

UNIVERSITY of CALIFORNIA
SANTA CRUZ

**DATA STRUCTURES FOR A MINI-THREADING ALGORITHM
FOR PROTEIN STRUCTURE PREDICTION**

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Sugato Basu

September 2000

The thesis of Sugato Basu is approved:

Associate Professor Kevin Karplus, Chair

Associate Professor Richard Hughey

Professor Charlie McDowell

Dean of Graduate Studies

Copyright © by

Sugato Basu

2000

Contents

List of Figures	v
List of Tables	vi
Abstract	vii
Dedication	viii
Acknowledgements	ix
1 Introduction	1
1.1 Biological Background	2
1.1.1 Proteins	2
1.1.2 Composition of Proteins	3
1.1.3 Protein Structure	6
1.1.4 Protein Homology and Sequence Alignments	8
1.1.5 The Protein Folding Problem	12
1.1.6 Methods of Protein Structure Prediction	13
1.2 Mini-threading	14
1.3 Thesis Organization	15
2 Problem Description	16
2.1 Overall Picture	16
2.1.1 Undertaker	16
2.1.2 Slicer	18
2.2 Problem Formulation	19
3 Transform Class	20
3.1 Motivation	20
3.2 Theoretical Background	20
3.2.1 Overview of Quaternions	21
3.2.2 Optimal Superposition	22
3.3 Implementation	22

3.3.1	Blas, Clapack and Lapack++ Packages	22
3.3.2	Class Design	23
3.3.3	Enhancements for Efficient Transformation	23
4	Aligned Fragments Class	25
4.1	Motivation	25
4.2	Data Structure Design	25
4.2.1	Requirements	25
4.2.2	Tree Structure	26
4.3	InsertFragments Operation on Segment Tree	27
4.4	Semantics: Atomic Operations	30
4.4.1	DeleteEdge	31
4.4.2	InsertEdge	31
4.4.3	InsertAndSearch	32
4.4.4	ReplaceCoordinates	32
4.5	Semantics: InsertFragments Operation	33
4.6	Implementation Details	37
4.6.1	Design of Segment	37
4.6.2	Design of Segment Tree	38
4.7	Integration with Existing Programs	39
4.7.1	Modifications to Slicer	39
4.7.2	Integration with Undertaker	40
4.8	Results	40
5	Conclusion and Future Work	47
5.1	Conclusion	47
5.2	Future Work	48
A	User Guide	50
A.1	Sample Commands	50
B	Class Descriptions	54
B.1	Transform and Related Classes	54
B.1.1	XYZpoint	54
B.1.2	Pointlist	55
B.1.3	Quaternion	56
B.1.4	Transform	57
B.2	AlignedFragments and Related Classes	59
B.2.1	History	59
B.2.2	Edge	59
B.2.3	EdgeSet	60
B.2.4	Segment	60
B.2.5	SegmentList	62
B.2.6	AlignedFragments	63
	Bibliography	66

List of Figures

1.1	An amino acid	3
1.2	Chemical structures of some of the amino-acids [48]	5
1.3	A polypeptide chain of three amino acids	6
1.4	3D structure of Enolase [17, 18]	7
1.5	The alpha helix [17, 18]	9
1.6	The beta sheet [17, 18]	9
1.7	Interactions in a tertiary structure [17, 18]	10
1.8	Quaternary structure of F-1 ATPase [17, 18]	11
4.1	The random conformation of 1ede	41
4.2	The 3D structure of 1gpl	43
4.3	The 3D structure of 1tca	44
4.4	Fragments obtained from the alignment of 1ede to 1tca are applied to the random conformation of 1ede	45
4.5	Fragments obtained from the alignment of 1ede to 1gpl are applied to the intermediate conformation of Figure 4.4	46

List of Tables

1.1	The twenty amino acids found in proteins	4
3.1	Timing results for the Transform Class, collected on a 466 MHz Dec Alpha workstation	24

Abstract

Data Structures for a Mini-Threading Algorithm for Protein Structure Prediction

by

Sugato Basu

This thesis outlines the design and implementation of efficient data structures for a mini-threading protein structure prediction method called Undertaker.

Protein structure prediction is a critical and difficult problem in biology. Three basic techniques have been used in trying to solve this problem—comparative modeling, fold recognition (also known as threading) and ab initio techniques. Recently, mini-threading approaches, which try to combine the strengths of the threading and the ab initio methods, have shown promising results.

Undertaker uses mini-threading techniques to generate the tertiary structure of a target protein from its sequence, using information from a generic fragment library and alignments of the target to homologous templates of known structure. Undertaker generates predicted structures for fragments of the target sequence. It then pieces the fragments together, transforming the fragments in three-dimensions (3D) in the process, to form a predicted 3D conformation of the target sequence.

In this thesis, I have designed and implemented data structures for efficient 3D transformation of proteins and for modeling sets of protein fragments with tertiary relations. The thesis describes the theoretical ideas and the semantic correctness behind the data structures, and also gives implementation details.

I dedicate this thesis to my grandparents.

Acknowledgements

First of all, I am grateful to Prof. Kevin Karplus for his guidance, advice and continued support during the time I worked with him. He was always very approachable—he gave interesting theoretical insights, helped me find bugs in the code and gave tips on good programming style.

I would like to thank my loving wife Shalini for her untiring support, understanding and inspiration, and for bearing with me when the bugs in the code made me go crazy.

I am grateful to Eric Savage for developing the Slicer program, which was of great help for the thesis work. I would like to thank Melissa Cline for giving me her set of pairwise protein alignments, which I used to test the AlignedFragments code. I am grateful to Christian Barrett for pointing out important bugs in the Transform classes.

I would like to thank Prof. Charlie McDowell and Prof. Richard Hughey for taking the time to review this work.

It was a pleasure working in the BioInformatics and Machine Learning group at UCSC, with highly talented people like Mark, Rachel, Terry, Chuck, Nigel, Nguyet and Spencer.

I would like to thank Carol Mullane, Kristine Kilkenny and the other staff and faculty members of the CE/CS department of UCSC for making my MS program at UCSC a memorable and enjoyable experience.

Finally, I will always be grateful to Baba (Samir Basu), Ma (Nandita Basu), Kaku (Ashok Ghosh), Kakima (Rajyasri Ghosh), Putu (Indira Basu), Bhai (Rajib Ghosh) and my other relatives for their love, encouragement and inspiration.

Chapter 1

Introduction

This thesis outlines the design and implementation of efficient data structures for a mini-threading protein structure prediction method called Undertaker.

Protein structure prediction is a critical and difficult problem in biology. Three basic techniques have been used in trying to solve this problem—comparative modeling, fold recognition (also known as threading), and ab initio techniques. Recently, mini-threading approaches, which try to combine the strengths of the threading and the ab initio methods, have shown promising results.

Undertaker uses mini-threading techniques to generate the tertiary structure of a target protein from its sequence, using information from a generic fragment library and alignments of the target to homologous templates of known structure. Undertaker generates predicted structures for fragments of the target sequence. It then pieces the fragments together, transforming the fragments in three-dimensions (3D) in the process, to form a predicted 3D conformation of the target sequence.

In this thesis, I have designed and implemented data structures for efficient 3D

transformation of proteins and for modeling sets of protein fragments with tertiary relations. The thesis describes the theoretical ideas and the semantic correctness behind the data structures, and also gives implementation details.

1.1 Biological Background

This section is an introduction to some biological terms and ideas related to proteins. It is not a comprehensive biological background to proteins—it is only an attempt to explain some concepts that will be used in the rest of the thesis.

1.1.1 Proteins

Proteins regulate a variety of activities in living organisms and form the building blocks of life. Proteins perform diverse functions in a living organism, e.g., transport, storage, signaling, movement, and defense. They are responsible for regulating the cellular machinery and, consequently, the phenotype of an organism.

The function of a protein is closely related to its three-dimensional interactions with RNA, DNA, and other proteins [50]. Thus, knowledge of the structure of a protein is essential to gain a thorough understanding of its function. Knowledge of the sequence, structure, and function of proteins will have far-reaching effects on the design of new specific and stable proteins in bio-engineering, rational drug design in the pharmaceutical industry, etc. [46].

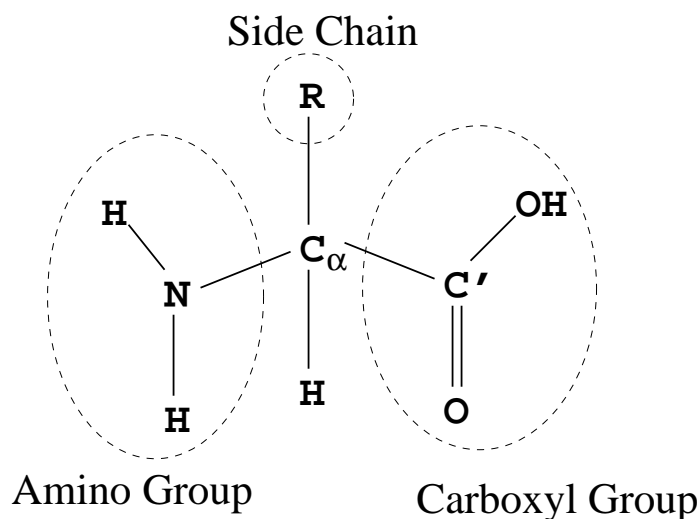


Figure 1.1: An amino acid

1.1.2 Composition of Proteins

A protein is composed of a contiguous chain of amino acids. All amino acids have a common central carbon atom, called the C_{α} atom, to which a hydrogen atom (H), a carboxy group ($COOH$), and an amino group (NH_2) are attached. What distinguishes one amino acid from another is the side chain, also attached to the C_{α} atom, as shown in Figure 1.1. There are twenty different side-chains that can attach to the C_{α} atom, giving twenty types of amino acids. The different types of amino acids are shown in Table 1.1 [8], and the chemical structures of some amino acids are shown in Figure 1.2 [48].

Amino acids are joined end-to-end during protein synthesis by the formation of peptide bonds. The carboxy group of the first amino acid has a condensation reaction with the amino group of the next, resulting in the elimination of a water molecule and formation of a peptide bond, as shown in Figure 1.3. The start of a protein is called the *N terminus*, while the end is called the *C terminus*. The series of N , C_{α} , and C' (of the

Amino Acid	Three-letter code	One-letter code
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartic acid	Asp	D
Cysteine	Cys	C
Glutamic acid	Glu	E
Glutamine	Gln	Q
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Methionine	Met	M
Phenylalanine	Phe	F
Proline	Pro	P
Serine	Ser	S
Threonine	Thr	T
Tryptophan	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

