character of the six possible, since it will be directly opposite of the one before it. For a fold of length i, that means we must evaluate no more than $(\frac{5}{6})^{i-1}$ of the total number of folds which interestingly approaches 0 as $i \to \infty$. For a fold of length 10, this upper bound is already below 20%. The actual proportions of valid folds for smaller lengths are shown in Figure 4.3.

The search space of BFSP is still exponential when using the skipping scheme. A simple argument is that any fold consisting only of, say, `UP` and `RIGHT` is valid and cannot be skipped. There is therefore at least 2^n legal folds, so the time is still exponential.

Note that for BFSP, the score must be evaluated on each candidate, which means the running time of the algorithm depends on the running time of score counting algorithm. Therefore the next section will quickly visit the subject of how to do this in linear time using hashmaps.

Length	Folds	Valid	%
1	6	6	100%
2	36	30	83%
3	216	150	70%
4	1296	726	57%
5	7776	3534	46%
6	46656	16926	37%
7	279936	81390	30%
8	1679616	387966	24%
9	10077696	1853886	19%
10	60466176	8809878	15%
11	362797056	41934150	12%
12	2176782336	198842742	9%

Figure 4.3: For strings of significant length, a considerable fraction of the possible folds can ruled out as illegal.

4.2.2 Score Counting in Linear Time

One of the obvious "human" methods for score counting may be to first draw the folded protein. Then going through the H's in order, one could add up their HH-connections while marking each H already visited. This simple approach is much like the one we use for BFSP:

First, the fold descriptions are translated into a set of points, and the hashes for each point are computed and used as indexes into a hash-table. Then we store their respective amino acids on those indexes in the table, putting 1 for "H" and 0 for "P". For any H stored, we can then produce the coordinates that correspond to its neighbors, and add up the numbers stored at those indexes. Removing the current H after scoring it amounts to marking it as "used".

The score is the total sum of HH-connections found this way, minus the number of "HH" occurrences in the sequence, including overlap.

It should still be possible to make score counting algoirthm multiple times faster. For one thing, using BFSP and the scoring function in this way means that the hashmaps are destroyed and created between each fold evalutation, though most of the time they represent almost identical coordinates. An idea is for BFSP to have a continuous representation of the point sets, that is then updated between each new fold. Perceivably one could also save sub-results of the score counting, so that only the changed part has to be counted again.

With BFSP in position to be used as a benchmark, we will now investigate how it compares to CPSP.

4.2.3 Running times

The algorithms were tested on different types of strings. Some of them are actual peptides picked from the "PepBank" by Massachusetts general Hospital [7]. Some were HP sequences used by Istrail to benchmark scores of 2D algorithms, [8] and some were generated randomly using `Random.org`.

The peptides were translated using data from the website of Sigma Aldrich [1]. The translation table is shown in Table 4.1.

Amino Acid	Hydrophobicity
F	H
I	H
W	H
L	H
V	H
M	H
Y	H
C	H
A	H
T	H
H	H
G	P (Actually neutral)
S	P
Q	P
R	P
K	P
N	P
E	P
P	P
D	P
Z (E/Q)	P
B (N/D)	P
X (Unknown)	Discarded Sequence

Table 4.1: Hydrophobicity chart at pH=7, based on the chart on the website of Sigma Aldrich. [1] Glycine (G) is neutral, but was arbitrarily assigned to P. The two ambiguous amino acid codes are both hydrophilic. Two sequences had unknown constituents and were discarded.

I based the table on a reference chart published as a learning resource by the chemical and biotechnology company Sigma Aldrich. [1] I used the `Hydrophobicity Index for Common Amino Acids` and the column for `pH`= 7.

Results Here we interpret the results of the CPSP expteriments. Table 4.2, p. 50, shows all running times for BFSP and CPSP. They are also plotted in Figure 4.4, p. 51.

First we explain the table and comment on the some of the data.

The columns, from left to right, contain:

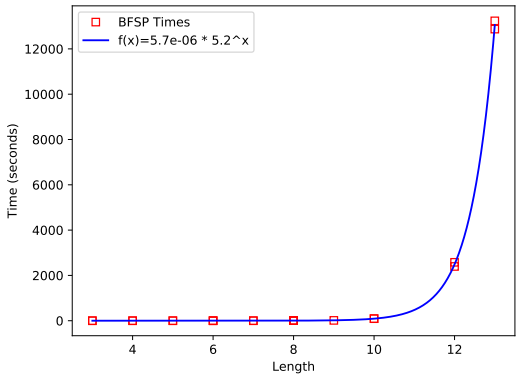
- **Origin:** The origin of the string. If it was an actual peptide, the sequence of the peptide is indicated.
- **Sequence:** The HP sequence that was tested on.
- **Length:** The length of the sequence.
- **BFSP:** The BFSP running time.
- **CPSP:** The CPSP running time.
- **Cores:** The number of cores considered by CPSP.
- **Threaded:** The number of threadings performed by CPSP.
- **%:** The ratio of threadings to cores considered by CPSP.

