FITTING PROTEIN CHAINS TO LATTICE USING INTEGER PROGRAMMING APPROACH

by

Xiaorong Huang
M.Econ, Southwestern University of Finance and Economics, 2001
B.Eng, Southwestern University of Finance and Economics, 1998

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

In the School of Computing Science

© Xiaorong Huang 2007

SIMON FRASER UNIVERSITY

Spring 2007

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.
Name: Xiaorong Huang
Degree: Master of Science
Title of Thesis: Fitting Protein Chains to Lattice Using Integer Programming Approach

Examinining Committee:
Chair: Ramesh Krishnamurti
Professor of Computing Science

Arvind Gupta
Senior Supervisor
Professor of Computing Science

Binay Bhattacharya
Supervisor
Professor of Computing Science

Daya Gaur
External Examiner
Associate Professor of Mathematics & Computer Science
University of Lethbridge

Date Defended/Approved: Dec 18, 2006
DECLARATION OF PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the "Institutional Repository" link of the SFU Library website <www.lib.sfu.ca> at: <http://ir.lib.sfu.ca/handle/1892/12>) and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada
ABSTRACT

Fitting Protein chains to Lattice problem (FPL) can be formulated as follows. Given the 3D-coordinates of all $C_{\alpha}$ atoms in a protein fold, find the optimal lattice approximation (self-avoiding path in the lattice) of the fold. The FPL problem is proved to be NP-complete for cubic lattice with side length 3.8Å when the coordinate root mean-square deviation (cRMS) is used as a similarity measure between original and approximated fold. We design three Integer Programming (IP) formulations for FPL problem, and generate a serial of algorithms which combine dynamic programming and backtracking techniques aiming to reduce the search space, while guaranteeing to find optimal solutions. Experiments show that optimal lattice approximations in cubic lattices with side length 3.8Å using cRMS measurement can be found in feasible time by ILOG CPLEX 9.1 for all proteins in a randomly selected group of proteins (longest of length 1014 residues).
ACKNOWLEDGEMENTS

Thanks are due to all members of my supervisory committee: Dr. Arvind Gupta, Dr. Binay Bhattacharya and Dr. Daya Gaur. I thank my senior supervisor Dr. Arvind Gupta for the guidance, advice and support he has given me all through my training program. Dr. Binay Bhattacharya supervised my thesis research, which was benefited from his insightful questions. I owe thanks to Dr. Daya Gaur and Dr. Ramesh Krishnamurti for their inspirations and penetrating comments which helped me improve the IP models. I owe particular thanks to Dr. Jan Manuch for his enlightening ideas, tremendous suggestions, unbelievable patience and unfailing encouragement that helped me in all the time of my research and writing of this thesis. I am also grateful to Dr. Robert Benkoczi of Queen’s University, whose comments and discussion let me have better understanding of linear programming and help to improve the IP models.

Special thanks are owed to my parents, whose love and support throughout my education years has been unconditional and unbroken. I am also grateful to my boyfriend Dennis for his spiritual support and understanding.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Approval</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iv</td>
</tr>
<tr>
<td>Table of Figures</td>
<td>v</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vi</td>
</tr>
<tr>
<td>Chapter 1: Introduction</td>
<td>vii</td>
</tr>
<tr>
<td>1.1 Molecular Biology Background</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Drug Design and HP Model</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Motivation</td>
<td>3</td>
</tr>
<tr>
<td>Chapter 2: Overview of Fitting Protein Chains to Lattice Problem</td>
<td>6</td>
</tr>
<tr>
<td>2.1 3D FPL Problem</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Overview of Existing Results</td>
<td>8</td>
</tr>
<tr>
<td>Chapter 3: Basic LP Formulations</td>
<td>13</td>
</tr>
<tr>
<td>3.1 ILOG CPLEX</td>
<td>13</td>
</tr>
<tr>
<td>3.2 IP Vertex Model</td>
<td>17</td>
</tr>
<tr>
<td>3.3 IP Edge Model I</td>
<td>27</td>
</tr>
<tr>
<td>3.4 IP Edge Model II</td>
<td>30</td>
</tr>
<tr>
<td>3.5 Efficiency of the IP Models</td>
<td></td>
</tr>
<tr>
<td>Chapter 4: Narrowing Down the Search Space</td>
<td>34</td>
</tr>
<tr>
<td>4.1 Sight Distance and Sight Sets</td>
<td>34</td>
</tr>
<tr>
<td>4.2 Revising IP Models by Using Sight Sets</td>
<td>36</td>
</tr>
<tr>
<td>4.2.1 Revised IP Vertex Model</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2 Revised IP Edge Model I</td>
<td>37</td>
</tr>
<tr>
<td>4.2.3 Revised IP Edge Model II</td>
<td>39</td>
</tr>
<tr>
<td>4.3 Determining Safe Sight Distances</td>
<td>40</td>
</tr>
<tr>
<td>4.3.1 Finding ( c_{dist} ) Upper Bound for the Optimal Lattice Approximation</td>
<td>42</td>
</tr>
<tr>
<td>4.3.2 Finding ( i-th ) Complimentary ( c_{dist} ) Lower Bound for the Optimal Lattice Approximation</td>
<td>46</td>
</tr>
<tr>
<td>4.3.2.1 Use of Closest Lattice Vertices</td>
<td>46</td>
</tr>
<tr>
<td>4.3.2.2 Use of Tuples</td>
<td>47</td>
</tr>
<tr>
<td>4.3.2.3 Use of Mega Tuples</td>
<td>49</td>
</tr>
<tr>
<td>4.3.2.4 Use of Groups of Tuples and Iterations</td>
<td>52</td>
</tr>
</tbody>
</table>
4.4 Refining Sight Sets .................................................. 56
  4.4.1 Refining Sight Sets by Checking Neighbourhood Connections ........................................ 57
  4.4.2 Refining Sight Sets by Backtracking - Dynamic Programming Algorithm ......................... 58
     4.4.2.1 General Strategy .................................................. 59
     4.4.2.2 Basic Techniques .............................................. 62
     4.4.2.3 Use of Seeds .................................................... 70
  4.5 Determining Sizes of Lattices ................................... 77

Chapter 5: Experimental Results ........................................ 81
  5.1 Database Schema and Testing Set Selection .................. 81
  5.2 Lattice Generation ................................................... 83
  5.3 Performance Comparison .......................................... 86

Chapter 6: Conclusions and Future Work ............................ 89

Appendices ........................................................................... 90
  Appendix A: Definitions .................................................. 90
     A.1 Linear Programming ................................................ 90
     A.2 Graph Theory ....................................................... 91
     A.3 Miscellaneous ....................................................... 93
  Appendix B: Algorithms ................................................... 94
     B.1 Description of Algo(4 - 1) .................................... 94
     B.2 Description of Algo(4 - 3) ................................. 102
     B.3 Description of Algo(4 - 6) ................................. 103
     B.4 Description of Algo(4 - 7) ................................. 108
     B.5 Description of Algo(4 - 8) ................................. 112

Reference List ......................................................................... 115
LIST OF FIGURES

Figure 3.1 Number of lattice vertices \((Q)\) is linear to the number of residues in protein \((P)\) ................................................................. 32

Figure 4.1 An example of tuples and mega tuples in a protein ...................... 50

Figure 4.2 Change of the average sight distances according to the mega tuple size ................................................................. 53

Figure 4.3 General Strategy of BT- DP Algorithm ............................................. 62

Figure 4.4 Illustration on how to compute \(LCT(t_{\alpha}, \star)\), where \(t_{\alpha} = p_{\alpha_1} \ldots p_{\alpha_k}\) .......... 69

Figure 4.5 Illustration on how to compute \(\text{dist}^{\text{corrected}}(0, P - I)\), where \(q_j \in \text{vsight}(p_i), p_i \in t_{\alpha} \) .............................................. 70

Figure 4.6 Number of tuples and boundaries of tuples before and after rearranging ................................................................. 74

Figure 5.1 Distribution of lengths of chains over all proteins in PDB .............. 81

Figure 5.2 Distribution of lengths of chains over all proteins in database .......... 82

Figure 5.3 \(\text{MAX}\{s_d(p_i) \mid p_i \in V_G\}\) in all proteins \(G_L\) in the testing set .......... 84

Figure 5.4 Maximum, average and minimum span along all axes of all protein spaces in the testing set and their corresponding lattice spaces ................................................................. 85

Figure 5.5 Ratio of average span of lattice space to that of the corresponding protein space ................................................................. 86

Figure 5.6 Comparison of cRMS values of all the proteins in the test set by the greedy algorithm and \(\text{Algo}(4-10)\) ..................................................... 87

Figure 5.7 Improvement of cdist value got by \(\text{Algo}(4-10)\) compared to greedy algorithm ................................................................. 88

Figure 5.8 Improvement of computational time used by \(\text{Algo}(4-10)\) compared to greedy algorithm ................................................................. 88

Figure 5.9 Data structures used in \(\text{Algo}(4-1)\) and their relations ............... 95

Figure 5.10 Illustration of SDIST\((p_i, p_j, \star, \star)\) .............................................. 104

Figure B.1 Data structures used in \(\text{Algo}(4-1)\) and their relations ............... 95

Figure B.2 Illustration of SDIST\((p_i, p_j, \star, \star)\) .............................................. 104

Figure B.3 Illustration of SDIST\((p_i, p_j, \star, \star)\) .............................................. 104
LIST OF TABLES

Table 3.1 $N_{va}$, $N_{cont}$ and $N_{occu}$ in the three IP models .......................... 31
CHAPTER 1: INTRODUCTION

1.1 Molecular Biology Background

In chemistry, a monomer is a small molecule that may become chemically bonded to other monomers to form a polymer. Amino acids are natural monomers which polymerize to form proteins. An amino acid is any molecule that contains both amine and carboxyl functional groups. In biochemistry, the term amino acid is used to refer to alpha amino acids: molecules where the amino and carboxylate groups are attached to the same carbon, the α-carbon, which also represented by Cα. Proteins are large organic compounds made of amino acids arranged in a linear chain and joined together between the carboxyl atom of one amino acid and the amine nitrogen of another. This bond is called a peptide bond. The portion in between any two consecutive peptide bond which corresponds to a amino acid is called a residue. The primary structure of a protein is the amino acid sequence. In an aqueous environment, a protein usually folds into a 3-dimensional structure that we call the spatial conformation of the protein. A protein’s biological functions are strongly determined by its spatial conformation, and the spatial conformation is mainly determined by the protein’s primary structure. Studies show that a protein tends to fold into a spatial conformation that has the minimum global energy. The conformation with minimum energy is called the native conformation or ground state. A protein is
considered stable if its native conformation is unique (i.e., there are no two native conformations with the same global energy). Natural proteins are usually stable.

1.2 Drug Design and HP Model

The Protein folding problem is to determine a protein’s native conformation based on its amino acid sequence. For many applications such as drug design, we are interested in the complementary problem: protein design, or inverse protein folding (IPF), which starts with a given structure or shape of a target protein, determines an amino acid sequence whose native conformation is stable and satisfies the given structure.

Many forces come into play in determining the native conformation into which protein fold, including hydrogen bonding, van der Waals interactions, intrinsic propensities, ion pairing, and hydrophobic interaction. Among these, hydrophobic interactions are by far the largest force accounts for as much as 80% the conformation. This observation inspired Dill to introduce the hydrophobic-polar (HP) model as a method of simplifying the protein folding problem. In HP model the 20 amino acids are represented as either hydrophobic (H) or polar (P) monomers depending on their affinity to water, thus greatly decreasing the complexity of the protein folding problem. Two monomers are said to be consecutive if a peptide bond occurs in between the two corresponding amino acid residues. Contact between any two non-consecutive hydrophobic monomers reduces the free energy of the resulting molecule. In the HP model, a protein reaches its ground state when the number of contacts between non-consecutive hydrophobic monomers is maximized. To further
simplify the problem, in Dill’s original HP model, the monomers are laid out on a 2D square lattice with each monomer occupying exactly one square, and neighbouring monomers occupying neighbouring squares. The HP model can be applied to any 2D or 3D lattice.

1.3 Motivation

Gupta et al. designed theoretical 2D proteins using the HP model applied to a 2D square lattice. A tiling structure is constructed using two kinds of base tiles that could be used to generate an approximate native conformation for any shape of 2D protein. The result is the motivation for an ongoing project to determine whether the results can be extended to 3D which is more biologically relevant. However, before designing a tiling structure, we must determine a 3D lattice that can be used in solving IPF on 3D HP model. This can be done by the following two steps:

Step 1: The attributes of 3D lattice most amenable to represent known protein 3D structures are determined. The 3D lattices which have these attributes are selected as candidate lattices. Usually a test set of protein 3D structures modelled in the Protein Data Bank (PDB) which satisfy some criteria is taken as representative of known protein structures.

Step 2: The candidate lattices are tested on their abilities to represent known protein structures by the following procedure:

2.1 Find the closest lattice representation of each protein such that the overall distance of the lattice representation to the protein, which is
measured by the coordinate root mean squared deviation (cRMS) or by the distance root mean squared deviation (dRMS), is minimized. This problem is also called “the discretization of a protein backbone”, “lattice approximation of 3D structure of a chain molecule”, “discrete state model fitting to X-ray structures”, “fitting of a protein chain to a lattice” or “protein chain lattice fitting problem”. In this thesis, we call it the fitting protein chains to lattice (FPL) problem;  

2.2 Compute the average cRMS (or dRMS) value over all proteins in the test set. The smaller the average cRMS (or dRMS) value, the better the lattice represents the known protein structures.

The work in Step 1 has been the subject of an extensive study in Mead et al., in which attributes of 3D lattices most suitable to represent known natural proteins have been considered statistically on more than 3000 protein structures from the PDB. A qualifying lattice must have the following attributes:

1. Uniform edge lengths of 3.8Å;
2. Minimum distance between any two vertices of at least 3.8Å;
3. Mainly 90° and 120° angles between consecutive triplet of lattice points;
4. Periodic structure.

The candidate lattices which satisfy the above criteria which were considered are:

- Truncated octahedron lattice
- Hexagonal prism lattice
• Truncated tetrahedron lattice
• Simple cubic lattice
• Cuboctahedron lattice
• Three commonly used lattices based on simple cubic lattice: face centred cubic (FCC) lattice, side+FCC (S+FCC) lattice, and extended FCC (e-FCC) lattice.

Mead et al. also attempted to solve the FPL problem by providing a greedy algorithm to compute the closest lattice representation of the given protein structures. Although the greedy algorithm does not guarantee that the lattice approximation that minimizes the overall cRMS (or dRMS) distance is found, it showed that FCC and e-FCC lattice are the best to represent protein structures in PDB.

In this thesis, we give an exact solution to the FPL problem by providing a backtracking dynamic programming algorithm to reduce the search space, formulating integer programming models for the FPL problem, and solving the IP models over the reduced search space by using the optimization package ILOG CPLEX 9.1, which produces the optimal solution for the problem.
CHAPTER 2: OVERVIEW OF FITTING PROTEIN CHAINS TO LATTICE PROBLEM

2.1 3D FPL Problem

A protein chain is an ordered sequence of residues, each residue is represented by the 3D coordinates of its \( C_\alpha \) atom. Starting from the first residue, each residue, except the last one, is connected by a peptide bond to the residue immediately succeeding it in the given order. Given a protein chain and a lattice graph, the fitting protein chain to lattice (FPL) problem is to find a corresponding lattice vertex for each residue in the protein chain, such that walking along the path of the lattice vertices in the same order as their corresponding residues appear in the protein chain forms a path that is the most similar to the original protein chain. This similarity can be measured in several different ways, among which \( cRMS \) and \( dRMS \) are most common. In this work we only consider \( cRMS \).

Formally, FPL is defined as the following:

Given:

\( \text{(1) A protein chain is represented by a finite directed graph } G_p = (V_p, E_p) \text{ where residues are represented as a set of vertices } V_p = \{ p, 0 \leq i \leq P - 1 \} \text{ in 3D Euclidean space. The 3D coordinates } (x_p, y_p, z_p) \text{ of each vertex } p \text{ are given.} \)

The ordering of the residues is preserved by having \( p_i \) as the immediate
successor of $p_i$. Peptide bonds are represented as a set of directed edges

$$E_p = \{ b_i, 0 \leq i \leq P - 1 \}$$

with each edge $b_i = (p_i, p_{i+1})$.

(2) A 3D lattice is represented by a finite directed graph $G_L = (V_L, E_L)$ where $V_L$ is a set of vertices $\{q_i, 0 \leq i \leq Q - 1\}$ in 3D Euclidian space. The 3D coordinates $(x_i, y_i, z_i)$ of each vertex $q_i$ are given. $E_L$ is a set of directed edges $\{e_i, 0 \leq i \leq M - 1\}$. We will assume that if there is a directed edge $e_i = (q_i, q_{i+1})$, there must also exist a directed edge $e_i = (q_{i+1}, q_i)$. The degree of the lattice is represented by $d$. The neighbours of a lattice vertex $q_i$ is represented by $N(q_i) = \{ q_j \mid (q_i, q_j) \in E_L \}$.

**Task:**

Find a map $\phi : V_p \rightarrow V_L$ which maps each protein vertex $p_i$ to a unique lattice vertex correspondent $q_i$, such that the following two properties are satisfied:

- Two different protein vertices are not mapped to the same lattice vertex.

  \[ (2 - 1) \]

- Any two adjacent protein vertices are mapped to two adjacent lattice vertices.

  \[ (2 - 2) \]

- The $\text{cdist}(\phi) = \text{cdist}((p_i), (\phi(p_i)))$ between the protein and its lattice approximation is minimized.

  \[ (2 - 3) \]

We call any map which satisfies (2 - 2) a **walk map**, and any map which satisfies (2 - 1) and (2 - 2) a **vertex map**. A vertex map $\phi$ is an injective graph.
homomorphism. In other words, a sequence \((\phi(p_0), \phi(p_1), \ldots, \phi(p_n))\) obtained by applying mapping \(\phi\) on path \((p_0, p_1, \ldots, p_n)\) is a path in \(G_i\). In the FPL problem, we are looking for a vertex map \(\phi\) that also satisfies (2.3), when the sequence \((\phi(p_0), \phi(p_1), \ldots, \phi(p_n))\) is called the **lattice approximation** for protein chain \(G_p = (V_p, E_p)\), the lattice vertex \(q_i = \phi(p_i)\) is called the **lattice image** of protein vertex \(p_i\), and \(p_i\) is called the correspondent protein vertex for lattice vertex \(q_i\).

The FPL problem for cubic lattice with side length 3.8Å and cRMS distance measure was shown to be NP-complete by Manuch et al\(^{22}\).

We need to make the volume of the finite 3D space defined by the lattice graph \(G_i = (V_i, E_i)\) big enough to guarantee that the optimal solution can be found. The method of generating such lattices is described in Section 4.5. Let us assume now that the lattice graph \(G_i\) is large enough to guarantee an optimal solution.

### 2.2 Overview of Existing Results

Various methods have been studied by different authors to solve FPL problem.

In a paper by Covell et al\(^1\), all possible conformations are enumerated by placing each residue into the available nearest neighbours on the lattice at each stepwise residue addition, and the best conformation is picked as global optimal solution.
Godzik et al. presented the "i-th vector" method which starting from the first residue of the protein chain, maps the proper lattice vertex to the residue such that the cRMS of the local lattice approximation is minimized. Proceeds in this fashion till the mapping of the last residue is determined. This was based on the assumption that a global optimal fit of a particular lattice to a protein chain is likely to be composed of shorter segments which are locally optimal.

Park and Levitt extended the above idea to build global good fits by keeping a small number (500) of the best local lattice representations at each stepwise residue addition. They selected good local fits in terms of lowest cRMS deviation from the appropriate portion of the protein chain.

Based on the above approach of Park and Levitt's, another greedy algorithm to find lattice approximation of a protein is designed by Mead et al. This algorithm generate good lattice approximations relatively fast, and its experimental results are compared against that of our algorithm in this work. The greedy algorithm procedure consists of 3 phases: Phase 1: For each vertex in the protein, the closest lattice vertex is found and a cRMS distance is obtained as the sum of the distances between each protein vertex and its closest lattice vertex. Note this cRMS distance gives a lower bound for cRMS distance of any lattice approximation of the protein. Phase 2: Consider the mapping that maps each protein vertex to its closest lattice vertex. A chain is defined as a self-avoiding (each vertex is visited at most once) walk along the edges of the lattice. The longest chain is identified in the mapped sequence of the lattice vertices. Phase 3: The longest chain is considered as the partial lattice approximation and
is extended on both ends until the whole protein is fitted to the lattice. To extend the partial lattice approximation, an improved \( i \)-th vector method\(^2\) is used: to find the best mapping for the \((k + 1)\)-th protein vertex, consider all paths with length \( \text{ahead} \) and having the lattice image of the \( k \)-th protein vertex as the starting point. Of all such paths, choose the one with the smallest local increment to the distance value (\( cRMS \) or \( dRMS \)). The lattice image of the \((k + 1)\)-th protein vertex is chosen as the first lattice vertex on the best path. The higher the \( \text{ahead} \) value, the more precise the result, but the computational time grows exponentially with the \( \text{ahead} \) value. According to the experiments, the \( \text{ahead} \) value does not improve significantly after 3-4 for \( cRMS \).

A dynamic programming approach is also applied by several authors. Godzik et al\(^2\) presented two methods which generate a random initial conformation, and then use Monte Carlo simulated annealing to bring the lattice approximation closer to the protein chain in terms of either inter-residue distances or \( cRMS \). Reva et al\(^3\) and Rykunov et al\(^4\) presented dynamic programming algorithms to find the lattice approximation which minimize the error function for a given lattice and the lattice-protein orientation (i.e., the relative position between the protein and the lattice in 3D Euclidean space), and tested the algorithm on 12 proteins with up to 445 residues. Rabow and Scheraga\(^5\) presented a dynamic programming algorithm to fit protein chain to a lattice, subject to bond length, bond angle and overlap constrains. Hinds and Levitt\(^6\) approximated protein structures using a self-avoiding path of fitting every other residue to a vertex of a tetrahedral lattice with edge length 4.95 Å. The search
space was reduced by a bounded volume which is at least 50% larger than the
turn occupied by the protein X-ray structure. Good global solutions were
computed using dynamic programming, and the set of mutually compatible
moves with lowest predicted final energy are produced. The algorithm
successfully predicted native-like conformations of 5 small proteins range from
52 to 68 residues.

Koehl and Delarue\(^4\) presented a completely different approach using self-
consistent mean field theory. For a given lattice, all possible conformation of four
consecutive lattice vertices are generated and each of them is mapped to the first
four \(C_\alpha\)'s of protein residues, such that the \(cRMS\) between the four residues and
the corresponding lattice vertex is minimized. In this case, the protein-lattice
orientation is bonded. The remaining mappings are generated by viewing each
residue as independent and can occupy any possible lattice vertex with a weight
stored in lattice probability matrix. The energy function is iteratively minimized
with respect to the lattice probability matrix. The final matrix describes the
conformational space available to the protein.

All above algorithms either exhaustively enumerate all conformations or
produce only approximation solutions. Most only apply to very short protein
chains. A polynomial and exact algorithm can be developed inspired by the
algorithm in Dal Palu et al\(^{14,15}\), who applied constraint logic programming over
finite domains (CLP(FD)) techniques to the protein structure prediction problem
on the FCC lattice model, in which the global energy of the folding is minimize by
simply observing contacts between amino acids. To improve efficiency,
reorganization of constraint structure, heuristics and exploitation of parallelism are applied. The improved algorithm can solve the protein folding problem with up to 84 amino acids in less than 8 minutes. They also showed how these results can be employed to solve the folding problem for larger proteins containing sub-sequences whose conformations are already known.
CHAPTER 3: BASIC LP FORMULATIONS

3.1 ILOG CPLEX

ILOG CPLEX is a powerful and popular optimization package which solves LP models, quadratic programming models where the objective function is quadratic instead of linear, and certain kinds of quadratically constrained problems. ILOG CPLEX also solves MIP and IP. The size of the problem solvable by ILOG CPLEX is not limited by maximum number of variables, maximum number of constraints, or the available disk space, but by the available memory.