Table 1.1: The twenty amino acids found in proteins

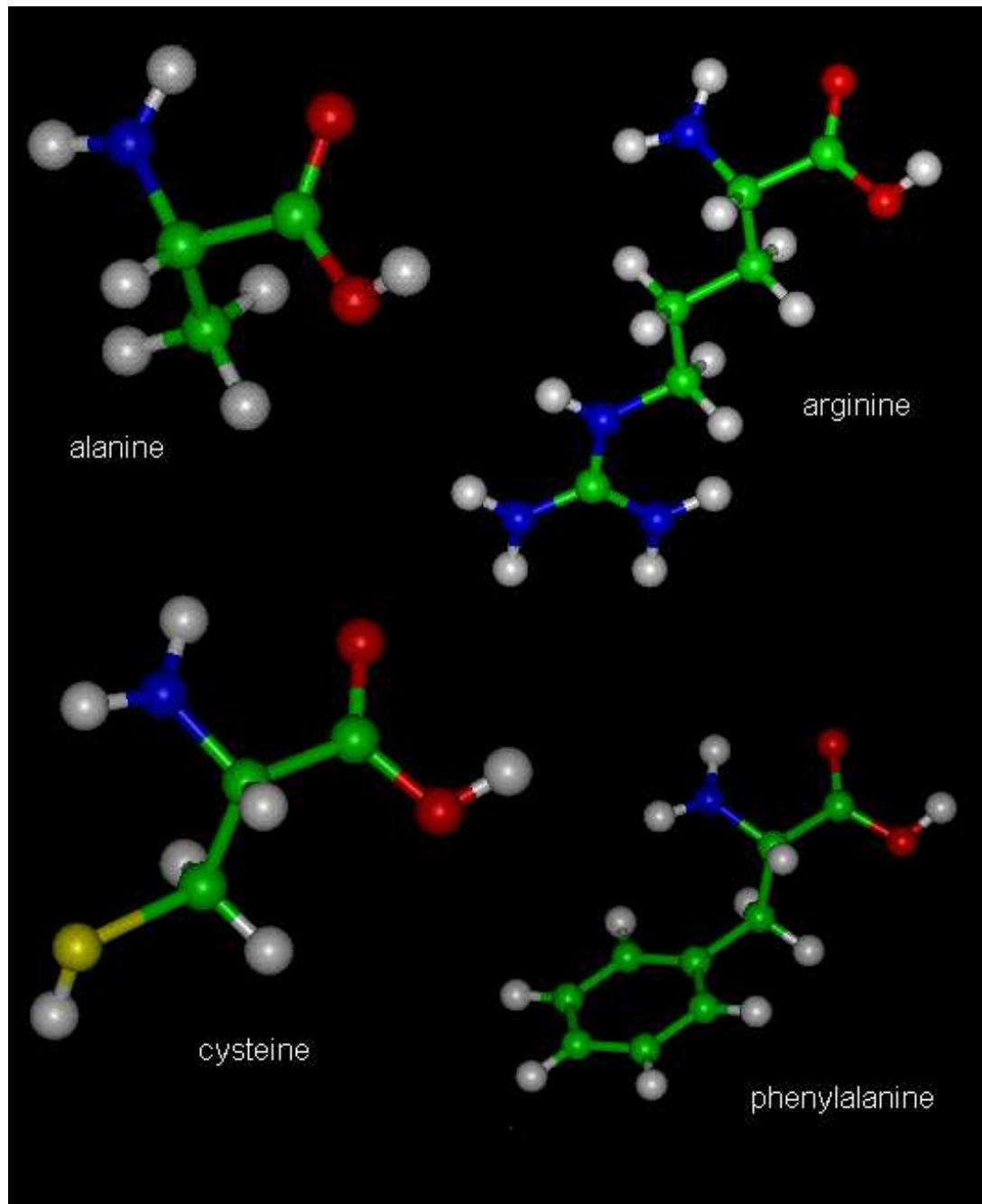


Figure 1.2: Chemical structures of some of the amino-acids [48]

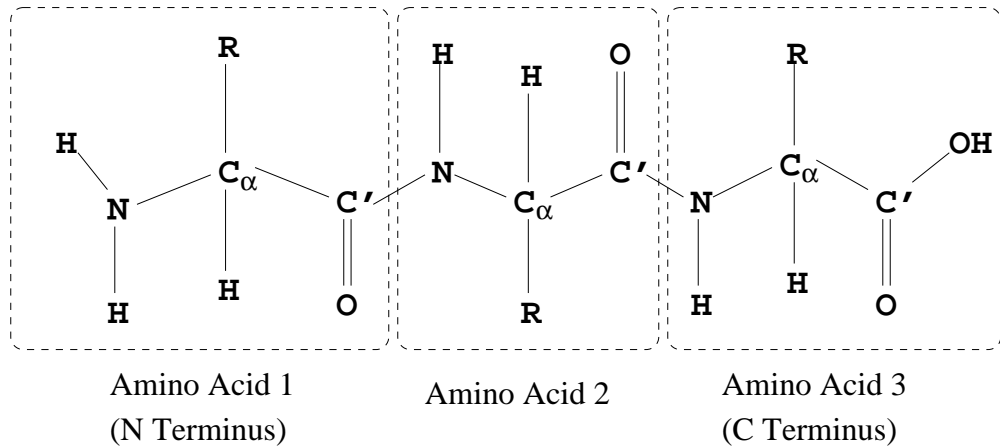


Figure 1.3: A polypeptide chain of three amino acids

carboxy group) atoms is referred to as the *backbone*, while the amino acids in the protein are referred to as *residues* [9].

1.1.3 Protein Structure

There are various forces of interaction between the amino acids in a protein, e.g., hydrophobic forces, hydrophilic forces, and pairwise interactions of residues (e.g., S-S bond in cysteine). The interaction of these forces makes the protein fold up into a complex, compact 3D structure. Figure 1.4 shows one example of a protein 3D structure, for the acid-denaturing kinase protein Enolase.

The structure of a protein is defined at four levels: *primary*, *secondary*, *tertiary*, and *quaternary* structure.

The primary structure of a protein is simply its linear amino acid sequence. It is usually represented as a string of the one-letter codes of the amino acids (Table 1.1) that occur along the polypeptide chain.

The secondary structure is a region of the protein with a regular local formation,

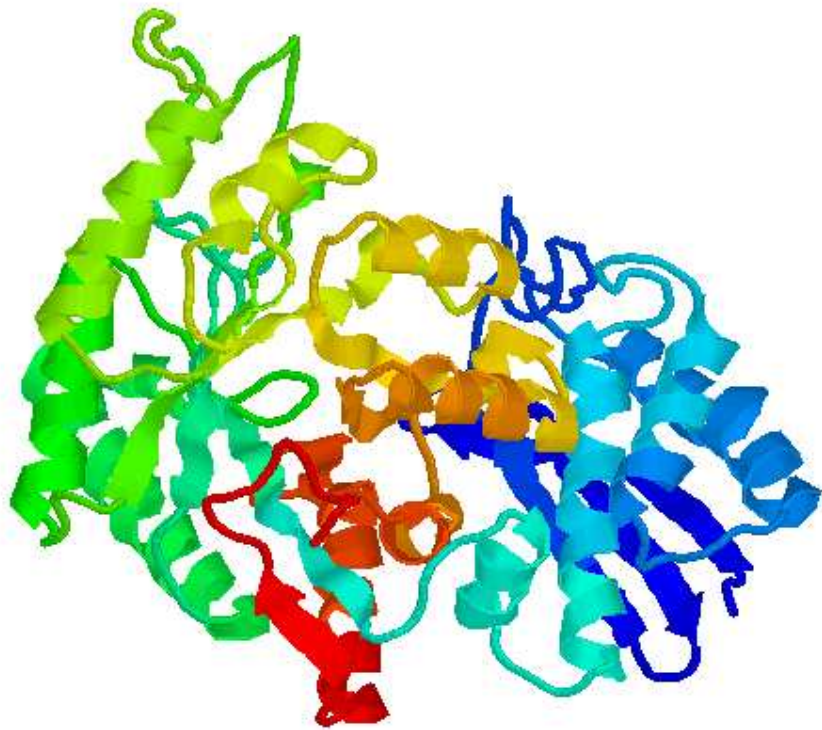


Figure 1.4: 3D structure of Enolase [17, 18]

characterized by specific bond angles. There are two types of secondary structure— α -*helices* and β -*sheets*, while random coils of residues are called *loops*. Figure 1.5 shows an example of an alpha helix, and Figure 1.6 shows an example of a beta sheet.

Tertiary structure is the compact 3D structure of the protein formed by packing the secondary structures. Different types of bonds that create tertiary structures are shown in Figure 1.7.

Some proteins have a single polypeptide chain, but a large number of proteins have multiple chains that interact to form a stable 3D structure. This is called the quaternary structure. Figure 1.8 shows that quaternary structure of the F-1 ATPase protein.

The figures in this section are taken from the web-site of the biochemistry book of Garrett and Grisham [17, 18].

1.1.4 Protein Homology and Sequence Alignments

Proteins can be classified into various families, according to their evolutionary, functional, structural, or sequence similarities. Such families of similar proteins are said to be *homologs*. Proteins are generally compared with their homologs using *alignments*—textual arrangements of two or more proteins arranged to indicate regions of sequence or structural similarity. Each *column* of an alignment has a vertical arrangement of amino acids from different proteins.

A *multiple alignment* contains an alignment between multiple proteins, while *pairwise alignments*, as the name suggests, aligns two proteins. A sample pairwise alignment is shown below:

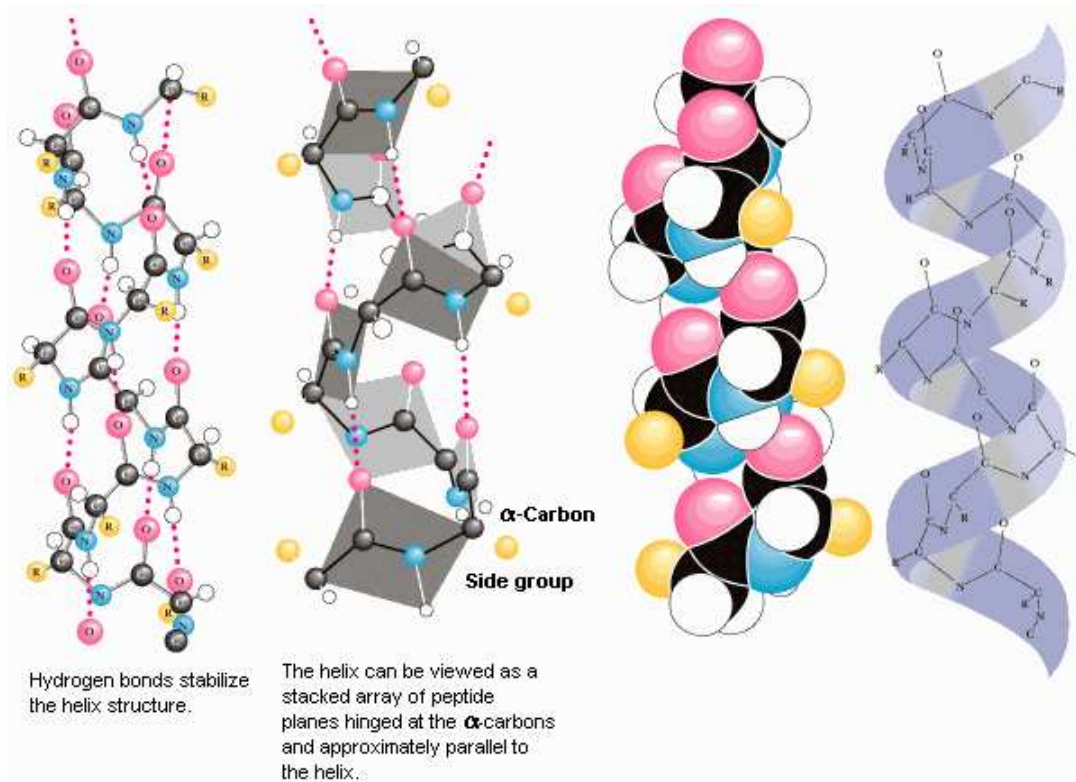


Figure 1.5: The alpha helix [17, 18]

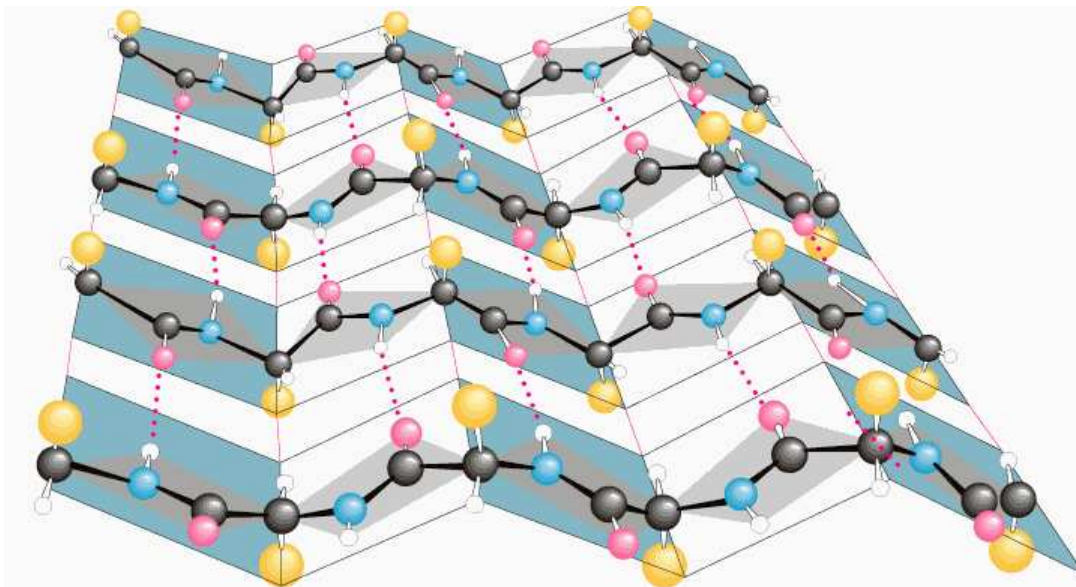


Figure 1.6: The beta sheet [17, 18]

