Capturing Atomic Interactions with a Graphical Framework in Computational Protein Design

by
Andrew Leaver-Fay

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2006

Approved by:

Jack Snoeyink

Fred Brooks

Brian Kuhlman

Jan Prins

Alex Tropsha

ABSTRACT
ANDREW LEAVER-FAY: Capturing Atomic Interactions with a Graphical Framework in Computational Protein Design.
(Under the direction of Jack Snoeyink)

A protein's amino acid sequence determines both its chemical and its physical structures, and together these two structures determine its function. Protein designers seek new amino acid sequences with chemical and physical structures capable of performing some function. The vast size of sequence space frustrates efforts to find useful sequences.

Protein designers model proteins on computers and search through amino acid sequence space computationally. They represent the three-dimensional structures for the sequences they examine, specifying the location of each atom, and evaluate the stability of these structures. Good structures are tightly packed but are free of collisions. Designers seek a sequence with a stable structure that meets the geometric and chemical requirements to function as desired; they frame their search as an optimization problem.

In this dissertation, I present a graphical model of the central optimization problem in protein design, the side-chain-placement problem. This model allows the formulation of a dynamic programming solution, thus connecting side-chain placement with the class of NP-complete problems for which certain instances admit polynomial time solutions. Moreover, the graphical model suggests a natural data structure for storing the energies used in design. With this data structure, I have created an extensible framework for the representation of energies during side-chain-placement optimization and have incorporated this framework into the Rosetta molecular modeling program. I present one extension that incorporates a new degree of structural variability into the optimization process. I present another extension that includes a non-pairwise decomposable energy function, the first of its kind in protein design, laying the ground-work to capture aspects of protein stability that could not previously be incorporated into the optimization of side-chain placement.

ACKNOWLEDGMENTS

I would like to thank my adviser, Jack Snoeyink, for his enthusiasm for computer science and learning, his support of my research, and his patience with my missteps. His keen intellect and academic curiosity inspired me to try harder and dig deeper when approaching the many challenges open in computational structural biology. He continues to inspire me as he straddles two disciplines, both structural biology and computational geometry, while maintaining a mastery of both. If I could have as many good ideas as Jack does, then I would have graduated a lot sooner. If I did not have Jack as an adviser, I would have graduated a lot later. Jack has greatly improved my writing, keeping the bar held high, and providing rapid and copious comments on the documents I have handed him.

I would like to thank my collaborator in Biochemistry and committee member, Brian Kuhlman, for his interest in my work, for the time spent meeting with me and answering questions, for allowing me access to Rosetta and granting me permission to modify it, and for his patience – especially when I would break his code. Brian is a giant in the field of protein design, and an inspiration. You can ask him a question that takes five minutes to state, he'll think for a few seconds, and give an illuminating one sentence answer. It has been a pleasure working with him, and I look forward to beginning a post-doc under his guidance in the coming weeks.

I would like to thank my committee members, Fred Brooks, Jan Prins and Alex Tropsha, for their support and interest in my research. I would especially like to thank Alex for founding the Bioinformatics and Computational Biology program here at UNC which inspired me to pursue my dual interests in computer science and biology. Alex introduced me to the field of computational structural biology, and showed me how rich and wonderful it is. I would like to thank Fred for his diligence and thoughtfulness in editing my writing, both in the classroom and through his role on my committee. I appreciate the genteel green felt-tip pen he uses to annotate my documents – much softer on the eyes than the red ink more commonly used. I would like to thank Jan for

his encouragement and enthusiasm, and, outside of his committee responsibilities, for his help in learning about compilers.

I would like to thank Jane and Dave Richardson from Duke University for their encouragement, interest and support in working with their program, REDUCE. They are wonderful teachers and wonderful people, and I am grateful for the opportunity to work with them. I would also like to thank Michael Word from GlaxoSmithKline, and Ian Davis and Bryan Arendall from Duke for their input and help in incorporating dynamic programming into Reduce and testing that the new version behaves as it should.

I would like to thank Hans Bodlaender from the University of Utrecht for his help in computing the treewidth of many protein design interaction graphs and for his pleasant company in Mallorca. Unfortunately he could not give me any good stories about my advisor from when they worked together at the University of Utrecht, except that Jack had taught him how to make tacos. I am sure Jack would like to thank Hans too.

I would like to thank David Baker from the University of Washington for his enthusiasm for the rotamer trie and the interaction graph and for his subtle prods to wrap up half-finished projects, usually in the form of an email sent to sixty people announcing that I will complete them shortly.

I would like to thank Loren Looger for his help in comparing dead-end elimination to dynamic programming and for the many fruitful conversations on protein design, rotamer library size, and combinatorial optimization.

I would like to thank the BioGeometry group from UNC, Duke, Stanford, and NC-A&T for providing a forum for the computational and geometric aspects of structural biology. I benefited greatly from the enthusiastic leadership of Patrice Koehl, Leonidas Guibas, Michael Levitt, Jean-Claude Latombe, Herbert Edelsbrunner, Pankaj Agarwal, and my advisor, Jack Snoeyink.

I would like to thank my wife, Lauren, for her love and support and her patience through the long and lonely journey I have been on. I'll be home to be with you soon.

I would like to thank my parents for their love and encouragement. Throughout the entirety of my schooling, they have encouraged me to keep challenging myself and to enjoy the learning process. I certainly couldn't have made it without their help. I would also like to thank my sister, Jeanie, for making me laugh and keepinng me sane.

I would like to thank the friends that I have made here at UNC: Ted Kim, Karl Gyllstrom, Brian Salomon, Brad Davis, Andrew Nashel, Yuanxin "Leo" Liu, Sharif Razzaque, Rebecca Pernell, Tetyana Schvydko, Xueyi Wang, Kimberly Noonan, David

O'Brien, Deepak Bandyopadhyay, Andrea Mantler, Jun "Luke" Huan, Vince Scheib, Deanne Sammond, Kristin Dang, Xiaozhen "Jenny" Hu, Glenn Butterfoss, Xavier Ambroggio, Doug Renfrew, Yi Liu, and Suja Thomas. Their company and conversation have meant so much to me over the years.

I would like to thank the staff in the computer science department, especially Janet Jones, Karen Thigpen, Mike Stone (who helped me replace every component on my laptop except for the CD drive, the hard drive, and the keyboard), Alan Forest, Fred Jordan, Mike Carter, Tammy Pike and Sandra Neely.

Finally, I would like to thank my funding sources, NSF and DARPA.

Contents

List of Figures

List of Tables

Glossary

amino group a chemical group containing a nitrogen, **9**

apolar an atom or chemical group lacking either a formal or a partial charge, **18**

carbonyl group a chemical group with an oxygen forming a double bond to a carbon where the carbon is not also bound to a second oxygen, **10**

DEE dead end elimination, a combinatorial optimization technique similar to branch and bound where theorems are used to prune the search space, **48**

dynamic programming an efficient optimization algorithm that stores subproblem solutions which would otherwise be repeatedly computed by a recursive (exponential) algorithm, **62**

enthalpy energy; electrostatics and gravitation both describe the enthalpy of a system, **17**

entropy the disorder of a state; the log of the number of microstates in a state ensemble, **17**

free energy the combination of enthalpy (energy) and entropy (disorder) that describes the distribution of time an entity will spend in different states as well as the distribution of a population of entities between different states at a single point in time, **16**

GMEC the global minimum energy conformation; in the side chain placement problem, the assignment of rotamers that minimizes the energy function, **38**

heavy atom an atom of any element other than hydrogen; *e.g.* nitrogen, carbon, or oxygen, **17**

hydrogen bond the energetically favorable interaction between an electron-deficient hydrogen atom and an electron-rich heavy atom where the atoms' van der Waals spheres overlap by as much as 0.4 Å, **17**

hydrophilic "water loving," dissolving well in water, **17**

hydrophobic "water fearing," dissolving poorly in water, oily, **17**

hydrophobic effect the driving force stabilizing protein structures wherein hydrophobic atoms are sequestered from water in protein cores, **18**

hydroxyl group a chemical group of the form XOH where X is bound to neither nitrogen nor another oxygen; also called an alcohol, **30**

isomer compounds or chemical groups with the same elemental compositions that differ in their chemical structure, **15**

organic acid — *a.k.a.* a carboxyl group. a chemical group with one carbon chemically bound to two oxygen atoms and to one other atom where the carbon forms a double bond to one of the two oxygen atoms, **9**

protein backbone — the atoms that form the linear tip-to-tip path along the protein, as well as the carbonyl oxygen; the chemical backbone structure stays constant under amino-acid substitution (mutation) as the backbone atoms are shared by all twenty naturally-occurring amino acids, **10**

protein side chains — the chemical groups bound to the backbone Cα atom of each residue, **10**

residue — a position in a sequence of amino acids where the position's amino-acid identity is not necessarily known. *E.g.* in design, the identity of residue 54 could be any one of the 20 amino acids, **10**

rotamer — *rota*tional iso*mer*, a conformation for an amino acid side chain described in terms of torsional (dihedral) parameters only; the other components of internal geometry – bond lengths and bond angles – are fixed, often to their ideal values, **15**

simulated annealing — a stochastic optimization technique that makes random changes according to the Metropolis criterion. Simulated annealing starts from a high temperature which is gradually lowered to bias the random changes the Metropolis criterion accepts to those that lower the energy, **52**

solvent-accessible surface the surface traced by the center of a sphere, representing a water molecule, as it rolls across the molecular (van der Waals) surface of a molecule; equivalently, the surface defined by expanding the radii of the van der Waals spheres of a molecule by 1.4 Å, the effective radius of a water molecule, **26**

Chapter 1

Introduction

Nature controls the processes of life with molecular machines called *proteins*. She communicates cell events with signal proteins, she catalyzes chemical reactions with protein *enzymes*, she propels organisms with muscle proteins, she even controls protein synthesis with other regulator proteins. Proteins represent the most efficient motors man has ever seen; the proton pump in mitochondria operates with near Carnot efficiency.

Scientists want to create new protein behavior. They want to alter existing protein/protein interactions to perturb and thereby understand biochemical pathways within cells. They want to create their own machines – machines to catalyze reactions or bind small molecules – so that they can perform tasks for which Nature has not given them tools.

Although protein design has many practical ends, it does not take its worth only from the support it lends other scientific disciplines: it is itself a worthy scientific endeavor. Protein design offers a unique way of testing theoretical models for protein energetics. Design insists the designer gather all of what is understood about protein structure, all structural preferences and energy models, and integrate that understanding to make hypotheses. The designer then asserts positive hypotheses of the form "protein X will have behavior Y." These hypotheses can be validated by synthesizing protein X and measuring behavior Y. The results of that synthesis and measurement then add to the body of knowledge surrounding structural biology. Protein design thus advances understanding of protein energetics and protein structure.

Current protein design techniques optimize protein sequences by computationally optimizing the energy of the physical structures those sequences would adopt. This optimization is NP-Complete (Pierce and Winfree, 2002). Even so, designers have had remarkable successes in designing proteins by approaching the task computationally.