In the following subsections, we built three different IP models for FPL problem. Later we will show how ILOG CPLEX 9.1 (referred to as CPLEX from now on) package for Sun Solaris platform is used to solve FPL problem using these models and study which one of them is the most efficient.

3.2 IP Vertex Model

In FPL problem, the IP formulation is searching through all vertex maps $\phi$ for the one which minimize $c_{\text{dist}}$ cost between protein and its lattice approximation. To design an IP model for the FPL problem, define a binary variable $X_{i,j}$ for every $p_i \in V_p$ and $q_j \in V_q$ as:

$$X_{i,j} = \begin{cases} 
1, & \text{if } \phi(p_i) = q_j; \\
0, & \text{otherwise.}
\end{cases}$$
The objective function is defined to minimize the c-RMS distance between the protein chain and its lattice approximation.

\[
\text{Minimize } \sum_{i,j} c_{ij} \cdot X_{ij} \tag{*}
\]

Where \( c_{ij} = d^2(p_i, q_j) \) is the square of Euclidean distance between protein vertex \( p_i \) and \( q_j \).

Subject to constraints:

1. Each protein vertex \( p_i \) maps to an unique lattice vertex correspondent \( q_j \):
   \[
   \forall p_i \in V_p : \sum_{j} X_{ij} = 1 \tag{c1}
   \]

2. Every lattice vertex can be mapped to maximum one protein vertex:
   \[
   \forall q_j \in V_q : \sum_{i} X_{ij} \leq 1 \tag{c2}
   \]

3. Any two adjacent protein vertices are mapped to two adjacent lattice vertices. Equivalently, any two adjacent protein vertices cannot be mapped to two lattice vertices which are not adjacent in \( G_L \):
   \[
   \forall p_i \in V_p, \forall q_j \in V_q : X_{ij} + \sum_{q_k, q_{k+1} \in G_L} X_{k+1} \leq 1 \tag{c3}
   \]

An alternative way of enforcing the adjacency is to use the following constraint \((vm \cdot c3 \cdot v2)\) instead of \((vm \cdot c3)\).

\[
\forall p_i \in V_p, \forall q_j \in V_q : \sum_{q_k, q_{k+1} \in G_L} X_{k+1} \geq X_{ij} \tag{c3 \cdot v2}
\]

The approach in \((vm \cdot c3 \cdot v2)\), which used by Hart et al\cite{har94}, greatly reduces the number of non-zeros in the coefficient matrix.
With the bounds: $X_{ij}$ are binary variables.

We claim that mapping $\phi$ satisfies the requirements (2 - 1) and (2 - 2) of the FPL problem, if and only if the variables $X_{ij}$ defined by $(vm - x)$ satisfy constraints $(vm - c1)$, $(vm - c2)$ and $(vm - c3)$ (or $(vm - c3 - v2)$). We prove it in Proof 3-1.

Proof 3-1: Mapping $\phi$ satisfies the requirements (2 - 1) and (2 - 2) of the FPL problem, if and only if the variables $X_{ij}$ defined by $(vm - x)$ satisfy constraints $(vm - c1)$, $(vm - c2)$ and $(vm - c3)$ (or $(vm - c3 - v2)$).

First, we are going to show that if the variables $X_{ij}$ defined by $(vm - x)$ satisfy constraints $(vm - c1)$, $(vm - c2)$ and $(vm - c3)$ (or $(vm - c3 - v2)$), then map $\phi$ satisfies the requirements (2 - 1) and (2 - 2) of the FPL problem.

- Given values for $(X_{ij})$, consider relation $\rho \subseteq J \times J'$ defined by $(vm - x)$. Constraint $(vm - c1)$ guarantees that every vertex in the protein has a unique lattice vertex correspondent, which means that $\rho$ is a proper mapping.

- Constraint $(vm - c2)$ guarantees that for every lattice vertex, either one or none protein vertex is mapped to this lattice vertex, which means $\phi$ is injective.

- Constraint $(vm - c3)$ guarantees that any two adjacent protein vertices are aligned to two adjacent lattice vertices. Indeed, suppose that there are two adjacent protein vertices that are mapped two non-adjacent lattice vertices,
i.e., there exist \( p_i \in V_p \), such that \((q_j, q_k) \notin E_L\) where \( q_j = \phi(p_j) \), \( q_k = \phi(p_{k+1}) \).

Then \( X_{i,j} + X_{i,k} = 2 \), which violates the requirement of constraint \((vm - c3)\), since \( X_{i,j} + \sum_{a \in \mathcal{V}(q_j)} X_{i,a} \geq 2 \).

- Constraint \((vm - c3 - v2)\) also guarantees that any two adjacent protein vertices are mapped to two adjacent lattice vertices. Suppose that there are two adjacent protein vertices that are mapped to two non-adjacent lattice vertices, i.e., there exist \( p_i \in V_p \), such that \( q_j = \phi(p_j) \), \( q_k = \phi(p_{k+1}) \) and \( q_i \notin \mathcal{N}(q_j) \). Then \( X_{i,j} = 1 > \sum_{a \in \mathcal{V}(q_j)} X_{i,a} = 0 \), which violates the requirement of constraint \((vm - c3 - v2)\).

Second, we are going to show that if map \( \phi \) satisfies the requirements (2-1) (2-2) of the FPL problem, then the variables \( X_{i,j} \) defined by \((vm - x)\) satisfy constraints \((vm - c1)\), \((vm - c2)\) and \((vm - c3)\) (or \((vm - c3 - v2)\)).

- For given \( p_i \in V_p \), let \( \phi(p_i) = q_i \). Then \( \sum_{q \in \mathcal{V}_q} X_{i,j} = 1 \) since \( X_{i,j} = 0 \) for \( j \neq k \) and \( X_{i,j} = 1 \) for \( j = k \), i.e., \((vm - c1)\) is satisfied.

- Let us suppose that \((vm - c2)\) is not satisfied, i.e., there exists \( q_j, q_k \in V_L \) such that \( \sum_{j \in \mathcal{V}_q} X_{i,j} > 1 \). There exist at least two protein vertices \( p_i, p_j \in V_p \), \( p_i \neq p_j \) such that \( X_{i,j} = 1 \) and \( X_{j,k} = 0 \). By \((vm - x)\), \( \phi(p_i) = q_i = \phi(p_j) \), which violates (2-1).
Let us suppose that \((vm - c3)\) is not satisfied, i.e., there exist \(p_i \in V_p, q_j \in V_L\) such that \(X_{ij} + \sum_{q \in N(q_j)} X_{iq} > 1\). Then either \(X_{ij} = 1\) and there exist at least one lattice vertex \(q_i \in V_L - N(q_j)\), such that \(X_{ii} = 1\), or there exist at least two lattice vertices \(q_i, q_j \in V_L - N(q_j), q_i \neq q_j\), such that \(X_{ii} = 1\) and \(X_{ij} = 1\). In the first case, by \((vm - x)\), edge \((p_i, p_{i+})\) is mapped to a pair \((q_i, q_j) \in E_{L}\), which violates \((2-2)\). In the second case, by \((vm - x)\), \(\phi(p_{i+}) = q_j\) and \(\phi(p_i) = q_j\), a contradiction.

Let us suppose that \((vm - c3 - v2)\) is not satisfied, i.e., there exist \(p_i \in V_p, q_j \in V_L\) such that \(\sum_{q \in N(q_j)} X_{iq} < X_{ij}\). Then \(X_{ij} = 1\) and \(\sum_{q \in N(q_j)} X_{iq} = 0\), which means two adjacent protein vertices \(p_{i+}, p_i \in V_p\) are not mapped to two adjacent lattice vertex \(q_j, q_i \in V_{L}\), which violates \((2-2)\).

### 3.3 IP Edge Model I

We will call a map \(\phi : E_p \rightarrow E_L\) an **edge map** if

- Any two adjacent protein edges are mapped to two adjacent lattice edges.
  \[ (3-1) \]

- Any two different protein edges are not mapped to the same lattice edge.
  \[ (3-2) \]

- Any two different lattice images of protein edges have two different terminal vertices.
  \[ (3-3) \]
If \((z, V) \in E_L\) is the image of \((p, p_1) \in E_r\), then no other protein edge is mapped to a lattice edge whose terminal vertex is \(u\).

(3-4)

The above four requirements can be stated in an equivalent way, namely, if walk along the images of protein edges in the following order, \(\varphi((p, p_1))\), \(\varphi((p_1, p_2))\), \(\varphi((p_2, p_3))\), \(\varphi((p_3, p_4))\), then the image of the path \((p, p_1), (p_1, p_2), \ldots, (p_{n-2}, p_{n})\) by \(\varphi\) is again a path in \(G_L\).

Given a vertex map \(\phi : V_r \rightarrow V_L\), define a relation \(\varphi : E_r \times V_L \times V_L \rightarrow E_L\) as

\[\varphi((p, q_1, q_2)) = (\varphi(p), \varphi(q_1), \varphi(q_2))\] Map \(\varphi\) will be called an **edge correspondent of map** \(\phi\), represented as \(\varphi = \varepsilon(\phi)\). Conversely, given an edge map \(\phi : E_r \rightarrow E_L\), we can define a map \(\phi : V_r \rightarrow V_L\) using initial vertices for every \(i = 0, \ldots, P-3\) as \(\phi(p) = q_i\) if \(\phi((p, p_1, p_2, \ldots, p_{n})) = (q_1, q_2, \ldots, q_{n})\) for some \(q_i \in V_L\), and using terminal vertices for every \(i = 1, \ldots, P-2\) as \(\phi(p) = q_i\) if \(\phi((p, p_1, p_2, \ldots, p_{n})) = (q_1, q_2, \ldots, q_{n})\) for some \(q_i \in V_L\). By \(4(1)\), \(\phi(p)\) is defined using the initial vertex and the terminal vertex coincide for \(i = 1, \ldots, P-3\), i.e., \(\phi\) is properly defined. \(\phi\) will be called a **vertex correspondent of map** \(\phi\), represented as \(\phi = \gamma(\phi)\).

Let us define \(c_{dist}(\phi) = c_{dist}(\phi_1, V_L, \varphi((p, p_1)))\). For \(i \in \{0, P-2\}\), let \(q_i\) be the initial vertex and \(q_i\) be the terminal vertex of \(\phi((p, p_1, p_2, \ldots, p_{n}))\), then

\[c_{dist}(\phi)=d^2(p_0, q_0)+\frac{1}{2}d^2(p_1, q_1)+\ldots+\frac{1}{2}d^2(p_{n-2}, q_{n-2})+d^2(p_{n-1}, q_n)\]
We claim that if $\phi$ is a vertex map, then its edge correspondent $\varphi = \varepsilon(\phi)$ is an edge map, and \( c\text{dist}(\varphi) = c\text{dist}(\phi) \). It is also easy to see that $\gamma(\varepsilon(\phi)) = \phi$. On the other hand, if $\varphi$ is an edge map, then its vertex correspondent $\phi = \gamma(\varphi)$ is a vertex map, and \( c\text{dist}(\phi) = c\text{dist}(\varphi) \). It is easy to see that $\varepsilon(\gamma(\varphi)) = \varphi$. We prove it in Proof 3-2.1 and Proof 3-2.2.

**Proof 3-2.1:** If a vertex map $\phi$ satisfies (2.1) and (2.2), then its edge correspondent $\varphi = \varepsilon(\phi)$ is an edge map, and \( c\text{dist}(\varphi) = c\text{dist}(\phi) \).

Consider a protein edge $(p_i, p_{i+}) \in E_p$, since $\phi$ is a graph homomorphism, $\phi(p_i)$ and $\phi(p_{i+})$ must be adjacent, i.e., $\varphi((p_i, p_{i+})) = (\phi(p_i), \phi(p_{i+})) \in E_e$, which means every protein edge is mapped to a lattice edge, i.e., $\varphi[E_p] \subseteq E_l$.

Since $\varphi$ is an injective map, for two different protein edges $(p_i, p_{i+})$, $(p_j, p_{j+}) \in E_p$, $(i \neq j)$, $\varphi(p_i) \neq \varphi(p_j)$, therefore $\varphi((p_i, p_{i+})) = (\varphi(p_i), \varphi(p_{i+}))$ and $(\varphi(p_j), \varphi(p_{j+}))$, i.e., $\varphi((p_i, p_{i+})) \neq \varphi((p_j, p_{j+}))$, which means (3-2) holds.

Consider two adjacent protein edges $(p_i, p_{i+})$, $(p_{i+}, p_{i+2}) \in E_p$, they are mapped to two lattice edges $(\varphi(p_i), \varphi(p_{i+}))$ and $(\varphi(p_{i+}), \varphi(p_{i+2}))$ by $\varphi$ respectively. Obviously, $(\varphi(p_i), \varphi(p_{i+}))$ and $(\varphi(p_{i+}), \varphi(p_{i+2}))$ are adjacent, which means (3-1) holds.
Suppose \((3 - 3)\) does not hold, i.e., \(\varphi((p_i, p_{i+1})) = (q_i, q_{i+1})\) and 
\[ \varphi((p_j, p_{j+1})) = (q_j, q_{j+1}) \] for some \(i \neq j\). We have \(p_i \neq p_j\) and 
\[ \varphi(p_{i+1}) = q_i = \varphi(p_{i+1}) \], which contradicts \((2 - 1)\). Therefore \((3 - 3)\) holds.

Suppose \((3 - 4)\) does not hold, there exists a protein edge \((p_i, p_{i+1}) \in E_r\), such that \(\varphi((p_i, p_{i+1})) = (\varphi(p_i), \varphi(p_{i+1}))\), then \(\varphi(p_{i+1}) = \varphi(p_{i+1})\), which again contradicts \((2 - 1)\). Therefore \((3 - 4)\) holds.

Recall that \(\text{cdist}(\emptyset) = \text{cdist}(\emptyset(p_i)) = \sum_{i=1}^{n} d_i^{2}(p_i, \varphi(p_i))\). For 
\[ i \in [0, P - 2] \], let \(q_i\) be the initial vertex and \(q_j\) be the terminal vertex of 
\(\varphi((p_i, p_{i+1}))\), then 
\[ \text{cdist}(\emptyset) = \sum_{i=1}^{n} d_i^{2}(p_i, \varphi(p_i)) \]
\[ = d_i^{2}(p_i, \varphi(p_i)) + \frac{1}{2} d_i^{2}(p_i, \varphi(p_{i+1})) + \frac{1}{2} \sum_{i=1}^{n} [d_i^{2}(p_i, \varphi(p_{i+1})) + d_i^{2}(p_{i+1}, \varphi(p_{i+2}))] \]
\[ + \frac{1}{2} d_i^{2}(p_{i+1}, \varphi(p_{i+3})) + \frac{1}{2} \sum_{i=1}^{n} [d_i^{2}(p_{i+1}, q_i) + d_i^{2}(p_{i+1}, q_{i+1})] \]
\[ + \frac{1}{2} d_i^{2}(p_{i+1}, q_{i+1}) + d_i^{2}(p_{i+1}, q_{i+2}) \]
\[ = \text{cdist}(\emptyset) \]

Therefore \(\text{cdist}(\emptyset) = \text{cdist}(\emptyset)\).

**Proof 3-2.2:** Given an edge map \(\varphi\), its vertex correspondent \(\emptyset = \varphi(p)\) satisfies \((2 - 1)\) and \((2 - 2)\), and \(\text{cdist}(\emptyset) = \text{cdist}(\emptyset)\) .
For $i = 0, \ldots, P - 3$, $\phi(p_i)$ was defined in two different ways, one using initial vertex, the other using terminal vertex. Assume that these two definitions disagree for some $i$, i.e., there are $q_i, q_i' \in V$, $q_i \neq q_i'$, such that $\phi(p_{i+1}, p_i) = (q_i, q_i')$ and $\phi(p_{i+1}, p_i) = (q_i, q_i')$. Since $\phi$ satisfies (3-1), the above situation can not happen since $(q_i, q_i)$ and $(q_i, q_i')$ are not adjacent. Therefore $\phi$ is properly defined.

Since $\phi$ is a map $E_j \to E_{j+1}$, then adjacent protein vertices are mapped to adjacent lattice vertices, which means (2-2) holds.

Suppose $\phi$ does not satisfy (2-1), i.e., $\phi$ is not injective, then either $i > 0$:
then $\phi(p_{i+1}, p_i) = (q_i, q_i')$ and $\phi(p_{i+1}, p_i) = (q_i, q_i')$, which contradicts (3-3). Or $i = 0$: then $\phi((p_0, p_i)) = (q_0, q_i)$ and $\phi(p_{j}, p_i) = (q_i, q_i')$, which contradicts (3-3).

Recall that for $i \in [0, P - 2]$, let $q_i$ be the initial vertex and $q_i'$ be the terminal vertex of $\phi((p_i, p_{i+1}))$, then

\[
\text{cdist}(\phi) = d^2(q_i, q_i') + \frac{1}{2} d^2(p_i, q_i') + \frac{1}{2} d^2(p_i, q_i') + \frac{1}{2} d^2(p_i, q_i') + \frac{1}{2} d^2(p_i, q_i')
\]
Therefore \( \text{cdist}(\phi) = \text{cdist}(\tilde{\phi}) \).

Our first IP edge model is defined as follows:

Let \( \phi \) be an edge map, the IP formulation is searching through all edge maps for the one which minimize the \( \text{cdist} \) between protein and its lattice approximation. Define a binary variable \( X_{i,x} \) for every \( h \in E_p, (q_x, q_y) \in E_L \) as:

\[
X_{i,x} = \begin{cases} 
1, & \text{if } \phi(h) = (q_x, q_y); \\
0, & \text{otherwise.}
\end{cases}
\]

The objective function in IP is defined to minimize the distance between the protein chain and its lattice approximation.

\[
\text{Minimize } \sum_{i \in P, x \in \mathcal{A}} c_{i,x} X_{i,x}
\]

where

\[
c_{i,x} = d^2(p_x, q_x) + \frac{1}{2} d^2(p_y, q_y)
\]

\[
c_{i,x} = \frac{1}{2} d^2(p_i, q_i) + \frac{1}{2} d^2(p_{i+1}, q_{i+1}), i \in [1, P-1]
\]

\[
c_{i-1, x} = \frac{1}{2} d^2(p_{i-1}, q_i) + d^2(p_p, q_i)
\]

Subject to constraints:

1. Every protein edge maps to an unique lattice edge:

\[
\forall (p_i, p_{i+1}) \in E_p : \sum_{x \in \mathcal{A}} X_{i,x} = 1
\]

2. Any two adjacent protein edges are mapped to two adjacent lattice edges.

Equivalently, any two adjacent protein edges cannot be mapped to two non-adjacent lattice edges.
\[ \forall i = 0, \ldots, P - 3, \forall q_i \in V_i^c : \sum_{(i, h, s) \in \Lambda_i} X_{i, h, s} + \sum_{(i, h, s, v) \in \Lambda_i} X_{i, h, s, v} = 1 \quad \text{(em1 \cdot c2)} \]

Similar to IP vertex model, an alternative way of enforcing the adjacency is to use the following constraint \((\em1 \cdot c2 \cdot v2)\) instead of \((\em1 \cdot c2)\).

\[ \forall i = 0, \ldots, P - 3, \forall (q_i, q_j) \in E_i : \sum_{q_i \in V_i} X_{i, q_i} \geq X_{i, q_j} \quad \text{(em1 \cdot c2 \cdot v2)} \]

3. No merging lattice edges in the solution set, that is, no two different lattice edges in the mapping can have the same terminal vertex. Equivalently, the total number of edges in the mapping that terminate at vertex \(q_i \in V_i\) can not be more than one:

\[ \forall q_i \in V_i^c : \sum_{(i, h, s) \in \Lambda_i} X_{i, h, s} \leq 1 \quad \text{(em1 \cdot c3)} \]

4. No edge in the mapping ends at the vertex which maps to the first protein vertex:

\[ \forall (q_i, q_j) \in E_i : X_{i, q_i} + \sum_{(i, h, s, v) \in \Lambda_i} X_{i, h, s, v} \leq 1 \quad \text{(em1 \cdot c4)} \]

With the bounds, \(X_{i, h, s}\), are binary variables.

We claim that conditions \((\em1 \cdot c1), (\em1 \cdot c2)\) (or \((\em1 \cdot c2 \cdot v2)\)), \((\em1 \cdot c3)\) and \((\em1 \cdot c4)\) are satisfied, if and only if \((3 - 1)\) to \((3 - 4)\) are also satisfied. We prove it in Proof 3-3.

**Proof 3-3:** Map \(\phi\) satisfies requirements \((3 - 1)\) to \((3 - 4)\) if and only if the variables \(X_{i, h, s}\), defined by \((\em1 \cdot x)\) satisfy constraints \((\em1 \cdot c1), (\em1 \cdot c2)\) (or \((\em1 \cdot c2 \cdot v2)\)), \((\em1 \cdot c3)\) and \((\em1 \cdot c4)\).
First, we need to prove if map \( \varphi \) satisfies requirements (3 - 1) to (3 - 4), the variables \( X_{i,s} \), defined by (3 - 5) satisfy constraints (3 - 1), (3 - 2), (3 - 3), and (3 - 4).

- Given protein edge \((p_i, p_i) \in E_p\), let \( \varphi((p_i, p_i)) = (q_i, q_i) \in E_e \). Then
  \[ X_{i,s} = 1 \]
  and for all lattice edges other than \((q_i, q_i)\), i.e.,
  \[ (q_i, q_i) \in E_e, (q_i, q_i) \neq (q_i, q_i), \sum_{i \in E_e, i \neq i} X_{j,s} = 0. \text{ Thus (3 - 1) holds.} \]

- Consider protein edge \((p_i, p_i) \in E_p\) and a lattice vertex \( q_i \in V_L \):

  1) First, if \( \varphi((p_i, p_i)) = (q_i, q_i) \), i.e., \( \sum_{i \in E_e, i \neq i} X_{j,s} = 1 \). Then by (3 - 2),
  \[ (p_i, p_i) \in E_p \]
  must be mapped to an adjacent lattice edge \((q_i, q_i) \in E_e\),
  thus \( \sum_{i \in E_e, i \neq i} X_{j,s} = 0 \). Constraint (3 - 2) holds in this case.

  2) Second, assume \( \varphi((p_i, p_i)) = (q_i, q_i) \) where \( q_i \neq q_i \). Then \( \sum_{i \in E_e, i \neq i} X_{j,s} = 0 \).
  By (3 - 2), \( \varphi((p_i, p_i)) = (q_i, q_i) \), i.e., \( \sum_{i \in E_e, i \neq i} X_{j,s} = 1 \). Constraint (3 - 2) holds in this case as well.

- Consider protein edge \((p_i, p_i) \in E_p\) and a lattice edge \((q_i, q_i) \in E_e\):

  1) First, if \( \varphi((p_i, p_i)) = (q_i, q_i) \), i.e., \( X_{i,s} = 1 \), then by (3 - 2), \( (p_i, p_i) \in E_p \)
  must be mapped to an adjacent lattice edge \((q_i, q_i) \in E_e\),
  thus \( \sum_{i \in V_L, i \neq i} X_{j,s} = 1 \). Constraint (3 - 2) holds in this case.
2) Second, assume \( \phi((p_{i,j}, p_{u,v})) \neq (q_{i,j}, q_{u,v}) \) then \( X_{i,j} = 0 \leq \sum_{q_{i,j}, q_{u,v}} X_{i,j,u,v} \).