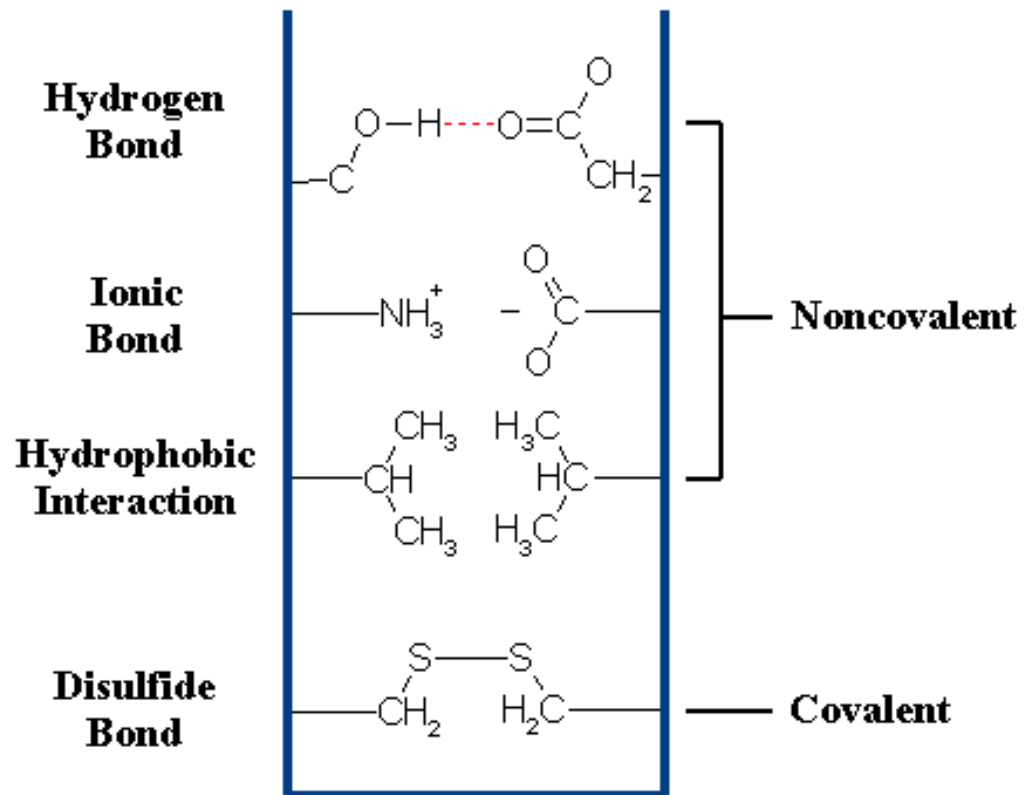


Figure 1.7: Interactions in a tertiary structure [17, 18]

F₁-ATPase

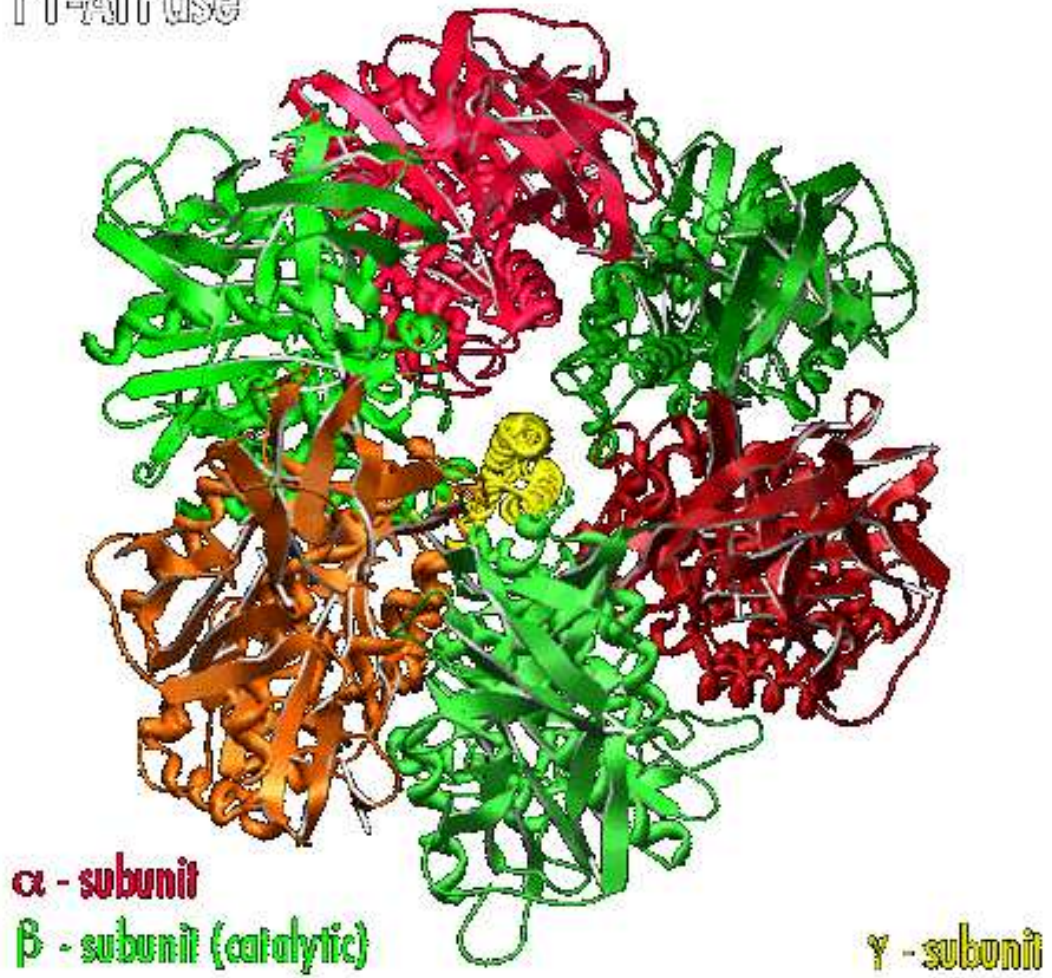


Figure 1.8: Quaternary structure of F-1 ATPase [17, 18]

```

>Sequence1
ayskSGAN---SDFTLLELN
>Sequence2
...SGGIVAGSDLAILKLN

```

The above alignment is in the a2m format. The alignment information is encoded using uppercase and lowercase characters, and the special gap character “-”. Uppercase characters and “-” represent alignment columns, and there must be exactly the same number of alignment columns in each sequence. Lowercase characters (and spaces or “.”) represent insertion positions between alignment columns or at the ends of the sequence [54, 34].

1.1.5 The Protein Folding Problem

DNA sequencing methods and statistical gene-finding techniques have provided tools for the rapid determination of gene sequences, from which the amino-acid sequence of a protein can be found by direct inference. But experimental methods for determination of the native three-dimensional (3D) structure of a protein are laborious. The two main types of experimental methods, namely *X-Ray Crystallography* and *Nuclear Magnetic Resonance (NMR) spectroscopy*, often require months or years of laborious work to determine the detailed structure of a protein. Structures of proteins determined by the experimental methods are stored systematically in a central repository called the Brookhaven Protein Data Bank (PDB) [6]. The number of known protein structures archived in the PDB at present is about 13,000 (June 7, 2000).

Christian Anfinsen discovered in 1972 that the information of the unique 3D structure of a protein is specified entirely by its sequence [2]. Ever since, a large amount

of effort has been devoted to the problem of finding the 3D structure of a protein from its sequence information alone. This problem has proved to be extremely difficult, and success has so far been limited.

1.1.6 Methods of Protein Structure Prediction

Currently there are three main methods of protein structure prediction that performed with varying levels of success at the CASP3 contest. These are comparative modeling, fold recognition, and ab-initio prediction. CASP (Community-Wide Experiment on Critical Assessment of Techniques for Protein Structure Prediction) is a meeting of research groups from around the world to assess and compare performances of current prediction methods on blind tests [14, 43, 37].

Comparative modeling exploits the fact that evolutionarily related proteins with similar sequences have similar structures. First, the target sequence is aligned to a template sequence of known structure. This sequence alignment is used to construct an initial model of the target sequence by copying over main chain and side-chain atoms corresponding to the aligned columns. A lot of groups are working in comparative modeling [16, 7, 22, 39, 52, 53], and their performance in CASP3 has been good overall [13, 5]. The drawback of this method is that it is viable only for proteins with known close homologs. The effectiveness of this method is limited by the extent of the structural similarity of the target protein and the homolog.

Fold recognition or threading uses a database of known three-dimensional structures to match sequences without known structure with protein folds in a template library. A scoring function is used to assess the quality of the fit (threading) of a target sequence

to a template fold, using sophisticated techniques, e.g., Gibbs Sampling, dynamic programming, and Hidden Markov Models. One of their major drawbacks of fold-recognition methods [38, 36, 35, 45, 51, 27, 32], is their computational complexity, but these methods have also performed well in CASP3 [33, 42, 31].

Ab-initio techniques work from first principles. They try to search the protein structure space for a 3D conformation that minimizes fundamental biophysical and biochemical forces. Different approaches have been used to design effective score functions and conformation sampling methods for recognizing good native folds, e.g., molecular dynamics, genetic algorithms, and lattice-based studies [19, 15, 20], with competitive performance in CASP3 [41, 40, 44]. A major drawback of these methods is the computational complexity because of the huge number of potential conformations to explore.

1.2 Mini-threading

A new approach called *mini-threading* has evolved in protein structure prediction in the last few years, which tries to combine the good features of threading methods [51, 38] and ab initio methods [41, 15]. Mini-threading methods are basically ab initio methods that rely on the structure knowledge base. The success of the mini-threading method of Simons and Baker [49] in CASP3 has encouraged researchers to look into this method in more detail. Simons's method generates short residue fragments of known structure having similar sequence to parts of the target sequence. It then assembles these fragments into a protein tertiary structure using the Monte Carlo simulated annealing technique.

1.3 Thesis Organization

This thesis is divided into 5 chapters and an appendix. Chapter 2 gives the detailed description of the problem that this thesis addresses. It formulates the problem and explains its relation to past and ongoing work in the BioInformatics research group at UCSC. In Chapter 3, the theoretical and implementation aspects of the fast 3D transformation classes are discussed. Chapter 4 explains the motivation and semantic correctness of the `AlignedFragments` class. It also gives an outline of the implementation of the `AlignedFragments` and related classes, and their integration with the Undertaker mini-threading program. Chapter 5 concludes the thesis and suggests relevant future work on this problem. Finally, Appendix A contains a guide to using the modified Undertaker program, and Appendix B contains descriptions of the C++ classes implemented.

Chapter 2

Problem Description

2.1 Overall Picture

The overall aim is to develop a mini-threading method to predict the tertiary structure of a given protein, called the *target protein*. The given information is the amino-acid sequence of the target protein, called the *target sequence*. Other available information includes the alignment of the target protein sequence to possibly homologous protein sequences of known structure, called *template sequences*, and information from a statistical knowledge-base of fragments of protein chains, called the *generic fragment library*.

2.1.1 Undertaker

A mini-threading method called *Undertaker* [30] is being developed by Prof. Kevin Karplus at the BioInformatics research group at UCSC. Undertaker uses a combination of the ideas of Dunbrack's method [7, 13] and Simons's method [49].

For a given protein sequence of unknown structure, the Undertaker method first

uses SAM-T99 [32, 25, 26] to identify close and remote homologs and to align the target sequence to possible template sequences. It then selects parts of the template structure whose sequence aligns with the target sequence and uses Dunbrack's tool SCWRL [7] to generate fragments from the template structure by side chain replacements of the corresponding residues. SCWRL strips the side chains from a homologous sequence and replaces them with side chains from the target sequence. Undertaker also uses fragments from a generic fragment library, which is a set of short (1- to 4-residue) 3-dimensional (3D) structures indexed by residues in the fragment (1 to 4 residues long), built from numerous actual high-resolution protein structures.

Undertaker creates a protein chain by assembling fragments from the generic fragment library, having the same residue sequence. The fragments, which may be in different 3-dimensional (3D) frames of reference, are transformed to the same frame of reference and pieced together to generate a random complete conformation for the target protein.

Single fragments, corresponding to gapless alignments to template proteins, can then be inserted into a complete conformation by splicing. The conformation can also be refined by various techniques, e.g., adjustment of rotamers for better packing.

Thousands of possible 3D protein conformations of the target protein, generated by this fragment insertion technique, are scored according to steric packing, water-exposure, rotamer preferences, and other factors. A genetic algorithm is used to create a pool of conformations, from which a set of high-scoring conformations is selected. The high-scoring conformations are potentially good predictions of the tertiary structure of the protein, and are suitable for further detailed analysis.