They rely on stochastic optimization techniques as well as some exact techniques to perform their optimizations. Computer scientists have made and can make worthy contributions to protein design by advancing the understanding of design's combinatorial optimization problem, and by creating useful optimization algorithms.

In addition to contributing to optimization, a computer scientist can make worthy contributions by improving the computation and representation of the energy models that designers rely upon. Designers are currently hindered from incorporating certain kinds of energy models, either from their computational expenses or their representational challenges. Representation has also proven a significant hurdle in attempts to solve optimization variations that sample larger regions of protein conformation space.

In this dissertation, I detail several instances where good computer science assists protein design.

1.1 Thesis Statement

The interaction graph is an expressive model for the side-chain-placement problem, providing deeper insight into the problem's complexity than previous models, and allowing extensions that model local backbone flexibility and non-pairwise decomposable energy functions, thereby improving the designs produced by protein design software.

1.2 Main Results

In this dissertation, I define the side-chain-placement problem, the interaction graph model for this problem, and demonstrate the interaction graph's *expressiveness* by showing it is a *better model* for the problem than the model used to prove it is NP-Complete. I do this by proving the connection between the problem's complexity and a graph property called *treewidth*. This connection cannot be proven without a graphical formulation of the problem. The connection's proof is constructive and takes the form of a dynamic programming algorithm defined in terms of operations on an interaction graph. In addition to standard dynamic programming, I also present a modification to dynamic programming that runs faster and uses less memory by introducing controlled amounts of error into the energy function.

An interaction graph can be used as a data structure in protein design software. As a data structure, it captures design's memory requirements in an intuitive way. Vertices in the graph correspond to residues in the protein; edges in the graph correspond to

interactions between pairs of residues. The amount of memory an edge requires depends on properties of the vertices the edge is incident upon. The amount of memory required for the whole graph is the sum of the memory required by the vertices and edges.

Replacing the previous data structure for the representation of energies with an interaction graph in the design modeule of the Rosetta molecular modelling program (Simons et al., 1999b; Simons et al., 1999a; Kuhlman and Baker, 2000), improves Rosetta. The interaction graph replaces a monolithic data structure that was difficult to allocate and difficult to work with. Breaking down this monolithic structure reduces memory usage for rotamer-pair energies in Rosetta by 12%. Moreover, in replacing the previous data structure, I uncovered and fixed a bug in the storage of some rotamer-pair energies. Some pair energies were mistakenly left out as a direct result of the difficulty of that data structure.

The interaction graph offers a guiding principle for how new kinds of interaction energies should be stored. The graph allows extensions to cover variations on the side-chain-placement problem. In particular, I show how an interaction graph accommodates local backbone flexibility, and how an interaction graph can be used to represent non-pairwise decomposable energetics.

I present a rapid algorithm for the computation of interaction energies used for side-chain placement. This algorithm has sped the slowest step in the Rosetta's fixed-backbone side-chain-placement subroutine by a factor of 3.5. This algorithm also allows rapid computation of the interaction energies needed when designing with local backbone flexibility.

I demonstrate the *extensibility* of the interaction graph framework by extending the graph in two ways. First, I present an interaction graph for the incorporation of local backbone flexibility into the design process. This interaction graph efficiently stores the energies needed. I present the graph to store these energies, the algorithm to compute them, and a simulated annealing algorithm to optimize both side chain and backbone conformation. Local backbone flexibility yeilds an *improvement* – structures designed with local backbone flexibility have a lower energy than than those designed with a fixed backbone.

Second, I demonstrate the *extensibility* of the interaction graph framework by incorporating a non-pairwise decomposable energy function into Rosetta's design module. This function relies on a measure of two surface areas of the protein. I present a novel algorithm for the maintanence of these surface areas during the course of a simulated annealing trajectory. Simulated annealing proceeds remarkably fast considering the

amount of computation required to maintain the two surfaces. The speed is a result of the tremendous amount of data caching, made easy by the intuitive nature of the interaction graph.

This non-pairwise decomposable model yeilds an improvement – it improves recapitulation of native sequences in whole protein redesign and in protein/protein interface redesign, and produces structures that are more like native proteins than those structures produced in the function's absence.

1.3 Organization

- Chapter 2 introduces the elements of protein structure and of protein energetics. It introduces the side-chain-placement problem and reviews the literature in protein design. Finally, it describes and compares the computational techniques that structural biologists have brought to bear on the problem.

- Chapter 3 outlines the three challenges faced in protein design, tying together the literature on successful and less-than-successful protein designs. These challenges are: the difficulty in finding a perfect fit for side chains into the core of a protein, which necessitates backbone conformational sampling; the difficulty of including non-pairwise decomposable energy functions into design software; and the difficulty inherent in the side-chain-placement problem. After outlining these three challenges, the chapter relates the work in this thesis toward overcoming these challenges.

- Chapter 4 introduces the interaction graph model of the side-chain-placement problem, hypergraphs as a generalization of graphs, and treewidth as a graph characteristic. From these definitions, the chapter proves that an individual instance of the side-chain-placement problem can be solved with dynamic programming in time exponential in the treewidth of its interaction graph. If the treewidth of an instance's interaction graph can be bounded by a constant, then the dynamic programming algorithm runs in polynomial time for that instance. This proof connects side-chain placement with the class of NP-hard problems for which certain problem instances admit polynomial time solutions. The chapter also defines a variation on dynamic programming, adaptive dynamic programming (ADP). Adaptive dynamic programming solves the side-chain-placement problem using less memory and less time than dynamic programming at the ex-

pense of introducing controlled amounts of error. This chapter concludes with a description of a similar problem to the side-chain-placement problem, the hydrogen placement problem. Dynamic programming on an interaction graph solves this problem efficiently. I have incorporated the dynamic programming algorithm into an existing piece of software for hydrogen placement, REDUCE, originally written by Michael Word in Dave and Jane Richardon's laboratory at Duke (Word et al., 1999b). The new version of REDUCE is now being distributed.

- Chapter 5 describes the how the interaction graph can serve as a data structure. The interaction graph has been part of Rosetta and in use for nearly a year. This chapter describes the previous data structure in Rosetta that stored rotamer-pair energies before the interaction graph replaced it; this data structure, though difficult to work with, was both efficient in amount of memory it used and efficient in its speed of energy retrieval during simulated annealing. The interaction graph, reduces memory use by $\sim$12%, and achieves the same speed during simulated annealing. This chapter also describes the class hierarchy for the object-oriented framework added to Rosetta.

- Chapter 6 presents a fast algorithm for rotamer-pair-energy computation that serves both locally-flexible-backbone side-chain placement, and fixed-backbone side-chain placement. This algorithm is now part of Rosetta, where it has been in use for fixed-backbone side-chain placement for over a year, speeding both protein design (3.5$\times$ faster) and protein folding (2$\times$ faster).

- Chapter 7 describes the incorporation of local backbone flexibility into the side-chain-placement problem. There are three aspects of this system described: the calculation of the energies needed when the backbone moves, the storage of those energies in an interaction graph, and the optimization of backbone and rotamer placement in simulated annealing. First, a simple extension to the fast rotamer-pair-energy computation algorithm from Chapter 6 efficiently computes the energies needed in flexible-backbone design. Second, an extension to the interaction graph framework stores those energies. Third, a new version of simulated annealing interfaces the interaction graph to perform new kinds of rotamer substitutions not used by simulated annealing in fixed-backbone design.

- Chapter 8 describes an efficient algorithm to update the solvent-accessible surface during side-chain placement. Unlike other aspects of protein energies, the degree

of burial from solvent cannot be decomposed into the sum of pair interactions. Any energy function that depends upon surface area will be non-pairwise decomposable. I describe how to integrate surface area computations into design using an interaction graph. Optimizing the surface-area-based energy function inside side-chain placement produces designs with a better surface-area-based energy.

- Chapter 9, concludes the thesis, discusses limitations, and notes several interesting directions for future work.

1.4 Assumed Background

This thesis describes protein chemical structure and protein physical structure in significant detail. It describes in significant detail protein energetics and protein stability and the approaches biophysicists have taken in modeling energies and stabilities. It explains the theory behind the dynamic programming algorithm presented in Chapter 4 by delving into an area of graph theory: this thesis defines the concepts of *treewith* and *tree decompositions* as well as *partial* k-*trees* before using these definitions to analyze the behavior of dynamic programming.

The reader should have a firm understanding of complexity analysis; of NP-Completeness and of Big-O notation. The reader should be aware generally of techniques for combinatorial optimization; the thesis describes in detail dynamic programming and a stochastic technique called *simulated annealing*, but does not describe branch-and-bound or integer linear programming, for instance. The thesis provides citations to relevant publications in protein design where these alternative optimization techniques are used, and the curious reader is encouraged to follow these citations. The reader should also have a general sense for graph theory; the thesis contains definitions for the parts of graph theory that are somewhat obscure (e.g. a k-tree), but does not contain definitions for the parts of graph theory that are common (e.g. a k-clique). I would recommend *Pearls in Graph Theory* by Hartsfield and Ringel (1990) for the reader that is unfamiliar with graph theory as well as the reader who wants to better learn graph theory.

The thesis does not presume the reader has a deep understanding of protein chemical or physical structure, or of protein energetics. It does presume the reader has some familiarity with organic chemistry, for example, the reader should be able to identify the chemical structures from the skeletal formulas in Figures 2.2 and 2.3. A tutorial on

reading and drawing chemical structures can be found at `http://www.chemguide.co.uk/basicorg/conventions/draw.html`. There are several pieces of chemical nomenclature the reader should know: for example, an oxygen that forms a double bond with a carbon is a *carbonyl* oxygen, an oxygen that forms a single bond with a carbon and a single bond with a hydrogen is a *hydroxyl* oxygen. The reader may find the glossary helpful in refreshing the chemical jargon contained in this thesis. In addition a useful tutorial on organic nomenclature can be found at `http://chemistry.boisestate.edu/rbanks/organic/nomenclature/orgaincnomenclature1.htm`.

The thesis also presumes that the reader has previously encountered the ideas of *energy* and *free energy*, specifically *Gibb's free energy*. The thesis describes the concepts of *enthalpy* and *entropy*, but at a level appropriate for a review and not as an introduction to these concepts. The reader wishing to know more about free energy should consult a textbook for any introductory undergraduate course in chemistry (either organic or inorganic chemistry). *Organic Chemistry* by Maitland Jones (1998) is a good text.

Finally, this thesis presumes that the reader has encountered electron orbital hybridization theory and can easily distinguish the geometry of sp^2 hybridized orbitals and sp^3 hybridized orbitals. Again, Maitland Jones' textbook would be a good place to start if the reader is unfamiliar with these concepts.