Constraint \((\text{cm}1 \cdot c2 \cdot v2)\) holds in this case as well.

- Consider a lattice vertex \( q_{i,j} \in V_i \), assume \( \sum_{i,j,u,v} X_{i,j,u,v} \geq 2 \). Then there exist \( i, j, u, u', i \neq j, u \neq u' \), such that \( X_{i,j} = 1 \) and \( X_{j,u'} = 1 \), i.e.,
  \( \phi((p_{i,j}, p_{u,v})) = (q_{i,j}, q_{u,v}) \) and \( \phi((p_{j,u'}, p_{u,v})) = (q_{j,u'}, q_{u,v}) \). By \((3 \cdot 3)\), \( i = j \). We have \( u = u' \). Therefore \((\text{cm}1 \cdot c3)\) holds.

- Consider \( (q_{i,j}, q_{u,v}) \in E_{i,j} \), assume \( X_{i,j,u,v} = 1 \), then \( \sum_{i,j,u,v} X_{i,j,u,v} = 0 \);
  Otherwise if \( X_{i,j,u,v} = 0 \), then by \((\text{cm}1 \cdot c3)\), the second term
  \( \sum_{i,j,u,v} X_{i,j,u,v} \leq 1 \). In both cases \((\text{cm}1 \cdot c4)\) holds.

Second, we need to prove if the variables \( X_{i,j,u,v} \) defined by \((\text{cm} \cdot x)\) satisfy constraints \((\text{cm}1 \cdot c1), (\text{cm}1 \cdot c2)\) (or \((\text{cm}1 \cdot c2 \cdot v2)\)), \((\text{cm}1 \cdot c3)\) and \((\text{cm}1 \cdot c4)\), map \( \phi \) satisfies requirements \((3 \cdot 1)\) to \((3 \cdot 4)\).

- Given values for \( X_{i,j,u,v} \), consider relation \( \phi \subseteq E \times E \) defined by \((\text{cm} \cdot x)\).

Constraint \((\text{cm}1 \cdot c1)\) guarantees that every edge in the protein has a unique lattice edge correspondent, which means that \( \phi \) is properly defined for every \((p_{i,j}, p_{u,v}) \in E_{p} \).
If two adjacent protein edges \((p_i, p_{i+1}) \in E_p\) are mapped to two non-adjacent lattice edges \((q_1, q_2), (q_{i+1}, q_i) \in E_E\), respectively, then
\[
\sum_{q_1, q_2 \in E_E} X_{q_1 q_2} + \sum_{q_{i+1}, q_i \in E_E} X_{q_{i+1} q_i} \geq 1 + 1 - 2, \text{ which violates } (\text{cm} 1 \cdot c2). \text{ Therefore } (3 \cdot 2) \text{ is satisfied, i.e., } \phi \text{ is an edge map.}
\]

If two adjacent protein edges \((p_i, p_{i+1}) \in E_p\) are mapped to two non-adjacent lattice edges \((q_1, q_2), (q_{i+1}, q_i) \in E_E\), respectively. Then
\[
X_{q_1 q_2} = 1. \text{ By } (\text{cm}1 \cdot c2 - v2), \ X_{q_1 q_2} = 1 \leq \sum_{q \in E_E} X_{q q}, \text{ therefore there must exist } q \in N(q_1) \text{ such that } X_{q q_1} = 1. \text{ Protein edge } (p_i, p_{i+1}) \in E_p \text{ has two different lattice images } (q_1, q_2), (q_{i+1}, q_i) \in E_E, \text{ which is a contradiction. Therefore } (3 \cdot 2) \text{ is satisfied, i.e., } \phi \text{ is an edge map.}
\]

If two protein edges \((p_i, p_{i+1}) \in E_p\) are mapped to two lattice edges \((q_1, q_2), (q_{i+1}, q_i) \in E_E\), that share the same terminal vertex, i.e.,
\[
X_{q_1 q_2} = 1 \text{ and } X_{q_{i+1} q_i} = 1, \text{ then } \sum_{q_{i+1}, q_i \in E_E} X_{q_{i+1} q_i} \geq 2, \text{ which violates } (\text{cm}1 \cdot c3). \text{ Therefore } (3 \cdot 3) \text{ is satisfied.}
\]

If \((p_i, p_{i+1}) \in E_p \text{ is mapped to } (q_1, q_2) \in E_E\), and there is a protein edge \((p_i, p_{i+1}) \in E_p\) mapped to \((q_1, q_2) \in E_E\), i.e.,
\[
X_{q_1 q_2} = 1 \text{ and } X_{q_{i+1} q_i} = 1, \text{ then } \sum_{q_{i+1}, q_i \in E_E} X_{q_{i+1} q_i} \geq 2 \text{ which violates } (\text{cm}1 \cdot c4). \text{ Therefore } (3 \cdot 4) \text{ is satisfied.}
\]
• If there are two different protein edges \((p_i, p_{i+1}), (p_j, p_{j+1}) \in E_P(p, \mathcal{P})\) which are mapped to the same lattice edge \((q_i, q_j) \in E_L\), then it violates (3-2).

Therefore (3-1) is satisfied.

3.4 IP Edge Model II

We have the second IP edge model in which the binary variables \(X_{i, j, k, l}\), the objective function and the bounds are defined same as the first IP edge model in Section 3.2, while the constraints are defined as follows:

Subject to constraints:

1. The first edge in protein \((p_i, p_j) \in E_P\) maps to a unique edge \((q_i, q_j) \in E_L\) in lattice:

\[
\sum_{i, j, k, l} X_{i, j, k, l} = 1 \quad (cm2 \cdot c1)
\]

2. Any two adjacent edges \((p_i, p_j), (p_m, p_n) \in E_P\) must be aligned to two adjacent edges \((q_i, q_j) \in E_L\) and \((q_m, q_n) \in E_L\). Equivalently, for every vertex \(q_i \in V_L\), the total number of edges which map to a protein edge \((p_i, p_j)\) equals the total number of edges which map to the adjacent protein edge \((p_m, p_n)\), if both the terminal vertices of the mapping lattice edge for \((p_i, p_j)\) and the initial vertex of the mapping lattice edge for \((p_m, p_n)\) are \(V_i, V_j, V_m, V_n\):

\[
\forall i = 0, ..., P - 3, \forall q_i \in V_L: \sum_{i, j, k, l} X_{i, j, k, l} - \sum_{i, j, k, l} X_{i+1, j, k, l} = 0 \quad (cm2 \cdot c2)
\]

Similarly, constraint \((cm2 \cdot c2)\) can be replaced by \((cm1 \cdot c2 \cdot v2)\).
3. The same as (em1- c3).

4. The same as (em1- c4).

We claim that conditions (em1- c1) to (em1- c4) are satisfied, if and only if (em2- c1), (em2- c2), (em1- c3) and (em1- c4) are also satisfied. We prove it in Proof 3-4.

Proof 3-4: The variables $X_{i,a}$ defined by (em - x) satisfy constraints (em1- c1) to (em1- c4), if and only if $X_{i,a}$ satisfy constraints (em2- c1), (em2- c2), (em1- c3) and (em1- c4).

First, we need to prove if the variables $X_{i,a}$, defined by (em - x) satisfy constraints (em1- c1) to (em1- c4), $X_{i,a}$ also satisfy constraints (em2- c1) and (em2- c2).

- Since (em1- c1) holds, so does $\sum_{x_i(a,b,c,d,k,\ell)} X_{x,i,a} = 1$ hold for the first protein edge, i.e., (em2- c1) holds.
- Consider $(p_i, p_{i-1}):(p_{i-1}, p_{i+1}) \in E_i$ and $q_1 \in F_i$, we will consider two cases depending on whether $\phi(p_{i-1}) = q_1$.

1) If $\phi(p_{i-1}) = q_1$, by (em1- c1), $\sum_{(x_i(a,b,c,d,k,\ell))} X_{x,i,a} = 1$. By (em1- c2),

$\sum_{(x_i(a,b,c,d,k,\ell))} X_{x,i,a} = 0$. Since

$l = \sum_{(x_i(a,b,c,d,k,\ell))} X_{x,i,a} = \sum_{(x_i(a,b,c,d,k,\ell))} X_{x,i,a} + \sum_{(x_i(a,b,c,d,k,\ell))} X_{x,i,a}$, we have

28
\[ \sum_{i=0}^{p-2} X_{i+1, p} = 1. \text{ Thus } \sum_{i=0}^{p-2} X_{i+1, p} = 0 = \sum_{i=0}^{p-2} X_{i+1, p} - X_{0, p} - X_{p, p}, \text{ i.e., } \]

\[ (cm2 \cdot c2) \text{ holds.} \]

2) In the other case, \( \phi(p_{in}) \neq q_i \). Then

\[ \sum_{i=0}^{p-2} X_{i+1, p} = 0 \quad \text{and} \quad \sum_{i=0}^{p-2} X_{i, p+1} = 1. \text{ Since} \]

\[ 1 = \sum_{i=0}^{p-2} X_{i+1, p} = \sum_{i=0}^{p-2} X_{i, p+1} + \sum_{i=0}^{p-2} X_{i+1, p} = 1, \text{ then} \]

\[ \sum_{i=0}^{p-2} X_{i+1, p} = 0 \quad \text{and} \quad \sum_{i=0}^{p-2} X_{i+1, p} - \sum_{i=0}^{p-2} X_{i, p+1} = 0 - \sum_{i=0}^{p-2} X_{i, p+1} - \sum_{i=0}^{p-2} X_{i+1, p}, \text{ i.e., } \]

\[ (cm2 \cdot c2) \text{ holds as well.} \]

Second, we need to prove if the variables \( X_{i, p} \) defined by \( cm \cdot x \) satisfy constraints \((cm2 \cdot c1), (cm2 \cdot c2), (cm1 \cdot c3), \) and \((cm1 \cdot c4)\). \( X_{i, p} \) also satisfy constraints \((cm1 \cdot c1) \) and \((cm1 \cdot c2)\).

- We will show that for every protein edge \((p_i, p_j) \in E_{pi}, \sum_{i=0}^{p-2} X_{i, p} = 1. \text{ Since} \]

\[ \sum_{i=0}^{p-2} X_{i, p} = \sum_{i=0}^{p-2} X_{i, p} = \sum_{i=0}^{p-2} X_{i, p} = \sum_{i=0}^{p-2} X_{i, p} = \sum_{i=0}^{p-2} X_{i, p}, \text{ and} \]

\[ \sum_{i=0}^{p-2} X_{i, p} = 1, \text{ by induction, } \sum_{i=0}^{p-2} X_{i, p} = 1 \text{ for every } i = 0, \ldots, p-2. \text{ Thus we have } (cm1 \cdot c1) \text{ hold.} \]

- Consider \((p_i, p_j) \in E_{pi} \) and some \( q_i \in V_{p} \), we will consider two cases depending on whether \( \phi(p_{in}) = q_i \).
1) If \( p_i r_f - q_i \), then \( \sum_{i \in \mathcal{I}, f \in \mathcal{F}, \mathcal{A}, \mathcal{B}} X_{i,f,a,b} = 1 \). By (em1-c1) and (em2-c2),

\[
\sum_{i \in \mathcal{I}, f \in \mathcal{F}, a \in \mathcal{A}, b \in \mathcal{B}} X_{i,f,a,b} = 1.
\]

Therefore, by (em1-c1) again,

\[
\sum_{i \in \mathcal{I}, f \in \mathcal{F}, a \in \mathcal{A}, b \in \mathcal{B}} X_{i,f,a,b} + \sum_{i \in \mathcal{I}, f \in \mathcal{F}, a \in \mathcal{A}, b \in \mathcal{B}} X_{i,f,a,b} \leq 1,
\]

thus \( \sum_{i \in \mathcal{I}, f \in \mathcal{F}, a \in \mathcal{A}, b \in \mathcal{B}} X_{i,f,a,b} = 0 \),

therefore \( \sum_{i \in \mathcal{I}, f \in \mathcal{F}, a \in \mathcal{A}, b \in \mathcal{B}} X_{i,f,a,b} = 1 + 0 = 1 \), we have

(\text{em1-c2}) holds;

2) In the other case, \( \sum_{i \in \mathcal{I}, f \in \mathcal{F}, a \in \mathcal{A}, b \in \mathcal{B}} X_{i,f,a,b} = 0 \). By (em1-c1), \( \sum_{i \in \mathcal{I}, f \in \mathcal{F}, a \in \mathcal{A}, b \in \mathcal{B}} X_{i,f,a,b} = 1 \),

then by (em2-c2),

\[
\sum_{i \in \mathcal{I}, f \in \mathcal{F}, a \in \mathcal{A}, b \in \mathcal{B}} X_{i,f,a,b} = 1,
\]

we have

\[
\sum_{i \in \mathcal{I}, f \in \mathcal{F}, a \in \mathcal{A}, b \in \mathcal{B}} X_{i,f,a,b} + \sum_{i \in \mathcal{I}, f \in \mathcal{F}, a \in \mathcal{A}, b \in \mathcal{B}} X_{i,f,a,b} = 0 + 1 = 1.
\]

Therefore (em1-c2) holds as well.

Since we have shown that (em1-c2) is equivalent to (em1-c2-v2), we claim that conditions (em1-c1) to (em1-c4) are satisfied, if and only if (em2-c1), (em1-c2-v2), (em1-c3) and (em1-c4) are also satisfied.

### 3.5 Efficiency of the IP Models

The efficiency of these IP models is judged in terms of the computational time needed to handle the same protein on CPLEX. If all other conditions are the same, the computational time is affected only by the size of the model, which is measured by three criteria:

- Number of constraints in the model, which is denoted by \( N_{\text{cont}} \);
- Number of variables in the model, which is denoted by $N_{va}$.
- Number of occurrences of variables in the model, which is denoted by $N_{occu}$.

Table 3.1 shows the $N_{va}$, as well as the $N_{cont}$ and $N_{occu}$ for each type of constraints in each IP model.

<table>
<thead>
<tr>
<th></th>
<th>$N_{va}$</th>
<th>$N_{cont}$</th>
<th>$N_{occu}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C1</td>
<td>C2</td>
<td>C3</td>
</tr>
<tr>
<td></td>
<td>PQ</td>
<td>P</td>
<td>Q</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>Q</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>(P-1)Q</td>
<td>(P-1)M</td>
<td>(P-1)Q</td>
</tr>
<tr>
<td></td>
<td>(P-1)Q</td>
<td>(P-1)M</td>
<td>(P-1)Q</td>
</tr>
<tr>
<td></td>
<td>(P-1)Q</td>
<td>(P-1)M</td>
<td>(P-1)Q</td>
</tr>
</tbody>
</table>

We did experiments to test the efficiency of the three IP models:
- IP vertex model with constraints ($vm_1 - c_1$), ($vm - c_2$) and ($vm - c_3$);
- IP edge model I with constraints ($em_1 - c_1$), ($em_1 - c_2$), ($em_1 - c_3$) and ($em_1 - c_4$).
- IP edge model II with constraints ($em_2 - c_1$), ($em_2 - c_2$), ($em_1 - c_3$) and ($em_1 - c_4$).

Figure 3.1 shows that the number of lattice vertices $Q$ is growing linearly with the number of residues in protein $P$, where $Q$ is counted from the 3D cubic lattice which is generated to have the same boundaries of $x$, $y$, $z$ coordinates as that of the protein’s in our testing dataset (the detail of testing dataset and lattice generation are described in the later sections). The ratio of $Q/P$ ranges
from 2.4 to 16.1, and the average ratio is 7.4427. The number of lattice edges \( M \) roughly equals to \( Q \times d \), where \( d \) is the degree of the lattice which is a constant.

**Figure 3.1** Number of lattice vertices \((Q)\) is linear to the number of residues in protein \((P)\)

The \( N_{va} \) is \( O(P^2) \) in all models, with the smallest constant in the vertex model. The \( N_{cont} \) is \( O(P) \) in the edge model I, and \( O(P^3) \) in the other two models, where there is a smaller constant for \( N_{cont} \) in vertex model than in the edge model I. The \( N_{cont} \) is \( O(P) \) in the edge model II, and \( O(P^3) \) in the other two models, where the constant is smaller in the vertex model than in the edge model I.

Based on the observation of our experiments, edge model II performs the best. It is clear that although \( N_{va} \) and \( N_{cont} \) are important factors, it seems that \( N_{va} \) is the most dominating term. For instance, edge model II performs better than vertex model even though it has larger \( N_{va} \) and \( N_{cont} \). It was the reason...
why in formulations we merged several constraints to one in order to minimize $N_{cont}$. 
CHAPTER 4: NARROWING DOWN THE SEARCH SPACE

We tried to run the most efficient IP edge model II using CPLEX. We observed that even with IP edge model II which CPLEX would easily run out of memory on protein of 263 residues; even for smaller sized proteins of 150 residues, CPLEX would run for an unacceptable long time (longer than an hour) before producing a solution. This is due to the large size of IP model. Since there the size of the protein is fixed, the only way to make the problem solvable by CPLEX is to reduce the value of $Q$ and therefore also the value of $M$.

4.1 Sight Distance and Sight Sets

We tried to reduce the size of our IP model by only mapping each protein vertex $p_i$ to a set of lattice vertices within a short distance (called sight distance) away from $p_i$. For each protein vertex, we determine a safe value for the sight distance such that in an optimal lattice approximation for the protein, every lattice vertex that maps to the corresponding protein vertex is within the corresponding sight distance.

The set of lattice vertices within the sight distance from a protein vertex will be called the sight set of the vertex. Sight distance which associates to protein vertex $p_i$ is denoted by $sd(p_i)$, and the sight set which are within $sd(p_i)$ distance away from $p_i$ is denoted as $v_{\text{sight}}(p_i)$, formally defined in (4.1).
\[ \text{vsight}(p_i) = \{ q_j \in V_l : \text{dist}(p_i, q_j) \leq \text{sd}(p_i) \} \] (4 - 1)

We will determine \( \text{sd}(p_i) \) in such a way that \( \text{cdist} \) of a vertex map \( \phi : V_p \rightarrow V_l \) such that if \( p_i \) were mapped to some lattice vertex \( q_j \in V_l \setminus \text{vsight}(p_i) \), the solution would not be optimal. If \( \text{sd}(p_i) \) has the above property, then it is called safe. Because \( \text{sd}(p_i) \) is determined to be safe, the search space for each protein vertex can be reduced to \( \text{vsight}(p_i) \), and the search space for all protein vertices can be reduced to the union of \( \text{vsight}(p_i) \) over all protein vertices, represented by \( \bigcup_{p_i \in V_p} \text{vsight}(p_i) \) as follows.

\[ \text{vsight}(V_p) = \bigcup_{p_i \in V_p} \text{vsight}(p_i) \] (4 - 2)

The set of lattice edges whose initial and terminal vertices are within the sight distances from the corresponding initial and terminal vertices of a protein edge will be called the sight set of the edge. Using sight set of protein vertices, we define sight set of protein edge \( b_j \), \( \text{esight}(b_j) \) as follows.

\[ \text{esight}(b_j) = \{ e_j = (q_k, q_k) \in E_l : q_k \in \text{vsight}(p_i), q_k \in \text{vsight}(p_m) \} \] (4 - 3)

It is easy to see that the search space can be reduced to \( \text{esight}(b_j) \) for each protein edge \( b_j \), and to the union of \( \text{esight}(b_j) \) for all protein edges, which is denoted by \( \text{esight}(E_v) \) as formally defined in (4 - 4).

\[ \text{esight}(E_v) = \bigcup_{b_j \in E_v} \text{esight}(b_j) \] (4 - 4)
4.2 Revising IP Models by Using Sight Sets

We revised the three IP models in Chapter 3 by using sight sets as shown in the following subsections.

4.2.1 Revised IP Vertex Model

In the revised IP vertex model, all contents are the same as the original IP vertex model except as specified below.

Let $\phi$ be a vertex map. Define a binary variable $X_{i,j}$ for every $p_i \in V_p$ and $q_j \in \text{vsight}(p_i)$ as:

$$X_{i,j} = \begin{cases} 1, & \text{if } \phi(p_i) = q_j; \\ 0, & \text{otherwise}. \end{cases}$$

The objective function is:

$$\text{Minimize } \sum_{p_i \in V_p, q_j \in \text{vsight}(p_i)} c_{i,j} \cdot X_{i,j}$$

Subject to constraints:

1. Each protein vertex $p_i$ maps to an unique lattice vertex $q_j$ within its sight set $\text{vsight}(p_i)$:

$$\forall p_i \in V_p: \sum_{q_j \in \text{vsight}(p_i)} X_{i,j} = 1$$

2. No lattice vertex is mapped to any two different protein vertices:

$$\forall q_j \in \text{vsight}(V_p): \sum_{p_i \in V_p, q_j \in \text{vsight}(p_i)} X_{i,j} \leq 1$$

3. Any two adjacent protein vertices are not mapped to two lattice vertices which are not adjacent in $G_j$.
We claim that if $sd(p_i)$ are determined safe, then conditions (vm - c1), (vm - c2) and (vm - c3) (or (vm - c3 - v2) ) are satisfied, if and only if (vms - c1), (vms - c2) and (vms - c3)(or (vms - c3 - v2)) are satisfied. To prove this, notice that the only difference between IP vertex model (referred to as the original model) and revised IP vertex model (referred to as the revised model) is, in the original model, each protein vertex $p_i$ can map to any lattice vertex in the whole lattice; while in the revised model, each protein vertex can only map to lattice vertex within its sight set $\text{vsight}(p_i)$. Suppose for a protein $G_i = (V_{i}, E_{i})$, the solution produced by original model $\phi$ is different from the solution produced by revised model $\phi'$ in the way that there is a protein lattice $p_i$ such that $\phi(p_i) \in V_{i} - \text{vsight}(p_i)$, then either $sd(p_i)$ is not safe, or $\phi(p_i)$ is not the $i$-th vertex in the optimal solution. Since neither of the two cases is true, the original model is equivalent to the revised model.

4.2.2 Revised IP Edge Model I

In the revised IP edge model I, all contents are the same as the original IP edge model I except as specified below.
Let \( \varphi \) be an edge map. Define a binary variable \( X_{i,j,b} \) for every \( b \in E_\varphi \),

\[
(q_i,q_j) \in \text{esight}(b) \quad \text{as:}
X_{i,j,b} = \begin{cases} 
1 & \text{if } \varphi(b) = (q_i,q_j); \\
0 & \text{otherwise}.
\end{cases}
\]

The objective function is:

\[
\text{Minimize } \sum_{(b,b') \in \text{esight}(b)} X_{i,j,b} \quad (\text{ems} \cdot x)
\]

Subject to constraints:

1. Every protein edge maps to an unique lattice edge within its sight set \( \text{csight}(b) \):

\[
\forall b_i = (p_i,p_{i+1}) \in E_\varphi : \sum_{(b,b') \in \text{esight}(b)} X_{i,j,b} = 1 \quad (\text{ems} \cdot c1)
\]

2. Any two adjacent protein edges are mapped to two adjacent lattice edges:

\[
\forall i = 0,...,P-3, \forall q_i \in \text{esight}(p_i) : \sum_{b_i \in b_i} X_{i,j,b} + \sum_{b_i \in b_i} X_{i,j,b} = 1 \quad (\text{ems} \cdot c2)
\]

or

\[
\forall i = 0,...,P-3, \forall q_i \in \text{esight}(b_i) : \sum_{b_i \in b_i} X_{i,j,b} \geq X_{i,j,b} \quad (\text{ems} \cdot c2 \cdot v2)
\]

3. No merging lattice edges in the solution set, i.e., no two different lattice edges in the lattice approximation solution can have the same terminal vertex:

\[
\forall q_i \in \text{esight}(p_i) : \sum_{b_i \in b_i} X_{i,j,b} \leq 1 \quad (\text{ems} \cdot c3)
\]

4. No lattice edge in the mapping solution ends at the lattice vertex which maps to the first protein vertex:

\[
\forall(q_i,q_j) \in \text{esight}(b) : \sum_{(b,b') \in \text{esight}(b)} X_{i,j,b} \leq 1 \quad (\text{ems} \cdot c4)
\]
As argued in Section 4.2.1 if $sd(p_i)$ are determined safe, then $(\text{ems1-c1}), (\text{ems1-c2})$ (or $(\text{ems1-c2-v2}))$, $(\text{ems1-c3})$ and $(\text{ems1-c4})$ are satisfied, if and only if $(\text{ems1-c1})$, (ems1-c2)(or (ems1-c2-v2)), (ems1-c3) and (ems1-c4) are also satisfied.