2.1.2 Slicer

The *Slicer* [47] program, developed by Eric W. Savage, is an automated method of generating possible fragments of the target sequence structure. The method has two steps: extraction and translation.

In the extraction step, the program looks at the alignment of the target sequence to the template sequence. It also reads in the template structure as a PDB file. If the template sequence in the alignment and the template PDB sequence do not match, it does a further alignment of these two sequences. The alignment of the target sequence and the template PDB sequence is passed on to the translation step.

The translation step looks at the template PDB file and extracts atom records from the PDB file, corresponding to the aligned residues obtained from the extraction step. These atoms and the alignment are passed as input to the translation tool SCWRL (Side Chain Replacement With a Rotamer Library), a program for adding side-chains to a protein backbone based on a backbone-dependent rotamer library [7]. A rotamer is one of many common conformations adopted by an amino acid side chain. The rotamer library contains information about the preferential angles at which side chain and the backbone atoms of the amino-acid bond to one another. SCWRL uses this information to replace the side chains with equivalent configurations, in which the 3D clashes between side-chain and backbone atoms are minimized. SCWRL outputs a PDB file with the backbone from the template structure corresponding to the aligned residues, with side chains from the target sequence at the aligned positions where the residues differ.

Slicer outputs this PDB file generated by SCWRL, along with an index file which

gives starting and ending indices of the contiguous residue fragments in the SCWRL-ed PDB file with respect to the target sequence.

2.2 Problem Formulation

This thesis addresses the following issues:

1. Design of a consistent and efficient data structure to store the intermediate 3D protein structures, obtained by the process of fragment insertion, and model dependencies between related fragments, e.g., fragments having a tertiary structure relation.
2. Design of a set of fast and efficient functions to transform protein structures in 3D. 3D Rotation and translation of protein fragments is repeatedly performed in the Undertaker method, and fast functions for these operations are essential for efficiency.
3. Modification of the Slicer program so that the multiple fragments it generates, by aligning the target protein to a template protein of known structure, are in a format compatible with the input format of the general data structure of fragments.
4. Design of an algorithm to insert multiple related fragments into the target protein conformation. It is important to preserve the tertiary relations between the fragments to be inserted and maintain the previously inserted tertiary relations, wherever possible. Before the work in this thesis, the Undertaker program allowed only a single fragment to be inserted into the conformation.
5. Integration of all these features and modifications with the Undertaker program.

Chapter 3

Transform Class

3.1 Motivation

An operation that is repeatedly performed in the Undertaker mini-threading method is the 3D transformation of protein fragments while joining the fragments to create a predicted structure. Functions that can rotate and translate protein structures in 3D efficiently can speed up the method by many times. This motivated the implementation of fast algorithms for 3D translation, rotation, and superposition of pointsets.

3.2 Theoretical Background

Several methods exist in the literature for finding optimal superposition between two 3D pointsets [4, 28, 29, 24]. I developed the transform class based on an efficient closed form solution of the problem given by Horn [23]. The solution uses quaternion operations to find the optimal rotation and translation to transform a set of points in one co-ordinate frame to a set of points in another co-ordinate frame, so that the root mean

square distance (RMSD) between the two pointsets is minimized.

In this method, the rigid-body transformation of the pointset is decomposed into a rotation and a translation. The method also addresses the problem of scaling, but I did not use that in developing the transformation classes, since the co-ordinate frames in which the structures are reported in the PDB files have the same scale.

3.2.1 Overview of Quaternions

Horn's method uses quaternions to handle rotation of pointsets. A quaternion $\hat{q} = q_0 + iq_x + jq_y + kq_z$ is a 4-dimensional (4D) vector, having one real component q_0 and three imaginary components q_x, q_y and q_z . A quaternion can be thought of as a complex number with three imaginary components. General operations defined on the quaternions, e.g., dot product of two quaternions, are the 4D analogues of the corresponding 3D vector operations. There is also a composite product operation for quaternions, similar to cross product for 3D vectors. If \hat{q} is a unit quaternion, i.e., a quaternion whose dot-product with itself is unity, then it can be shown that the composite product $r' = \hat{q}r\hat{q}^*$ is purely imaginary if r is purely imaginary [23]. Here, \hat{q}^* is the quaternion conjugate of \hat{q} , the 4D analogue of the complex conjugate.

A vector in 3D can also be represented as a purely imaginary quaternion. So, a point having 3D co-ordinates (x, y, z) can be equivalently represented as a quaternion $0 + ix + jy + kz$. It can be shown that rotation of a point about the unit vector $(\omega_1, \omega_2, \omega_3)$ by an angle θ is equivalent to the composite product of the quaternion of that point with the unit quaternion $\hat{q} = \cos(\theta/2) + \sin(\theta/2)(i\omega_1 + j\omega_2 + k\omega_3)$ and its conjugate [23].

3.2.2 Optimal Superposition

The problem of finding the optimal rotation quaternion to superpose two pointsets with minimum RMSD can be reduced to the problem of finding the eigenvector associated with the most positive eigenvalue of a symmetric 4×4 matrix N , where the entries of the matrix N can be calculated from the given points in linear time [23, 24]. The optimal rotation quaternion can hence be found very efficiently, since standard fast algorithms exist for computing maximum eigenvalues and eigenvectors of small constant size matrices.

The best translational offset is the difference between the centroid of one set of points and the rotated centroid of the points in the other co-ordinate system [23, 24]. This computation also runs in time linear in the number of points in the two sets.

3.3 Implementation

I developed the *Transform* class and other related classes, namely *XYZpoint*, *Pointlist*, and *Quaternion*, to provide fast transformation operations on pointsets, using Horn's method. I tested and incorporated the classes into the Ultimate library, a library of useful C++ functions maintained at UCSC by the BioInformatics group.

3.3.1 Blas, Clapack and Lapack++ Packages

lapack++ is a C++ library of functions for linear algebra routines (e.g., equation solving, matrix operations, eigen-value and eigen-vector computation) [11]. The *lapack++* drivers basically provide a C++ wrapper around the C routines of *lapack* [3] and *blas* [12]. I tested and installed these mathematical packages and used the optimized matrix and

eigen-value functions in these packages while developing the Transform class.

3.3.2 Class Design

The XYZpoint class defines a point in (x,y,z) co-ordinates and provides basic operations on points. The Pointlist class represents an array of XYZpoints and provides basic operations on this array. The Quaternion class defines a quaternion object as a 4D vector, with one real and three imaginary components. It also provides some basic operations on quaternions. Finally, the Transform class defines an object for transforming points, having a XYZpoint object for translation and a Quaternion object for rotation. Appendix B.1 contains detailed descriptions of these classes.

For finding the rotation quaternion and translational offset to optimally superpose two pointsets, the Transform class uses routines which compute the optimal rotation quaternion and translational offset between two pointsets using Horn's method [23]. I used the lapack++ expert driver *dsyevx* to find the eigenvector corresponding to the maximum eigenvalue of a 4×4 matrix, which is required for finding the optimal rotation quaternion.

3.3.3 Enhancements for Efficient Transformation

There are some additional features to the Transform class to make the transform operation more efficient:

- A *coplanar_trans* routine — This routine can be used to perform optimal rotation between two coplanar pointsets by direct geometric computation, without calculating the maximum eigenvector explicitly [23]. Use of this function speeded up the transformation of large test sets of coplanar points almost by a factor of five.

	1 point	200 points
Time to transform a point using rotation quaternion	1.33 microsecs.	0.74 microsecs.
Time to transform a point using rotation matrix	0.99 microsecs.	0.33 microsecs.

Table 3.1: Timing results for the Transform Class, collected on a 466 MHz Dec Alpha workstation

- Inplace operations — Inplace versions of rotation, translation, and transformation operations give the user the ability to directly modify the pointset that is passed into the functions, instead of creating a new pointset to pass back the result.
- A *quaternion_to_matrix* routine — This routine creates a rotation matrix from a rotation quaternion. Using the rotation matrix on single points gives no savings over the quaternion in speed of transformation, but for large point sets, the savings is about two fold.

The results of some timing tests run on the transform class is shown in Table 3.1.

The points were generated at random, and the timing information was collected by averaging over 1000 transformations.

Chapter 4

Aligned Fragments Class

4.1 Motivation

The mini-threading method used in Undertaker predicts a possible structure for a protein from its sequence by successively replacing sections of the residues of the protein by gapless *fragments* having similar residues and known structures. The key issue in the data structure design for such a method is to come up with a consistent and efficient way to store the intermediate 3D structures in this process of fragment replacement, while maintaining 3D tertiary relations between the fragments.

4.2 Data Structure Design

4.2.1 Requirements

The data structure would have to represent the contiguous set of residues in the structure, which we will refer to henceforth as *segments*. There may be gaps in the

protein backbone between segments in this intermediate structure, which will be filled up by fragment insertions in later stages. These breaks in the protein backbone have to be modeled by the data structure too. While generating the conformation by the successive insertion of fragments, two segments adjacent to a break may have to be merged into a single segment. While merging, the segments have to be transformed in 3D to maintain the proper length and orientation of the peptide bonds at the segment boundaries. So, the data structure would have to maintain information about the reference frame and transformation of each segment. Other dependencies between related segments also have to be modeled by the data structure. Examples of such related segments include segments having tertiary structure relations (e.g., S-S bond of a cysteine bridge) that have to be kept in the same frame of reference. During transformation of a segment, the data structure must ensure that other segments related to it undergo the same transformation.

4.2.2 Tree Structure

Keeping all these design issues in mind, Prof. Karplus and I came up with a solution involving a tree-based data structure. For the purpose of implementation efficiency, we represent the segments as nodes of a tree, and the relations between related segments as edges in the segment tree. We chose this representation to ensure efficient implementation of tree operations, such as insertion of alignments and transformation of segments. For proving the consistency of the tree structure under these operations, we consider an underlying semantic tree. This semantic tree models each residue in the protein as a node. The semantic tree represents peptide bonds in the backbone as *chain edges* and tertiary connections between related segments as *tertiary edges*. The operations on the segment

tree are equivalent to compositions of atomic operations on the underlying semantic tree. By proving that each of these atomic operations preserves the tree property and maintains a consistent 3D frame of reference of the overall protein structure, we show that the operations on the actual segment tree are correct—they preserve the peptide bonds and maintain the relations between related segments.

4.3 InsertFragments Operation on Segment Tree

The function *insertFragments(fragments)* is an important function in the Aligned-Fragments class. It inserts a set of fragments into a target conformation. The various steps in inserting the fragments are outlined below.

1. Connect the fragments to be inserted by tertiary edges, if required, to make a spanning tree. This spanning tree does not connect all the segments in the data structure—only the fragments that will be inserted are connected into a tree.
2. Create a set of protected edges, consisting of the existing edges of the segments and the edges between the fragments to be inserted.
3. If any of the fragments to be inserted does not have N_atoms and is not at the N-terminus of the chain, shrink the size of the fragment from the start and use the extra residue at the beginning to calculate the N_atoms. In case the fragment does not have C_atoms and is not at the C-terminus of the chain, shrink the end of the fragment. This is done to ensure that all segments have N_atoms and C_atoms, so that the transform for each segment can be calculated when the fragments are being

inserted (Note: This heuristic might create problems with segments that are 1 or 2 residue long).

4. Insert breaks at the two boundaries of each fragment.
5. If a fragment to be inserted spans multiple segments in the target structure, merge the segments it spans, without calculating any transforms. This merge operation has the following steps.
 - (a) By traversal of the tree, find the tertiary edge on the path in the tree between the two segments, such that this edge does not belong to the set of protected edges and is the longest (unprotected) edge in the path. The longest edge is chosen because it is least likely to represent a true tertiary relationship (Note: This heuristic might create problems with bad Conformations having a lot of clashes).
 - (b) If such an edge exists, delete it. Else, goto Step (c).
 - (c) End.
6. Remove tertiary edges incident on segments to be overwritten and set the locations of the atoms on which the edges are incident to a point far away. This is a hack to prevent edges incident on those points from coming into consideration when later calculations are done to find the minimum spanning tree joining the segments.
7. Insert minimum distance tertiary edges to chain together segments, using Kruskal's algorithm and the Union-Find method to find edges of the minimum spanning tree [10].