Chapter 2

Protein Structure and the Side-Chain-Placement Problem

Structural biochemists have discovered a host of problems to pique the interest of any computer scientist. A high-level formulation of these problems that abstracts away the biochemistry can be understood by anyone with a basic knowledge of algorithms. At one level, a computer scientist can contribute to structural biochemistry by solving problems that the structural biochemists present. On a deeper level, a computer scientist that understands the biochemistry can unearth and solve new questions – possibly ones that the biochemists had initially wanted to ask but held back because there seemed no plausible solution. This thesis describes a combination of these two contributions; places where good algorithms have improved the speed or accuracy of the software biologists initially built, and places where an understanding of the underlying biochemistry has lead to the formulation of new problems. In order to provide a fuller understanding of the contributions, I begin with a detailed description of structural biochemistry.

2.1 Protein Chemical Structure

Chemically, proteins are composed of *amino acids*; the tail of one amino acid is linked to the head of the next amino acid to create a one dimensional chain, much like the Snap-Lock beads of my youth (Figure 2.1a). Different amino acids link to form proteins just as snap-lock beads link to form colorful chains with interesting shapes (Figure 2.1b).

Amino acids are made up of three parts: an amino group (NH_3^+), an organic acid ($C(=O)O^-$) and a side chain. All amino acids share the amino and acid groups; amino acids differ in the composition of their side chains. Nature uses only twenty different

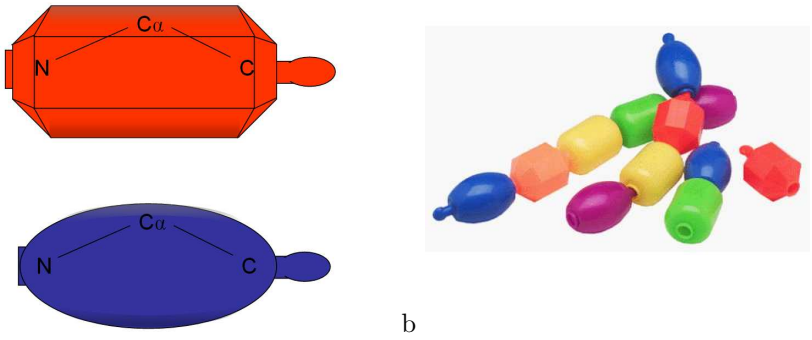


Figure 2.1: *Snap-Lock Beads.* a) Each snap-lock bead has a hole at one end (left) and a knob on the other (right). b) Chains of snap-lock beads are formed by connecting knobs and holes. Photo from Amazon.com

amino acids.

Amino acids are chemically linked head-to-tail to form chains. The chemical reaction that binds two amino acids together links the carbon from the acid of one amino acid to the nitrogen of the next amino acid. One of the two oxygens from the acid is converted into water and released as a product of the reaction. The chemical bond between the carbonyl carbon and the amide nitrogen is called a *peptide bond*. The stringing together of amino acids forms a repeating triad: -N-C-C-. From one tip of a protein to the other, there is a path that follows this N-C-C repeat. For this reason, these three atoms that make up the tip-to-tip path are often referred to as the *backbone* of the protein. The carbonyl oxygen is often included as a member of the backbone, though, it does not lie on the tip-to-tip path. The ambiguity over whether the carbonyl oxygen belongs in the backbone depends on which definition is meant by the word "backbone:" it could refer to the tip-to-tip path along the protein, or it can refer to everything that is not part of the side chain. In the rest of the document, I use "backbone" to mean all non-side-chain atoms.

The *side chains* attach to the first carbon in the N-C-C repeat. This carbon is one chemical bond away from the carbonyl carbon so it is said to be α to the carbonyl group. It is therefore called Cα. The rest of the atoms branching from Cα are similarly named with increasing Greek letters. The carbon chemically bound to Cα is called Cβ as it is β to the backbone carbonyl carbon. Of the twenty amino acids (Figure 2.3), all but one, glycine, have a Cβ atom. Gylcine's side chain is a single hydrogen atom bound to Cα.

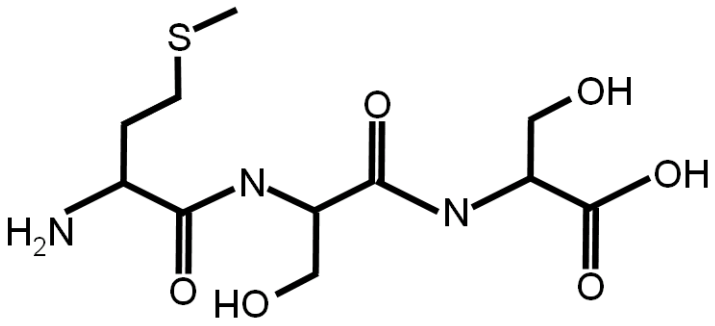


Figure 2.2: *A Short Protein.* Three amino acids, methionine, serine and serine again, form this short protein. The one dimensional sequence for this protein is MSS. Note that the first residue has an NH_2 group, making it the N-terminus, while the last residue has an acid group, making it the C-terminus. These figures are skeletal formulas; for a review of skeletal formulas, see `http://www.chemguide.co.uk/basicorg/conventions/draw.html`

The sequential list of the amino acids that compose a protein captures completely the protein's chemical structure. The convention is to list the amino acids starting from the N- (or amino-) terminus and going to the C- (or carboxy-) terminus. Each position in a protein's sequence is referred to as a *residue*. It is more proper to talk about the fifty-fourth residue of a protein than to talk about the fifty-fourth amino acid. In Figure 2.2 for example, the N-terminus is residue one and is a methionine amino acid. The residue it is bound to is residue two and is a serine amino acid. The one-dimensional sequence of the amino acids is called the *primary structure*. A protein's chemical structure captures only a small part of its three-dimensional structure. Biologists consider four levels of protein structure. The three levels I have not yet described all pertain to three-dimensional structure.

2.2 Protein Stability

Naturally occurring proteins adopt a compact geometric configuration when placed in water. This compact geometric configuration is called a protein's *fold*. When folded, a protein will usually have a handful of chemical groups positioned in such a way that the protein can accomplish some task. For instance, the "catalytic triad" of serine proteases is made up of the side-chain atoms from only three residues; the rest of the protein's

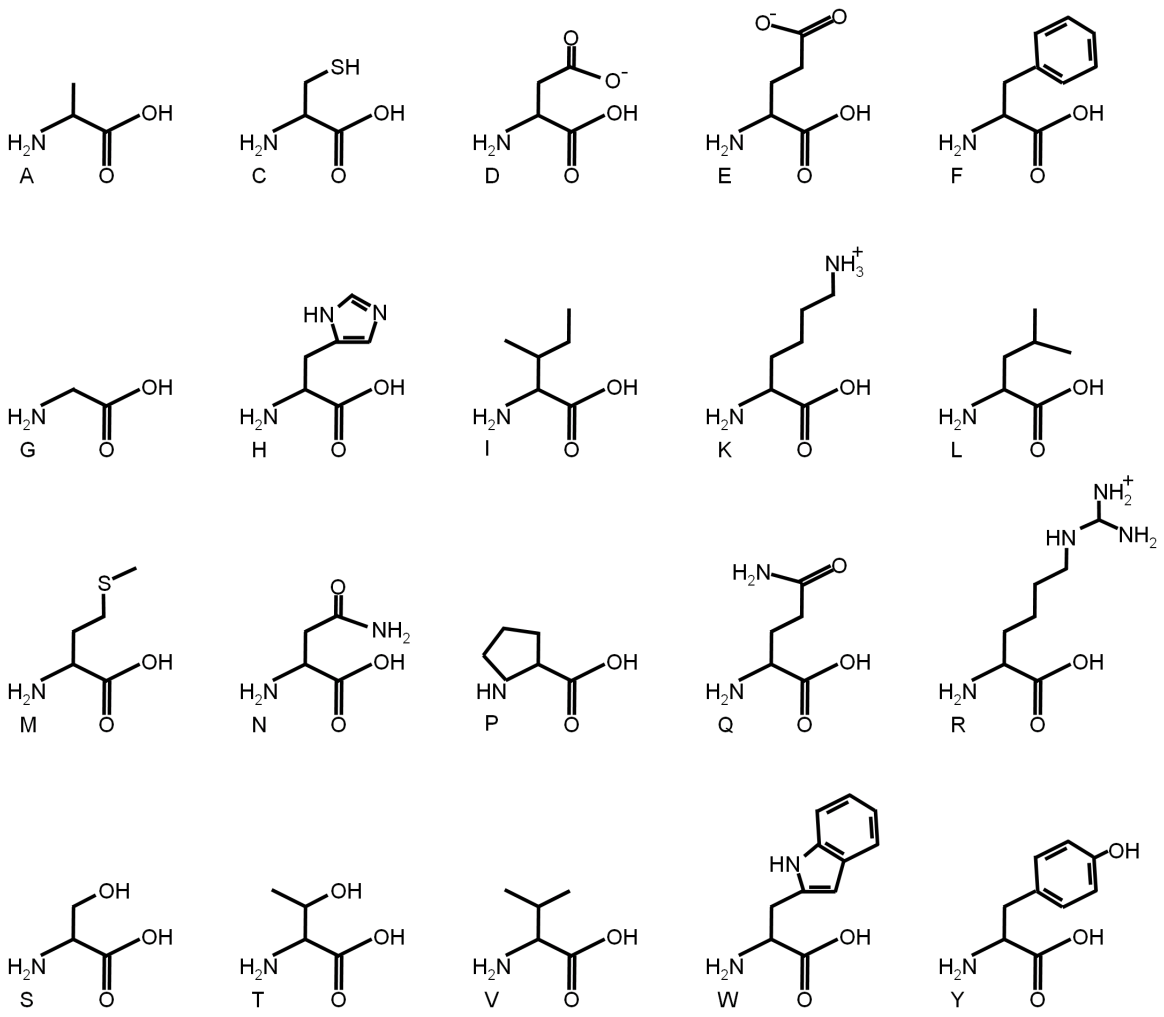


Figure 2.3: The twenty amino acids found in nature. Amino acids are often referred to by three letter or one letter codes. From left to right, with both three letter and one letter codes: *row 1* Alanine (ALA, A), Cysteine (CYS, C), Aspartic Acid (ASP, D), Glutamic Acid (GLU, E), Phenylalanine (PHE, F), *row 2* Glycine (GLY, G), Histidine (HIS, H), Isoleucine (ILE, I), Lysine (LYS, K), Leucine (LEU, L), *row 3* Methionine (MET, M), Asparagine (ASN, N), Proline (PRO, P), Glutamine (GLN, Q), Arginine (ARG, R), *row 4* Serine (SER, S), Threonine (THR, T), Valine (VAL, V), Tryptophan (TRP, W), and Tyrosine (TYR, Y).

structure can be viewed as providing the structure that holds these residues in just the right relative position and orientation. Biochemists wishing to fully understand protein function must understand protein structure. A central, open problem in computational structural biology is to predict a protein's fold given its sequence. This is called the *protein folding problem*.