4.2.3 Revised IP Edge Model II

In the revised IP edge model II, all contents are the same as the original IP edge model II except as specified below.

Subject to constraints:
1. The first protein edge $b_i = (p_i, p_i) \in E_i$ maps to an unique edge $(q_i, q_i) \in \text{esight}(b_i)$ in lattice.
   \[ \sum_{(q_i, q_i) \in \text{esight}(b_i)} X_{b_i, q_i} = 1 \]  
   (ems2-c1)
2. Any two adjacent edges $(p_i, p_{i+1}), (p_{i+1}, p_{i+2}) \in E_i$ must be aligned to two adjacent edges $(q_i, q_{i+1}) \in E_i$ and $(q_{i+1}, q_{i+2}) \in E_i$.
   \[ \forall i = 0, \ldots, P-3, \forall q_i \in \text{esight}(V_i) \colon \sum_{(q_i, q_i) \in \text{esight}(V_i)} X_{b_i, q_i} - \sum_{(q_i, q_i) \in \text{esight}(V_i)} X_{b_i, q_i} = 0 \]  
   (ems2-c2)
   or use constraint (ems1-c2-v2).
3. The same as (ems1-c3).
4. The same as (ems1-c4).
Similarly if \( \text{sd}(p_i) \) are determined safe, then \((\em1 - c1), (\em2 - c2),(\em1 - c2 - v2)), \((\em1 - c3)\) and \((\em1 - c4)\) are satisfied, if and only if \((\em2 - c1), (\em2 - c2),(\em1 - c2 - v2)), \((\em1 - c3)\) and \((\em1 - c4)\) are also satisfied.

### 4.3 Determining Safe Sight Distances

In order to reduce the search space, we want to minimize \( \text{sd}(p_i) \) such that it is not smaller than the distance between the protein vertex \( p_i \) and its lattice vertex image in the optimal lattice approximation, otherwise the resulting model would not produce the optimal lattice approximation for the whole protein chain.

Given a lattice approximation of protein \( G_r = (V_r, E_r) \), let us define \( q_{w_i} \) as a lattice vertex which maps to a protein vertex \( p_i \).

Given a protein \( G_p = (V_p, E_p) \), if \( \{\phi(p_i)\}_{i=0}^{n} \) is the optimal lattice approximation for the whole protein \( G_p \), then we denote the \( \text{cdist}(\{p_i\}_{i=0}^{n}, \{\phi(p_i)\}_{i=0}^{n}) \) for the protein substring \( \langle p_u, p_v \rangle \ (v, p_v, \in V_p, \ u < v) \) by \( \text{cdist}_{u,v}(u,v) \), a partial cost of global optimal solution for protein \( G_p \). I.e.,

\[
\text{cdist}_{u,v}(u,v) = \sum_{i=0}^{n} d^i(p_i, \phi(p_i));
\]

where

\[
\sum_{i=0}^{n} d^i(p_i, \phi(p_i)) = \min \{d^2(p_{j+1}, q_{\alpha_{1}}) + d^2(p_{j+1}, q_{\alpha_{2}}) + \ldots + d^2(p_{j}, q_{\alpha_{n}}) ; q_{\alpha_{n}} \in N(q_{\alpha_{n}}), \forall i \neq j, q_{\alpha_{n}} \neq q_{\alpha_{n}} \} \ u < v
\]

If \( \{\phi(p_i)\}_{i=0}^{n} \) is the optimal lattice approximation for the protein substring \( \langle p_u, p_v \rangle \ (p_u, p_v, \in V_r, \ u < v) \), then we denote \( \text{cdist}(\{p_i\}_{i=0}^{n}, \{\phi(p_i)\}_{i=0}^{n}) \) by
In this case, \( \{p(p)\}_{\text{opt}} \) can be viewed as a local optimal solution for protein \( G \), i.e.,

\[
\text{cdist}_{\text{opt}}(u,v) = \min \left\{ d^2(p_u,q_u) + d^2(p_v,q_v) + \ldots + d^2(p_u,q_v) \right\},
\]

\( q_u \in V, q_v \in V(q_u), \forall i \neq j, q_u \neq q_v \) \( u < v \) (4 - 6)

Take any lattice approximation \( \{p(p)\}_{\text{lat}} \) of the whole protein \( G \), let

\[
\text{cdist}_{\text{lat}}(u,v) = \text{cdist}\{p(p)\}_{\text{lat}}(u,v),
\]

as it can be viewed as an upper bound for \( \text{cdist}_{\text{opt}}(u,v) \) and \( \text{cdist}_{\text{opt}}(u,v) \). Depending how the lattice approximation \( \{p(p)\}_{\text{lat}} \) is computed, \( \text{cdist}_{\text{lat}}(u,v) \) is denoted by \( \text{cdist}_{\text{greedy}}(u,v) \) if its value is computed by using greedy algorithm, or by \( \text{cdist}_{\text{CPLEX}}(u,v) \) if its value is computed by using CPLEX to solve IP models with fixed sight distance of 10Å.

The later algorithm will be further explained in later sections.

For a protein vertex \( p_i \in V \), let \( \{p(p)\}_{\text{opt}} \) be the global optimal lattice approximation for the whole protein \( G \). Let

\[
\text{sd}(p_i) = \sqrt{\text{cdist}_{\text{opt}}(0,P-1) - (\text{cdist}_{\text{opt}}(0,i-1) + \text{cdist}_{\text{opt}}(i+1,P-1))}
\]

We now prove that \( \text{sd}(p_i) \)'s are safe if being determined as (5 - 7). Assume \( \phi \) is an optimal vertex map for the whole protein. It is easy to see that

\[
\text{cdist}_{\text{opt}}(0,P-1) = \sum_{i=0}^{P-1} d^2(p_i,\phi(p_i)) = d^2(p_0,\phi(p_0)) + \sum_{i=1}^{P-1} d^2(p_i,\phi(p_i))
\]

\[
= \text{cdist}_{\text{opt}}(0,i-1) + d^2(p_i,\phi(p_i)) + \text{cdist}_{\text{opt}}(i+1,P-1)
\]

\[
\leq \text{cdist}_{\text{lat}}(0,P-1)
\]

41
Therefore, 
\[ d^2(p, \phi(p)) \leq \text{cdist}_{\text{opt}}(0, P-1) - (\text{cdist}_{\text{opt}}(0, i-1) + \text{cdist}_{\text{opt}}(i+1, P-1)) \]
\[ \leq \text{cdist}_{\text{opt}}(0, P-1) - (\text{cdist}_{\text{opt}}(0, i-1) + \text{cdist}_{\text{opt}}(i+1, P-1)) \]
\[ = \text{sd}^2(p) \]

Therefore, 
\[ d(p_{\text{opt}}, \phi(p)) \leq \text{sd}(p) \] (4-8)

for any optimal vertex map \( \phi \), i.e., \( \text{sd}(p) \)'s defined by (4-7) are safe.

Since \( \text{cdist}_{\text{opt}}(0, i-1) \) and \( \text{cdist}_{\text{opt}}(i+1, P-1) \) can be hard to compute if the protein is large, or if \( p \) is close to the ends of the protein chain, when computing the optimal approximation for a larger part of the protein is almost equal to computing the optimal approximation for the whole protein. To solve this problem, we present (4-9) as a relaxation of (4-7).

\[ \text{sd}(p) = \sqrt{\text{cdist}_{\text{opt}}(0, P-1) - \text{cdist}_{\text{opt}}(0, i, P-1)} \] (4-9)

We will call \( \text{cdist}_{\text{opt}}(0, i, P-1) \) the \( i \)-th complementary lower bound for \( \text{cdist}_{\text{opt}}(0, P-1) \) (abbreviated as the \( i \)-th complementary lower bound) if it has the following property:

\[ \text{cdist}_{\text{opt}}(0, i, P-1) \leq \text{cdist}_{\text{opt}}(0, i-1) + \text{cdist}_{\text{opt}}(i+1, P-1) \] (4-10)

Obviously, the above proof shows that \( \text{sd}(p) \)'s defined by (4-9) are safe.

Methods for computing \( \text{cdist}_{\text{opt}}(0, i, P-1) \) and \( \text{cdist}_{\text{opt}}(0, i, P-1) \) are stated in the following subsections.

4.3.1 Finding \( \text{cdist} \) Upper Bound for the Optimal Lattice Approximation

The cost \( \text{cdist} \) of any lattice approximation can be viewed as an upper bound for \( \text{cdist} \) of the optimal lattice approximation. A good and feasible upper bound
for $c_d(i)$ the optimal lattice approximation can be obtained from a lattice approximation which is very close to the optimal, and can be computed efficiently. Various methods have been used to find lattice approximation of protein. Due to our need to analyze a large number of structures in a short amount of time, the greedy algorithm by Mead et al. is the most suitable and the one we adopted. A feasible $\text{ahead}$ value for cRMS value was chosen as 10 in the Phase 3 in the greedy algorithm. The $c_d(i)$ of the lattice approximation for protein $G_r = (V_r, E_r)$ computed by the above greedy algorithm is represented by $c_d_{\text{greedy}}(0, P - 1)$.

From experiments, we found it is necessary to compute different sight distances (i.e., $s_d(p_i)$) values for different protein vertices, since assigning a same value to all $s_d(p_i)$'s results in IP models too large to be solved by CPLEX. For example, CPLEX fails to efficiently solve a protein with 263 residues when all $s_d(p_i)$'s are assigned to a fixed value of 20Å. We also observed from the experiments that for every protein in our database, CPLEX is able to solve the IP edge model II generated by assigning a fixed 10Å value to $s_d(p_i)$'s for all protein vertices, and find the lattice approximation for that protein successfully within time 1 second to 31 minutes, depending on the length of the protein. Since we cannot prove that such $s_d(p_i)$'s are safe, the $c_d(i)$ of the lattice approximation for protein $G_r = (V_r, E_r)$ found by this method, which is represented by $c_d_{\text{CPLLEX}(10\text{Å})}(0, P - 1)$, can only be used as the upper bound for $c_d(i)$ of the optimal lattice approximation.
The above two algorithms are used to produce approximate solutions. We can find the exact solution by using a backtracking algorithm as described in Algo(4-I). In Algo(4-I), for a protein substring $p_s,...,p_e$ of length $l$, we try to generate paths in the lattice of the same length as lattice approximations for the protein substring. The lattice approximation with minimum $cdist$ cost is the optimal lattice approximation. To generate a candidate path in the lattice, we map every lattice vertex $q_j \in V_L$ to the first protein vertex $p_s$, and start generating the rest of the path from $q_j$. An one-dimensional integer array $neih$ of size $l$ is used and candidate paths are generated according to the value of the elements in array $neih$. Each lattice vertex in the neighbour set of a lattice vertex is assigned with an unique index number between 0 to $d-1$. Given the mapping of the $i$-th protein vertex $q_i$, and the value of the $i$-th element in array $neih$, i.e.,

$neih[i] = \{0 \leq n \leq d-1\}$, the mapping of the $(i+1)$-th protein vertex is the neighbour of $q_i$ with index number $n$. Several techniques are applied to speed up the process by avoiding enumerating all possible paths in the lattice. The detailed description of Algo(4-I) can be found in Appendix B.1. Since in most cases, computing exact solution for the whole protein chain by Algo(4-I) is not feasible, we only use Algo(4-I) to compute the exact solutions for short protein substrings, as applied in Section 4.3.2.2.

**Algo(4-I): Compute $cdist_{st}(s,c)$ by backtracking approach**

**Input:** A protein substring specified by the starting index $s$ and ending index $c$. 

44
Output \( \text{cdist}_{v_p}(s,e) \)

1: \( \text{cdist} \leftarrow \infty \quad \text{//During the execution of the algorithm, the variable stores the best \( \text{cdist}(s,e) \) encountered by far. At the end of the execution, it stores the value of \( \text{cdist}_{v_p}(s,e) \).} \)

2: compute the length of protein substring \( l \)

3: if \( (l \leq 0) \) then

4: \( \text{cdist} \leftarrow 0.0 \)

5: else

6: if \( (l = 1) \) then

7: \( \text{cdist} \leftarrow \text{Square of distance between protein vertex } s \) and its closest lattice vertex

8: else

9: for all \( q_j \in V_l \) do

10: \( \text{//Phase 1: Generate a valid path in the lattice} \)

11: Map \( p_i \) to \( q_j \).

12: if \( \text{(the value of the last element } neib[l-1] \text{ is 0)} \) then

13: Starting from \( q_j \), generate a path in the lattice as the lattice approximation of the protein substring by finding the candidate lattice image for the next protein vertex. The neighbour sets information of all lattice vertices are stored in an one-dimensional array \( N \) of size \( Q \). \( N[k] \) points to another one-dimensional array with size up to \( d-I \), in which the neighbours of lattice vertex \( q_j \) are recorded. If protein vertex \( p_{i+1} \) maps to the lattice vertex \( q_j \), then the candidate lattice image for protein vertex \( p_{i+1} \) is the lattice vertex \( q_j \), where \( \nu = N[k][\text{neib}[i]] \). If \( q_j \) is already the image of a protein vertex before \( p_{i+1} \), we record that it is required to carry out a Fast Increase at current position \( i \). We also need to make sure that the cdist value of current partial path from \( q_j \) up to \( q_j \) does not exceed the minimum cdist value for the whole protein and its lattice approximations we have got by far. Otherwise we record that it is required to carry out a Fast Increase at current position \( i \).

14: If we have generated a complete valid path, we compute its corresponding cdist cost and use it to update the minimum value of cdist cost in our record.

15: \( \text{//Phase 2. Increasing Phase} \)
if it is required to do fast Increase then
   Do Fast Increase: regenerate candidate path starting from the recorded position
else
   Do Normal Increasing: regenerate candidate path starting from the last position
end if

Phase 3: Refreshing Phase

Step 7: Check the values of every element in array neib: if the value of neib[i] is larger than \( d - 1 \), refresh its value to be 0, and increase the value of neib[i-1] by 1. If the value of neib[0] is larger than \( d - 1 \), refresh its value to be 0, and refresh the value of the last element neib[f-1] to be 1.

Step 2: Increase the value of the second last element neib[f-2] by 1.

end if

end for

end if

end if

return cdist

4.3.2 Finding \( i \)-th Complimentary \( cdist \) Lower Bound for the Optimal Lattice Approximation

In the following subsections, we describe the algorithms to find the \( i \)-th complimentary lower bound \( cdist_{i,\theta}(0,i,P-1) \) as defined in (4.9).

4.3.2.1 Use of Closest Lattice Vertices

We adopt the same strategy as in the Phase 1 of the greedy algorithm developed by Mead et al. For a protein substring \( (p, ..., p) \), \( p \in V, s < e \), find the closest lattice vertex for each protein vertex, and use the \( cdist \) between all
protein vertices and their corresponding closest lattice vertices as the lower bound of \( c_{\text{dist}}(s,c) \).

Let \( C(p_i) \in V_s \) be the closest lattice vertex for protein vertex \( p_i \in V_s \), and \( c_{\text{dist}}(i) \) be the square of distance between protein vertex \( p_i \) and its closest lattice vertex, i.e.,

\[
c_{\text{dist}}(i) = d^2(p_i, C(p_i)) = \min\{d^2(p_i, q_j); q_j \in V_s\} \tag{4-11}
\]

Let

\[
c_{\text{dist}}^{(0,i,P-1)} = \sum_{j=1}^{i} c_{\text{dist}}(j) + \sum_{j=i+1}^{P-1} c_{\text{dist}}(j) \tag{4-12}
\]

It is easy to see that \( c_{\text{dist}}^{(0,i,P-1)} \leq c_{\text{dist}}^{(0,P-1)} + c_{\text{dist}}^{(i+1,P-1)} \).

Hence, \( c_{\text{dist}}^{(0,i,P-1)} \) defined above is indeed an \( i \)-th complementary lower bound.

The computation of all \( C(p_i) \)'s and \( c_{\text{dist}}(i) \)'s can be done in time \( O(PQ) \).

### 4.3.2 Use of Tuples

The method given in the previous subsection is naive and does not necessarily produce good lower bounds. We will further improve it by dividing the protein substring \( p_s, \ldots, p_s, p_e \in V_s, s < e \) into tuples. Starting from the first vertex in the protein chain, \( p_s \), we view a certain number of consecutive protein vertices as a tuple. We call the number of protein vertices in a tuple the tuple size, \( r \), and denote the number of tuples in a protein as \( T \). All tuples but the last one contain the tuple size protein vertices, i.e., tuple \( t_i = p_s, \ldots, p_{s+i-1}, i \in [0, T-2] \), and the last tuple is \( t_{T-1} = p_s, \ldots, p_{s+T-2} \). For convenience, we also represent each tuple as \( t_i = p_{s+i}, i \in [0, T-1] \). Note
that the last protein vertex $p_{9r}$ in each tuple $t_i = p_{0}, p_{1}, ..., p_{r}, i \in [0, T - 2]$ is adjacent to the first protein vertex $p_{9l}$ in the next tuple $t_{i+1} = p_{9l}, ..., p_{9r}, s.t.

\[ p_{9l+1} = p_{9r} \]

Let us represent the $cdist$ for tuple $t_i$, $cdist_{\text{ext}}(St_i, Et_i)$ as $cdist_{\text{ext}}(t_i)$. More formally,

\[
\begin{align*}
\text{cdist}_{\text{ext}}(t_i) &= \text{cdist}_{\text{ext}}(St_i, Et_i) = \text{cdist}_{\text{ext}}(i \times \tau, (i + 1) \times \tau - 1), i \in [0, T - 2] \\
\text{cdist}_{\text{ext}}(t_{i+1}) &= \text{cdist}_{\text{ext}}((T - 1) \times \tau, P - 1)
\end{align*}
\]

We can compute $\text{cdist}_{\text{ext}}(t_i)$ for every tuple $t_i$, $\text{cdist}_{\text{ext}}(St_i, i - 1)$ and $\text{cdist}_{\text{ext}}(i + 1, Et_i)$ by using $\text{Algo}(4 - 1)$. Then the $i$-th complementary lower bound can be computed as follows.

\[
\begin{align*}
\text{cdist}_{\text{ext}}(0, i, P - 1) &= \sum_{j=0}^{i-1} \text{cdist}_{\text{ext}}(t_j) + \text{cdist}_{\text{ext}}(St_j, i - 1) \\
&\quad + \text{cdist}_{\text{ext}}(i + 1, Et_i), \text{where } p_i \in t_i, p_{i+1} = p_{9l}, ..., p_{9r}.
\end{align*}
\]

Since $\text{Algo}(4 - 1)$ is a backtracking algorithm, the time is growing exponentially to the tuple size. To compute each $\text{cdist}_{\text{ext}}(0, i, P - 1)$, we need to call $\text{Algo}(4 - 1)$ $T + 1$ times, which decreases linearly as tuple size $r$ increases. Therefore by increasing tuple size, we significantly increase the computational time for $\text{cdist}_{\text{ext}}(t_i)$, $\text{cdist}_{\text{ext}}(St_i, i - 1)$ and $\text{cdist}_{\text{ext}}(i + 1, Et_i)$, while gain a slightly better $\text{cdist}_{\text{ext}}(0, i, P - 1)$ value. In total this causes a significant increase of the computational time for processing the whole protein. A feasible tuple size $r$ is less than 20.
4.3.2.3 Use of Mega Tuples

From our experiments, we found that \( c_{dist}^\text{opt}(0, i, P - 1) \) has larger values when computed using tuples. This does help to solve proteins with more vertices than before, but it still is not efficient enough to solve the largest proteins. For example, CPLEX ran out of memory when processing protein 1ARB_A1 with 263 vertices, even though we set its tuple size to be 20. But we did observe that CPLEX solved in a short time the protein IDW_9_A1 with 56 vertices, when its tuple size was set to 8. This inspires us to further group a number of consecutive protein vertices into **mega tuples**, whose size is much larger than the tuple size. Each mega tuple is processed just like a single protein. The intuition is that after obtaining sight distances by using tuples, one IP model is generated for each mega tuple, and solved by CPLEX to find the \( c_{dist} \) of the optimal lattice approximation for that mega tuple. Then the \( c_{dist} \)'s for mega tuples and the \( c_{dist} \)'s for tuples would be used to compute smaller sight distances, which might enable us to process longer proteins.

Starting from the first protein vertex, every \( \zeta \) protein vertices form a mega tuple. \( \zeta \) is called the **mega tuple size**. The total number of mega tuples is denoted by \( \Omega \). From the first mega tuple to the second last one, each mega tuple is represented as \( m_{i} = \{v_{i}, \ldots, v_{(\Omega - 2)i}, t_{i} \} \), and the last mega tuple is \( m_{(\Omega - 1)i} = \{v_{(\Omega - 2)i}, \ldots, v_{n} \} \). If the first and the last protein vertices of mega tuple \( m_{i} \) happen to be the first protein vertex of tuple \( t_{j} \) and the last protein vertex of tuple \( t_{k} \) respectively, then we can represent \( m_{i} \) by \( m_{i} = \{t_{j}, \ldots, t_{k} \} \). If \( m_{i} = \{t_{j}, \ldots, t_{k} \} \),
then \( t_i \) and \( t_j \) are those tuples inside which the mega tuple begins and ends. We represent the \( \text{cdist} \) of the optimal lattice approximation for mega tuple \( m_t \) by \( \text{cdist}_{opt}(m_t) \).

We will compute \( \text{cdist}_{opt}(0, i, P - 1) \) as follows.

\[
\text{cdist}^{\text{opt}}_{opt}(0, i, P - 1) = \sum_{t \in \text{mt}_{i}, t \neq t_i} \text{cdist}_{opt}(m_t) + \sum_{t \in \text{mt}_{i}, t = t_i} \text{cdist}_{opt}(m_t, (t_i, ..., s_l, ..., t_j), w_x z, \text{vt}_{t_i}, n_z) + \text{cdist}_{opt}(s_l, (t_i, ..., s_l, ..., t_j), (u \times z, \text{vt}_{t_i}, n_z)) \quad (4-15)
\]

Equation (4-15) shows the situation where \( p_i \) in tuple \( t_i \), and \( t_j \in m_t \), \( m_t = (t_i, ..., t_j) \), as illustrated in Figure 4.1.

**Figure 4.1** An example of tuples and mega tuples in a protein

The algorithm to compute \( \text{cdist}^{\text{opt}}_{opt}(0, i, P - 1) \) by using mega tuples is described in Algo(5-2).
Algo(5-2): Compute $c_{\text{dist}}(0, i, P - 1)$ using tuples and mega tuples

Input: $G_i = (V_i, E_i)$, $G_{i+1} = (V_{i+1}, E_{i+1})$

Output: $c_{\text{dist}}(0, i, P - 1)$

1. Divide all the protein vertices into tuples according to tuple size $r$.
2. Compute $c_{\text{dist}}(t_i)$ for every tuple.
3. Compute $c_{\text{dist}}(0, i, P - 1)$ by using $c_{\text{dist}}(t_i)$ as defined in (4 - 14).
4. Compute sight distances for all protein vertices.
5. Compute $\text{sight}$ for all protein vertices to be used later in building IP vertex models, or compute $\text{esight}$ for all protein edges to be used later in building IP edge models.
6. For all $i = 0, \ldots, \Omega - 1$
7. Build IP model: either use $\text{sight}$ to build one IP vertex model for mega tuple $m_{ti}$, or use $\text{esight}$ to build one of the IP edge models for mega tuple $m_{ti}$.
8. Solve the IP model built above by CPLEX.
9. Compute the $c_{\text{dist}}(m_{ti})$.
10. End for
11. Compute $c_{\text{dist}}(0, i, P - 1)$ as defined in (4 - 15).