8. Delete all tertiary edges incident on the residues whose co-ordinates have been changed.
9. Insert the new atom co-ordinates into the pointlist locations corresponding to the fragments.
10. Connect the fragments to be inserted by tertiary edges, if required, to make a spanning tree. This spanning tree does not connect all the segments in the data structure—only the fragments that have been inserted are connected into a tree.
11. Corresponding to each tertiary edge connecting the fragments, insert a new tertiary edge between the segments in the data structure.
12. Merge the segments adjacent to each break iteratively, calculating the transforms simultaneously. This merge operation has the following steps.
 - (a) By traversal of the tree, find the tertiary edge on the path in the tree between the two segments, such that this edge does not belong to the set of protected edges and is the longest (unprotected) edge in the path. The longest edge is chosen because it is least likely to represent a true tertiary relationship.
 - (b) If such an edge does not exist, goto Step (f). Else, continue.
 - (c) Calculate the transform between the reference frames of the segments using the N_atoms and the C_atoms of the two segments.
 - (d) Determine by tree traversal the sizes of the connected sets to which each segment belongs, where size is measured by the number of atoms and connectivity is determined by the tertiary edges.

- (e) Transform the atoms of the segment, belonging to the smaller connected set, using the transform parameters calculated. Do a Depth-First-Search (DFS) traversal rooted at the transformed segment, and set up the values required to later bring all segments in the DFS tree to the same reference frame. This is done by calculating the Transform matrix of each segment with respect to the reference frame of the DFS tree root, so that the atoms in the segment can later be transformed by applying this Transform. This method of delayed Transform is more efficient in terms of number of transformations—instead of transforming the atoms directly in a segment whenever the segment gets transformed, successive transformations with respect to the underlying reference frame are stored in the Transform field of the segment and the atoms are only transformed once at the end of the merging, by applying the Transform to the atom co-ordinates.
- (f) End.

Perform merges at all break positions, if possible.

13. Bring all segments to the same reference frame, by applying the Transform of each segment to the co-ordinates of the atoms in that segment.

4.4 Semantics: Atomic Operations

Semantically, the actual operations on the segment tree are equivalent to compositions of atomic operations on the underlying semantic tree. This section gives brief descriptions of the atomic operations of the semantic tree.

4.4.1 DeleteEdge

DeleteEdge(D): Delete edge D .

- Precondition—The data structure is a forest consisting of n trees, where $n \geq 1$.
- Postcondition—The data structure is a forest consisting of $n + 1$ trees.
- Steps—
 1. Check D is in the forest.
 2. Remove D .

4.4.2 InsertEdge

InsertEdge(I): Insert edge I .

- Precondition—The data structure is a forest consisting of n trees, where $n \geq 2$.
- Postcondition—The data structure is a forest consisting of $n - 1$ trees.
- Steps—
 1. Check that the two residues joined by I are not already connected in the forest.
 2. Insert I .
 3. If I is chain edge, transform one of the trees which get connected by insertion of I , to bring the residues in the two connected trees to the same frame of reference.

4.4.3 InsertAndSearch

InsertAndSearch(I, protected edges): Insert the edge I and search and delete an edge from the data structure, such that it does not belong to the *protected edges* set.

- Precondition—

1. The data structure is a forest with n trees, $n \geq 1$.
2. I is not in the tree.

- Postcondition— The data structure is a forest with n trees.

- Steps—

1. Search along the tree path between the two residues which would be joined by I and select the longest tertiary edge to be D , such that it does not belong to the *protected edges* set.
2. If such an edge D is found, then DeleteEdge(D) and InsertEdge(I). Else, do nothing.

4.4.4 ReplaceCoordinates

ReplaceCoordinates(R, New): R is an interval of residues whose atom coordinates are replaced by those of the interval of residues New .

- Precondition—

1. The data structure is a forest having n trees.
2. The residues in R are not connected to any other residues, by chain or tertiary edges.

- Postcondition—

1. The data structure is a forest having n trees.
2. The tree now has inconsistent reference frames. This has to be resolved by adding chain edges across the breaks at the ends of R , since insertion of these chain edges will force one of the trees (connected by the insertion of the edge) to get transformed in 3D, and will make the residues come to the same reference frame.

- Steps—

1. Replace all coordinates of the atoms of R by the coordinates of the corresponding atoms of New .

4.5 Semantics: InsertFragments Operation

InsertAndSearch(I, protected edges): Insert the edge I and search and delete an edge from the data structure, such that it does not belong to the *protected edges* set.

The InsertFragments operation on the Segment Tree can be shown to be equivalent to a set of atomic operations on the underlying Semantic Tree, as shown below.

- Precondition—

1. Data structure is a forest.
2. Residues in the data structure, whose co-ordinates will be replaced by co-ordinates in a different frame of reference, are marked as *Replaced* residues. The other residues in the data structure are marked as *Unreplaced* residues.

The residues, whose co-ordinates will replace the co-ordinates of the Replaced residues, are called the *Foreign* residues.

- Postcondition—

1. Data structure is a tree.
2. The new co-ordinates of the Replaced residues have been set, and the overall reference frame of the data structure is consistent.

- Steps—

1. Insert tertiary edges, if required, inbetween the Foreign residues, to ensure that they are connected by edges to form a single spanning tree. The tertiary edges are inserted between the Foreign residues by the **InsertEdge** function.
2. Mark the tertiary edges present between residues in the original data structure and the tertiary edges in the spanning tree connecting the Foreign residues as *protected edges*.
3. Delete chain edges with one end incident on a Replaced residue and another end on a Unreplaced residue, using the **DeleteEdge** function. This removes the peptide bonds connecting the Replaced residues with the Unreplaced residues.
4. Insert chain edges between adjacent Replaced residues not connected by chain edges, if possible, using the **InsertAndSearch** function. The protected edges from Step 2 are used in the **InsertAndSearch** function. This step ensures that peptide bonds exist inbetween the Replaced residues. Also, unprotected tertiary edges between the residues may be removed.

5. Delete all tertiary edges, with one or both ends incident on a Replaced residue, using the **DeleteEdge** function. This ensures that the subtree containing the Replaced residues is completely isolated from the rest of the forest.
6. Insert tertiary edges, if required, between the residues in the data structure, to ensure that the current data structure is a single spanning tree. The tertiary edges are inserted by the **InsertEdge** function. This step ensures that current data structure is a tree. The hack in Step 6 of Section 4.3 implies that some of the tertiary edges inserted in this step are temporary, and will be removed in Step 7.
7. For each Replaced residue, delete the tertiary edges incident on it, using the **DeleteEdge** function. This removes the tertiary edges incident on residues whose co-ordinates will change in Step 8, since these tertiary relations are bogus. This also removes the temporary edges inserted in Step 6.

The overall effect of Steps 6 and 7 is to ensure that all Unreplaced residues are in a single tree, disjoint from the Replaced residues.

8. For each Replaced residue in the data structure, apply the **ReplaceCoordinates** function to replace its co-ordinate by the co-ordinate of the corresponding Foreign residue.
9. Insert tertiary edges, if required, inbetween the Foreign residues, to ensure that they are connected by edges to form a single spanning tree. The tertiary edges

are inserted between the Foreign residues by the **InsertEdge** function.

10. Corresponding to each tertiary edge connecting the Foreign residues in Step 9, use the **InsertEdge** function to insert a new tertiary edge between the residues in the data structure.

The overall effect of Steps 8, 9 and 10 is to ensure that all Replaced residues are in a single tree, disjoint from the Unreplaced residues.

So, at this stage, the forest of the current data structure can at most consist of two trees. The only case when the data structure consists of one tree is when there is no Unreplaced residue, i.e., when the co-ordinates of all the residues are replaced. Each of these trees is in a consistent reference frame, but the overall reference frame may not be consistent.

11. If the data structure after these operations is a single tree, then the reference frame is already consistent. Else, if the data structure is a forest with two trees, insert a chain edge connecting the two trees, using the **InsertAndSearch** function. The **InsertAndSearch** operation inserts a chain edge between the two trees and deletes a tertiary edge on the path between the residues joined by the chain edge, such that the deleted edge is not protected. Such a chain edge and such a tertiary edge to be deleted always exist. This is because, in the worst case, there must be one chain edge connecting the two trees, and there cannot be a unprotected edge joining the two trees (since they are disjoint). Other chain

edges are inserted in the tree, if possible across all the breaks, using the **InsertAndSearch** function. This removes all tertiary edges which are not protected.

This ensures that Step 11 makes the data structure a tree. The chain edge inserted to merge the two trees transforms one of the trees, and the overall reference frame of the data structure becomes consistent.

The correctness of each of these atomic operations ensures that the `InsertFragments` operation on the Segment Tree is also correct.

4.6 Implementation Details

I implemented the segment tree data structure as the C++ class *AlignedFragments*. Other related classes which had to be created are *History*, *Edge*, *EdgeSet*, *Segment* and *SegmentList*. This section outlines the design of the important data structures and functions. Appendix B.2 contains detailed descriptions of these classes.

4.6.1 Design of Segment

An important data structure in these classes is the segment, which is implemented in the `Segment` class. Each segment consists of a set of contiguous residues, which are stored efficiently by the first segment residue number and the first missing residue number. Each segment stores a set of edges incident on the segment. The segment maintains variables (traversal timestamp, rank, parent pointer, etc.) to aid operations on the segment, such as depth first traversal of the segment tree and finding the set of minimum edges to span

a set of segments.

The segment also stores two atom pointlists—*N_atoms* and *C_atoms*. *N_atoms* consists of the *C'* atom of the residue just before the segment and the *N* and *C_α* atom of the first residue in the segment. *C_atoms* consists of the *C'* atom of the last residue of the segment and the *N* and *C_α* atoms of the first residue of the next segment. These two pointlists are useful for merging adjacent segments. When a break between two adjacent segments is removed by merging the two segments at the two sides of the break, the atoms in the two segments may be in two different reference frames and have to be brought to the same frame of reference. The transform parameters required to transform the atoms of one segment to bring it to the reference frame of the other segment are calculated by doing a coplanar superposition of the *N_atoms* of the segment after the break with the *C_atoms* of the segment before the break. Whenever a segment is transformed, its *N_atoms* and *C_atoms* must also be transformed.

4.6.2 Design of Segment Tree

The segment tree is implemented in the `AlignedFragments` class. The `AlignedFragments` class is derived from the `Conformation` class in the `Undertaker` program, with which it has many features in common. The `Conformation` class is a data type for storing protein conformations, but has no representation for chain breaks or tertiary relations. The `Undertaker` program also has a `ChainsResiduesAndAtoms` class, which is suitable for storing a set of chains or fragments. `ChainsResiduesAndAtoms` stores a list of atoms, with types and locations, indexed by residue number [30]. The `Conformation` class has a `Pointlist` object to store the current 3D co-ordinates of the atoms in the conformation. It

also has a `ChainsResiduesAndAtoms` object called `Master`, which has the residue identification and the indices into the array of atoms.

The `AlignedFragments` class contains the set of the segments in the tree and the set of tertiary edges connecting the segments. The underlying `Conformation` base class stores the 3D co-ordinates of the atoms in the fragments.

4.7 Integration with Existing Programs

I integrated these classes with the `Undertaker` program and tested the insertion of fragments into a conformation, using the output of the `Slicer` program. I made certain changes to the `Slicer` code, to make the output format compatible with the input required for the `AlignedFragments` `insert_fragments` function.

4.7.1 Modifications to Slicer

I made the following modifications to the `Slicer` code.

1. The index file, giving the location of the fragments with respect to the target protein, originally gave just the start residue and the first omitted residue for each segment. In the modified index file, I added two flags to the specification of each fragment, to indicate whether the fragment had `N_atoms` and `C_atoms`.
2. I modified the `ChainsResiduesAndAtoms` file of the fragments, which `Slicer` gave as an output, to include one residue before and one residue after each fragment, wherever possible. This was done so that the `AlignedFragments` function reading the fragments from this file could extract the `N_atoms` and the `C_atoms` of each

segment, depending on the flags given in the index file.

Prof. Karplus did further substantial modifications to the Slicer code to handle the extensions for different cases.

4.7.2 Integration with Undertaker

I integrated the `Edge`, `EdgeSet`, `Segment`, `SegmentList` and `AlignedFragments` classes with the Undertaker program. I added the following two commands to the set of commands in Undertaker.

1. *ReadFragments*— Read a set of fragments from a `ChainsResiduesAndAtoms` file, using the specifications given in an index file. Store the set of fragments in a global `AlignedFragments` data structure.
2. *ApplyFragmentsToConform*— Insert the set of fragments stored in the global `AlignedFragments` data structure to the current conformation in the program.