Biochemists are able to indirectly observe protein structure. In a multi-step process called *X-ray crystallography*, biochemists 1) form protein crystals, 2) diffract X-rays off of the crystals and obtain the amplitudes of the diffracted X-rays, 3) transform diffraction data into electron density, and 4) thread the chemical structure through the density. In fact, the data gathered from the second stage is incomplete, and so stages three and four are often cycled between. The end product is a model containing Euclidean coordinates for the nucleus of each atom visible in the electron density. The resolution of most X-ray-determined structures is on the order of 2 Å. The highest resolution structures have a resolution of 0.7 Å.

In another multi-step process called *nuclear magnetic resonance (NMR) spectroscopy*, biochemists 1) measure distances between all protein hydrogen atoms, 2) assign all the distances to specific atoms, and 3) compute a set of protein structures that satisfy the distance constraints. The resolution of NMR-determined structures is on the order of 4 Å. Both techniques require considerable labor and time. Nonetheless, biochemists are willing to invest time and energy in both techniques. Protein structure is that important.

X-ray crystallography may take six months from end to end. It can take months to synthesize enough protein. It can take months to crystallize the protein once there is enough to be crystallized. The X-ray diffraction step proceeds comparatively rapidly (once access to busy synchrotrons or other X-ray sources has been obtained), but the diffraction data (which describes the amplitudes of diffracted x-rays) can take months to transform (where the transformation process requires both amplitudes and phases). Model fitting following transformation is no small problem. Indeed, the absence of phase data from the diffraction step means that phases are unknown, biochemists can justifiably change the phases used in the first transformation to result in different electron densities. The stages 3 and 4 are iteratively combined: a fitted model is used to refine the phases and then to fit new models.

Preparing for the first stage in NMR takes less time than the first stage on X-ray crystallography since less protein is required to measure hydrogen distances than is needed to form crystals. However, the distances that come back from NMR are not

connected to particular atoms; the second step in NMR structure determination is to figure out a correspondence between the hydrogen atoms in the protein and the distances that have been measured. This assignment problem has been shown to be NP-Complete (Xu et al., 2002). Following assignment, the distances provide only a basic description of a protein's structure. Distance geometry techniques are used to invert the incomplete distance matrix to produce a three-dimensional structure (Pardi et al., 1988; Crippen and Havel, 1988). What results is not a single structure, but an ensemble of structures all of which are consistent with the distance data. The ensemble is sometimes argued to reflect motion within the protein's structure; the fewer the distance constraints on a region of the protein, the more models will be consistent with the data. On the other hand, it is not easy to argue that all the models consistent with the data reflect actual conformations the protein adopts.

Three sets of parameters describe the internal three-dimensional geometry of a protein: bond lengths, bond angles, and bond dihedrals. Computational structural biochemists who model proteins often restrict their models to idealized bond geometries; that is they fix the bond lengths and bond angles. Biochemists restrict the source of conformational flexibility to bond dihedrals because the energetic penalties for stretching bonds and flexing bond angles far exceed the penalty in rotating dihedrals. As I discuss in detail later, the folded structure of a protein is only marginally stable; it is unreasonable to suspect that a folded protein strains bond lengths and angles at all. Restricting the conformation space available to models simplifies them without sacrificing their quality. The bond lengths and angles used come from empirical observations of small molecule crystal structures (Engh and Huber, 1991).

The backbone for each residue contributes two degrees of freedom from two dihedral angles, ϕ and ψ (Figure 2.4a). The ϕ and ψ dihedrals are measured from C_i-N_{i+1}-$C\alpha_{i+1}$-C_{i+1} and from N_i-$C\alpha_i$-C_i-N_{i+1} where the subscripts on these atoms denote which residues they come from. The four atoms $C\alpha_i$-C_i-N_{i+1}-$C\alpha_{i+1}$ would define a third dihedral for conformational flexibility; however, the structure of the molecular orbitals along the peptide bond keeps these four atoms planar. (The backbone carbonyl oxygen forms a double bond to the carbonyl carbon; this double bond requires both the backbone oxygen and the backbone carbonyl carbon adopt an sp^2 hybridization state – in an sp^2 hybridization an atom has three of its sp^2 orbitals in the plane, and has its one p orbital perpendicular to the plane. Because the carbonyl carbon is sp^2, the set of atoms $C\alpha_i$, C_i, O_i and N_{i+1} all lie in the same plane. The amide nitrogen, N_{i+1}, also adopts an sp^2 character so that its p orbital overlaps well with the p orbital of the

carbonyl carbon. The nitrogen's sp^2 character places its sp^2 orbitals in the same plane as the other four atoms, so that its bonded Cα, that is Cα_{i+1}, lies in this same plane. All together, the set of atoms Cα_i, C$_i$, O$_i$, N$_{i+1}$, and Cα_{i+1} all lie in the plane. For more on the geometry of electron orbitals and of orbital hybridization, consult *Organic Chemistry* by Jones (1998)) Thus this third dihedral, ω, is either 0° or 180°, with 90% of observed ω dihedrals at 180° (Branden and Tooze, 1999). Most structural biologists restrict their models to adopt planar ω angles, eliminating ω as a degree of freedom.

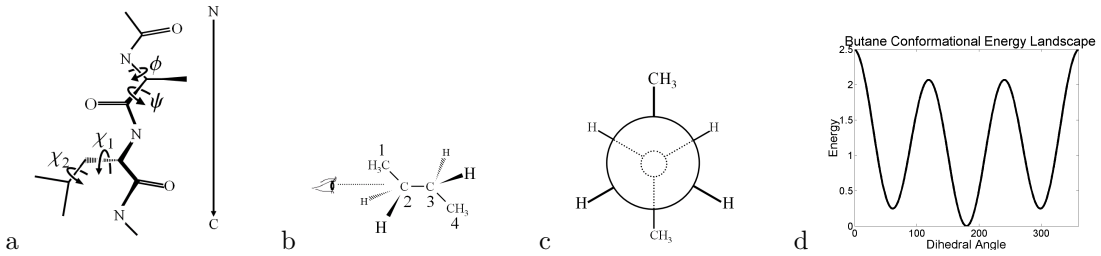


Figure 2.4: Protein Geometry. *Protein structural flexibility results from rotatable backbone (ϕ and ψ) and side chain (χ_1, $\chi_2 \ldots$) dihedrals (a). Side-chain rotamers can be understood through an example molecule, butane. Looking down the bond connecting carbon-2 and carbon-3 in butane (b) creates a Newman projection (c). The dihedral angle can be measured as the angle between the two methyl groups in the Newman projection. Butane's low energy conformations stagger the chemical groups bound to carbon-2 and carbon-3, staggering carbon-2's groups relative to the groups on carbon-3. (d).*

Side-chain conformations are also expressed in terms of dihedral angles, as biochemists also keep side-chain bond lengths and angles fixed. Leading away from the backbone, the side-chain dihedrals are enumerated as χ_1, $\chi_2 \ldots$. The most limber side chains, lysine and arginine, have four flexible dihedrals. Biochemists do not give side-chain dihedrals complete freedom, but instead limit the dihedrals to discrete values. Combining the chemical idea of *isomers* – compounds or chemical groups with the same elemental compositions that differ in their chemical structure – and the fact that side-chain conformers are superimposable by bond dihedral rotation, scientists refer to discrete side-chain conformations as *rotamers*, rotational isomers.

Structural biologists started analyzing observed side-chain conformations as soon as protein crystal structures appeared (Janin et al., 1978; Bhat et al., 1979) and their conclusions further helped x-ray crystallographers fit protein models into observed electron density. Crystallographers, among others, now use lists of the χ dihedrals most commonly observed for protein side chains – they call these lists *rotamer libraries* as they catalog side-chain-conformation space. The first rotamer library was presented by

Ponder and Richards [1987] when they observed a sharpening of the distributions of χ dihedral angles present in the newly deposited crystal structures which had higher resolutions than earlier structures. The early crystal structures were not sufficiently resolved to show dihedral value preferences, however as crystallography improved and the resolution increased, the dihedral angle preference became apparent. The general angle distribution shows clusters in three regions, at 60, 180 and -60 degrees. These values correspond to the staggered conformations for butane which lie at the bottom of shallow energy wells (Figure 2.4d). Ponder and Richards reported dihedral values at the bottom of these observed shallow energy wells and estimates of the ranges of these wells.

Current rotamer libraries derive from high-resolution crystal structures (Dunbrack and Karplus, 1993; Dunbrack and Cohen, 1997; Bower et al., 1997; Lovell et al., 2000; Dunbrack, 2002) and from quantum mechanical calculations (Butterfoss and Hermans, 2003). The sharpening of the dihedral angle distributions that Ponder and Richards initially observed continued as crystallographic resolution improved; however, that sharpening has stopped. The standard deviation in dihedral angles that Lovell reported in 2000 were much smaller than the standard deviations reported by Ponder and Richards. However, the most highly resolved crystal structures do contain dihedral angles in a range surrounding the energy well bottoms and not solely at their bottoms. The presence of these off-optimal dihedral values suggests that the energetic destabilization of off-optimal dihedral angles can be compensated for by improving other portions of the protein's energy. Computational structural biochemists should therefore sample the dihedral wells at points other than their bottoms to accurately model side-chain conformational flexibility. Indeed, protein designers have had the most success when they sample dihedral space at sub-optimal points (De Maeyer et al., 1997; Gordon and Mayo, 1998; Looger and Hellinga, 2001). As could be expected, the more rotamer samples, the more difficult the computational problems become.

Biochemists say that proteins in solvent adopt two conformations or states: an unfolded state (an ensemble of extended, non-compact structures) and a folded, highly ordered, compact state. At any point in time, a certain percentage of a protein population lies in the unfolded state, the rest lies in the folded state. A *free energy* diagram for a protein's conformation would include a low-energy unfolded state, a set of higher-energy partially-folded states, and a low-energy folded state (Figure 2.5). The difference in free energy (also called the Gibb's free energy) between the folded and unfolded states represents the stability of the protein. If one were able to know the absolute free en-

ergy of a folded structure, one would still need to know the absolute free energy of the unfolded state in order to measure the stability of the protein. The distinction between absolute free energy of a conformation (G) and the stability of a conformation (ΔG) is crucial for protein design, as designers seek the most stable structure for some purpose and not simply the structure with the lowest free energy. Biophysicists have come up with a set of energy models that can be used to compute the energy of a folded protein structure, (described in greater detail in Section 2.4); to evaluate the stability of a folded protein structure requires an idea of the protein's energy in its unfolded state. It is insufficient to examine the interactions between protein atoms alone to determine the protein's stability.

The free energy of a state is always a negative number, and nature prefers low free energy. This can be summed up with a variation on the old stock market saying: "buy low, sell lower." In this dissertation, all optimizations of free energy are of minimizing free energy. The free energy of a state, G depends on the temperature (T), the enthalpy (H), and the entropy (S)

$$G = H - T * S.$$

The enthalpy of a state is what is most commonly called its *energy* – a distinct idea from free energy, to be sure. Gravity describes the enthalpy of two bodies: when two bodies are brought nearer together, they have a lower enthalpy. However, enthalpy is only half of free energy's story. The entropy of the state is a function of the disorder in that state, or how many micro-states that the state ensemble consists of. The more disorder, the higher the entropy; the less disorder, the lower the entropy. The effect of entropy on the free energy of a state increases as temperature increases.