To compute each $c_{\text{dist}}(0, i, P - 1)$, we need to call Algo(4 - 1) $T + S$ times in the worst case, which decreases linearly as tuple size $r$ increases, and we need to run CPLEX $\Omega - 1$ times, which decreases linearly to mega tuple size $S$.

From our experiments, we observed that using mega tuples helps reduce the search space significantly such that we can find the optimal lattice approximation for protein 1AR8_A1 with 263 residues, when tuple size is 8 and mega tuple size is 16 (note that protein 1AR8_A1 could not be solved by the previous algorithms).
4.3.2.4 Use of Groups of Tuples and Iterations

It is easy to see that if we obtain better bound for sight distances, we can solve longer proteins. Initially, the bound obtained by using tuples is quite bad and therefore mega tuples have to be quite short. After we get a better bound for sight distances, we can use this bound to run Algol(4-2) again with larger mega tuple size to get even better bounds for sight distances. Reiterating this process can improve the bounds significantly.

We observe that CPLEX would run out of memory on the longest proteins if the sight distances are fixed to 20Å, while it has no problem finding the lattice approximations for the longest proteins if the sight distances are fixed to a value smaller than 16Å. This led us to iteratively apply the algorithm in the previous section. In each iteration, we increase the mega tuple size, and solve the resulting larger mega tuples by CPLEX to get possibly smaller sight distances. We repeat the iteration until average sight distances falls below a certain threshold (16Å in our experiments), which we believe enable CPLEX to find solutions to even the longest proteins in our database.

From our experiments, we observed that the tuple size and mega tuple size need to be determined carefully, otherwise, sight distances might not decrease in the later iterations when the mega tuple size increases. For example, take a protein with 128 residues, we set the tuple size to 8 and the mega tuple size to 16 at the beginning, and then increase the mega tuple size by 8 after each iteration. As shown in Figure 4.2, the average sight distances keeps decreasing until iteration IV, since the gain for having more vertices in the first
The address the above problem, we divide a protein chain into tuples by the algorithm \( \text{Algo}(4 \cdot 3) \), whose detailed description is shown in Appendix B.2. In \( \text{Algo}(4 \cdot 3) \), we first determine the number of tuples, then determine the boundaries of each tuple. To determine the number of tuples, we divide the protein chain into two segments whose lengths are as close as possible to each other, and check if the length of the longer segment is larger than the desired length; if it is, then continue dividing both segments into two and check the length of the longest among all resulting segments to determine whether a future division is needed; if it is not, then the current number of segments equals the number of tuples. The idea is to divide the whole protein into \( T \) tuples (where \( T \) is a power of 2) such that each tuple contains approximately the same number of
protein vertices, i.e., the difference between the sizes of any two tuples is no larger than 1. We prove it in Proof 4-1.

Proof 4-1: If the number of tuples and the boundaries of tuples are determined by the algorithm described in Alg(4-3), then the difference between the size of any two tuples is no larger than 1.

Let us prove by induction. Given a protein \( G_p = (V_p, E_p) \) whose total number of vertices is \( P \).

Base case: The whole protein is one tuple. No division needed.

Induction Step: Assume that after \( k \) divisions, tuples have sizes \( s \) and \( s + 1 \). Consider the situation after \( k + 1 \) divisions.

- If \( s = 2k \) (\( k \neq 0 \) and \( k \) is integer), then every tuple of size \( s \) is divided into two smaller tuples of the same size \( k \); And every tuple of the same size as \( s + 1 \) is divided into two smaller tuples, one of which is of size \( k \), and the other is of size \( k + 1 \). Therefore the above statement holds.

- If \( s = 2k - 1 \) (\( k \neq 0 \) and \( k \) is integer), then every tuple of size \( s \) is divided into two smaller tuples, one of which is of size \( k \), and the other is of size \( k - 1 \); And every tuple of the same size as \( s + 1 \) is divided into two smaller tuples of the same size \( k \). Therefore the above statement holds.

Instead of using mega tuples, we use **groups of tuples**. Starting from the first tuple, every \( \theta \) consecutive tuples form a group. \( \theta \) is called the group size. At the beginning of first iteration, \( \theta \) can be set to a small number which is a power
of 2 (2 for example). The group size \( \theta \) is doubled after each iteration. The total number of groups is denoted by \( \Theta \). Each group is represented as \( g_i = \mathcal{P}_i, \ldots, \mathcal{P}_{i-1} \).

\[ p_{n-i} \quad i \in [0, \Theta - 1]. \]

Now the \( i \)-th complimentary lower bound can be computed as follows:

\[
\text{dist}_{m}^\text{comp}(0, i, \mathcal{P} - 1) = \sum_{g \in \mathcal{G}} \text{dist}_{m}(g_i) + \sum_{t \in \mathcal{T}, \mathcal{T} \neq t} \text{dist}_{m}(t_i) + \text{dist}_{m}(S(t_i, i - 1) + \text{dist}_{m}(i + 1, \mathcal{E}_t)), \quad i \in [0, \Theta - 1]. \tag{4-16}
\]

The algorithm for computing the optimal lattice approximation by using tuples, groups of tuples and iterations is described in Alg(4 - 4) as below.

**Alg(4 - 4): Compute \( \text{dist}_{m}^\text{comp}(0, i, \mathcal{P} - 1) \) using tuples, groups of tuples and iterations**

**Input:** \( G_p = (V_p, \mathcal{E}_p) \), \( G_t = (V_t, \mathcal{E}_t) \)

**Output:** Global optimal lattice approximation of protein and its corresponding \( \text{dist}_{m}^\text{comp}(0, \mathcal{P} - 1) \)

1. Compute \( \text{dist}_{m}^\text{comp}(0, \mathcal{P} - 1) \).
2. Divide all the protein vertices into tuples using Alg(4 - 3).
3. Compute \( \text{dist}_{m}(t_i) \) for every tuple
4. Compute \( \text{dist}_{m}^\text{comp}(0, i, \mathcal{P} - 1) \) by using \( \text{dist}_{m}(t_i) \) as defined in (4-14)
5. Compute sight distances for all protein vertices.
6. Compute average sight distance.
7. While average sight distance > 16Å do
8. Compute all sight sets: compute \( \text{vsight} \) for all protein vertices to be used later in building IP vertex models, or compute \( \text{esight} \) for all protein edges to be used later in building IP edge models.
9. for all group \( g_i \) do
10. Build a IP model for group \( g_i \); either use \( \text{vsight} \) to build IP vertex model for group \( g_i \), or use \( \text{esight} \) to build one of the IP edge models for group \( g_i \).
Solve the above IP model by CPLEX.

Compute $c_{\text{dist}}(g_i)$.

end for

Compute $c_{\text{dist}}^{\text{pre}}(0, i, P - 1)$ by using $c_{\text{dist}}(g_i)$ and $c_{\text{dist}}(g_j)$ as in (4 - 16).

Compute sight distances for all protein vertices, and compute average sight distance.

$g \leftarrow 2 \times g$ /* Double the group size*/

Compute sight distances for all protein vertices, and compute average sight distance.

Re-compute starting vertex and ending vertex of each group.

end while

Compute $v_{\text{site}}$ for all protein vertices and build IP vertex model for the whole protein; Or compute $e_{\text{site}}$ for all protein edges and build one of the IP edge model for the whole protein.

Solve the above IP model by CPLEX.

return the optimal lattice approximation for the whole protein, and $c_{\text{dist}}(0, P - 1)$.

Algorithm 4.4 could not fit long proteins to lattice efficiently. In our experiments, optimal lattice approximation for protein 1E8M_A1 with 710 residues has been found after two interactions. In the final iteration, there are about 22 vertices in a tuple, about 89 vertices in a group, and the average sight distance was reduced to 15.09Å. The total computational time was close to 22 hours.

4.4 Refining Sight Sets

Having $s_d(p_i)$, we can determine $v_{\text{site}}(p_i)$ by formula (4 - 1) and then $e_{\text{site}}(p_j)$ by formula (4 - 4). However, the sight sets computed by the method stated in the previous sections contain too many vertices, and the resulting IP models were too large to be handled by CPLEX. Therefore we developed two methods to further refine sight sets for vertices $v_{\text{site}}$: (1) refining sight sets by checking neighbourhood connections, and (2) by BT_DP algorithm which applies
backtracking and dynamic programming approaches. Sight sets for edges $\text{vsight}(b_i), i = 0, ..., P - 2$ can be obtained from the refined sight sets for vertices $\text{vsight}(p_i), i = 0, ..., P - 1$.

### 4.4.1 Refining Sight Sets by Checking Neighbourhood Connections

We can refine $\text{vsight}(p_{+,i})$ by only keeping those lattice vertices $q_j \in \text{vsight}(p_{+,i})$ for which there is an edge between $q_j$ and a lattice vertex $q_k \in \text{vsight}(p_i)$, and also an edge between $q_j$ and a lattice vertex $q_l \in \text{vsight}(p_{-,i})$. All $q_j \in \text{vsight}(p_{+,i})$ that satisfy the above condition form a set $\text{new\_vsight}(p_{+,i})$, as defined more formally in (4-21).

Let us extend the definition of $N$ from a lattice vertex to a set of lattice vertices $S$,

$$N(S) = \{q_j \in N(q_j) \mid q_j \in S\} \quad (4-20)$$

Then $\text{new\_vsight}(p_i)$ can be defined as follows:

$$\text{new\_vsight}(p_0) = \text{vsight}(p_0) \cap N(\text{vsight}(p_0))$$
$$\text{new\_vsight}(p_i) = N(\text{vsight}(p_{+,i}) \cap \text{vsight}(p_i) \cap N(\text{vsight}(p_{+,i})) \cap [0, P - 2] \quad (4-21)$$
$$\text{new\_vsight}(p_{-,i}) = N(\text{vsight}(p_{-,i}) \cap \text{vsight}(p_{-,i}))$$

If the values of all $\text{new\_vsight}(p_i)$'s are the same as the values of the corresponding $\text{vsight}(p_i)$'s, we assign the values of all $\text{new\_vsight}(p_i)$'s back to the corresponding $\text{vsight}(p_i)$'s, and terminate the refining process. Otherwise, update the values of $\text{vsight}(p_i)$'s by the value of $\text{new\_vsight}(p_i)$'s, and repeat the refining procedure. The refining procedure is defined more formally in Alg (4-5).
Algorithm 4.5: Refining sight sets by checking neighbourhood connections

Input: vsight(p_i)
Output: Refined vsight(p_i) after checking neighbourhood connections

1: while ∃p_i, new_vsight(p_i) ≠ vsight(p_i) do
2:     ∀p_i, vsight(p_i) ← new_vsight(p_i)
3:     Compute vsight(p_i) new_vsight(p_i) as in (5.21)
4: end while
5: return vsight(p_i)

4.4.2 Refining Sight Sets by Backtracking - Dynamic Programming

Algorithm
We further refine sight sets for vertices by pruning the lattice vertices in vsight(p_i) using the following strategy.

For a protein vertex p_i in a protein substring p_i,...,p_j, and a lattice vertex q_j, if a walk map φ satisfies the following conditions:

- φ(p_i) = q_j
- ∀p_k, k = i,..., j, φ(p_k) ∈ vsight(p_k)
- cdist(φ(p_i)) ≤ cdist^stop(s,e)

we call φ a \textit{p_i,q_j anchored insight lower bound walk map} (abbreviated as p_i,q_j \textit{anchored walk}) for protein substring p_i,...,p_j, and the cost of the p_i,q_j anchored walk, cdist(φ(p_i)) ≤ cdist^stop^a(s,e), is denoted by cdist^stop^a(s,e), since it is a lower bound for cdist^stop(s,e).
It is obvious that $\text{cdist}_{\text{lin}}(0, P - 1) \leq \text{cdist}_{\text{seg}}(0, P - 1) \leq \text{cdist}_{\text{seg}}(0, P - 1)$. If there is no anchored walk for pair $(p_i, q_j)$, then any lattice approximation mapping $p_i$ to $q_j$ cannot be optimal. In such a case we can remove $q_j$ from $\text{v sight}(p_i)$. Furthermore, we can obtain $\text{cdist}_{\text{lin}}(0, P - 1)$ with higher value by applying the following two steps: (1) Divide the whole protein into segments, for each segment, find the vertex maps such that the mapping of each protein vertex is within its sight set. (2) Try to connect the paths in the previous step to form $p_i, q_j$ anchored walk, which we call segmented $p_i, q_j$ anchored insight lower bound walk map (abbreviated as segmented $p_i, q_j$ anchored walk). Since the costs of the vertex maps in the previous step would have higher values, it is more difficult to satisfy the third condition in the definition of segmented $p_i, q_j$ anchored walk, and therefore more likely that there is no segmented $p_i, q_j$ anchored walk is found. If there is no segmented anchored walk for pair $(p_i, q_j)$, then we can remove $q_j$ from $\text{v sight}(p_i)$. This idea is applied in the backtracking-dynamic programming (BT-DP) algorithm described in the following subsections.

4.4.2.1 General Strategy

Given a protein $G_p = (V_p, E_p)$, we divide protein into tuples, and compute $\text{cdist}_{\text{lin}}(0, i, P - 1)$ as in (4 - 14) or (4 - 16), then compute $\text{sd}(p_i)$ for all $p_i \in V_p$ as in (4 - 9), and then compute $\text{v sight}(p_i)$ as in (4 - 1). Having sight sets for all protein vertices and segments are tuples, we try to compute the segmented $p_i, q_j$ anchored walk with the minimum $\text{cdist}$, which is denoted by
If there is no such segmented $p_i,q_i$ anchored walk, we remove $q_i$ from $\text{vsight}(p_i)$.

The data structures used in the BT-DP algorithm are defined below.

For a protein vertex $p_i$, let $s_i$ represents a lattice vertex in $\text{vsight}(p_i)$.

Given a protein substring $p_1,...,p_j$, for each $s_i \in \text{vsight}(p_i)$ and $s_j \in \text{vsight}(p_j)$, **substring cdist** $SDIST(p_i,p_j,s_i,s_j)$ stores the minimum cdist of the any mapping $s_i,s_1,...,s_{j-1},s_j$ which starts from $s_i$ and ends at $s_j$. More formally,

$SDIST(p_i,p_j,s_i,s_j) = \min\{d^2(p_i,s_i),SDIST(p_i,p_j,s_1,s_j)\}$

$$SDIST(p_i,p_j,s_i,s_j) = \min\{d^2(p_i,s_i),SDIST(p_i,p_j,s_1,s_j)\}$$

$$+ d^2(p_j,s_j); \forall s \neq y, s_i \neq s_j, s_i, s_j \in N(y), i \neq j$$

For a tuple $t_i = p_{i_1},...,p_{i_k}$, let $s_{i_k}$ represents a lattice vertex in $\text{vsight}(p_{i_k})$.

For every protein vertex $p_i \in t_i$, $t_j = p_{i_1},...,p_{i_k}$, **left sub-tuple cdist** $LST(p_i,s,.,s_{i_k})$ is the minimum cdist of the any path $s_{i_1},s_{i_2},...,s_{i_{k-1}},s_{i_k}$.

$LST(p_i,s_{i_k},s_i) = SDIST(p_{i_k},p_i,s_{i_k},s_i)$

and **right sub-tuple cdist** $RST(p_i,s,.,s_{i_k})$ is the minimum cdist of the any path $s_{i_1},s_{i_2},...,s_{i_{k-2}},s_{i_{k-1}},s_{i_k}$.
minimum cdist of the any path \( s_{i_1}, s_{i_2}, s_{i_3}, \ldots, s_{i_k} \),

\[
\text{TDIST}(t_i, s_{i_1}, s_{i_2}) = \text{RST}(p_{i_1}, s_{i_1}, s_{i_2}) \tag{4-24}
\]

For every tuple \( t_i = p_{i_1}, \ldots, p_{i_k} \), **tuple cdist** TDIST \((t_i, s_{i_1}, s_{i_2})\) is the

minimum cdist over all the segmented \( p_{i_1} \rightarrow s_{i_2} \) anchored walks

\[
\text{TDIST}(t_i, s_{i_1}, s_{i_2}) = \text{LST}(p_{i_1}, s_{i_1}, s_{i_2}) = \text{RST}(p_{i_1}, s_{i_1}, s_{i_2}) \tag{4-25}
\]

For every tuple \( t_i = p_{i_1}, \ldots, p_{i_k} \), **left combined tuple cdist** LCT \((t_i, s_{i_1})\) is

the minimum cdist over all the segmented \( p_{i_1} \rightarrow s_{i_2} \) anchored walks

\[
s_{i_1}, s_{i_1}, s_{i_1} + 1, \ldots, s_{i_1} + T - 1, i.e., \text{cdist}_{\text{seq}}^{s_{i_1} \rightarrow \text{vsite}(0, E_{i_1})}.
\]

\[
\text{LCT}(t_i, s_{i_1}) = \min \{ \text{TDIST}(t_i, s_{i_1}, s_{i_2}); \forall s_{i_2} \in \text{vsite}(p_{i_1}) \}
\]

\[
\text{LCT}(t_i, s_{i_1}) = \min \{ \text{TDIST}(t_i, s_{i_1}, s_{i_2}) + \text{LCT}(t_i, s_{i_2});
\forall s_{i_2} \in \text{vsite}(p_{i_1}), \forall s_{i_2} \in \text{vsite}(p_{i_2}), (s_{i_2}, s_{i_2}) \in E_{i_1} \} = 1, \ldots, T - 1 \tag{4-26}
\]

For every tuple \( t_i = p_{i_1}, \ldots, p_{i_k} \), the **right combined tuple cdist**

\[
\text{RCT}(t_i, s_{i_1}) \text{ is the minimum cdist over all the segmented \( p_{i_1} \rightarrow s_{i_2} \) walks}
\]

\[
s_{i_1}, s_{i_1} + 1, \ldots, s_{i_1} + T - 1, s_{i_1} + T, i.e., \text{cdist}_{\text{seq}}^{s_{i_1} \rightarrow \text{vsite}(0, P - 1)}.
\]

\[
\text{RCT}(t_i, s_{i_1}) = \min \{ \text{TDIST}(t_i, s_{i_1}, s_{i_2}); \forall s_{i_2} \in \text{vsite}(p_{i_1}) \}
\]

\[
\text{RCT}(t_i, s_{i_1}) = \min \{ \text{TDIST}(t_i, s_{i_1}, s_{i_2}) + \text{RCT}(t_i, s_{i_2});
\forall s_{i_2} \in \text{vsite}(p_{i_1}), \forall s_{i_2} \in \text{vsite}(p_{i_2}), (s_{i_2}, s_{i_2}) \in E_{i_1} \} = T, \ldots, 0 \tag{4-27}
\]

For every protein vertex \( p_i \in t_i \) where \( t_i = p_{i_1}, \ldots, p_{i_k} \), we want to
determine whether to remove a lattice vertex \( s_{i} \) from \( \text{vsite}(p_{i_k}) \) by checking if

\[
\text{cdist}_{\text{seq}}^{s_{i_1} \rightarrow \text{vsite}(0, P - 1)} \text{ is greater than } \text{cdist}_{\text{seq}}^{s_{i_1} \rightarrow \text{vsite}(0, P - 1)}. \text{ We can compute}
\]

61
$cd_{\text{def}}^{\text{DP}}(0, P-1)$ by the following formula, and the general strategy is illustrated in Figure 4.3.

$$
\begin{align*}
&cd_{\text{def}}^{\text{DP}}(0, P-1) = \min \{LCT(j_1, s_{j_0}), LST(p, s_{j_0}, s_j) \\
&+ RST(p, s_{j_0}, s_{j_1}) - d^2(p, s_j) + RCT(j_1, s_{j_0}, s_{j_1}) \mid (s_{j_0}, s_{j_1}) \in E_i, (s_{j_0}, s_{j_1}) \in E_j \}
\end{align*}
$$

Figure 4.3 General Strategy of BT-DP Algorithm

After reducing the search space for every $p_i \in V$, in the way described above, we will get new sets of $v_{\text{sight}}(p_i)$ with much smaller sizes. Then CPLEX is capable of finding optimal lattice approximations in the reduced search space for a set of randomly selected proteins from PDB with up to 10^14 residues (please see Chapter 6 for detailed experimental results).

4.4.2.2 Basic Techniques

Before describing the algorithms used to compute the data structures defined in the previous section, let us first explain how backtracking and dynamic programming approaches are applied in the BT-DP algorithm.
Backtracking approach is a systemic depth first search to go through all the possible configurations until the right one is found. During the search in each step, a valid partial solution is extended by choosing a value for a new element, and the extended partial solution is tested to see whether it is still valid. If it is not, the search backtracks to the choice point and an alternative value is assigned to that new element. If none of the values in current choice point can be part of the valid solution, the search further backtracks to the previous choice point and go through other alternatives over there. If there is no more choice point, the search fails. The strength of backtracking algorithm we use lies in its ability to reject subsets of possible solutions even before they have been tested. This is done by performing a) forward checking: deletes that values that conflict with current partial solution from the domains of unassigned elements, and b) constraint propagation: deletes that values that violate any constraint from the domains of unassigned elements. In our BT-DP algorithm, we use backtracking approach to compute SDIST's (as described in Alg(4-6)), in which forward checking technique is used when situation “duplicated vertices” happens, and constraint propagation is applied in “early long cdist” and “out of sight” situations, as well as by using “cdist upper bound for tuples”.

Dynamic programming is a powerful method to solve optimization problem which has the properties of overlapping subproblems and optimal substructure by memorization. If the same subproblems are used to solve several larger subproblems, the overall problem is said to have the overlapping subproblems property. If the optimal solution of subproblems can be used to compute the
optimal solution of the overall problem, the overall problem is said to have the optimal substructure property. Dynamic programming method breaks a problem into several subproblems, and find optimal solutions for subproblems which will be used to compute the overall optimal solution. During the computation, the solutions of the subproblems are saved only if they will be used again in solving other subproblems. This technique is called memorization. We apply dynamic programming approach in computing LCT's, RCT's and cdist \( (0, \ldots, P-1) \).

1. Computing Substring cdist SDIST \((p_i, p_j, s_i, s_j)\)

To compute the values of the data structures defined in Section 4.4.1 more efficiently, we use the backtracking algorithm Algo(4-6) to compute a set of substring cdist SDIST \((p_i, p_j, s_i, s_j)\) simultaneously instead of computing SDIST \((p_i, p_j, s_i, s_j)\) one at a time. SDIST \((p_i, p_j, s_i, s_j)\) is a sequence of cdist's of maps where \(s_j\) is fixed to different values from \(\text{vsight}(p_j)\). Algo(4-6) is based on Algo(4-1) with some modifications. A detailed description of Algo(4-6) can be found in Appendix B.3. The major modification in Algo(4-6) is as follows.

- During the process of generation a path in the lattice as the lattice approximation for the protein substring, we need to make sure that the current candidate lattice vertex \(q\) is in \(\text{vsight}(p_i)\), where \(k = i + 1, \ldots, j\).