4.8 Results

At first, I generated a random conformation for the 1ede target protein using fragments from the generic fragment library. The random conformation was loosely packed and did not have pronounced secondary structures or tertiary connections, as shown in Figure 4.1.

I used the Slicer program to generate fragments corresponding to the alignments of 1ede to two homologs – 1tca and 1gpl. The 3D structure of 1gpl is shown in Figure 4.2,

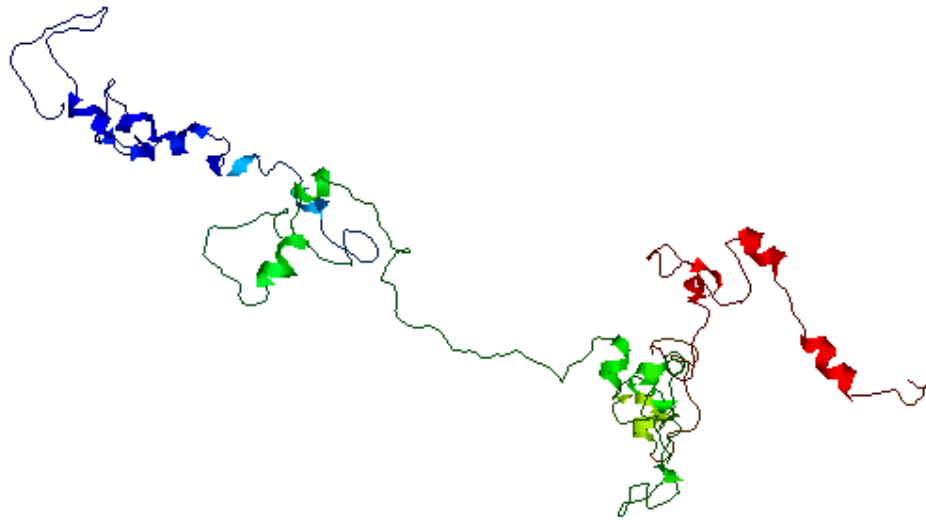


Figure 4.1: The random conformation of 1ede

and the 3D structure of 1tca is shown in Figure 4.3. Slicer generated four fragments for the lede-1gpl pair and four fragments for the lede-1tca pair.

The fragments obtained from the alignment of lede and 1tca are applied to the random conformation of lede, to generate the set of connected fragments as shown in Figure 4.4. The fragments obtained from the alignment of lede and 1gpl are further applied to this intermediate structure to give another set of connected segments, as shown in Figure 4.5.

Appendix A.1 contains details of the commands of the modified Undertaker program that were used to generate these results.

I collected profiling data for Undertaker by running it on the script in Appendix A.1, using the program *prof*. Prof gave the profiling information for the part of the program run on the serial Alpha 466 MHz processor. The profile does not include the time spent during the optimization process, run on the parallel Kestrel processor [21]. The profiling reveals that Undertaker spent about 50% of the time in creating the spanning tree connecting the segments. Most of this time was spent in finding the length of the edges while calculating the set of minimum distance edges required to connect the segments. This distance computation can be further optimized, by using techniques similar to that used by Prof. Karplus for clash detection [30].

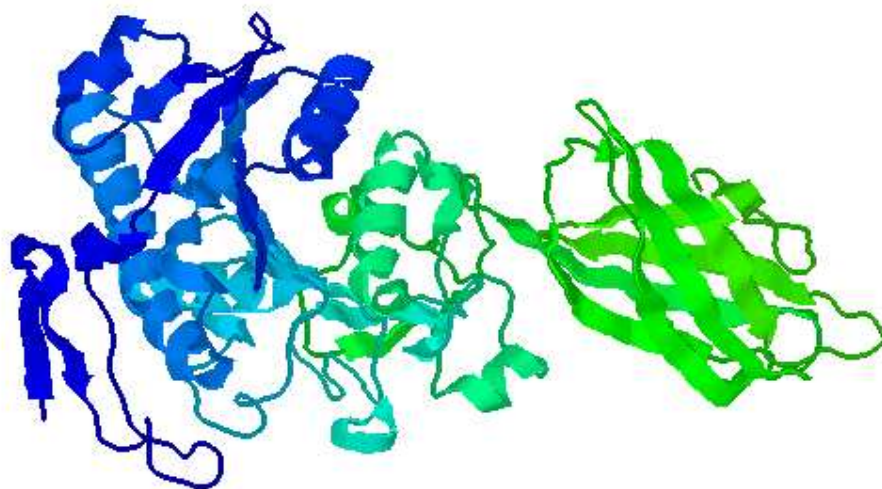


Figure 4.2: The 3D structure of 1gpl

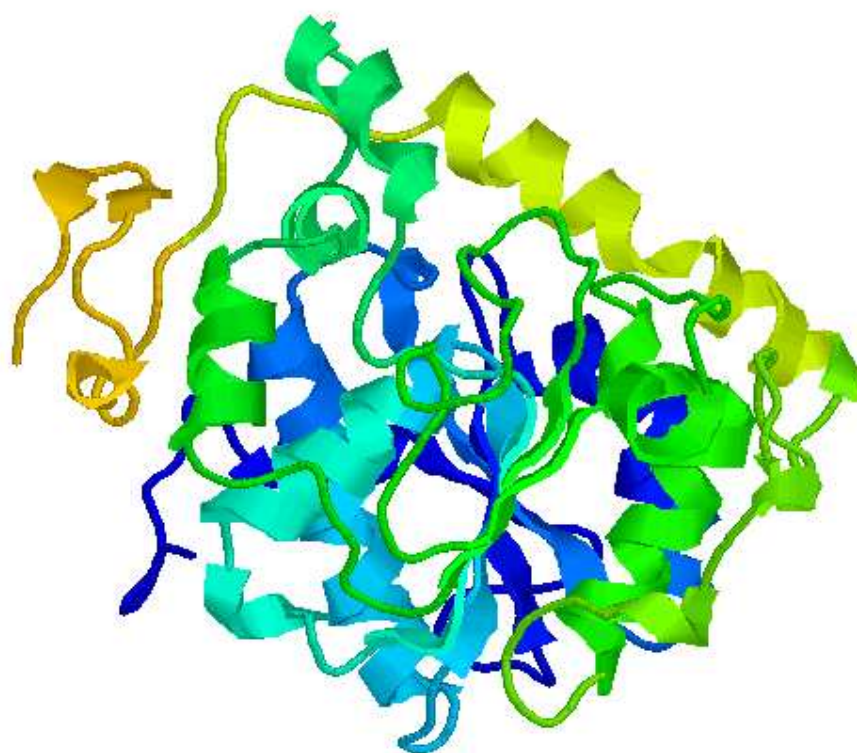


Figure 4.3: The 3D structure of 1tca

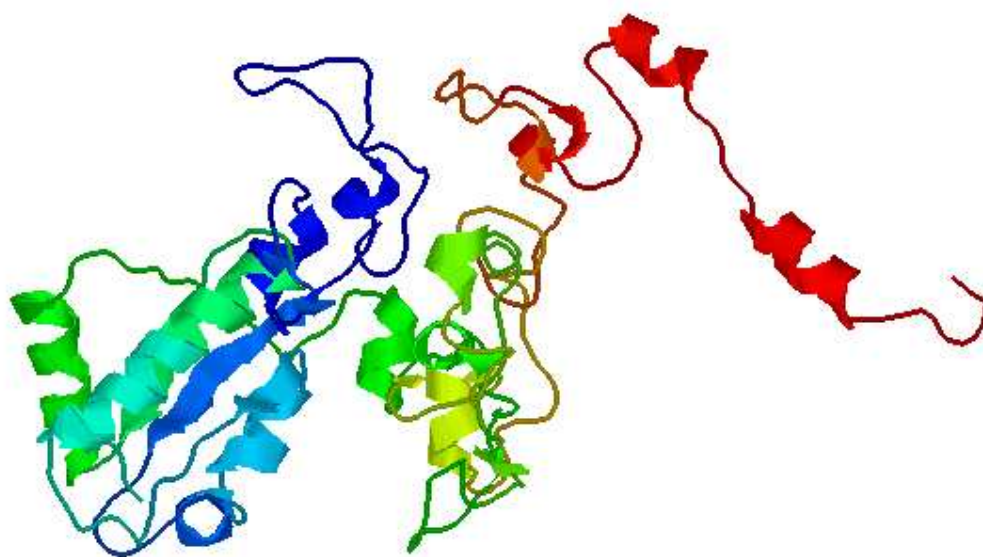


Figure 4.4: Fragments obtained from the alignment of 1ede to 1tca are applied to the random conformation of 1ede

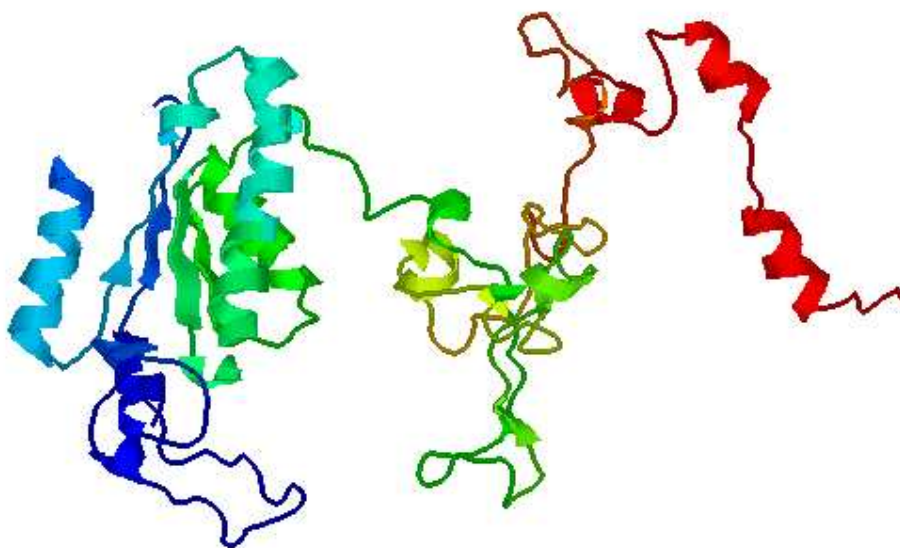


Figure 4.5: Fragments obtained from the alignment of 1ede to 1gpl are applied to the intermediate conformation of Figure 4.4

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this thesis, I accomplished the following.

1. Design and implementation of the Transform and related classes for efficient transformation of 3D pointsets
2. Creation of AlignedFragments and related classes to efficiently store and manipulate the intermediate protein conformations in the Undertaker method
3. Modification of the Slicer code to generate fragments from an alignment in a format compatible to the input format of the AlignedFragments class
4. Integration of the AlignedFragments class with the Undertaker program
5. Running test scripts on the merged code, which gave expected results

5.2 Future Work

To extend the work in this thesis, work can be done along the following lines.

1. The modified Undertaker program stores the structure of a target protein as a complete Conformation, on which an AlignedFragments object, representing a set of fragments, can be applied. The structures can be stored instead as AlignedFragments, which would enable the structures to be represented in a more general format, as a set of connected fragments. The single complete Conformation would be a special case of this, namely a single fragment. Storing the AlignedFragments results is an easy modification and has been done by Prof. Karplus. More substantial changes are needed to provide virtual functions for Conformation operations, such as `insert_fragment` and `peptide_splice`, that do the correct operation on AlignedFragments (already done by Prof. Karplus).
2. The optimization routines need to be rewritten to keep track of the regions inserted from an alignment and protect them (at least sometimes) from fragment replacement.
3. Hooks were provided to keep track of the history of conformations, but have not been used as yet.
4. Some new Conformation operators may be needed to try to choose chain breaks.
5. The AlignedFragments class currently handles only one chain in a protein structure. This can be generalized to make the class handle proteins with multiple chains.
6. The data structures of Undertaker do not currently handle ligands and ions, though they are essential for the correct structure prediction of many proteins.

7. Slicer is now a separate program. It can be modified to be a part of Undertaker.
8. A Union-Find (UF) algorithm that does not need to keep track of the rank of each node in the UF tree could be added [1]. This will reduce the memory requirement of the Segment data structure.

Appendix A

User Guide

A.1 Sample Commands

This section contains a set of commands to generate fragments from the Slicer program and insert them into conformations in the Undertaker program.