A precise understanding of the forces involved in protein stability is required to predict a protein's fold, or to design novel proteins. While a precise understanding remains elusive, biochemists have gleaned several general rules of folded proteins. One rule is that the side chains of hydrophobic amino acids (PHE, ILE, LEU, MET, VAL, TRP) are buried from solvent in the core of a protein, while the remaining hydrophilic amino acid side chains are located towards the surface of the protein. The burial of hydrophobic amino acids, called the *hydrophobic effect* (Kauzman, 1959), is commonly thought to play the greatest role in stabilizing a protein's fold. The hydrophobic effect on the molecular level stems from preferential orientation of water atoms that surround hydrophobic amino acids (Dill, 1990). This deserves explanation.

Water molecules form *hydrogen bonds* with each other and with solvated proteins.

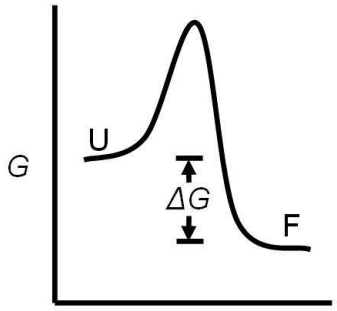


Figure 2.5: *Folding coordinate diagram* The X-axis represents the "folding coordinate" where the unfolded state (U) is on the left and the folded state (F) is on the right. The Y-axis represents the absolute Gibb's free energy of each point in the folding coordinate. The stability of a protein is the difference in free energy between its unfolded and folded states, ΔG. In protein design, the optimization problem is often one where the most stable structure is desired, as opposed to the structure with the lowest free energy. Designers must take into account the free energy of the unfolded state (which is never explicitly modeled) as well as the free energy of the folded state.

A hydrogen bond forms when any electron-rich *heavy atom* – where a heavy atom is any atom that is not hydrogen – shares its electrons with an electron-deficient hydrogen atom. The heavy atom to which the hydrogen is bound *donates* its hydrogen to the electron rich heavy atom that *accepts* it. The donor in a hydrogen bond is the heavy atom chemically bound to the hydrogen; the acceptor is the electron-rich heavy atom that interacts with the hydrogen.

The electron-deficient hydrogen atoms in proteins are those chemically bound to oxygen, nitrogen, or sulfur. Hydrogen atoms bound to carbon atoms cannot be donated to form hydrogen bonds. Electron-rich heavy atoms in proteins are oxygen, nitrogen, and sulfur. Carbon atoms cannot accept hydrogen bonds. The majority of the atoms in proteins are either carbon atoms or hydrogen atoms bound to carbon.

Dill provides an atom-level interpretation of the hydrophobic effect as follows (Dill, 1990). A water molecule that sits next to an *apolar* – neither electron rich nor electron deficient – solute atom will pay an enthalpic cost if it orients itself in a direction that points one of its hydrogen atoms toward the solute: if the hydrogen atom points toward bulk solvent, then it can form a hydrogen bond, whereas if the hydrogen atom points toward the solute atom, then it cannot form a hydrogen bond. The solute-pointing orientation of the hydrogen atom is less stable. Therefore, water prefers to

orient its hydrogen atoms toward bulk solvent. However, this orientational restriction has an entropic cost. Thus, there is a fight between entropy and enthalpy. At low temperatures, enthalpy wins and the water suffers a loss of conformational entropy: the water cannot orient itself in as many positions as it would if not next to the solute atom. At high temperatures, entropy wins and the water suffers a loss of enthalpy: the water will give in to entropy and end up pointing its hydrogen atoms at the solute atom at the energetic expense of not forming a hydrogen bond. At body temperature, enthalpy wins, and entropy loses.

Why does the hydrophobic effect stabilize a protein? The unfolded state of the protein exposes more hydrophobic surface area to water than the folded state. The more surface area a structure exposes to water, the higher the free energy of that structure. The folded state is stable because the unfolded state is at a higher energy. For example, if the free energy of the unfolded state, G_u in Figure 2.5 were to increase from where it is drawn, and the free energy of the folded state G_f were to stay the same, then the $\Delta G = G_f - G_u$ would decrease, meaning the protein would be more stable.

Entropy's contribution toward the destabilization of the unfolded state can be observed through "cold-unfolding." As temperature decreases, the effect of entropy of the free energy of the unfolded state decreases; the unfolded state is at lower free energy at lower temperatures. The decrease in the energy of the unfolded state creates a decrease in the stability of the protein; below certain temperatures, proteins unfold. At high temperatures, the protein will also unfold, however, this is due not to water, but to the low entropy of the folded state in comparison to the entropy of the unfolded state, and the greater importance of entropy at higher temperatures.

While solvent interactions drive hydrophobic amino acids to the core of the protein, they often keep hydrophilic amino acids near the surface. A hydrogen bonding group in the protein will form a hydrogen bond with water when the protein is in the unfolded state. If the folded state of the protein buries the group away from water, it pays an energetic cost to break the hydrogen bond of that group. The folded state can compensate for this expense by forming an intra-molecular hydrogen bond between the now-buried hydrogen bonding group and another hydrogen bonding group. Indeed early surveys of protein structures suggested that 90 percent of buried hydrogen bonding groups participated in intra-molecular hydrogen bonds (Baker and Hubbard, 1986). Rose recently re-examined the backbone hydrogen bonding groups from a large set of protein structures and found that the supposed ten percent of buried, unsatisfied

hydrogen bonding groups were in fact accessible to solvent (Fleming and Rose, 2005). "The difference between ∼90% and ∼100%," he argues, "is the difference between a trend and a rule." Buried hydrogen bonding groups form intra-molecular hydrogen bonds.

Folded proteins are also subject to electrostatic forces. Coulomb's law expresses the interaction energy of two charged particles i and j with charges q_i and q_j as

$$\mathcal{E}_{electrostatics}(i,j) = \frac{q_i q_j}{\epsilon_1 d_{ij}} \tag{2.1}$$

where d_{ij} is the distance between the two, and ϵ_1 is the dielectric constant. The function describing Coulomb's law has a peculiar shape. The interaction that atom i has with nearby atom j has more influence on i's energy than the interaction it has with atom k further away. However, because the drop off is a function of $\frac{1}{d}$, the interactions that atom i has with all atoms at distance d_1 contribute less than the interactions i has with all atoms at distance $d_2 > d_1$, since there are more atoms at distance d_2. This can be understood by looking at the volume of a spherical shell of radius d and thickness Δ, which is $4\pi d^2\Delta$, and the influence contributed by all atoms inside this shell (assuming a uniform density of ρ atoms per unit volume, and that each atom j interacting with i at this distance has a charge of q_j):

$$4\pi d^2\Delta\rho\epsilon_1\frac{q_i * q_j}{d} = 4\pi\Delta\rho\epsilon_1 q_i q_j d.$$

The influence contributed from each shell increases with distance! In general, any energy function that drops off according to d^{-k} is short ranged if $k > 3$ and long-ranged if $k < 3$ (Bromberg and Dill, 2003). At $k = 2$, all spherical shells contribute the same toward i's energy. At $k = 3$, the contribution for the shell at radius d drops off as a function of $\frac{1}{d}$ – this drop off is slow enough that such energy functions are described as long ranged. (The sum $\sum_{x=1}^{n} x^{-1}$ does not converge as n goes to infinity – the sum $\sum_{x=1}^{n} x^{-2}$ on the other hand does converge.)

The degree to which electrostatic interactions contribute to protein stability is debated. The dielectric constants vary inside the protein core, and the presence of polarizable elements (electron orbitals) could shorten the reach of an electrostatic potential. One one hand, statistical analysis of protein structures suggest that protein side chains orient themselves in ways predicted by Coulomb's law (Bryant and Lawrence, 1991). On the other hand, recent successes in protein folding where electrostatic terms have

been left out entirely suggest long range electrostatics are unnecessary (Bradley et al., 2005). At the very least, close interactions between oppositely charged side chains, (*e.g.* LYS^+ and GLN^-), are often observed near the surface of proteins. These interactions are called *salt bridges* as they resemble the close interactions of oppositely charged ions in salts.

Finally, proteins are also stabilized by London dispersion interactions, which are also called van der Waals interactions. Uncharged, freely rotating dipoles attract each other with a function proportional to d_{ij}^{-6} (for the proof, see [Bromberg and Dill, 2003]). The Lennard-Jones 6-12 potential approximates the van der Waals interactions for atoms i and j with radii R_i and R_j by

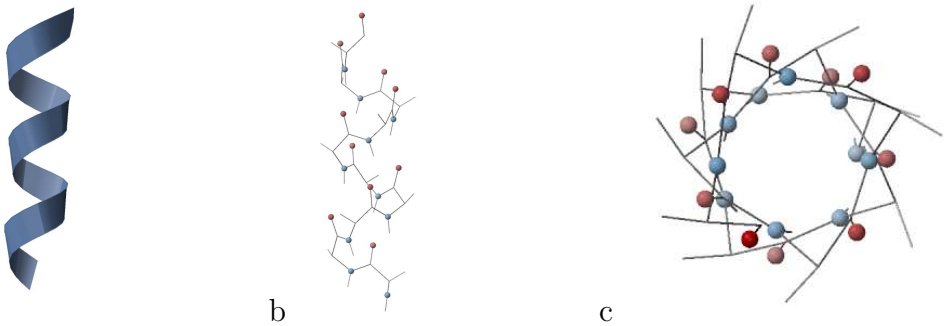
$$\mathcal{E}_{vdW}(i,j) = \epsilon\left(\left(\frac{R_i + R_j}{d_{ij}}\right)^{12} - \left(\frac{R_i + R_j}{d_{ij}}\right)^{6}\right) \tag{2.2}$$

where ϵ is the energy at the bottom of the contact well. The d_{ij}^{-6} term captures the attractive forces of dipole interactions from statistical thermodynamics; the d_{ij}^{-12} term is meant to capture the repulsive forces preventing non-bonded atoms from colliding. The principle reason the d_{ij}^{-12} term is used is its computational convenience as the square of the d_{ij}^{-6} term.

2.3 Hierarchy of Protein Structure

Biochemists talk about protein structure on four levels: primary, secondary, tertiary and quaternary. Primary structure, as mentioned earlier, is the linear sequence of amino acids that compose the protein. Primary structure can be represented as a string, where each amino acid in the protein is represented by its one-letter code (Figure 2.3).