Some comments about the data in Table 4.2:

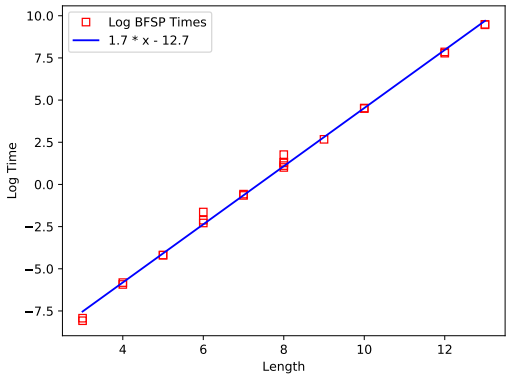
- Two of the peptide sequences had unknown monomers, denoted by the amino acid code `X`. I discarded them from the test, and found other sequences of the same length.
- Some fields are marked with `N/A`. This means one of the following: "The data is not available because the string was discarded.", "The data is not available because the experiment took too long to carry out", or "The ratio is not applicable because no threading was performed." The latter happens when the optimal score is 0.
- A notable sequence was the third random sequence, `PPPPHHPH` that took unexpectedly long to thread with CPSP. Only one second was spent on the successful threading, the rest was used trying to thread the first core. It seems like some types of threadings are particularly slow due to reasons unclear to me.
- One of the real peptide strings skipped 231 cores before threading. This indicates that it is possible to get unlucky with your cores using real data, and not just some contrived example that was dreamed up by an adversary. If we had been using a different lattice, the skipping scheme is likely not applicable either, meaning that it could be possible to have a realistic application scenario where one has to thread a large number of cores before finding a solution. If no bound on the needed number of threadings exists, then there is no way of knowing whether the solution is imminent, or firmly out of reach. There is therefore a possibility that some real proteins, may be much more difficult than average to fold with CPSP.