- Let us use the two-dimensional array \(\text{sight}\) to record the sight set for each protein vertex. For an example, lattice vertex \(\text{sight}[j][k]\) is the \(k\)-th element in \(\text{vsight}(p_j)\). Assume in an one-dimensional array \(\text{sdist}, \text{sdist}[k]\)
stores the minimum cost among all the paths which end at,
s_j = \text{sight}[j][k] that we encountered by far. At the beginning of the
computation, all elements in array \text{sdist} are set to infinity. If there is a path
which starts from \( s_i \) and ends at \( s_j \), the value of \text{sdist}[k] is not infinity at
the end of the computation.

Since \( \text{cdist}_{\text{up}}(t_i) + \sum_{j=0}^{P-1} \text{cdist}_{\text{up}}(t_j) \leq \text{cdist}_{\text{op}}(0,P-1) \),
therefore \( \text{cdist}_{\text{up}}(t_i) \leq \text{cdist}_{\text{up}}(0,P-1) - \sum_{j=0}^{P-1} \text{cdist}_{\text{up}}(t_j) \).

Let \( \text{cdist}_{\text{up}}(t_i) = \text{cdist}_{\text{up}}(0,P-1) - \sum_{j=0}^{P-1} \text{cdist}_{\text{up}}(t_j) \) \tag{4 - 29}
We call \( \text{cdist}_{\text{up}}(t_i) \) \textit{upper bound for tuple} \( t_i \).

Since for any \( s_{\text{op}} \in \text{vsight}(p_{\text{op}}) \) and \( s_{\text{up}} \in \text{vsight}(p_{\text{up}}) \),
\[ \text{cdist}_{\text{up}}(t_i) \leq \text{TDIST}(t_i,s_{\text{op}},s_{\text{up}}) \leq \text{cdist}_{\text{up}}(t_i) \]
Therefore, if \( s_{\text{op}} \) and \( s_{\text{up}} \) do not make the inequality
\[ \text{TDIST}(t_i,s_{\text{op}},s_{\text{up}}) \leq \text{cdist}_{\text{up}}(t_i) \] hold, then any mapping which maps \( p_{\text{op}} \) to
\( s_{\text{op}} \), and \( p_{\text{up}} \) to \( s_{\text{up}} \) cannot be optimal and therefore we are not interested
in its value. When computing \( \text{TDIST}(t_i,s_{\text{op}},s_{\text{op}}) \), we can speed up the
computation by initializing all elements of array \text{sdist} to be \( \text{cdist}_{\text{up}}(t_i) \),
whose value is the better upper bound than infinity and is used to avoid
generating paths that are for sure not part of the optimal solution. In this
case, an one-dimensional Boolean array \text{realPath} of the same size as
array \textit{sdist} is used to keep tracking whether we have found a path whose \textit{cdist} cost equals to the value of the corresponding element in array \textit{sdist}.

\begin{align*}
\text{Algo}(4\cdot 6): \text{ Compute SDIST}(p_i, p_j, s, \textit{\*})
\end{align*}

\begin{itemize}
\item \textbf{Input:} \(i, j, k\) where \(s_i = q_k, v\) where \(s_j = \text{sight}[i][j]\), Array \textit{sdist} of size \(|\text{sight}|^2\).
\item When using this algorithm to compute \(TDIST(t_i, s, \textit{\*})\), initialize all elements in array \textit{sdist} to be \textit{cdist},(l) before passing it as input.
\end{itemize}

\begin{itemize}
\item \textbf{Output:} the value of \(SDIST(p_i, p_j, s, \textit{\*})\) as stored in array \textit{sdist}.
\end{itemize}

*To avoid stating the same contents as in Algo(4 - 1), only the important modification is shown*

\begin{aligned}
\text{getSTdist_allInOne}(i, j, k, v, \textit{sdist})
&: \text{ Phase 1: Generating valid path in the lattice}^*
& \text{ for all } \text{ pos } = i+1, \ldots, j \text{ do }
& \text{ Generate current candidate vertex } q, \text{ for current position } \text{ pos } + 1 \text{ in the path in the lattice by checking the neighbours of the lattice vertex at previous position } \text{ pos } \text{ the same way as used in Algo(4 - 1).}
& \text{ if } ((q, \text{ is a neighbour of the lattice vertex at previous position } \text{ pos }) \text{ and } (q, \text{ is not used in the current partial path}) \text{ and } (q, \text{ is within the sight set of the protein vertex which it maps to}) \text{ then }
& \text{ if } (q, \text{ is the image of the last protein vertex } p_i, \text{ in substring } p_i, \ldots, p_j; (q, = \text{sight}[j][v]) \text{ then }
& \text{ Mark is } q, \text{ as used in the current path.}
& \text{ if } (\text{the cdist cost of the current path } \leq \text{ minimum value of cdist cost recorded}) \text{ then }
& \text{ Update the minimum value recorded.}
& \text{ Mark the current minimum value as come from the cost of real path in the lattice.}
& \text{ end if }
& \text{ pos } ++
& \text{ else } \text{ if } q, \text{ is not image of the last protein vertex } p_i, ^*/
\end{aligned}
if (the cost of the current partial path < maximum value of all cost recorded) then
    Mark q, as used in the current partial path.
    pos ++
else
    Mark that it is necessary to generate the next candidate path from position pos.
end if
end if
else
Mark that it is necessary to generate the next candidate path from position pos.
end if
/*Got a valid path*/
end for

2. Computing Combined Tuple cdist LCT's and RCT's

Due to limited memory space, it is infeasible to pre-compute and store the values of all LST's, RST's and TDIST's. Indeed, to store all the values of TDIST's is not feasible. Otherwise it could greatly speed up the whole computation, since LST(p, s, *) can be computed faster if the values of LST(p, s, *) are available, and RST(p, s, *) can be computed faster if the values of RST(p, s, *) are available, and all LST's and RST's are used to compute cdist (0, q, p-1). Also TDIST's are used twice to compute LCT's and RCT's. In order to accomplish the computation within the limits of
computational time and space, time efficiency is sacrificed for the gain in space efficiency. Specifically, we only pre-compute and store the values of all LCT’s and RCT’s. The value of LST’s, RST’s and TDIST’s are computed directly from SDIST whenever necessary.

We use algorithm Algo(4-7) to compute the values of all LCT’s and RCT’s. The ideas used to compute LCT’s and RCT’s are very similar. Let us take the way to compute LCT’s as example. We first compute the values of all LCT($t_i,^*$) simultaneously by using Algo(4-6). Then starting from the first tuple, we can compute the values of tuple exist of the next tuple, i.e., TDIST($t_i, s_{t_i}, ^*$) simultaneously. Since the values of LCT($t_i, s_{t_i}$) are available, we can compute LCT($t_{i+1}, ^*$) as defined in formula (4-26). Figure 4.6 demonstrates the idea on how to compute LCT($t_{i+1}, ^*$). None of the values of TDIST($t_i, s_{t_i}, ^*$) are saved for future usage. Therefore, the memory space used to compute current TDIST is released at the end of computation and can be reused in computing the next TDIST. A detailed description of Algo(4-7) can be found in Appendix B.4.
Figure 4.4 Illustration on how to compute \( \text{LCT}(t_{i+1},*) \), where \( t_{i+1} = p_i, \ldots, p_r \).

3. Computing \( c \text{dist}^{\text{supr, and}, \neg \neg \neg}_{\text{vis}}(0, P-1) \)

For a lattice vertex \( q_j \in \text{vsight}(p_i), p_i \in t_{i+1}, c \text{dist}^{\text{supr, and}, \neg \neg \neg}_{\text{vis}}(0, P-1) \) is computed as defined in formula (4.28) by the algorithm described in \text{Algo}(4.8).

The idea is, starting from the negative value of the distance square between \( p_i \) and \( q_j \), i.e., \(-d^2(p_i, q_j)\), adds up the values of corresponding \( \text{LCT}(t_{i+1}, s_{i+1}), \text{RCT}(t_{i+1}, s_{i+1}, q_j), \text{LST}(p_i, q_j), \text{RST}(p_i, s_{i+1}, q_j) \) one at a step in this order, and at the end of every step, makes sure the current value of the sum is smaller than \( c \text{dist}_{\text{vis}}(0, P-1) \). If it is not, mark that \( q_j \) needs to be removed from \( \text{vsight}(p_i) \). Figure 4.7 illustrates how to compute \( c \text{dist}^{\text{supr, and}, \neg \neg \neg}_{\text{vis}}(0, P-1) \). A detailed description of \text{Algo}(4.8) can be found in Appendix B.5.
Figure 4.5 Illustration on how to compute $\text{cdist}_{\text{approx}}(0, P - 1)$, where $q_i \in \text{vsight}(p_i), p_i \in I_i$

Our experiments show that the BT-DP algorithm, which consists of \text{Algo(4-6)}, \text{Algo(4-7)} and \text{Algo(4-8)} fails to find lattice approximations for the largest proteins efficiently (computational time is over 24 hours). This is mainly because the procedure to compute $\text{cdist}_{\text{approx}}(0, P - 1)$ and decide whether to remove a lattice vertex from the sight set of a protein vertex runs extremely slow. However, the algorithm is very powerful in identifying bad candidate lattice vertices. For an example, after refining procedure, averagely only about 2% lattice vertices remain in the sight set of a protein vertex as good candidates.

4.4.2.3 Use of Seeds

We have noticed that for a boundary protein vertex (i.e., the first or the last protein vertex of a tuple), the decision on whether to remove a lattice vertex out of its sight set can be made much faster than for any non-boundary protein vertex, since we do not need to compute any LST's and RST's when computing $\text{cdist}_{\text{approx}}(0, P - 1)$ for a boundary protein vertex. Instead, it is enough to
simply combine the corresponding LCT’s and RCT’s, whose values are already
pre-computed and stored, as explained in the following formulas:

\[ \forall s_p \in \text{sight}(p) : \text{cdist}_{\text{min}}(s, (0, P-1)) = \text{RCT}(s, s_p) \]

\[ \forall s_p \in \text{sight}(p) : \text{cdist}_{\text{min}}(s, (0, P-1)) = \text{LCT}(s, s_p) \]

\[ \forall s_p \in \text{sight}(p) : \text{cdist}_{\text{min}}(s, (0, P-1)) = \min\left\{ \text{LCT}(s, s_p) + \text{RCT}(s, s_p) \right\} \]

We perform the first refinement by using the similar techniques as in
Algo(4 - 8) to refine sight sets for boundary protein vertices. Let us call the
reduced sized sight sets of boundary protein vertices refined by BT-DP algorithm
seeds. Then we perform the second refinement by checking neighbourhood
connections as defined in (4 - 21). During the second refinement, a chain-action-
like phenomenon is triggered by the seeds, which further reduces the sizes of the
sight sets for the non-boundary protein vertices.

However, the seeds’ are less powerful in reducing the sight set of a
protein vertex which locates further away from boundaries. Our experiments
showed that for large proteins, by performing the two-step refinement described
above is not enough, since the sight sets in the centre of tuples do not shrink
much. Therefore for each tuple, we need to refine sight sets of at least one vertex
which locates in the middle of the tuple. For example, for protein 1A41_A1 with
239 residues, the average number of vertices in the initial sight sets was 346
when tuple size was set to 14 or 15. Then we set tuple size to 7 or 8 before using
Algo(4-8) to refine sight sets of boundary vertices. Even though the average
number of vertices in sight sets of boundary vertices drops significantly to 5 after refining by BT-DP algorithm, the chain action of checking neighbourhood connections was only able to bring the average number of vertices in sight sets of all protein vertices down to 256. However, if for each tuple, we can further refine sight sets of one vertex which locates in the middle of tuple by using Algo(4 - 8), following by the procedure of checking neighbourhood connections, we are able to reduce the average number of vertices in the sight sets for all protein vertices to 13, which forms a small enough search space that enables CPLEX to find the optimal solution.

To refine sight set of a protein vertex which locates in the middle of a tuple can be done in the following two fashions:

**Fashion 1** For every tuple $t_i$, $i = 0, T - 1$, refine sight set of the middle vertex of $t_i$ by the BT-DP algorithm Algo(4 - 8). If the number of vertices in $t_i$ is odd, the middle vertex is $p_{\left\lfloor \frac{1}{2} (|t_i| + 1) \right\rfloor}$; otherwise, the middle vertex is $p_{\left\lfloor \frac{1}{2} |t_i| \right\rfloor}$.

**Fashion 2** Rearrange all protein vertices into new tuples by dividing every original tuple into left and right segments of similar sizes, whose difference is at most 1, and combining the right segment of the original tuple $t_i$ with the left segment of the original tuple $t_{i+1}$ to form the new tuple $t_{i+1}$. The left segment of the original tuple $t_i$ becomes the new tuple $t_{i+1}$, and the right segment of the original tuple $t_{i+1}$ becomes the new tuple $t_{i+2}$. The algorithm of determining
boundaries of new tuples is described in Algo(4-9). As shown in Figure 4.8, after rearranging, the number of tuples is exactly one more than before, and the boundary vertices are exactly those previously hiding at the centre of the original tuples. Then we can use the same strategy to refine the boundary vertices of the new tuples. In this way, the average tuple size is almost the same as before especially for large proteins, therefore the values of $c_{dist}(t_i)$ do not change much, which allows algorithm Algo(4-8) to perform as good on the new tuples as on the original tuples.

**Algo(4-9): Determining New Boundaries of Tuples**

**INPUT:** $T$, one dimensional array tupleSize  
**OUTPUT:** $T$, array tupleSize

getNewTupleBoundaries($T$, tupleSize)
1. $T \leftarrow T + 1$
2. Define array NewTupleSize[$T$]
3. NewTupleSize[0] \leftarrow \lceil \text{tupleSize}[0]/2 \rceil$
4. $t \leftarrow 0$
5. while $t < T - 2$ do
6. \hspace{1em} NewTupleSize[$t+1$] \leftarrow tupleSize[$t$] + $\lceil \text{tupleSize}[$t$/2] \rceil$
7. \hspace{1em} $t \leftarrow t + 1$
8. end while
9. NewTupleSize[$T-1$] \leftarrow tupleSize[$T-1$] + $\lceil \text{tupleSize}[$T$/2] \rceil$
10. return $T$, NewTupleSize
As we observed from our experiments, refine sight sets of middle protein vertices by fashion 2 takes much less computational time than fashion 1. It is because in fashion 1, the procedure to compute $c_{\text{dist}}^{\text{geom}}(x, (0, P - 1))$ and decide whether to remove a lattice vertex from the sight set of a protein vertex $p_i$ (where $p_i$ is non-boundary vertex) runs extremely slow. Fashion 2 is also better than fashion 1 since more sight sets of protein vertices are refined in fashion 2.

We can also apply iterations to fashion 2, in this case the information of original tuples must be stored throughout all iterations. For example, in the first iteration, divide the original tuples into two segments of $1/4$ and $3/4$ length, combine two neighbouring segments to form new tuples, and refine sight sets of boundary vertices in new tuples; In the second iteration, divide the original tuples into two segments each of $1/2$ length, then form new tuples and refine sight sets. In the
third iteration, divide the original tuples into two segments of 3/4 and 1/4 length, then form new tuples and refine sight sets. We can have as many iterations as necessary to get a small enough search space in the end.

Since the power of the "chain action" triggered by seeds during the refinement by checking neighbourhood connections is very sensitive to the length of tuples, we want to keep the tuple size small. The smaller the average tuple size is, greater effect the seeds on reducing the sight sets of non-boundary protein vertices. On the other hand, we want to have large tuple size such that the initial sight sets computed by using \( cdist^{(m)}_{N}(i, P - 1) \) as described in (4-14) would be small. Therefore, we using Algorithm (4-3) to determine a set of tuples with large sizes when computing initial sight sets by (4-14), and use the same algorithm to determine a new set of tuples with smaller sizes when refining sight sets by BT-DP algorithm. In our experiments, if the length of protein is less than 200, we set the maximum length of the first and the second set of tuples to be less than 10 and 6 respectively; otherwise the two numbers are set to be 20 and 10, respectively.

Algorithm described in Algorithm 4-10 presents the complete BT-DP algorithm, which reduces the search space to be small enough to enable CPLEX to handle all proteins with up to 1014 residues that are randomly selected from our database.

Algorithm 4-10: Compute optimal lattice approximations of protein chains using IP approach: complete BT-DP algorithm using tuples, sight sets and refinement with seeds

75
Input: \( G = (V, E) \), \( G_c = (V_c, E_c) \)

Output: Global optimal lattice approximation of protein and its corresponding

cdist\(_{opt}(0, P - 1)\)

1. Compute \( \text{cdist}_{\text{Cplex}(14)}(0, P - 1) \) as \( \text{cdist}_{\text{opt}}(0, P - 1) \).
2. Using Algo(4-3), divide all the protein vertices into initial tuples according to \( P \) and initial tuple size.
3. Compute \( \text{cdist}_{\text{opt}}(t, i) \) for every tuple \( t, i = 0, T - 1 \).
4. Compute \( \text{cdist}_{\text{opt}}(0, i, P - 1) \) by using \( \text{cdist}_{\text{opt}}(t, i) \) as in (4-14).
5. Compute sight distances for all protein vertices.
6. Compute sight sets: compute \( \text{vsight}(p_i) \) for all protein vertices \( p_i \in V_p \).
7. Refine all sight sets by checking neighbourhood connections by Algo(4-5).
8. Using Algo(4-3), divide all the protein vertices into a new set of tuples according to \( P \) and initial tuple size. The initial tuple size is set to be much smaller than in line 2 of this algorithm.
9. Compute \( \text{cdist}_{\text{opt}}(t, i) \) and \( \text{cdist}_{\text{opt}}(t, i) \) for every \( t, i = 0, T - 1 \).
10. Compute and store the value of \( \text{LCT}(t, s_{in}) \) and \( \text{RCT}(t, s_{in}) \) for every \( t, i = 0, T - 1 \) by Algo(4-7).
11. for all tuple \( t, i = 0, T - 1 \) do
12.    Refine sight sets of both boundary protein vertices: \( \text{vsight}(p_a) \) and \( \text{vsight}(p_b) \), using Algo(4-8)
13. end for
14. Refine \( \text{vsight}(p_i) \) for every protein vertex \( p_i \) by checking neighbourhood connections.
15. Determine new \( T \) and array tupleSize using Algo(4-9). /*Line 16-21 repeats the steps in Line 9-14*/
16. Compute \( \text{cdist}_{\text{opt}}(t, i) \) and \( \text{cdist}_{\text{opt}}(t, i) \) for every \( t, i = 0, T - 1 \).
17. Compute and store the value of \( \text{LCT}(t, s_{in}) \) and \( \text{RCT}(t, s_{in}) \) for every \( t, i = 0, T - 1 \) by Algo(4-7).
18. for all tuple \( t, i = 0, T - 1 \) do
19.    Refine sight sets of both boundary protein vertices: \( \text{vsight}(p_a) \) and \( \text{vsight}(p_b) \), using Algo(4-8)
20. end for
21. Refine \( \text{vsight}(p_i) \) for every protein vertex \( p_i \) by checking neighbourhood connections.
2. Build IP vertex model for the whole protein using \( \text{v} \text{is} \text{ght}(p_i), p_i \in V \). Or build IP edge model for the whole protein using \( \text{e} \text{is} \text{ght}(e_i), e_i \in E \), which are computed by using \( \text{v} \text{is} \text{ght}(p_i), p_i \in V \).

3. Solve the IP model by CPLEX and get the optimal lattice approximation for the whole protein.

4.5 Determining Sizes of Lattices

As mentioned in Section 2.1, we need to make the volume of the finite 3D space defined by the lattice graph big enough to guarantee that the optimal solution can be found. For this purpose, a set of margins is added to the boundaries of the maximum and minimum 3D coordinates of protein vertices \( V \).

Suppose the 3D space of bounded by the protein vertices is represented as \((\min(x), \max(x), \min(y), \max(y), \min(z), \max(z))\), then the 3D space bounded by adding the set of margins \((m_{g_i}, \ldots, m_{g_n})\) where \( m_{g_i} \geq 0, i = 0, \ldots, 5 \) to the 3D space of the protein is \((\min(x)-m_{g_i}, \max(x)+m_{g_i}, \min(y)-m_{g_i}, \max(y)+m_{g_i}, \min(z)-m_{g_i}, \max(z)+m_{g_i})\).

In our experiments, we first compute sufficient margins to guarantee optimality.

The margin can be computed using Algo(4 - 11). The idea is to start up with a initial lattice space which is obtained by placing initial margin on the boundaries of the protein. Then we compute safe sight distance for every protein vertex by (4 - 9), i.e., \( \text{s} \text{d}(p_i) = \sqrt{\text{c} \text{d} \text{i} \text{s} \text{t}(0, P - 1) - \text{c} \text{d} \text{i} \text{s} \text{t}(0, i, P - 1)} \). Since a lattice approximation can be found for every protein with fixed sight distance equals to 10Å, we replace \( \text{c} \text{d} \text{i} \text{s} \text{t}(0, P - 1) \) by \( \text{c} \text{d} \text{i} \text{s} \text{t}(\text{CPLEX/IOA}, 0, P - 1) \). Cost \( \text{c} \text{d} \text{i} \text{s} \text{t}(0, i, P - 1) \) is
replaced by \( \sum_{j=0}^{P-I} \text{cdist}_{\text{cp}}(j) \), since it is the only value we have at this moment to guarantee the resulting sight distances are safe. I.e.,

\[
\text{sd}(p_i) = \sqrt{\text{cdist}_{\text{cp}}(0, P-I) - \sum_{j=0}^{P-I} \text{cdist}_{\text{cp}}(j)} \tag{4-31}
\]

The boundaries of the lattice is the boundaries of the 3D space defined by placing the corresponding margin \( \text{sd}(p_i) \) on every protein vertex \( p_i \).

We set the initial margin to be the longest distance between any protein vertex and its closest lattice vertex. For cubic lattice it would be the distance form the centre of cube to the vertex, which is \( \frac{\sqrt{3}}{2} \times 3.8 \AA \approx 3.3 \AA \). For some sparse cubic lattice it could be larger, but surely less than \( 2 \times 3.8 \AA = 7.6 \AA \). In our experiments, it is set to be 4\AA.

**Algorithm (4-11): Compute precise boundaries of lattice**

**Input:** A protein \( G_p = (V_p, E_p) \), \text{cdist} upper bound comes from using CPLEX to solve the protein with fixed sight distance equals to 10 \AA \text{cdist} of the closest lattice vertex to a protein vertex \( p_i \) \( \text{cdist}_{\text{cp}}(i, i), i = 0, ..., P-1 \)

**Output:** A finite 3D lattice \( G_L = (V_L, E_L) \)

1. \( m \leftarrow 4\AA \)
2. define the boundaries of \( G_L \) as the boundaries of \( G_p \) with margin value equals to \( m \), i.e., the boundaries of \( G_L \) whose boundaries are redefined as \( \max(x, x+\text{m}) \), \( \max(y, y+m) \), \( \max(z, z+m) \), \( \min(x, x-m) \), \( \min(y, y-m) \), \( \min(z, z-m) \)
3. for all \( p_i \in V_p \) do
4. Given \( G_p \) and \( G_L \), compute square root of sight distance \( \text{sd}(p_i) \) as defined in (4-31).
5. end for
6. Return $G_{r}$ whose boundaries are redefined as $\text{MAX}(x_{r_i} + \text{sd}(p_{j})),
\text{MAX}(y_{r_i} + \text{sd}(p_{j})),
\text{MIN}(x_{r_i} - \text{sd}(p_{j})),
\text{MIN}(y_{r_i} - \text{sd}(p_{j}))$.

For all the proteins we have studied, the largest sufficient margin computed using $\text{Alg}(4-11)$ was as large as 82Å. Since it is infeasible to generate and work with such a large lattice, we have to reduce the lattice margins to small enough to be handled by our computational resources. In this case, 30Å is the maximum.