Secondary structure is characterized by the presence of intra-molecular hydrogen bonds between backbone elements. Secondary structure includes α-helices, β-sheets, and tight turns. In all three forms of secondary structure the backbone nitrogen donates its bound hydrogen to some other residue's backbone-carbonyl oxygen. Except for proline, whose backbone nitrogen does not have a hydrogen to donate, any amino acid can be in a helix or a sheet. This means that a random mutation to a helix or a sheet residue will likely preserve the residue's ability to form that secondary structural element. The key structural features of secondary structure depend on the backbone and not on the side chains. The seeming independence of structure from sequence is part of why α-helices and β-sheets gets their low rank in the hierarchy of protein structure;



a b c

Figure 2.6: Ubiquitin's Alpha Helix. a) Cartoon and b) ball and stick representations of the α-helix in ubiquitin. The carbonyl oxygens atoms (red) point in the direction towards the C-terminus (up). The nitrogen atoms (blue) point their bound hydrogen atoms towards the N-terminus (down). In c) the outward splaying of the carbonyl oxygens is apparent when viewed down the center of the helix.

secondary structure does not suggest much about side-chain conformation. The other part of α-helices' and β-sheets' low rank is simply their commonality. Helices and sheets are found everywhere; complex structures are built from various arrangements of secondary structures.

In an alpha helix (Figure 2.3), the backbone is wrapped in a tight coil. The carbonyl oxygen on residue i forms a hydrogen bond with the backbone nitrogen on residue $i+3$. The hydrogen bond between i and $i+3$ positions the carbonyl oxygen on residue $i+1$ so that it can form a hydrogen bond with residue $i+4$. The helices are almost always right-handed – as Jane Richardson would say, as you climb the spiral staircase of the helix, you use your right hand to hold the rail. Left-handed helices have been observed in protein structures, but they are very short – 4 or 5 residues. For a residue in the middle of the helix – that is, not near the ends of the helix – both the residue's carbonyl oxygen and the nitrogen groups form hydrogen bonds with the backbone nitrogens and oxygens of other helix residues. For a residue at either end of the helix, one of its backbone hydrogen bonding groups is left to find some non-backbone hydrogen bonding partner – either water, or a side-chain hydrogen-bonding group.

In a beta sheet (Figure 2.7), the backbone is extended in a long strand. Two β-strands can either align in parallel (Figure 2.8) where the N-to-C direction is the same for both strands, or anti-parallel (Figure 2.9) where the N-to-C direction of one strand is opposite the N-to-C direction of the other. Both parallel and anti-parallel arrangements align the strands' backbone nitrogen and oxygen atoms so that they hydrogen bond. A collection of β-strands together form a β-sheet.

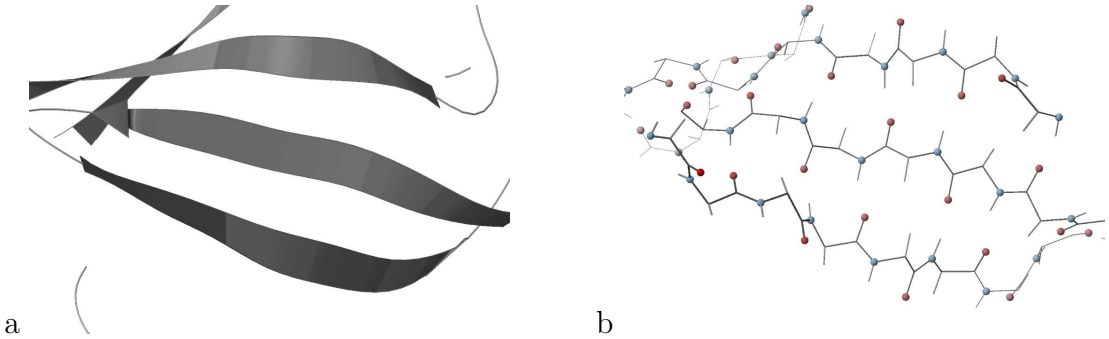


Figure 2.7: Ubiquitin's Beta Sheet. a) Cartoon and b) ball and stick representations of ubiquitin's beta sheet. The top strand and the middle strand show a parallel arrangement, the middle strand and the bottom strand show an anti-parallel arrangement.

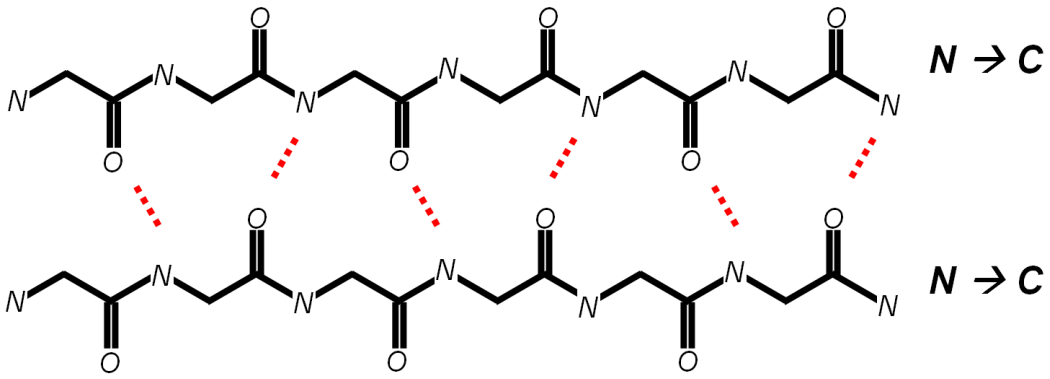


Figure 2.8: Parallel Beta Strands: If two β-strands align so that their N-to-C orientations point in the same direction, they are able to form a regular pattern of hydrogen bonds. The hydrogen atom bound to each backbone nitrogen is not shown.

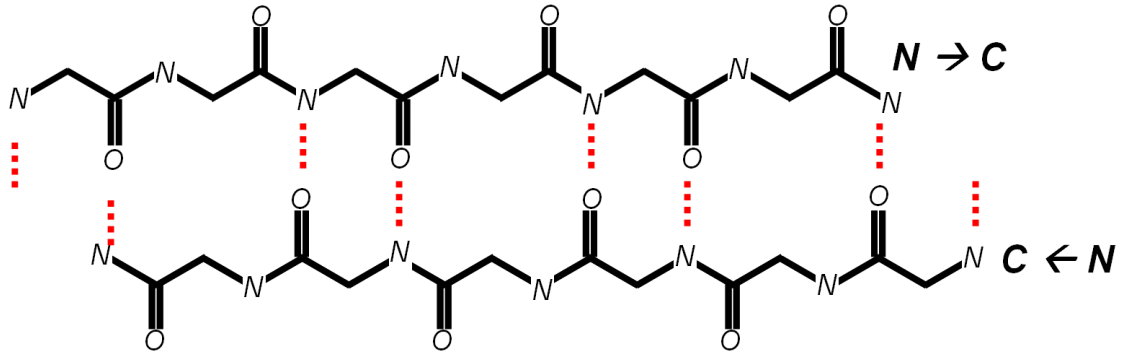


Figure 2.9: Anti-Parallel Beta Strands. If two β-strands align so that their N-to-C orientations point in opposite directions, they are able to form a regular pattern of hydrogen bonds. The hydrogen atom bound to each backbone nitrogen is not shown.

Sequence analysis of secondary structures in proteins reveals that some amino acids are more likely to form α-helices than others, while others are more likely to form β-sheets (Chou and Fasman, 1978; Garnier et al., 1978; Richardson and Richardson, 1988; O'Neil and DeGrado, 1990; Munoz and Serrano, 1994). The statistics can be interpreted as natural propensities for amino acids to form helices or sheets. Biochemists have used these propensities to predict protein secondary structure and from there the complete folded structure (Cohen et al., 1982).

Tertiary structure depends significantly upon side chains. Tertiary structure is a little frustrating to define; certainly specifying the location for all atoms in the protein defines its tertiary structure, however, biochemists often refer to the topological arrangement of secondary structures as tertiary structure (Richardson et al., 1992), as well as some specific types of interactions between pairs of side chains. Structural elements described as elements of tertiary structure include the protein's fold classification (Murzin et al., 1995), the hydrogen bonding pattern of β-sheets, side-chain salt bridges, the tight packing of hydrophobic side chains in protein cores and disulfide bonds (chemical bonds formed between the sulfur atoms in cysteine side chains). Disulfide bonds constitute the exception to the rule that protein chemical structure is captured completely by its primary structure – these chemical bonds are not captured in a one-dimensional sequence.

Quaternary structure describes the geometry of two or more chains together as they interact. Quaternary structure differs from tertiary structure only in the number of chains involved.

2.4 Energy Functions

Computational biochemists must be able to evaluate the energy of a particular structure. They express energy as a function of the coordinates of the protein. As biochemists manipulate the structure of a protein, they want to know if their manipulations improve (decrease) or degrade (increase) the protein's energy. Expressing the energy of a structure is both extremely important and extremely difficult, and has been the study of biophysicists for decades.

One form of energy function comes from the field of molecular dynamics. Molecular dynamics attempts to model protein motion using numerical integration of Newtonian-like functions of protein energetics. Several "force fields" for molecular dynamics have been developed such as Amber (Weiner et al., 1986; Cornell et al., 1995) and CHARMM (Brooks et al., 1983; MacKerell et al., 1998). The shape of the functions in these force fields are consistent across all force fields: what varies are the coefficients chosen to fit the fields to experimental data. The energy of an entire system can be described by the sum of five energy terms. The first three terms, bond length stretch, bond angle flex and dihedral twist, reflect the energetics between atoms separated by very few chemical bonds. The last two terms are applied to those atoms separated by at least three chemical bonds. Specifically, these terms look like

$$\begin{aligned} \mathcal{E}_{total} &= \sum_{i,j \ bonded} \mathcal{E}_{stretch}(i,j) + \sum_{i,j,k \ neighboring} \mathcal{E}_{flex}(i,j,k) \\ &+ \sum_{i,j,k,l \ neighboring} \mathcal{E}_{twist}(i,j,k,l) \\ &+ \sum_{i,j \ non-bonded} \mathcal{E}_{electrostatics}(i,j) \\ &+ \sum_{i,j \ non-bonded} \mathcal{E}_{vdW}(i,j) \end{aligned} \tag{2.3}$$

where the individual atoms are iterated over in each sum.

The van der Waals and electrostatic terms, the last two terms in this model, sum over all pairs of non-bonded atoms in the system. Because the function can be decomposed into the sum of pair interactions, it is called *pairwise decomposable*. The interactions between pairs of atoms are independent of the context in which they occur. The pairwise decomposability of the function lends itself to computational efficiency in a number of ways. It allows tabulation of all the pair energies in the protein; if part of the protein changes, then the interactions between the changing part and the unchanging part are all that need updating; interactions within the unchanging part will remain the same, and interactions within the changing part will also remain the same, so long as the distances have not changed between atoms in the changing part. In molecular

dynamics, pairwise decomposability of the energy function implies that the energy derivative for an atom can be computed analytically as the sum of the derivatives for each of its interactions.

Pairwise decomposability plays a large role in the Rosetta molecular modeling program. Rosetta contains a gradient-based minimizer that works directly in torsion space; the minimizer computes derivatives for bond dihedrals (Abe et al., 1984). These derivatives for dihedrals rely on the pairwise decomposability of the energy function. Torsional derivatives allow Rosetta to preserve ideal bond lengths and angles throughout protein folding trajectories, unlike molecular dynamics simulations which are also able to relax a structure, but require that bond angles and bond lengths be flexible. In protein design, pairwise decomposability allows the precomputation of residue pair energies so that the optimization phase need not compute any energies. Chapter 5 discusses pairwise decomposability's role in protein design in greater detail.