Origin	Sequence	length	BFSP	CPSP	Cores	Threaded	%
AGG	HPP	3	0.00031 sec	0.001 sec	0	0	N/A%
EGR	PPP	3	0.00036 sec	0.001 sec	0	0	N/A%
AGFL	HPHH	4	0.0030 sec	0.24 sec	2	2	100%
AGPF	HPPH	4	0.0066 sec	0.11 sec	1	1	100%
ETKRS	PHPPP	5	0.15 sec	0.001 sec	0	0	N/A%
CRPPR	HPPPP	5	0.15 sec	0.001 sec	0	0	N/A%
DYKXXD	Discarded	6	N/A	N/A	N/A	N/A	N/A
CARPAR	HHPPHP	6	0.10 sec	0.13 sec	1	1	100%
DVAGGLG	HPPPPHP	7	0.52 sec	0.18 sec	1	1	100%
AEEGGTS	PHHPPHP	7	0.56 sec	0.16 sec	1	1	100%
GTVGLVLV	PPPHHPPP	8	2.7 sec	0.14 sec	1	1	100%
DPRATPGS	PHHPHHHH	8	3.6 sec	0.42 sec	2	2	100%
CRNSWKPNC	HPPPHPPPH	9	14 sec	0.001 sec	0	0	N/A%
ACTNQFLQQC	HHHPPHHPPH	10	1.5 min	0.14 sec	1	1	100%
AFNIRGDHWVPD	HPPHHHHPHPHH	12	43 min	4.7 sec	1	8	12%
AGDLVTFKLRLH	HHPHPPPHHHPP	12	40 min	3.5 sec	1	5	20%
AIEVQTVTLIPGD	HHPHPHHHHHPPP	13	3.67 h	0.19 sec	1	8	12%
AHSGWYSKPPPPIP	HHPPHHPPPPPPHP	14	17.4 h	3.1 sec	1	1	100%
AHSDAVTPLPARSKV	HHPPHHHPHPHPPPH	15	N/A	0.35 sec	1	232	0.4%
ACVNRSDGMCGGSYK	HHHPPPPPHHPPPHP	15	N/A	0.66 sec	1	1	100%
EMVLGPPVPPKRGTVV	HPPPHPPHHPPPPPPP	16	N/A	0.26 sec	1	4	25%
AGSRHPPTLPPKESGG	PHHHPPPHPPPPPHHH	16	N/A	9.5 sec	1	26	3%
AIDYDEDEDGRPKVHVDAR	HHPHPPPPPPPPPHHHPHP	19	N/A	15 h	1	25	4%
AIAGLTPPKESKAFAKXEKN	Discarded	20	N/A	N/A	N/A	N/A	N/A
ZPIDLMGKVFVFDKELSPBI	PPHPHHPPHHHHPPPHPPPH	20	N/A	1.4 h	3	3	100%
Random	HHPP	4	0.0038 sec	0.11 sec	1	1	100%
Random	HHPPHH	6	.12 sec	0.13 sec	1	1	100%
Random	PPPPHHPH	8	3.7 sec	7.3 min	2	2	100%
Random	PPHPHHPH	8	3.1 sec	19 sec	4	5	80%
Random	PHHPPPPPHH	10	93 sec	1.3 sec	1	1	100%
Random	HHHPPHHHHPHPP	13	3.5 h	1.06 sec	1	8	12.5%
Random	HHPPPHHHPPPHHH	14	17.3 h	0.16 sec	1	8	12.5%
Istrail site	HHPPPPPHHPPPHPPPHP	18	N/A	1.8 sec	1	5	20%
Istrail site	HPHPHHHPPPHHHHPPHH	18	N/A	3.1 min	1	22	4%
Istrail site	PHPPHPHHHPHHPHHHHH	18	N/A	23 min	3	3	100%
Istrail site	HPHPPHHPHPPHPHHPPHPH	20	N/A	1.3 h	24	36	66%
Istrail site	HHHPPHPHPHPPHPHPHPPH	20	N/A	23 min	7	12	58%
Istrail site	HHPPHPPHPPHPPHPPHPPHPPHH	24	N/A	8.3 sec	1	1	100%
Istrail site	PPHPPHHPPPPHHPPPPHHPPPPHH	25	N/A	27 h	1	1	100%

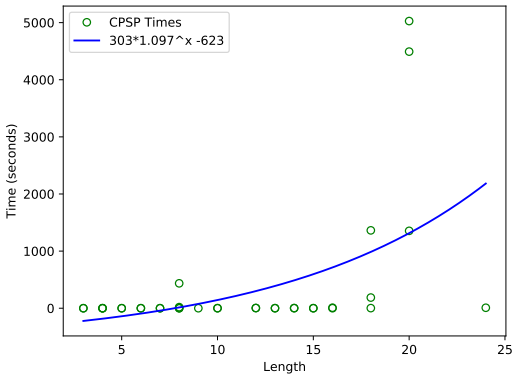
Table 4.2: Running times of BFSP vs CPSP. The blank fields were impractical to compute. Folding with S38 took less than 0.01 milliseconds on all of the above strings.



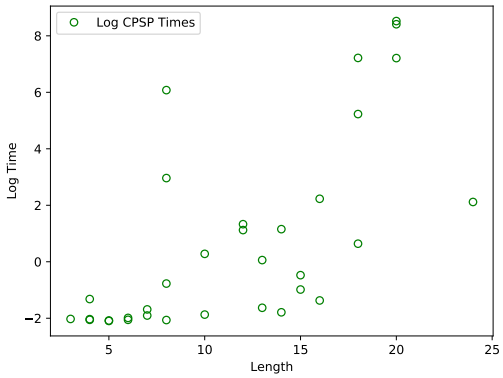
(a) Running times for BFSP



(b) Logarithm of running times for BFSP



(c) Running times for CPSP



(d) Logarithm of running times for CPSP

Figure 4.4: We expect the running times for both BFSP and CPSP to be exponential. The plots confirm that BFSP is exponential. For CPSP we cannot tell because the function is too complex and unpredictable, making the data "noisy".

Interpretation The plot in Figure 4.4a shows running time of BFSP, and Figure 4.4b shows the logarithm. The BFSP times closely follows an exponential curve. Since the algorithm works by entirely enumerating the search space, there are not many ways in which a fold can be found much faster or slower than the expected time. The measurements are therefore unsurprising. Though predictability of BFSP is an advantage, its running times are generally more restrictive than CPSP.

In Figure 4.4c and 4.4d we see the times and their logarithms for CPSP. Here we cannot tell whether the times are exponential because the data is too "noisy".