We observed from our experiments that usually, the longer protein chain, the larger sight distances computed by (4-31). For any protein with less than 60 residues, the maximum sight distance computed by (4-11) is less than 20Å. For any protein with less than 130 residues, the maximum sight distance computed by (4-31) is less than 30Å. Therefore, for any protein chain in our database, we can start with a lattice with 30Å margin, divide protein chain into tuples of tuple size less than 20, and group consecutive tuples into groups with maximum 130 residues per group, and then compute sight distances using the same way as described in Section 4.3.2.4, i.e.,

$$\text{sd}(p_{j}) = \sqrt{\text{cdist}_{\text{tupel}(0,i,P-1)} - \text{cdist}_{\text{tupel}(0,i,P-1)}}$$

(4-32)

Where $\text{cdist}_{\text{tupel}(0,i,P-1)}$ is defined in (4-16).

The limits in the tuple size and group size guarantee that the sight distances computed by (4-32) are safe, because the optimal lattice...
approximation for any tuple less than 20 vertices and any group less than 130 vertices found within the lattice of 30Å margin is the same optimal lattice approximation found within the lattice of as large as 82Å margin. Our experiments showed that for the largest protein in our database, the maximum sight distance computed by $(4 - 32)$ is less than 30Å. Therefore, it is sufficient to work with a lattice of 30Å margin for any protein in our database.

Instead of using a lattice of 30Å margin for the whole protein chain, we can further speed up the process by generating a much small sized lattice with 20Å margin for each tuple when computing the optimal lattice approximation for tuple $t$, to get $\text{cdist}_{30}(t)$, and generating a much smaller sized lattice with 30Å margin for each group when computing the optimal lattice approximation for group $g$, to get $\text{cdist}_{30}(g)$. 
CHAPTER 5: EXPERIMENTAL RESULTS

5.1 Database Schema and Testing Set Selection

As of November 21, 2006, there are 36906 protein structures in PDB. A protein can have as many as 60 chains. The longest protein chain in PDB contains 2999 residues (protein 1S11 chain 3). The distribution of lengths of chains over all the proteins in the PDB as of above date is shown in Figure 5.1.

Figure 5.1 Distribution of lengths of chains over all proteins in PDB.
A subset of PDB files is chosen based on a method similar to that used previously by Karplus.[10] Files are extracted from the Protein Data Bank (PDB) (version April 13, 2004) if their attributes include X-Ray diffraction, resolution of less than or equal to 1.75Å, and an R factor less than or equal to 20°. We eliminated files containing chains of length smaller than 2, resulting in 3704 PDB files. Each PDB file in our subset was parsed to extract Cα atom x, y, and z coordinates, amino acid type, sequence number etc. We call this subset of PDB the *source*.

All chains are parsed from each protein in the source. In a chain, if the distance between two consecutive residues is larger than 6Å, we broke the chain into two chain segments between these two residues. Therefore, in any chain segment, the distance between any two consecutive residues is no more than 6Å. Each chain segment is given an unique name, and will be fitted into lattice later. For an example, the second segment in chain A of protein 1AZO is named chain segment 1AZO_A2. This is because we are looking for the lattice approximation which is a path in the lattice. If the distance between two residues
is too large, the it would be natural not try to fit those two residues to two consecutive lattice vertices and minimize the overall $c_{\text{dist}}$, but instead break the chain into two independent chain segments, and try to find for each chain segment an approximation which is a path in the lattice s.t. the $c_{\text{dist}}$ is minimized. The set of all chain segment is used as the database in our experiment. There are total 9470 chain segments whose lengths range from 10 to 1014 residues in our database. The distribution of lengths of chain segments in the database is shown in Figure 5.2.

Based on the database, a test set of total 104 chains is formed by randomly select 10 chains from all chains in each length interval of 100 residues (there are only 2 chain segments which have between 900 and 1000 residues). Two of the three longest chain segments each has 1014 residues are also included in the test set.

5.2 Lattice Generation

We tested our protein chain on 3D cubic lattices. For any lattice which is regular and periodic (such as cubic lattice), every vertex is undistinguishable from the others. In the experiment, the orientation of the lattice is determined by placing a lattice vertex at the coordinates (0, 0, 0), and the 6 neighbours of the lattice vertex at the coordinates $(\pm 1, 0, 0), (0, \pm 1, 0)$ and $(0, 0, \pm 1)$.

We generate one 3D finite cubic lattice specifically for each protein chain in our testing set. The precise boundaries of x, y, z coordinates of the lattice is computed by using Algo(4 - 11). Figure 5.3 shows the $\max\{sd(p,) | p, \in V_p\}$ in all
proteins $G_r$ in the testing set. As it shows, the maximum margin for any lattice of a protein in the testing set is no more than 81.45 Å, and in general, for longer proteins, larger margins are needed for their lattices.

For a finite 3D space whose boundaries of x, y, z coordinates are represented as min(x), max(x), min(y), max(y), min(z), max(z), we call the value $\text{max}(x) - \text{min}(x)$ the span of the space along x-axis, the value $\text{max}(y) - \text{min}(y)$ the span of the space along y-axis, the value $\text{max}(z) - \text{min}(z)$ the span of the space along z-axis. The values of span give us an idea of the size of a 3D space. Figure 5.4 gives us an idea of the size of protein space and lattice space. Figure 5.5 shows how much larger the lattice space could be compared to the protein space. Any span in a lattice space is no more than 3.06 times of any span in the corresponding protein space. It is also clear from the plot that longer proteins...
require much larger lattice spaces, which is consistent with the conclusion we
drawn from Figure 5.3.

In our experiments, we generate a lattice with 30Å margin for the whole
protein chain, and compute the optimal lattice approximation for the protein chain
within the lattice.

Figure 5.4 Maximum, average and minimum span along all axes of all protein spaces in
the testing set and their corresponding lattice spaces
5.3 Performance Comparison

The configuration of machine we ran our experiment is Sun Fire V60z, dual 3.06GHz Intel Xeon, 2GB RAM, 8GB swap, SuSE 9.0.

The cRMS values obtained by using Algo(4-10) with CPLEX and by the greedy algorithm (with the $\alpha$ value equal to 10) described in Section 2.2 are shown in Figure 5.6. Algorithm Algo(4-10) produces optimal solutions, whose cRMS value is smaller than the approximate solutions obtained by the greedy algorithm. As shown in Figure 5.7, the improvement of $c_{dist}$ values is no more than 30%, with an average 7.33%. However, the computational time used for computing optimal solutions is 167 times that of the greedy algorithm on average (as shown in Figure 5.8). For example, for the longest protein chains in our
testing set, using \( \text{Alg}(4-10) \) takes about 5 to 8 hours to produce optimal solutions, while greedy algorithm uses about 1 minute to produce a solution.

We can also see from Figure 5.6 that \( cRMS \) values of the optimal solutions increase with the length of the proteins.

For almost all proteins in our test set, the approximate solutions produced by using CPLEX and fixed 10Å sight distance are exactly the same as the optimal solutions. There are only 3 proteins whose optimal approximations are slightly better than their CPLEX 10Å solutions. But computing the CPLEX 10Å solutions takes less than 10% of the computational time for producing optimal solutions. For an example, it takes less than 30 minutes to compute CPLEX 10 Å solution for the longest protein chain in our test set.

Figure 5.6  Comparison of \( cRMS \) values of all the proteins in the test set by the greedy algorithm and \( \text{Alg}(4-10) \).
Figure 5.7 Improvement of cdist value got by Algo(4 - 10) compared to greedy algorithm

Figure 5.8 Improvement of computational time used by Algo(4 - 10) compared to greedy algorithm
CHAPTER 6: CONCLUSIONS AND FUTURE WORK

In this work, we formulated three integer programming models for fitting protein chains to lattice problem, and designed an algorithm using backtracking and dynamic programming approach to reduce the search space. Using optimizer ILOG CPLEX 9.1, we are able to solve the integer programming models on the reduced search space, and find the optimal lattice approximations for all protein chains randomly selected from PDB with up to 1014 residues.

The approximate algorithms, such as greedy algorithm and CPLEX fixed 10Å sight distance algorithm, are much more efficient in computational time, yet still produce approximate solutions which are very close to the optimal solutions in term of CRMS measurement. They are best applied to the situations when the solutions are needed to be generated in a short time.

It would be interesting to use the algorithm developed in this work to fit proteins to the other candidate lattices mentioned in Section 1.3. As a more challenging problem would be to redesign our IP formulations and pre-processing algorithms to work with the other commonly used similarity measures, namely with dRMS. The algorithm can be applied to fit part of the protein, such as a docking site to a lattice, which is also biologically interesting. One could improve the lattice-protein orientation by introducing translation and rotation to the fitting procedure, which is expected to produce more favourable lattice approximations of proteins.

89
Appendix A: Definitions

A.1 Linear Programming

A linear programming (LP) problem can be written in the following general form:

Objective Function: \[ \max \min f(x_1, x_2, \ldots, x_n) = c_1x_1 + c_2x_2 + \ldots + c_nx_n \]

Subject to Constraints: \[ Ax = b \]

With the bounds: \[ l_i \leq x_i \leq u_i, i = 1, \ldots, n \]

Here \( x = (x_1, x_2, \ldots, x_n) \) is the vector of variables, \( c = (c_1, c_2, \ldots, c_n) \in \mathbb{R}^n \) is the objective function coefficients vector, \( A \in \mathbb{R}^{m \times n} \) is the constraint matrix, \( b \in \mathbb{R}^m \) is the right-hand side vector, \( l = (l_1, l_2, \ldots, l_n) \) is the vector of lower bounds, and \( u = (u_1, u_2, \ldots, u_n) \) is the vector of upper bounds. Individual elements of the bounds vectors can be negative infinity or infinity respectively. The matrix \( A \) is generally not square, hence an LP cannot usually be solved by inverting \( A \). Usually there are more columns than rows in \( A \), which makes the constraints very likely to be under-determined, thus giving an enormous choice from the values of the solution vector.
An integer programming (IP) problem is a LP problem which requires all variables to take integer values. While LP with continuous variables falls into the complexity class P, IP is NP-hard. However, because of its ability to model a wide assortment of practical problems, researchers have developed many techniques to solve IP formulations. These are mainly in the form of branch & cut, branch and bound, or branch and price techniques, which enumerate possible integer values of the variables according to certain variable order (branch). A branch is cut off without further splitting if provably it cannot provide better objective function values than the current best one.

Integer variables may be further restricted to the values 0 (zero) and 1 (one), in which case they are referred to as binary variables.

A mixed integer programming (MIP) problem may contain both integer and continuous variables. If a MIP problem contains an objective function with no quadratic term, i.e., a linear objective function, then the problem is termed a mixed integer linear programming (MILP). A MIP problem is termed a mixed integer quadratic programming (MIQP) if there is a quadratic term in the objective function. If the model has any constraint contains quadratic terms, regardless of the objective function, the problem is termed a mixed integer quadratically constrained programming (MIQCP).

A.2 Graph Theory

The following are definitions of several notions from the graph theory which we will use in this work (see [17] for more details).
A graph $G$ is specified by a finite set of vertices $V$ and a finite set of edges $E$, denoted by $G := (V, E)$ where edges are unordered pairs of distinct vertices $(u, v)$, $u, v \in V$. If $(u, v) \in E$, we say that $u$ and $v$ are connected by the edge $(u, v)$.

A directed graph is a graph in which all the edges are directed. Formally, a directed graph is a finite set of vertices $V$ and set of edges $E$, where each edge is an ordered pair $(u, v)$, $u, v \in V$; $u \neq v$. The vertex $u$ is the initial vertex and $v$ is the terminal vertex of edge $(u, v)$. An edge in the directed graph is also called an arc.

In an undirected graph, vertex $v$ is adjacent to vertex $u$ if they are connected by an edge. In a directed graph, edge $r = (u, v)$ and $s = (v, w)$ are adjacent if the terminal vertex of edge $r$ is the initial vertex of edge $s$, i.e., $u = v$.

In a directed or undirected graph, the set of neighbours $N(u)$ of a vertex $u$ is the set of all vertices which are adjacent to $u$, i.e., $v \in N(u)$ if two vertices $u, v \in V$ are connected by an edge.

In a directed or undirected graph, a walk is a sequence of adjacent vertices $(v_1, v_2, ..., v_n)$ such that $v_i$ and $v_{i+1}$ are adjacent, for every $1 \leq i \leq n - 1$. A path is a walk with no repeated vertices.

Graph homomorphism is a mapping between two directed or undirected graphs that maps adjacent vertices of one graph to adjacent vertices of the other.
More formally, a **graph homomorphism** $f$ from a undirected graph $G := (V, E)$ to a undirected graph $G' := (V', E')$, written $f : G \rightarrow G'$ is a mapping $f : V \rightarrow V'$ from the vertex set of $G$ to the vertex set of $G'$ such that $(f(u), f(v)) \in E'$ whenever $(u, v) \in E$. The above definition can be easily extended for directed graphs. Let $G := (V, E), G' := (V', E')$ be directed graph. A mapping $f : V \rightarrow V'$ is a homomorphism if for every arc $(u, v) \in E, (f(u), f(v)) \in E'$.

### A.3 Miscellaneous

Let $f$ be a function defined on domain of a set $A$ and taking values in range of a set $B$. Then $f$ is said to be an **injection** (or **injective map**, or **embedding**) if, whenever $f(x) = f(y)$, it must be the case that $x = y$.

Equivalently, $x \neq y$ implies $f(x) \neq f(y)$. In other words, $f'$ is an injection if it maps distinct objects to distinct objects. An injection is sometimes also called one-to-one.

Given two points in 3D Euclidean space, $p$ and $q$, whose 3D coordinates are $(x_p, y_p, z_p)$ and $(x_q, y_q, z_q)$ respectively. The **Euclidean distance** between $p$ and $q$ is given by $d(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2 + (z_p - z_q)^2}$.

Given two ordered sets of points $V = \{p_i | 1 \leq i \leq n\}$ and $V' = \{q_j | 1 \leq j \leq n\}$, the **coordinate root mean square deviation** is

$$ \text{cRMS}(V, V') = \sqrt{\frac{\sum_{i=1}^{n} d(p_i, q_i)^2}{n}} $$

Function cRMS measures how well two structures
\( V \) and \( V' \) are aligned. In this work, given \( V \) we are going to find \( V' \) from a certain set of valid structures (paths in a lattice) such that \( cRMS(V, V') \) is minimized. This problem is equivalent to the problem of minimizing the \( \text{cdist cost} \) \( c\text{dist}(V, V') \), which is defined as \( \text{dist}(V, V') = \sum_{i=1}^{n} d'(p_i, q_i). \) \textbf{Distance root mean square deviation} (\( dRMS \)) is defined as the intra-molecular distance matrix entries of the corresponding vertices in the two structures:

\[
dRMS = \sum_{i=1}^{n} \sum_{j=i+1}^{n} (d'(p_i, q_i) - d'(p_j, q_j))^2 / n(n-1)/2.
\]

Function \( dRMS \) measures how well the overall structure is preserved (note that mirror reflection of a structure is considered as fully preserves its original structure, and hence gives 0 value of \( dRMS \), even though the two structures cannot be aligned). Function \( cRMS \) and \( dRMS \) are two common measures on how two structures are aligned. In this work we only use \( cRMS \) measurement.

The degree \( (d) \) of the lattice is the number of neighbours of any particular vertex of the lattice. The degree of cubic lattice is 6. The degree naturally measures the complexity of the lattice.

\textbf{Appendix B: Algorithms}

\textbf{B.1 Description of Algo(4 - 1)}

The algorithm compute optimal approximation for a substring \( p_i, \ldots, p_e \), where \( s \) and \( e \) are given as an input. The most important data structures used in
Algo(4 - 1) are three arrays, \( I_v \), \( \text{neib} \) and \( \text{cost} \), which are all of the same size as the number of protein vertices in the substring. The relations among the three arrays are described below and shown in Figure B.1.

Figure B.1 Data structures used in Algo(4 - 1) and their relations

- **Array \( I_v \)** stores the indexes for the candidate lattice vertices. If the protein vertex \( p_{i+1} \) is mapped to \( q_j \), then \( I_v[j] = j \).

- Each element \( \text{neib}[i] \) indicates one neighbour of \( I_v[i] \), and any element in array \( \text{neib} \) ranges from 0 to \( d - 1 \) (\( d \) is the degree of the lattice). The value of \( I_v[i]+1 \) is determined to be the \( \text{neib}[i] \)-th neighbour of vertex \( I_v[i] \).
Array element \texttt{cost[i]} stores the accumulated \texttt{cdist} from the first protein vertex of the substring \( p \) up to the \( i \)-th protein vertex \( p_i \).

Beside the above four arrays, we also have another array \texttt{usedLV}, whose size is the same as the total number of vertices in the lattice. Because no lattice vertex can appear in a lattice approximation more than once, array \texttt{usedLV} is used to keep track of the lattice vertices which have been used in the mapping. If a lattice vertex with index \( i \) has been mapped to a protein vertex, then the value of \texttt{usedLV}[i] is set to 1 from 0. If the \( i \)-th lattice vertex is no longer being used in the mapping, then the value of \texttt{usedLV}[i] is set back to 0 from 1. Given a lattice vertex index, we can find out immediately (in constant time) by using array \texttt{usedLV} if the vertex has already been mapped to from a preceding protein vertex.

To speed up the computation, we use an array of arrays \texttt{dSqr[]} of size \( Q \). During the process of computing \texttt{vsight}(p), if \( q \in \text{vsight}(p) \), then we create an array of size \( P \) and store it in \texttt{dSqr[i]}. The element \texttt{dSqr[i][j]} will contain the square of distance between \( q \) and \( p \). If a lattice vertex \( q \) is not within sight distance of any protein vertex, then \texttt{dSqr[i]} points to null.

The basic idea of \texttt{Alg(4-i)} can be described in 3 phases:

Phase 1: Phase of Generating a Valid Path. Generate a candidate lattice approximation and check its \texttt{cdist}. It is done by the following:

- We generate a candidate lattice approximation for the substring and store it in array \texttt{lv} as a list of mapped lattice vertices. \texttt{lv[0]} looping through all the
lattice vertices (respectively, all vertices in \( \text{vsight}(p) \) if sight distance is used). And at the beginning of each iteration when \( M[0] \) get a new value, the remaining elements in array \( lv \), \( M[1] \) to \( M[e-s] \), are all set of 0, and all elements of array \( neib \) are set to 0.

- We generate the \( i \)-th vertex in the mapped lattice path by checking array \( lv \) and array \( neib \) at the current position \( i - 1 \), and we record the current position \( i - 1 \) in the variable \( neib\_pos \). When \( neib[neib\_pos] \) is trying to get the \( j \)-th neighbour of \( lv[neib\_pos] \), but the total number of neighbours that \( lv[neib\_pos] \) has is less than \( j + 1 \), which means that vertex \( lv[neib\_pos] \) does not have the \( j \)-th neighbour, we call the situation \( neib\text{-overflow} \).
  Whenever \( neib\text{-overflow} \) happens at position \( neib\_pos \), we stop generate the current candidate, and mark that it is necessary to do a procedure called \( \text{fast increase} \) starting from position \( neib\_pos \) so as to generate the next candidate (\( \text{fast increase} \) will be elaborated later).

- If the current candidate lattice vertex \( v \) (whose value equals to \( N[lv[neib\_pos]][neib[neib\_pos]] \) ) has been used in the mapping of previous positions, then there is no need to do further checking as we can not have \( v \) in the path more than once. We call the situation \( \text{duplicated-vertices} \). When duplicated-vertex happens, we mark that it is necessary to do \( \text{fast increase} \) procedure from position \( neib\_pos \).

- A variable \( cdist \) is used to store the minimum \( cdist \) over all lattice approximations we have encountered so far. At the end of the execution, the value of \( cdist \) is the value of \( cdist_{op}(s,e) \). During the process of
generating candidate from position 0 to position e-s, we check at every position \( i \) if the cost of the current partial candidate chain, \( cost[i] \) is smaller than the value of \( cdist \); if it is, we continue generate the next vertex mapping; otherwise, the current path can not be part of the optimal solution, therefore there is no need to further generate the next vertex on this path (We call this situation \textit{early-long-cdist}). Whenever early-long-cdist happens, we mark that it is necessary to do fast increase procedure from position \( neib\_pos \).

- If a new candidate is generated (\( neib\_pos \) reaches e-s) and current \( cdist \) is smaller than the value of \( cdist \), then we record this smaller value in \( cdist \).

\textbf{Phase 2: Phase of Increasing.} After a candidate lattice approximation is successfully generated and processed, we do preparation for generating the next candidate, i.e., do either fast increase, or normal increase.

- Do \textit{fast increase} if it is marked necessary. We have a variable called \( fastIncreasePos \) to keep track if it is necessary to do fast increase, and if it is, from which position in the \( neib \) to carry it out. The value of \( fastIncreasePos \) always equals to -1, unless a fast increase needs to be done, when the value of \( fastIncreasePos \) indicates the position from which to start fast increase. Fast increase happens in 3 cases: \( neib\)-overflow, duplicated-vertices and early-long-cdist. When we detect a fast increase request, we immediately increase \( neib[fastIncreasePos] \) by 1.
Do normal increase if fast increase is not necessary. We increase the current value of \( \text{neib}[e-s-1] \) by 1.

**Phase 3: Phase of Refreshing.** Refresh the value of some elements in array \( \text{neib} \) in 2 steps in order to be prepared for the next candidate.

- **Refreshing Step 1:** for all the \( \text{neib}[i] \) where \( 1 \leq i \leq \text{neib}_{\text{pos}} \), if its value is greater than \( d-I \), then set it to 0, and increase \( \text{neib}[i-1] \) by 1; mark the vertices in \( lV \) from position 1 to \( \text{neib}_{\text{pos}} \) to be unused. And set \( \text{neib}_{\text{pos}} \) to 0 if its current value if larger than 0.

- **Refreshing Step 2:** the \( \text{neib}[e-s] \) works as a stop sign. We set \( \text{neib}[e-s] \) from 0 to 1 if the value of \( \text{neib}[0] \) is greater than \( d-I \), which means all the possible valid candidates have been checked by now. Variable \( \text{neib}_{\text{pos}} \) should be reset to 0.

After finishing looping through all lattice vertices in \( lV[0] \), we have checked all the possibilities and the value of \( cdist \) is the value of \( cdist_{\text{opt}}(s,e) \).