2.4.1 Solvation Energy Functions

In many molecular dynamics simulations, water is explicitly modeled along with the protein. In the absence of explicit water, proteins modeled with the energy function in Equation 2.4 unfold over the course of a molecular dynamics trajectory. By including water along with the protein, one can maintain proteins' folded states. So in addition to the thousands of atoms in a single protein, a biophysicist must include tens of thousands of water molecules (Berne, 1977). Getting the right water model has been an important topic in molecular dynamics (Jorgenson et al., 1983; Hermans et al., 1988; Silverstein et al., 1998). Modeling water explicitly costs much more than modeling a protein by itself and considerable effort has been directed at simulating only as much water as is necessary (Brooks and Karplus, 1983; Brungler et al., 1985; Brooks and Karplus, 1989). In protein design, the expense of modeling water explicitly would be prohibitive. Designers do not run molecular dynamics simulations during the course of their design calculations. To avoid the costs associated with modeling water explicitly, attempts have been made to express water's effects implicitly.

The amount of buried hydrophobic surface area has proven the most accurate metric to gage the stability conferred by the hydrophobic effect (Dill, 1990; Dahiyat and Mayo, 1996; Kortemme et al., 1998). To measure surface burial requires a representation of the protein's *solvent-accessible surface* (SAS) (Lee and Richards, 1971). Surface burial cannot be accurately modeled by a pairwise decomposable function. Once one

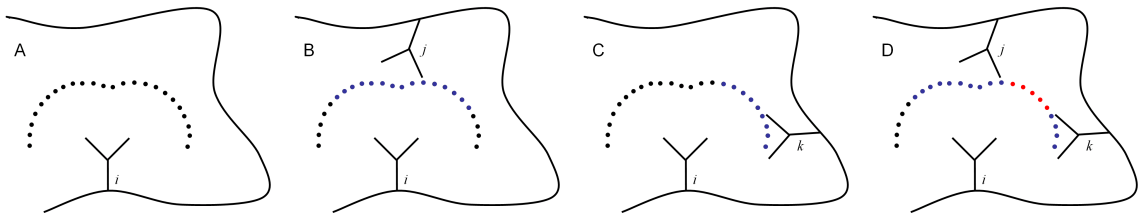


Figure 2.10: *Pairwise Surface Burial.* (a) The portion of residue i that is not buried from solvent by the backbone (b) Residue j buries some of i's surface from water (blue dots). (c) Residue k buries some of i's surface from water. (d) Together, residues j and k bury some of i's surface doubly (red dots). In Street and Mayo's pairwise decomposable energy function, this region is doubly counted, and over-counting in general is treated by scaling.

atom buries a certain portion of the surface of another atom, that surface portion cannot become more buried by the approach of a third atom. Because the solvent-accessible-surface area (SASA) is not decomposable into a sum of pairs, exact SASA-based solvation models have not been previously incorporated into the optimization step of protein design software.

Street and Mayo described an approximate SASA-based solvation model that is pairwise decomposable on the level of rotamer pairs (Street and Mayo, 1998). In their technique, they placed rotamer pairs onto a backbone from which all other side chains had been removed. They then measured the surface area that each rotamer buried of the other (Figure 2.10). Because the surface of one rotamer can be buried multiple times by other rotamers, this technique over-counts the degree of surface burial. They correct the over-counting of surface area burial by a scaling constant. Surfaces that are fully covered but only by a single other rotamer will be counted as somewhat exposed.

Zhang, Zeng and Wingreen improved upon Street and Mayo's approximate SASA-based solvation model. They placed rotamer pairs onto a backbone from which all other side chains had been replaced by large spheres that approximated the side chains' presence, and measured the surface area each rotamer buried of the other (Zhang et al., 2004). Because these extra spheres buried certain portions of each rotamer, there was less double-counting of surface burial. Instead their approximation under-counts burial in some areas and over-counts the burial in others. The average error in the approximation is neither positive nor negative; it is not biased towards under-counting or over-counting surface area. This is not to say that the approximation usually gets the right answer; their approximation regularly miscalculates SASA by 10 Å^2.

Lazaridis and Karplus proposed an implicit solvation model that is decomposable into the sum of atom pair interactions (Lazaridis and Karplus, 1999). They conceived of a field, F_i, that surrounds atom i where the field describes for a point in space the solvation energy contributed by that point if it were occupied by water. An atom j that approaches i excludes water from occupying a region of space surrounding i (Figure 2.11). They define the change in the free energy of solvation of atom i, ΔG_i^{slv}, caused by placing a single atom j in i's solvation field as the difference between a reference solvation energy, ΔG_i^{REF} and a volume integral over this field:

$$\Delta G_i^{slv} = \Delta G_i^{REF} - \int_{V_j} F_i(r)dr. \tag{2.4}$$

They define the change in free energy of solvation induced by all atoms j that surround i as

$$\Delta G_i^{slv} = \Delta G_i^{REF} - \sum_j \int_{V_j} F_i(r)dr \tag{2.5}$$

Having decided upon the form of a volume integral, they needed a function F to describe the field. They observed that the first solvation sphere surrounding an atom accounted for ∼84% of the solvation energy. They also noticed that the error function, which is given by

$$\text{erf}(y) = \frac{2}{\sqrt{\pi}} \int_0^y e^{-x^2} dx.$$

evaluates to 84% for erf(1). Thus to describe the field, the *solvation free energy density*, for a point at a distance of r from the center of atom i with radius R_i, with *correlation distance* λ_i, and with a field intensity of α_i they chose the following function:

$$F_i(r) = \frac{\alpha_i e^{-(\frac{r-R_i}{\lambda})^2}}{4\pi r^2} \tag{2.6}$$

Both α_i and λ_i depend on what the type of atom i is (aromatic carbon, carbonyl oxygen, etc). I should note here, though it I discuss it again in Chapter 8, that water is usually approximated as a sphere with a radius of 1.4 Å– the radius of an oxygen atom. Therefore, for most atoms, they set $\lambda_i = R_i +$ 2.8 Å so that the correlation distance represented the first solvation sphere surrounding atom i. If the entire volume surrounding atom i out to 2.8 Å past its van der Waals surface were occupied by other atoms, then the volume integral in Equation 2.5 would reflect an 84% loss in solvation free energy for the atom. Without getting into the details, the constants

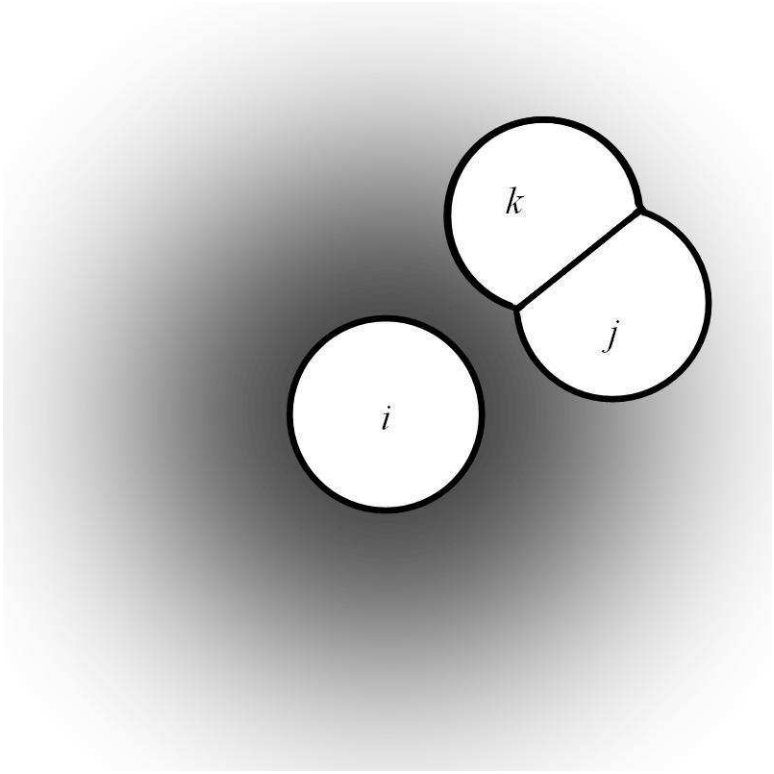


Figure 2.11: *Pairwise Decomposable Solvation Model.* A two-dimensional analogy for Lazaridis' and Karplus' solvation free energy density. The density is shown in gray scale surrounding the atom i in the center. Atoms j and k prevent water molecules from interacting with i in the regions j and k occupy, causing a loss of solvation energy. This loss is given by the integral of the density inside the volumes (areas) excluded by the two atoms. Atoms j and k effectively punch holes in the solvation field; one could imagine using a hole punch on this piece of paper to carve out some additional gray area. The volume of each atom is set so that volumes of overlapping (bonded) atoms are not double-counted.

α_i were chosen to penalize the burial of hydrophilic groups, and reward the burial of hydrophobic groups.

Lazaridis and Karplus tested their implicit solvation model with molecular dynamics. As mentioned before, in molecular dynamics simulations that do not include explicit or implicit solvent, the protein molecules tend to unfold over time. The force field described in Equation 2.4 is unable to keep a protein folded *in vacuo*. When Lazaridis and Karplus included their implicit solvation model, the proteins they simulated did not unfold.

2.4.2 Hydrogen Bond Energy Functions

Originally, hydrogen bonding in molecular dynamics was not modeled separately from the electrostatic terms. Recent force fields include a specific hydrogen bonding term given by a 10-12 potential:

$$\mathcal{E}_{hb}(i,j) = (\Big(\frac{R_{ij}}{d_{ij}}\Big)^{12} - \Big(\frac{R_{ij}}{d_{ij}}\Big)^{10})$$

defined similarly to the Lennard-Jones 6-12 potential, but shorter-ranged.

Kortemme *et al.* introduced a knowledge-based hydrogen bonding potential that aimed to capture the geometric specificity that comes from the quantum mechanics (Kortemme et al., 2003). Quantum mechanics predicts the electron orbital geometry of sp^2 hybridized atoms (such as the backbone carbonyl oxygen) to have the three sp^2 orbitals in the plane with 120° between each orbital, and to have their fourth orbital, a p orbital, perpendicular to that plane. Quantum mechanics also predicts that sp^3 hybridized atoms (such as serine's hydroxyl oxygen) will position their four sp^3 orbitals in a tetrahedral geometry, with 109° between them. Since hydrogen bonding occurs when a hydrogen atom is positioned inside the electron density of the acceptor atom, the geometry of the electron orbitals should play a role in the energy of the hydrogen bond. The better the overlap between the hydrogen atom and the electron orbital, the stronger the hydrogen bond. However, previous hydrogen bond models effectively treated the electron orbitals as if they were spherical by not including any degree of directional sensitivity.