Interestingly, the actual *noise* of the CPSP plots is not greater than that of the BFSP plots. The

unpredictability comes not from randomness, but from the fact that the function describing the running time is very complex. We can think of the unpredictability as noise, though.

We still expect the running time to roughly follow an exponential curve. In fact, the alternative would be *very* surprising. But given the noise, we need more data for the plot to confirm it.

In summary the CPSP algorithm is faster and more unpredictable the BFSP. The longest sequence folded with CPSP was 25, compared to 14 for BFSP.

Section 5

Conclusion

Protein structure prediction is a hard problem. It is currently not feasible to find exact solutions in the HP model, even though it is an extreme simplification of the problem.

Finding conformations that score within $\frac{3}{8}$ of the optimal is easy, but being within $\frac{3}{8}$ does not mean that the conformation looks anything like the optimal. For this type of algorithm to be useful, the ratio of approximation needs to be much better.

Currently with CPSP, exact solutions can be found for sequence lengths in the 200's. This requires algorithm engineering and CP expertise. Since most proteins have lengths in the hundreds, predicting them does not seem unrealistic if the algorithm can be further developed or combined with other methods.

Even if it were possible, the complexity of a real protein is much greater than what is modeled in HP. But the goal is not to capture all of the complexity of real proteins, but just enough that their *functions* can be predicted. There might be useful models and heuristics that satisfactorily predicts key aspects without taking the full complexity into account.

Perspective Advancements may come from the current popularity of AI and many metaheuristics. These are characterized by being robust to changes in the model. For instance, *evolutionary algorithms* is being been studied and refined for lattice protein prediction using with varying energy functions [13, 10]

New computing technologies, particularly quantum computing, might also bring us closer to the goal. For example, the Aspuru-Guzik research group at Harvard University implemented an algorithm for prediction in lattice models using quantum annealing. [12] As technology advances, the problem may be more tractable to solve with quantum computing than with classical computing.

If we get very good at predicting proteins structures and functions, it likely involves advancements in several areas, like computing technology and molecular biology. These fields are advancing very fast, so it is hard to say what the state of PSP will be in the future.

Bibliography

[1] Sigma Aldrich. Hydrophobicity index for common amino acids. `http://www.sigmaaldrich.com/life-science/metabolomics/learning-center/amino-acid-reference-chart.html`, 2017.

[2] Rolf Backofen and Sebastian Will. A constraint-based approach to fast and exact structure prediction in three-dimensional protein models. *Kluwer Academic Publishers*, 2006.

[3] Rolf Backofen and Sebastion Will. Excluding symmetries in constraint-based search. 2002.

[4] Luciano Brocchieri and Samuel Karlin. Protein length in eukaryotic and prokaryotic proteomes. 2005.

[5] William E. Hart and Sorin Istrail. Fast protein folding in the hydrophobic-hydrophilic model within three eighths of optimal. *Proceedings of the 27th Annual ACM Symposium on the Theory of Computing*, 1995.

[6] Martin Bjerrum Henriksen. The three eights approximation algorithm for HP structure prediction on the 3D cubic lattice. `https://www.dropbox.com/s/xhqyn2bmqeyymls/three-eights-approximation.pdf?dl=1`, 2017.

[7] Massachusetts General Hospital. Pepbank. `http://pepbank.mgh.harvard.edu`, 2017.

[8] Sorin Istrail. The proFolding project: HP2D Benchmarks. `http://www.brown.edu/Research/Istrail_Lab/hp2dbenchmarks`, Brown Univesity, 2017.

[9] Kit Fun Lau and Ken A. Dill. A lattice statistical mechanics model of the conformational and sequence spaces of proteins. *American Chemical Society*, 1989.

[10] Mijajlovic M, Biggs MJ, and Djurdjevic DP. On potential energy models for ea-based ab initio protein structure prediction. 2013.

[11] Ashwin Nayak, Alistair Sinclair, and Uri Zwick. Spatial codes and the hardness of string folding problems. *Journal of Computational Biology*, 1998.

[12] Alejandro Perdomo-Ortiz, Neil Dickson, Marshall Drew-Brook, Geordie Rose, and Alán Aspuru-Guzik. Finding low-energy conformations of lattice protein models by quantum annealing. *Nature*, 2012.

[13] Jyh-Jong Tsay and Shih-Chieh Su. An effective evolutionary algorithm for protein folding on 3d fcc hp model by lattice rotation and generalized move sets. 2013.