The results shown in Section 4.8 can be obtained in the following steps:

1. Copy the 1ede-1gpl and 1ede-1tca a2m files into the directory of the Slicer program.

Run the Slicer program with the commands:

```
slicer 1ede-1gpl-samA-w0.5-t98fssp.a2m 1ede 1gpl
slicer 1ede-1tca-samA-w0.5-t98fssp.a2m 1ede 1tca
```

2. Copy the .index and .cra files for 1ede-1gpl and 1ede-1tca, created in /tmp in the Slicer directory, to the Undertaker directory
3. Edit the .index files to set the correct path of the cra file. For example, I modified the file 1ede-1tca.index as follows:

```

ClassName = AlignedFragments
Target = 1ede
Template = 1tca
Alignment = 1ede-1tca-samA-w0.5-t98fssp.a2m
Atoms = tmp/1ede-1tca.cra <----- Changed to ./1ede-1tca.cra
NumResidues = 87
NumSegments = 4
Segments =
47 1 58 1
58 1 73 1
73 1 90 1
95 1 139 0
EndClassName = AlignedFragments

```

4. Create a .atoms file for the set of generic fragments to be used to create the fragment library. For example, I modified gx.atoms [30] and created gx-modified.atoms, a smaller subset of fragments, to reduce runtime.
5. Create a script to insert the fragments successively, using commands of the modified Undertaker program. For example, Prof. Karplus used the script apply-alignments-1ede.script [30].

```

InFilePrefix \
/projects/kestrel/users/karplus/burial/undertaker/atoms-inputs/
// sets the prefix of the input file
ReadTrainingAtoms dunbrack-925.atoms.gz
// reads fragments for generic library
ReadRotamerLibrary dunbrack-925.rot
// reads the rotamer library

```

```

InFilePrefix \
/projects/kestrel/users/karplus/burial/undertaker/spots/

ReadVDW initial.radii
ReadWetSpot dunbrack-925-wet-6.5.spot
ReadWetHist smoothed-dunbrack-925-wet-6.5.hist
ReadDrySpot dunbrack-925-dry-12.spot
ReadDryHist smoothed-dunbrack-925-dry-12.hist

```

```
SetCosts
// reads statistics of radii, wet spots, etc.
// from input files and sets costs

SetScoreParam \
    clash_penalty 10.0 wet_spot_weight 1.0 dry_spot_weight 1.0 \
    ss_bond_bonus 30.0 chain_break_penalty 5.0
// sets up parameters for scoring in various score
// and optimization commands

ClearIDs
AddId 1ede
// sets 1ede as target
ReadTargetPDBandapply

OutFilePrefix output/
ReportScore 1ede-6.5-12+rot.rdb

ClearIDs
AddTargetId
SetExclusion
MakeGenericFragmentLibrary
// creates generic fragment library

PatchTarget
// patches gaps in the target protein
SaveConformAsReal
// saves the current conformation as the REAL
// conformation for future comparisons

ConformFromSeq
// gets a random conformation using the generic library
NameConform 1ede-rand
PrintConformPDB 1ede-rand.pdb wet - dry -

OptConform 40 10 100
// creates a pool of conformations and does a genetic
// algorithm, generating a fixed number of new
// conformations in each generation

NameConform 1ede-rand-opt
PrintConformPDB 1ede-rand-opt.pdb wet - dry -
InFilePrefix \
/projects/kestrel/users/karplus/burial/undertaker/alignment-inputs/
```

```
ReadFragments 1ede-1gpl.index
// reads the fragments
ApplyFragmentsToConform
// applies the fragments
NameConform 1ede-rand-1gpl
PrintConformPDB 1ede-rand-1gpl.pdb wet - dry -

ReadFragments 1ede-1tca.index
ApplyFragmentsToConform
NameConform 1ede-rand-1gpl-1tca
PrintConformPDB 1ede-rand-1gpl-1tca.pdb wet - dry -

OptConform 40 10 100
NameConform 1ede-rand-1gpl-1tca-opt
PrintConformPDB 1ede-rand-1gpl-1tca-opt.pdb wet - dry -

ReadFragments 1ede-1tca.index
ApplyFragmentsToConform
NameConform 1ede-rand-1gpl-1tca-opt-1tca
PrintConformPDB 1ede-rand-1gpl-1tca-opt-1tca.pdb wet - dry -

OptConform 40 10 100
NameConform 1ede-rand-1gpl-1tca-opt-1tca-opt
PrintConformPDB 1ede-rand-1gpl-1tca-opt-1tca-opt.pdb wet - dry -

ScoreConformVsReal

quit
```

6. Run Undertaker on this script:

```
undertaker < apply-alignments-1ede.script
```

Appendix B

Class Descriptions

B.1 Transform and Related Classes

B.1.1 XYZpoint

This class defines a point in 3D (x,y,z) co-ordinates. It has three floats as public data members, representing the X, Y and Z co-ordinates of the point. The co-ordinates are stored as floats to save memory, and also because the precision provided by floats is sufficient for calculations on points.

The basic operators are

- $+$: Overloaded to add two points, or add an offset to a point,
- $-$: Overloaded to subtract two points, or to subtract an offset from a point,
- $\&$: Compute dot product of two points,
- $\%$: Compute distance between two points,

- `/`: Divide each co-ordinate by a scaling constant,
- `*`: Multiply each co-ordinate by a scaling constant, and
- `==`: Test if two points are equal.

The other operators are `+=`, `-=`, `/=`, `*=`, `>>` and `<<`.

The functions like `add`, `subtract`, `dot`, etc. are the function equivalents of the corresponding operators. There is also a “cross” function to find the cross-product between two points.

B.1.2 Pointlist

This class represents a list of XYZpoints. The pointlist is maintained internally as a private array of XYZpoints. The Pointlist class owns the XYZpoints and deletes them by an explicit destructor. A private function `Realloc` allocates more array space on demand.

The operators are

- `=`: Assignment operator for deep copy,
- `[]`: Index operator for the array of XYZpoints,
- `-=`: Subtract a point from each point in the list,
- `+=`: Add a point to each point in the list, and
- `<<`: Read a Pointlist from input.

The functions are

- `num_points`: Give the number of points in the array,
- `clear`: Set number of points in array to 0,
- `compute_centroid`: Find the centroid of the Pointlist,
- `add_point`: Add a new point to the Pointlist,
- `point(i)`: Return the i^{th} point from the Pointlist, and
- `square_dist`: Find the cumulative weighted square distance between the points in two pointlists.

B.1.3 Quaternion

This class defines a quaternion object as a 4D vector. It has 4 doubles as private data members, representing the 1 real and the 3 imaginary components of the quaternion.

The basic operators are

- `+`: Add two quaternions,
- `-`: Subtract two quaternions,
- `&`: Compute dot product of two quaternions,
- `*`: Overloaded to compute cross product of two quaternions, or cross-product between a unit rotation quaternion and a point, or multiply a quaternion by a scaling constant, and
- `/`: Divide a quaternion by a scaling constant.

Other operators are $+=$, $-=$, $/=$, $*=$, $>>$, and $<<$.

The functions like `add`, `subtract`, `dot`, `cross`, etc. are the function equivalents of the corresponding operators.

The other the functions are

- `conjugate`: Return the conjugate of a quaternion, and
- `axis_vector`: Return the axis-vector of a unit rotation quaternion.

This class has three versions of the frequently used operators and functions. The first returns the result by value, incurring a copy overhead. The second requires a reference to the destination to be passed in as an argument and returns the value by reference through this destination with no copy overhead. The third returns a pointer to a new object created with in the routine to store the result, with no copy overhead but with memory allocation overhead. The three versions of the operators and functions give the user the choice of minimizing copy overhead while developing application code using these classes.

B.1.4 Transform

This class defines an object for transforming points. Private data members are a Rotation quaternion and a translational Offset.

The basic operators are

- $*$: Compose two transformations: $(f * g)(\text{point}) = f(g(\text{point}))$,
- $/$: Compose two transformations: $(f/g)(\text{point}) = f(g^{-1}(\text{point}))$

Other operators are `*=`, `/=`, `<<`, and `>>`.

The functions like `times`, `divide`, etc. are the function equivalents of the corresponding operators.

The other functions are

- `inv`: Find inverse of a transform,
- `trans`: Transform a point or a pointset,
- `rotate`: Rotate a point,
- `shift`: Translate a point,
- `transIP`: Transform a point or a pointset inplace,
- `rotateIP`: Rotate a point inplace,
- `shiftIP`: Translate a point inplace,
- `rotation`: Set the rotation quaternion,
- `offset`: Set the translation offset,
- `optimal_transform`: Find the optimal rigid transformation of two Pointlists,
- `coplanar_trans`: Efficient computation of Rotation and Offset for coplanar points,
and
- `xyplane`: Compute the transform required to transform three points to the XY plane,
with one transformed to the origin.

The transform constructor can take as arguments two pointsets and a weight vector, computing the Rotation and the Offset to minimize the weighted distance between the pointlists, according to the point weighting specified by the weight vector.

B.2 AlignedFragments and Related Classes

B.2.1 History

This class maintains the history of the operations performed on the protein conformation e.g. inserting fragments, splicing, etc. It is currently a skeleton class that will be fleshed out during future modifications to the Undertaker program. History pointers should be available in base class Conformation.

Private data members are

- int EntryNumber: Index number of history entry
- History * Previous: Pointer to previous history entry
- Conformation * Conf: Pointer to Conformation associated with this history entry

B.2.2 Edge

This class models a tertiary relation between protein segments. It has AlignedFragments class as a friend. Private data members are

- const AlignedFragments * Owner: AlignedFragments owning this
- int FirstAtom: First atom on which edge is incident
- int SecondAtom: Second atom on which edge is incident

- float Length: Length of edge
- History * HistoryEntry: Pointer to entry in history table

The functions in this class provide access to the private data members and provide I/O.

B.2.3 EdgeSet

This class stores a set of edges. It has AlignedFragments and Segment classes as friends. Private data members are

- int NumEdges: Number of edges in set
- int AllocEdges: Number of edges allocated for set, reallocated on demand
- Edge ** EdgesInSet: Array of pointers to edges. The edges are not owned by EdgeSet — AlignedFragments is responsible for creating and destroying the edges.

The functions in this class provide access to the private data members and provide I/O.

B.2.4 Segment

This class models a set of contiguous residues in the protein conformation. It has AlignedFragments and SegmentList classes as friends. Private data members are:

- const AlignedFragments * Owner: AlignedFragments owning this
- int FirstSegmentResidue: Starting residue of segment

- `int FirstOmittedResidue`: Start of break associated with segment. `FirstOmittedResidue - 1` is the last residue of the segment. `FirstOmittedResidue` is always \leq `FirstSegmentResidue` of the next segment
- `EdgeSet * IncidentEdges`: Edges incident on the segment not owned by this
- `Segment * NextSegment`: Pointer to next Segment in linked list of `SegmentList`
- `Pointlist * N_atoms`: Store the locations of the carboxyl carbon atom of the residue just before the segment and the N and C_α atom of the first residue in the segment. This is useful for computing transform between segments
- `Pointlist * C_atoms`: Store the location of the carboxyl carbon atom of the last residue of the segment and the N and C_α atoms of the first residue of the next segment. This is useful for computing transform between segments
- `int TraversalTime`: Timestamp during segment traversal in `AlignedFragments` tree
- `Transform * SegmentTransform`: Store the relative transform of the atoms in the segment with respect to the underlying reference frame Applying `SegmentTransform` to the atoms in the segment puts them in the final co-ordinate reference frame
- `Segment * DFSParent`: Parent of segment in `DepthFirstSearch` (DFS) tree. DFS traversal is used to traverse the segment tree
- `Segment * UFParent`: Parent of segment in `UnionFind` (UF) tree. The UF method is used while connecting related segments by edges of a minimum spanning tree, using Kruskal's algorithm [10]

- `int Rank`: Rank of segment in the UF algorithm

Most of the functions in this class provide access to the private data members and provide I/O. Two important functions are

- `apply_transform`: Apply `SegmentTransform` to all the atoms in the segment and then sets `SegmentTransform` to identity
- `OK`: Check consistency of `C_atoms` and `N_atoms` with atoms in the segment

There are also functions such as `set_union`, `set_link`, etc. for implementing Union-Find with path compression

B.2.5 SegmentList

This class stores a list of segments. It has `AlignedFragments` class as a friend.

Private data members are:

- `AlignedFragments * Owner`: `AlignedFragments` owns this. `SegmentList` does not own the segments in its list; they are owned by `Owner`
- `int NumSegments`: Number of segments in list
- `Segment * HeadOfList`: Head of segment list
- `Segment * TailOfList`: Tail of segment list

Most of the functions in this class provide access to the private data members, and provide I/O.