---

**Algorithm:** Compute \( cdist_{\text{opt}}(s,e) \) by backtracking approach

**Input:** A protein substring specified by the starting index \( s \) and ending index \( e \), dimension of the lattice \( d \), \( cdist_{\text{opt}}(i), i = 0, \ldots, P-1 \) such that \( cdist_{\text{opt}}(i) = d'(P_i, C(P_i)) \)

**Output:** \( cdist_{\text{opt}}(s,e) \)

1. \( cdist \leftarrow \infty \) /*During the execution of the algorithm, the variable stores the best \( cdist(s,e) \) so far encountered. At the end of the execution, it stores the value of \( cdist_{\text{opt}}(s,e) \) */
2. \( \text{size} \leftarrow e-s+1 \) /*The length of the protein substring*/
if (size ≤ 0) then
   cdist ← 0.0
else
   if (size = 1) then
      cdist ← cdist_{opt} (s, S)
   else
      Define lv[size]
      Define usedLv[Q]
      Initialize every element in usedLv to be 0
      Define neib[size]
      neib_pos ← 0
      Define cost[size]
      Initialize the value of all elements in array cost to be 0
      fastIncreasePos ← -1
      Iv[0] ← 0
      while Iv[0] < Q do /* Phase 1*/
         cost[0] ← d2(s, Iv[0])
         Initialize all elements from the second element (with index 1) to the last element of array Iv and cost to be 0
         Initialize all elements in array Iv to be 0
         usedLv[Iv[0]] ← 1 /*Mark lattice vertex Iv[0] as being used*/
         Initialize all elements in array neib to be 0
         Initialize fastIncreasePos to be -1
         Initialize neib_pos to be 0
         while neib[size-1] < 1 do /*While not reach the stop sign*/
            while neib_pos < size-1 AND fastIncreasePos < 0 do
               if |N[lv[neib_pos]]| > neib(neib_pos) then /* No neib-overflow */
                  v ← N[lv[neib_pos]](neib[neib_pos])
               if usedLv[v] < 1 then /*No duplicated-vertices*/
                  cost[neib_pos+1] ← cost[neib_pos] + d2(s+neib_pos+1, v)
               if cost[neib_pos+1] < cdist then /* No early-long-cdist */
                  lv[neib_pos+1] ← v /*Mark v is the qualified candidate mapping of protein vertex at position neib_pos+1*/
                  usedLv[v] ← 1 /*Mark v as used*/
neib_pos++ /*Move to the next available position*/

else /*If early-long-cdist*/
fastIncreasePos ← neib_pos /*Do fast increasing*/
end if

else /*If duplicated-vertices*/
fastIncreasePos ← neib_pos /*Do fast increasing*/
end if
else /*If neib-overflow*/
fastIncreasePos ← neib_pos /*Do fast increasing*/
end if
end while /*end of while. We have got one valid candidate lattice approximation, which is stored in array lv*/

if fastIncreasePos > -1 then /*Phase 2. Do fast increasing*/

else /*Do normal increasing*/

end if

for all neib[l, 1 ≤ i ≤ size-2] do /*Phase 3 refresh step 1*/
if neib[i] > d-I then
neib[i] ← 0 /*Mark vertex Iv[i] as unused*/
neib[l-1]++
usedLv[lv[i]] ← 0 /*Mark lv[i] as unused*/
if i-I < neib_pos then
neib_pos ← i-I
end if
end if
end for
95: if nei[0] > d-1 then /*Phase3 refresh step 2*/
96:     nei[size-1]++ /*Refresh step sign*/
97:     nei_pos ← 0
98: end if
99: fastIncrasePos ← .1 /*Initialization for the next candidate path*/
100: end while /*End of while(nei[size-1] = 0)*/
101: lv[0]++ /*Starting from a new vertex, generate the next candidate*/
102: end while /*End of while(lv[0] < Q)*/
103: end if /*End of else*/
104: end if /*End of else*/
105: return cdist

B.2 Description of Algo(4 - 3)

Algo(4 - 3): Determining number of tuples and boundaries of tuples

Input: P
Output: The variable noTuple which stores the total number of tuples T, one dimensional array tupleSize, where tupleSize[i] stores the number of protein vertices in T_i.
Description: The boundaries of tuples can be determined by using noTuple and tupleSize.

1: if P < 200 then /*Determine iniTupleSize according to P*/
2:     iniTupleSize ← 10 /*Tuple sizes should be smaller than iniTupleSize, but as large as possible.*/
3: else
4:     iniTupleSize ← 20
5: end if
6: noTuple ← 1
7: s ← P
8: while s > iniTupleSize do /*Determine NoTuple according to P and iniTupleSize*/
9:     s ← ⌊s/2⌋ + 1
10: end while
11: noTuple ← noTuple x 2
12: Define tupleSize[noTuple] /*tupleSize[i] stores the size of the i-th tuple*/
13: Initialize all elements in array tupleSize to be 0
getTupleBoundaries(noTuple, 0, P, Q, noTuple-1, tupleSize)
14: return noTuple, tupleSize

getTupleBoundaries(noTuple, start, end, startTupleIndex, endTupleIndex, tupleSize[])
1: if startTupleIndex ≠ endTupleIndex then
2:  tupleSize[startTupleIndex] ← (end - start)/2 /*Divide current substring into 2 segments, and determine the length of the left segment*/
3:  leftSegmentSize ← tupleSize[startTupleIndex]
4:  tupleSize[endTupleIndex] ← (end - start) - leftSegmentSize /*Determine the length of the right segment*/
5:  getTupleBoundaries(noTuple2, start, start+leftSegmentSize, startTupleIndex, startTupleIndex+nTuple/2, end, tupleSize[]) /*Further divide the left segment recursively*/
6:  getTupleBoundaries(noTuple2, start+leftSegmentSize, end, startTupleIndex+nTuple/2, endTupleIndex, tupleSize[]) /*Further divide the right segment recursively*/
7: end if

B.3 Description of Algo(4-6)

Algorithm Algo(4-6) is based on Algo(4-1) but modified according to its special tasks. As in Algo(4-1), there are 3 phases in Algo(4-6), except that in the generating phase of Algo(4-6), we have the following modifications to speed up the process of generating valid paths.  

- For a protein substring \(p_i,...,p_j\), and a starting lattice vertex \(s_i\), we can compute all \(SDIST(p_i,p_j,s_i,*)\) simultaneously by generating all possible valid paths that start from \(s_i\) and end at any lattice vertex within \(vsite(p_j)\). Then \(SDIST(p_i,p_j,s_i,*)\) can be stored in an one-dimensional array \$ST\) of size equals to the number of vertices in \(vsite(p_j)\). Depending on the ending point of the path generated in a particular iteration that the
value in the sequence $SDIST(p_i, p_j, s_*, *)$ we will update. Since the value of $SDIST(p_i, p_j, s_*, s_j)$ would be the same as $SDIST(p_i, s_j, s_*)$ no matter whether the paths are generated forwardly or backwardly. This enables us to use algorithm Algol 4-6 to compute simultaneously not only values in $SDIST(p_i, p_j, s_*, *)$, as shown in Figure B.2, but also values in $SDIST(p_i, p_j, s_*, s_j)$, except that the paths are generated backwardly, as shown in Figure B.3.

Figure B.2 Illustration of $SDIST(p_i, p_j, s_*, *)$

![Figure B.2 Illustration of $SDIST(p_i, p_j, s_*, *)$](image)

Figure B.3 Illustration of $SDIST(p_i, p_j, s_*, s_j)$

![Figure B.3 Illustration of $SDIST(p_i, p_j, s_*, s_j)$](image)
• Let us use the two-dimensional array \texttt{vsight} to record the sight set for each protein vertex. For an example, lattice vertex \texttt{vsight}[j][k] is the \textit{k}-th element in \texttt{vsight}(p). Assume that ST[k] stores the minimum cost among all the paths ending at \( s_v = \texttt{vsight}[j][k] \) that we encountered so far. At the beginning of computation, all elements in array \texttt{ST} can be set to infinity. If there is a path which starts from \( s_v \) and ends at \( s_v \), then the value of \texttt{ST}[k] is not infinity at the end of the computation. When computing \texttt{TDIST}(t, \ldots, s, \ldots, \ast)\), we can speed up the computation by initializing elements in array \texttt{ST} to be cdist, whose value is the better upper bound than infinity and is used to avoid generating paths that is for sure not a part of an optimal solution. In this case, an one-dimensional Boolean array realPath of the same size as array \texttt{ST} is used to keep track whether we have succeeded in finding a path whose cost is smaller or equal to this upper bound. All elements in array realPath are initialized to be False. Only when the value stores in \texttt{ST}[k] is coming from the cost of a real path, would the value of realPath[k] be set to True. At the end of the procedure to compute \texttt{TDIST}(t, \ldots, s, \ldots, \ast), \texttt{TDIST}(t, \ldots, s, \ldots, \ast) is marked as infinity if \texttt{SV} = \texttt{vsight}[j][k] and realPath[k] is False, since any map which maps \( s_v \) to \( s_v \) and \( s_v \) to \( s_v \) can not be optimal, and therefore we are not interested in using the map to further generate segmented anchored walk of the whole protein.

• During the process of generation a path, we need to make sure that the current candidate lattice vertex \( q_v \) is in \texttt{vsight}(p), where \( k = i + 1, \ldots, j \). If \( q_v \) is not in \texttt{vsight}(p), we say \( q_v \) is \textit{out of sight}. In such a case, it is necessary to carry out Fast Increase at the corresponding position, in the same way as described in \texttt{Algo}(4-1). In order to check efficiently whether \( q_v \) is out of sight, we use a two-dimensional array \texttt{insight}, which is of size \( j - i + 1 \) in the first dimension and \( Q \) in the second dimension, to keep
track if a lattice vertex is in the sight set of a protein vertex, and if yes, what is the index of the lattice vertex in the sight set. For an example, if $q_i = \text{v}\text{sight}[k][v]$, then $\text{insight}[k][v] = i$; otherwise, $\text{insight}[k][v] = -1$.

- We use a variable $\text{maxST}$ to keep track of the maximum value of all elements in array ST. Once it is ensured that $q_i$ is not Out of Sight, we check whether $\text{maxST}$ is smaller than the cost of the path from the first vertex up to vertex $q_i$. If yes, then we say Early Long Cdist has happened, and we should do Fast Increase at this position in the same way as described in Alg(4-1). Since no matter which lattice vertex the current path will end at, the cost of the mapping would be larger than the corresponding value in array ST, and the mapping cannot be a part of a optimal solution. Therefore there is no need to further generate the next candidate lattice vertex along this path. If $\text{maxST}$ is larger than the cost of the path from the first vertex up to vertex $q_i$, then we continue generating the next candidate vertex along the path until the candidate lattice vertex $s_j$ (where $s_j = \text{v}\text{sight}[j][k]$) as the image of the last protein vertex in the substring has been generated. Then we check whether the cost of current mapping is smaller than the value of ST[$k$]. If it is smaller, then record the old value of ST[$k$], update the value of ST[$k$] by this smaller value, mark realPath[$k$] to be True, and recomputed $\text{maxST}$ if the old value of ST[$k$] equals to $\text{maxST}$.

---

**Alg(4-6): Compute $\text{SDIST}(p_i, p_j, s_i, s_j)$**

**Input:** $i, j, k$ where $s_i = q_i$, $v$ where $s_j = \text{v}\text{sight}[j][v]$, Array ST[$j$] of size $|\text{v}\text{sight}[j]|$.

When use this algorithm to compute $\text{TDIST}(l, s_i, s_j)$, initialize all elements in array ST to be $\text{cdist}_{i,j}(s_i)$ before passing it as input.

**Output:** Array ST as the value of $\text{SDIST}(p_i, p_j, s_i, s_j)$, and array realPath.

**Description:** For a protein substring $p_i, \ldots, p_j$, try to find vertex maps that map $p_i$ to $s_i$ and $p_j$ to $s_j$, where $s_i = \text{v}\text{sight}[j][v]$. If there exists such vertex maps, then record the
minimum cost of all such maps in \( ST[u] \), and mark \( realPath[u] \) to be True.

/*To avoid stating the same contents as in Algo(5-I), only the important part is shown below*/

\[ \text{getSTdist-allInOne}(i, j, k, v, ST) \]

27: \( \text{len} \leftarrow j - i + 1 \)

28: \( \text{maxST} \leftarrow \max\{ST[y]; y = 0, \ldots, \text{vsight}[j]\} \) /*maximum value of all elements in array \( ST[I] \)*/

29: Define Boolean array \( realPath[] \) of size \( \text{vsight}[j] \), and initialize all elements to be False.

30: for all \( \text{neib_pos} = i + 1, \ldots, j \) do /*Phase 1: Generating Phase*/

31: Generate current candidate \( q_i \), by checking \( LV[\text{neib_pos}] \) and \( \text{neib}[\text{neib_pos}] \) the same way as in Algo(5-I).

32: if \( q_i \) is not Neib Overflow then
33: if \( q_i \) is not Duplicated Vertices then
34: if \( q_i \) is not Out of Sight then
35: if \( q_i \) is the image of the last protein vertex in substring \( p_{i-1}, \ldots, p_j \) then
36: Mark \( q_i \), as used
37: Update \( \text{cost}[\text{len} - 1] \)
38: oldValue \( \leftarrow ST[\text{len} - 1] \)
39: if \( \text{cost}[\text{len} - 1] \leq \text{oldValue} \) then
40: \( ST[\text{len} - 1] \leftarrow \text{cost}[\text{len} - 1] \)
41: \( realPath[q_i] \leftarrow \text{True} \)
42: if \( \text{oldValue} = \text{maxST} \) then
43: \( \text{maxST} \leftarrow \max\{ST[y]; y = 0, \ldots, \text{vsight}[j]\} \)
44: end if
45: end if
46: end if
47: \( \text{neib_pos}++ \) /* if \( q_i \) is not mapping of the last protein vertex*/
Update cost[neib_pos+1]
if cost[neib_pos+1] ≤ maxST then  /* Not Early Long Cdist*/
Mark qi, as used
neib_pos++
else  /* Early Long Cdist */
fastIncreasePos ← neib_pos
end if

else  /* Early Long Cdist */
fastIncreasePos ← neib_pos
end if

else  /* If qi is out of sight*/
fastIncreasePos ← neib_pos
end if
else  /* If Neib Overflow happens*/
fastIncreasePos ← neib_pos
end if

end if

/* Got a valid path*/
return ST, realpath

B.4 Description of Algo(4 - 7)

Algo(4-7): Compute Combined Tuple Cdist LCT's and RCT's
Input: cdist,(ti), cdist,(O,P - 1)
Output: value of array LCT and RCT

1. left ← True  /* If left equals True, we are computing LCT; Otherwise, computing...
getCT(left, CT)

Input: left, two-dimensional array CT

Output: value of array CT

/* To compute LCT(t, s), the value of variable \(i, j, k\) is correspondent to the index of protein vertex \(p_i, p_j, p_k\) respectively. To compute RCT(t, s), the value of variable \(i, j, k\) is correspondent to the index of protein vertex \(p_i, p_j, p_k\) respectively. */

1. /* compute the first combine tuple exist LCT(t, s) or RCT(t, s) */
2. if left then /* if it is computing LCT(t, s) */
3. \(j \leftarrow S_t, \text{ where } t = p_{i_1}, \ldots, p_{i_k}\)
4. \(k \leftarrow E_t, \text{ where } t = p_{j_1}, \ldots, p_{j_k}\)
5. \(x \leftarrow 0 \text{ /* Variable } x \text{ keep track of the current tuple index */} \)
6. \(z \leftarrow T - 1 \text{ /* Variable } z \text{ store the index of the last tuple to be computed */} \)
7. else /* if it is computing RCT(t, s) */
8. \(j \leftarrow S_t, \text{ where } t = p_{i_1}, \ldots, p_{i_m}\)
9. \(k \leftarrow E_t, \text{ where } t = p_{j_1}, \ldots, p_{j_m}\)
10. \(x \leftarrow T - 1 \)
11. \(z \leftarrow 0 \)
12. end if
13. Define array CT[\(x\)] as an array of size \([\text{size}[k]]\), and initialize all elements in array CT[\(x\)] to be infinity.
14. Define array $T_{\text{dist}}[\|v_{\text{site}}[k]\|]$ For $T_{\text{DIST}}(t_1, s_{n_1}, t_2)^x$
15. for all $S_j = 0, \ldots, \|v_{\text{site}}[j]\|$ do
16. initialize all elements in array $T_{\text{dist}}$ to be $c_{\text{dist}}(t_1)$
17. Call getSTdist_allInOne($j$, $k$, $v_{\text{site}}[j]$, $S_j$, $T_{\text{dist}}$) in Alg(5-6)
18. for all $S_k = 0, \ldots, \|v_{\text{site}}[k]\|$ do
19. if realPath[$S_k$] = True AND $C[t][x][S_k] > T_{\text{dist}}[S_k]$ then
20. $C[t][x][S_k] \leftarrow T_{\text{dist}}[S_k]$
21. end if
22. end for
23. end for
24. /* compute all other combined tuple $c_{\text{dist}}$ LCT($t_1, s_{n_1}, t_2$) where $x = 1, \ldots, T-1$. or $RCT(t_1, s_{n_1}, t_2)$ where $x = T-2, \ldots, 0$ */
25. Define array $\min C[t]$
26. Define array sightindex($q$) /* To speed up the compuation, record the index of lattice vertex in $v_{\text{site}}[p,q]$, i.e., if $q_i = v_{\text{site}}[p_i][e]$, then sightindex[$j$] $\leftarrow$ $s$; if $q_i \neq v_{\text{site}}[p_i]$, then sightindex[$j$] $\leftarrow$ $-1$ */
27. for all $t_1, w = x, \ldots,$ do
28. if left then
29. $y \leftarrow w$ /*Variable $y$ keep track of the previous tuple index*/
30. $w \leftarrow w + 1$
31. $i \leftarrow k$
32. $j \leftarrow k + 1$
33. $k \leftarrow k + \text{tupleSize}[w]$
34. else
35. $y \leftarrow w$
36. $w \leftarrow w - 1$
37. $i \leftarrow k$
38. $j \leftarrow k - 1$
39. $k \leftarrow k - \text{tupleSize}[w]$
40. */
end if
 Define array Tdist[v\[j\]]
 Initialize all elements in array sightIndex to be -1
 for all Sj = 0, ..., \[v\[j\]\] do
     sightIndex[v\[j\][Sj]] ← Sj
 end for
 Define array minCT[v\[j\]], and initialize all elements to be infinity
 for all Sj = 0, ..., \[v\[j\]\] do
     for all Sn = 0, ..., \[N\][v\[j\]] do
         Sj ← sightIndex[N[v\[j\]][Sn]]
         if Sj ≠ -1 AND CT[y][Sj] < minCT[Sj] then
             minCT[Sj] ← CT[y][Sj]
         end if
     end for
 end for
 /*compute the value of current combined tuple cdist using minCT and Tdist*/
 Define CT[w] as an array of size \[v\[w\]\], and initialize all elements to be infinity
 for all Sj = 0, ..., \[v\[w\]\] do
     initialize all elements in Tdist to be cdist, (j, w).
     Call getSTdist alllnOne (j, k, v\[j\][Sj], Sj, Tdist) in Alg (5-6)
 for all Sk = 0, ..., \[v\[k\]\] do
     if realPath[Sk] = True AND CT[w][Sk] > minCT[Sj] + Tdist[Sk] then
         CT[w][Sk] ← minCT[Sj] + Tdist[Sk]
     end if
 end for
 end for
 return CT
B.5 Description of Algo(4 - 8)

Input: \( \text{cdist}_{\min}(t_t) \)

Output: Refined \( \text{vsight}(p_i) \) for all \( p_i \in \{t_0, \ldots, t_{T-1}\} \)

This algorithm demonstrates how to refine \( \text{vsight}(p_i) \) for all \( p_i \in \{t_0, \ldots, t_{T-1}\} \). If \( p_i \in t_t \), then there is no need to check LCT values; if \( p_i \in t_{T-t} \), then there is no need to check RCT values. In both cases, the same strategy applies but the algorithm needs to be modified accordingly.*

1. for all \( t_t, x \in \{1, T-2\} \) do
2. Define array \( \text{minLCT}[[\text{vsight}(S_t)]] \) and initialize all elements to be -1
3. Define array \( \text{minRCT}[[\text{vsight}(E_t)]] \) and initialize all elements to be -1
4. Define array \( \text{LST}[[\text{vsight}(S_t)]] \)
5. Define array \( \text{RST}[[\text{vsight}(E_t)]] \)
6. Define array \( \text{sightIndex1}[Q] \) and initialize all elements to be -1
7. for all \( Si = 0, \ldots, |\text{vsight}(E_t)| \) do
8. \( \text{sightIndex1}[[\text{vsight}(E_{t_0}, [S_i])] \leftarrow S_i \)
9. end for
10. Define array \( \text{sightIndex2}[Q] \) and initialize all elements to be -1
11. for all \( Si = 0, \ldots, |\text{vsight}(S_{t_0})| \) do
12. \( \text{sightIndex2}[[\text{vsight}(S_{t_0}, [S_i])] \leftarrow S_i \)
13. end for
14. for all \( p_i \in t_t \) do
15. for all \( q_j \in \text{vsight}(p_i) \) do
16. Initialize all elements in array \( \text{LST} \) to be -1
17. Initialize all elements in array \( \text{RST} \) to be -1
18. if \( \text{pruneSi}(i, x, j, \text{minLCT}, \text{minRCT}, \text{LST}, \text{RST}, \text{sightIndex1}, \text{sightIndex2}) \) = False then

112
Remove $q_i$ from $\text{vsight}(p_i)$.

end if

end for

end for

return refined sight sets

\text{pruneSi}

\text{INPUT:} i, \ s, \ t, \ p_i, \ e_i, \ j, \ \text{minLCT}, \ \text{minRCT}, \ \text{LST}, \ \text{RST}, \ \text{sightindex1}, \ \text{sightindex2}

\text{OUTPUT:} \text{True, if } q_i \text{ should be removed from } \text{vsight}(p_i); \text{ False otherwise.}

\text{DESCRIPTION:} \text{Compute } cdist_{\text{minLCT}, \text{minRCT}}(0, P - 1) \text{ and decide whether to remove } q_i \text{ from } \text{vsight}(p_i). \text{ The value of arrays minLCT and minRCT can be changed after the procedure returns.}

1: \text{for all } S_s = 0, \ldots, |\text{vsight}[St, I| \text{do}
2: \text{if } \text{minLCT}[S_s] < 0 \text{ then}
3: \text{min} \leftarrow \text{infinity}
4: \text{for all } S_n = 0, \ldots, |\text{vsight}[St, S_s]| \text{do}
5: \text{Si} \leftarrow \text{sightindex}[N[\text{vsight}[S_t, S_s][S_n]]]
6: \text{if } \text{Si} \neq -1 \text{ AND } \text{LCT}[x-1][S_i] < \text{min} \text{ then}
7: \text{min} \leftarrow \text{LCT}[x-1][S_i]
8: \text{end if}
9: \text{end for}
10: \text{minLCT}[S_s] \leftarrow \text{min}
11: \text{end if}
12: \text{if } \text{minLCT}[S_s] - \text{dSqr}[j][i] < \text{cdist}_{\text{minLCT}}(0, P - 1) \text{ then}
13: \text{for all } S_e = 0, \ldots, |\text{vsight}[Et, I| \text{do}
14: \text{if } \text{minRCT}[S_e] < 0 \text{ then } /*\text{if has not been computed yet, then compute it now}*/
15: \text{min} \leftarrow \text{infinity}
16: \text{for all } S_n = 0, \ldots, |\text{vsight}[Et, S_e]| \text{do}

Si ← sightIndex2([v, sight[i], [Se], [Ss]])
if Si ≠ -1 AND RCT(x+1)[Si] < min then
    min ← RCT(x+1)[Si]
end if
end for
minRCT[Se] ← min
end if
if minLCT[Se] + minRCT[Se] - dSqr[j][i] < cdist,(0, P - 1) then
    Compute q; q; = vsight[[i][v]]
    if LST[Se] < 0 then /*If it has not been computed yet, then compute it now*/
        Initialize all elements in LST to be infinity
        Call getSTdist_allinOne(i, Ss, j, v, LST) in Algo(5-6)
    end if
end if
if minLCT[Se] + minRCT[Se] + LST[Se] - dSqr[j][i] < cdist,(0, P - 1) then
    if RST[Se] < 0 then /*If it has not been computed yet, compute it now*/
        Initialize all elements in RST to be infinity
        Call getSTdist_allinOne(i, Et, j, v, LST) in Algo(5-6)
    end if
end if
if minLCT[Se] + minRCT[Se] + LST[Se] + RST[Se] - dSqr[j][i] ≤ cdist,(0, P - 1) then
    return True
end if
end if
end for
end for for all Se=0,...,|sight[Ps]| do */
end if
end for /*end for all Se=0,...,|sight[Ps]| do*/
return False
REFERENCE LIST


26. Protein. Retrieved December 1, 2006 from
http://en.wikipedia.org/wiki/Protein