B.2.6 AlignedFragments

This class models the segment tree of the protein conformation. It is derived from the Conformation class in the Undertaker program. Private data members are

- static IdObject ID: ID for Named Class—used for runtime type checking
- static NameToPtr *CommandTable: Command table of command-interface for reading input
- SegmentList * SegmentsAndBreaks: List of segments in the tree
- EdgeSet * TertiaryEdges: Set of edges in the tree

Some of the private functions are

- delete_edge: Delete an edge from the AlignedFragments tree. Check if the edge is a tertiary edge, find the segments the edge belongs to, delete the pointers in their edge lists, call the edge destructor and delete the edge pointer from the TertiaryEdges EdgeSet in AlignedFragments
- insert_edge: Insert an edge into the AlignedFragments tree. Check if the edge is a tertiary edge, find segments the edge belongs to, add pointers to their edge lists, add the edge pointer to the TertiaryEdges EdgeSet in AlignedFragments
- insert_segment: Create a segment and insert it at the end of the SegmentsAndBreaks list
- break_before(r): Insert a break before residue r, splitting the segment containing r

- `merge_segments`: Merges two segments only if they are adjacent. Removes the break between them, adjusting the `EdgeSets` to remove a tertiary edge on the path between the two merged segments. To maintain tree property, removes the edge specified in the optional argument, else searches and removes a tertiary edge from the path between the two segments. Transforms the two segments to the same reference frame if the flag in the argument list is set to 1. Propagates the transform to connected segments
- `DFS`: Implements recursive in-order depth-first search (DFS) of the tree
- `propagate_transform`: For every node visited during a DFS traversal, set the `SegmentTransform` of that segment such that the segment is in the same reference frame as the root of the DFS tree

Some of the public functions are

- `OK`: Does internal consistency checks on tree property
- `is_complete`: Checks to see that all residues are included in the segments
- `segment_of_residue(r)`: Find segment to which residue belongs
- `segment_of_atom(a)`: Find segment to which atom belongs
- `insert_fragments`: Insert fragments into the tree - add tertiary edges, merge segments and propagate transforms as required
- `bring_atoms_to_same_reference_frame`: Apply the transforms in each segment to the atoms of that segment, bringing all the atoms in the tree to the same frame of reference

- `traverse_subtree`: Traverse the `AlignedFragments` subtree, starting from the segment specified in the argument. Execute a function, specified in the argument list, on each segment visited. Return -1 if loop is detected
- `make_spanning_tree`: Add tertiary edges as needed to connect segments into a tree. Choose edges to minimize the total length of added edges

Bibliography

- [1] Peter K. Allen. Recitation 10: The Disjoint Set ADT. World Wide Web. <http://www.cs.columbia.edu/~allen/f98/Recitations/Recitation10/>.
- [2] C. B. Anfinsen. Principles that govern the folding of protein chains. *Science*, 181:223–230, 1973.
- [3] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: A portable linear algebra library for high-performance computers. In *Proceedings of Supercomputing '90*, pages 2–11. IEEE Computer Society Press, November 1990.
- [4] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-D point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(5):698–700, Sept 1987.
- [5] P. A. Bates and M. J. E. Sternberg. Model building by comparison at CASP3: using expert knowledge and computer automation. *Proteins: Structure, Function, and Genetics*, Supplement 3(1):47–54, 1999.
- [6] F.C. Bernstein, T. F. Koetzle, G. J. Williams, E. E. Meyer, M. D. Brice, J. R. Rodgers, O. Kennard, T. Shimanouchi, and M. Tasumi. The protein data bank: a computer-based archival file for macromolecular structures. *JMB*, 112:535–542, 1977.
- [7] Michael Bower, Fred Cohen, and Roland Dunbrack. Prediction of protein side-chain rotamers from a backbone-dependent rotamer library: A new homology modeling tool. *JMB*, 267:1268–1282, 1997.
- [8] C. Branden and J. Tooze. *Introduction to Protein Structure*. Garland Publishing, Inc., New York, USA, 1991.
- [9] Melissa Suzanne Cline. *Protein Sequence Alignment Reliability: Prediction and Measurement*. PhD thesis, University of California, Santa Cruz, June 2000.
- [10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Ronald Rivest. *Introduction To Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

- [11] J. Dongarra, R. Pozo, and D. Walker. Lapack++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings of Supercomputing '93*, pages 162–171. IEEE Computer Society Press, 1993.
- [12] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. Technical report, Argonne National Laboratory, September 1986.
- [13] R. L. Dunbrack. Comparative modeling of CASP3 targets using PSI-BLAST and SCWRL. *Proteins: Structure, Function, and Genetics*, Supplement 3(1):81–7, 1999.
- [14] David Eisenberg. Into the black of night. *Nature Structural Biology*, 4:95–97, Feb 1997.
- [15] A. Falicov and F. E. Cohen. A surface of minimum area metric for the structural comparison of proteins. *J. Mol. Biol.*, 258:871–892, 1996.
- [16] D. Fischer and D. Eisenberg. Protein fold recognition using sequence-derived predictions. *Protein Sci.*, 5(5):947–955, May 1996.
- [17] R. H. Garrett and C. M. Grisham. *Biochemistry*. Harcourt College Publishers, Texas, USA, 2nd edition, 1999.
- [18] R. H. Garrett and C. M. Grisham. Interactive Biochemistry. World Wide Web, 1999. <http://www.harcourtcollege.com/chem/biochem/GarrettGrisham/GGBYChapter.html>.
- [19] A. Godzik and J. Skolnick. Flexible algorithm for direct multiple alignment of protein structures and sequences. *CABIOS*, 10(6):587–596, 1994.
- [20] H. M. Grindley, P. J. Artymiuk, D. W. Rice, and P. Willett. Identification of tertiary structure resemblance in proteins using a maximal common subgraph isomorphism algorithm. *J. Biol.*, 229:707–721, 1993.
- [21] J.D. Hirschberg, R. Hughey, K. Karplus, and D. Speck. Kestrel: A programmable array for sequence analysis. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors*. IEEE Computer Society Press, 1996. THIS CITATION IS OUT OF DATE. USE hirschberg96.
- [22] Liisa Holm and Chris Sander. Fast and simple Monte Carlo algorithm for side-chain optimization in proteins: application to model building by homology. *Proteins: Structure, Function, and Genetics*, 14:213–223, 1991.
- [23] B. K. P. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America A (Optics and Image Science)*, 4(4):629–642, April 1987.

- [24] B. K. P. Horn, H. M. Hilden, and S. Negahdaripour. Closed-form solution of absolute orientation using orthonormal matrices. *Journal of the Optical Society of America A (Optics and Image Science)*, 5(7):1127–1135, July 1988.
- [25] Richard Hughey, Kevin Karplus, and Anders Krogh. SAM: Sequence alignment and modeling software system, version 3. Technical Report UCSC-CRL-99-11, University of California, Santa Cruz, Computer Engineering, UC Santa Cruz, CA 95064, October 1999. Available from <http://www.cse.ucsc.edu/research/compbio/sam.html>.
- [26] Richard Hughey and Anders Krogh. Hidden Markov models for sequence analysis: Extension and analysis of the basic method. *CABIOS*, 12(2):95–107, 1996. Information on obtaining SAM is available at <http://www.cse.ucsc.edu/research/compbio/sam.html>.
- [27] D. Jones and J. Thornton. Protein fold recognition. *J. Comput. Aided Mol. Des.*, 7:439–456, 1993.
- [28] W. Kabsch. A solution for the best rotation to relate two sets of vectors. *Acta Crystallographica, Section A (Crystal Physics, Diffraction, Theoretical and General Crystallography)*, A32:922–923, Sept 1976.
- [29] W. Kabsch. A discussion of the solution for the best rotation to relate two sets of vectors. *Acta Crystallographica, Section A (Crystal Physics, Diffraction, Theoretical and General Crystallography)*, A34:827–828, Sept 1978.
- [30] Kevin Karplus. “Private communication”, 1999-2000.
- [31] Kevin Karplus, Christian Barrett, Melissa Cline, Mark Diekhans, Leslie Grate, and Richard Hughey. Predicting protein structure using only sequence information. *Proteins: Structure, Function, and Genetics*, Supplement 3(1):121–125, 1999.
- [32] Kevin Karplus, Christian Barrett, and Richard Hughey. Hidden markov models for detecting remote protein homologies. *Bioinformatics*, 14(10):846–856, 1998.
- [33] K. K. Koretke, R. B. Russell, R. R. Copley, and A. N. Lupas. Fold recognition using sequence and secondary structure information. *Proteins: Structure, Function, and Genetics*, Supplement 3(1):141–8, 1999.
- [34] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. *JMB*, 235:1501–1531, February 1994.
- [35] Michael Levitt. Competitive assessment of protein fold recognition and alignment accuracy. *Proteins: Structure, Function, and Genetics*, Supplement 1(1):92–104, 1997.
- [36] A. Marchler-Bauer and S. Bryant. Measures of threading specificity and accuracy. *Proteins: Structure, Function, and Genetics*, Supplement 1(1):134–139, 1997.

- [37] A Marchler-Bauer, S. Bryant, and C. Hogue. <http://www.ncbi.nlm.nih.gov/structure/casp2/index.html>. World Wide Web, June 1998.
- [38] A. Marchler-Bauer, M. Levitt, and S. Bryant. A retrospective analysis of CASP2 threading predictions. *Proteins: Structure, Function, and Genetics*, Supplement 1(1):83–91, 1997.
- [39] C. Orengo and W. Taylor. SSAP: Sequential structure alignment program for protein structure comparison. *Methods Enzymol.*, 266:617–35, 1996.
- [40] C.A. Orengo, J.E. Bray, T. Hubbard, L. LoConte, and I. Sillitoe. Analysis and assessment of ab initio three-dimensional prediction, secondary structure, and contacts prediction. *Proteins: Structure, Function, and Genetics*, Supplement 3(1):149–170, 1999.
- [41] A. R. Ortiz, A. Kolinski, P. Rotkiewicz, B. Ilkowski, and J. Skolnick. Ab initio folding of proteins using restraints derived from evolutionary information. *Proteins: Structure, Function, and Genetics*, Supplement 3(1):177–185, 1999.
- [42] A. Panchenko, A. Marcher-Bauer, and S. Bryant. Threading with explicit models for evolutionary conservation of structure and sequence. *Proteins: Structure, Function, and Genetics*, Supplement 3(1):133–40, 1999.
- [43] Elizabeth Pennisi. Teams tackle protein prediction. *Science*, 273:426–428, 26 July 1996.
- [44] E. Huang R. Samuralla, Y. Xia and M. Levitt. Ab initio protein structure prediction using a combined hierarchical approach. *Proteins: Structure, Function, and Genetics*, Supplement 3(1):194–198, 1999.
- [45] B. Rost. Protein fold recognition by merging 1D structure prediction and sequence alignments. World Wide Web, 1996. <http://www.embl-heidelberg.de/~rost/Var/aqua/aqua.html>.
- [46] Ram Samudrala. Protein folding and protein structure prediction. Outline for ISMB Tutorial on the World Wide Web, 2000. <http://ismb2000.sdsc.edu/tutorials/samudrala.html>.
- [47] Eric W. Savage. “Private communication”, 1999-2000.
- [48] Alexander Sczyrba and Georg Fuellen. Some of the 20 amino acids. World Wide Web. <http://merlin.mbc.bcm.tmc.edu:8001/bcd/ForAll/Basics/aminos.html>.
- [49] Kim T. Simons, Rich Bonneau, Ingo Ruczinski, and David Baker. Ab initio protein structure prediction of CASP III targets using ROSETTA. *Proteins: Structure, Function, and Genetics*, Supplement 3(1):171–176, 1999.

- [50] A. P. Singh and D. L. Brutlag. Heirarchical protein structure superposition using both secondary structure and atomic representations. *ISMB*, 5:284–293, 1997.
- [51] M. Sippl. Knowledge-based potentials for proteins. *Curr. Opin. Struct. Biol.*, 5:229–235, 1995.
- [52] W. R. Taylor and C. A. Orengo. A holistic approach to protein structure comparison. *Protein Engineering*, 2:505–519, 1989.
- [53] W. R. Taylor and C. A. Orengo. Protein structure alignment. *JMB*, 208(1):1–22, 1989.
- [54] Spencer Tu. Description of a2m alignment format. World Wide Web, March 2000. <http://www.cse.ucsc.edu/research/compbio/a2m-desc.html>.