Kortemme *et al.*'s knowledge-based function explicitly includes directional sensitivity. Their function is the sum of three terms:

$$\mathcal{E}_{hb}(i,j,i_b,j_b) = \mathcal{E}_{hb-dist}(i,j) + \mathcal{E}_{hb-\theta}(i,j,i_b) + \mathcal{E}_{hb-\phi}(i,j,j_b)$$

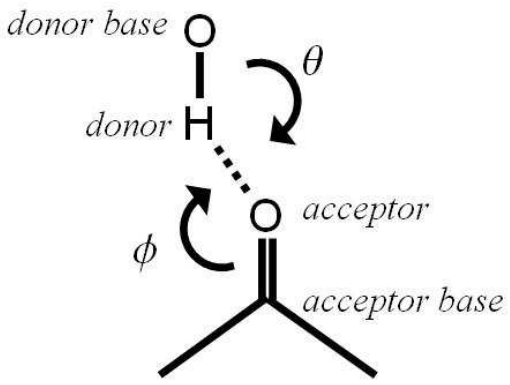


Figure 2.12: *Orientation Dependent Hydrogen Bond Energy Function.* Kortemme *et al.* defined a hydrogen bond function in terms of three parameters the donor/acceptor distance, the donor/acceptor/acceptor-base angle (ϕ), and the acceptor/donor/donor-base angle (θ). The first angle reflects the location of the electron orbitals. For example, carbonyl oxygen atoms are sp2 hybridized, with their two lone pairs in the plane of the carbon and the two atoms bound to that carbon. These orbitals are 120° away from the C-O bond vector. The statistics Kortemme *et al.* mined from the PDB agreed with the geometric intuition from quantum mechanics. The second angle, Θ, captures the orientation of the dipole, giving a preference for hydrogen bonds where Θ is 180°.

where i represents the donor hydrogen and j represents the acceptor, i_b is the *donor-base*, the heavy atom to which i is bonded, and j_b is the *acceptor-base*, a heavy atom to which j is bound that serves to orient j (Figure 2.12). The first term in this function, $\mathcal{E}_{hb-dist}(i,j)$ is a function of the separation distance between i and j. The next term, $\mathcal{E}_{hb-\theta}(i,j,i_b)$ is a function of the angle defined by the donor-base the donor-hydrogen and the acceptor and prefers to be at 180°. The last term, $\mathcal{E}_{hb-\phi}(i,j,j_b)$ is a function of the angle defined by the donor-hydrogen the acceptor and the acceptor-base. When the acceptor is a carboxy oxygen, this last function prefers an angle of 120°. When the acceptor is a hydroxyl oxygen, this last function prefers an angle of 109°.

Kortemme *et al.* mined high resolution structures from the PDB to obtain distributions of donor-acceptor distances and of ϕ and θ angles. By assuming that the PDB represents a Boltzmann ensemble, they inverted the observed distributions to produce energies. Kortemme *et al.*'s hydrogen bond energy function is what is used in Rosetta.

2.4.3 Non-Pairwise Decomposable Energy Functions

Although pairwise decomposability offers computational efficiency, several contributors to protein stability cannot be described as the sum of pair interactions. In particular, the effects of solvent, both in the way it forces hydrophobic groups to the core, and in the way it forces buried hydrophilic groups to form intra-molecular hydrogen bonds, cannot be accurately described as a sum of pairs. This is unfortunate for protein design in that solvent plays a prominent role in protein stability. Optimizing protein sequence for stability requires either approximating solvent's effects with a pairwise decomposable energy function or optimizing sequence using a non-pairwise decomposable energy function.

As mentioned before, the stability provided by the hydrophobic effect is proportional to the amount of buried hydrophobic surface area. The amount of buried surface area cannot be quantified without knowing the location of all atoms in the protein. Even once all of the atom locations are known, computing the SAS is still challenging. Lee and Richards defined the solvent-accessible surface as the surface produced by enlarging the van der Waals radius of the protein atoms by the radius of a water molecule, 1.4 Å (Lee and Richards, 1971). This is mathematically equivalent to the Minkowski sum as defined within robotics.

Suppose a robot is asked to move through some region that contains obstacles (Figure 2.13). The robot must compute where it can fit without colliding with the obstacles. It finds the regions it can fit into by considering the accessible regions of a new space called *configuration space*. In configuration space, the robot has been shrunk to a single point – its center, perhaps – and all of the obstacles are larger than they were in the original space. The size and shape of the obstacles in configuration space is defined mathematically by computing the Minkowski sum of the obstacles and the robot.

If both robots and obstacles are made of circles in 2D or spheres in 3D, the computation of configuration space is easy: the Minkowski sum produces a new set of configuration space obstacles which are themselves circles or spheres with radii that are the sum of the robot's radius and original obstacles' radii.

Lee and Richards' definition of the SAS follows when water is considered a spherical robot and the protein molecule a set of spherical obstacles. The Minkowski sum of the protein atoms and a water molecule is a set of spheres in configuration space where the radius of each configuration-space sphere is the sum of the van der Waals radius of the

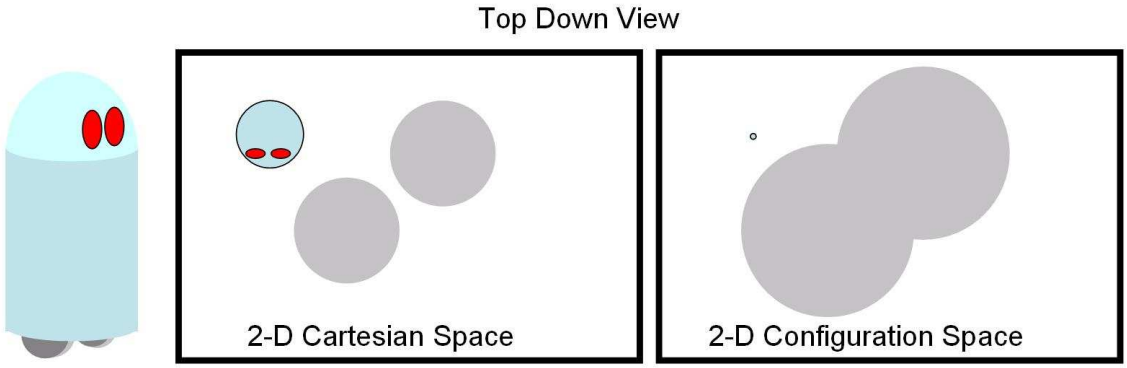


Figure 2.13: *Configuration Space* The Minkowski sum of a circular robot (blue) and circular obstacles (gray) produces obstacles in configuration space whose radii are the sum of the robot's and the obstacle's radii. A point on the surface of the union of the configuration space obstacles represents a position the center of the robot could be in Cartesian space without colliding with the obstacles. Because the configuration space obstacles overlap, the robot may not pass between them.

atom it originated from and the 1.4 Å radius of water. The surface of the union of the configuration-space spheres is the solvent-accessible surface (Figure 2.14b). Each point on the configuration space surface maps to a point in 3-D where the center of a water molecule can fit without colliding with the protein.

To compute the SAS requires computing the surface of a set of overlapping spheres. This is non-trivial. Recent techniques in computing the SAS include maintaining a spherical arrangement (Eyal and Halperin, 2005) for the enlarged spheres. With a spherical arrangement, the solvent-accessible surface can be computed exactly. Unfortunately, the computational expense of maintaining a spherical arrangement for a moving structure is large. Shrake and Rupley proposed a numerical approximation technique for computing the SAS (Shrake and Rupley, 1973). They sampled the surface of each enlarged sphere with 92 dots. Each dot was the representative of a surface patch. Summing the surface area for each patch whose representative dot is not contained inside any other enlarged sphere yielded the SASA. (They tested using up to 400 dots but decided that 92 were sufficient).

The technique of sampling dots on the molecular surface appears frequently in analysis of protein conformations. For example, Connolly created a software visualization package for molecular surfaces (Connolly, 1983). He defined what is sometimes called the *solvent-excluded surface* (SES), and what is often called the *Connolly surface*, by considering a probe sphere that rolls across the van der Waals spheres of the protein

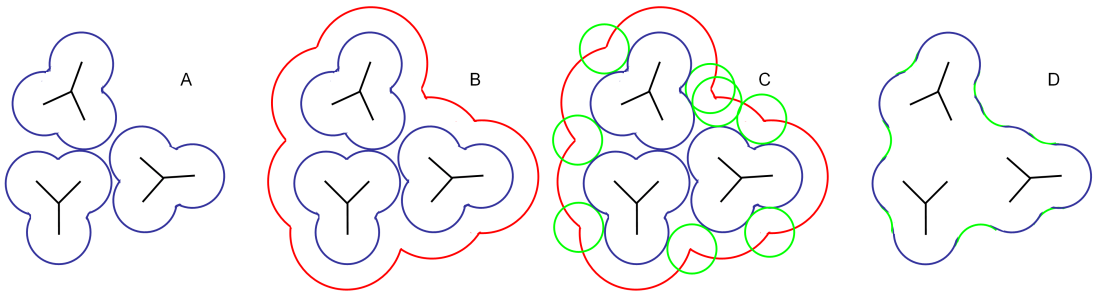


Figure 2.14: *Solvent-Accessible Surfaces.* (a) The molecular surface for three groups of atoms. (b) The solvent-accessible surface (SAS) in red, defined by increasing the van der Waals radii by the radius of water. (c) The intersections of enlarged spheres define points where a water molecule (green) would contact two atoms simultaneously. (d) The solvent-excluded surface (SES) consists of the *contact surface* (blue) and the *reentrant surface* (green).

atoms. The probe sphere deposits dots on the van der Waals surface at points where the probe contacts exactly one atom (blue curves in Figure 2.14d). He calls the surface defined by this set of dots the *contact surface*. In the regions where the probe contacts two or three atoms, the probe leaves dots along its surface in the region between the atoms (green curves in Figure 2.14d). He calls the surface defined by this second set of dots the *reentrant surface*. The contact surface is mathematically related to the SAS – the SAS is obtained in Connolly's rolling-probe framework if the probe sphere were to leave dots at its center instead of on the van der Waals surface. Connolly argues that SES better conveys the cavities inside proteins than the SAS, citing a handful of cases where biochemists benefited from his program. Molecular graphics has received considerable attention here at UNC as well (Varshney et al., 1994).

Dave and Jane Richardson have developed a suite of molecular graphics software based on dot-surface analysis (Richardson and Richardson, 2001). Unlike Connolly who designed software for examining surfaces, they designed software to analyze protein packing (Word et al., 1999a; Word et al., 2000). Like Connolly, they roll a small probe (0.25 Å radius) across the protein's van der Waals surface. While Connolly's technique deposited dots on the molecular surface when the probe sphere is not in contact with any other atom, the Richardsons' technique deposits dots on the molecular surface when the probe sphere makes contact (Richardson et al., 1992). For a pair of non-bonded atoms that come within 0.5 Å of each other, their visualization software deposits cool blue and green dots along the atoms' van der Waals surfaces in the regions where the the atoms come close (Figure 2.15a). When two atoms overlap, the cool blue dots

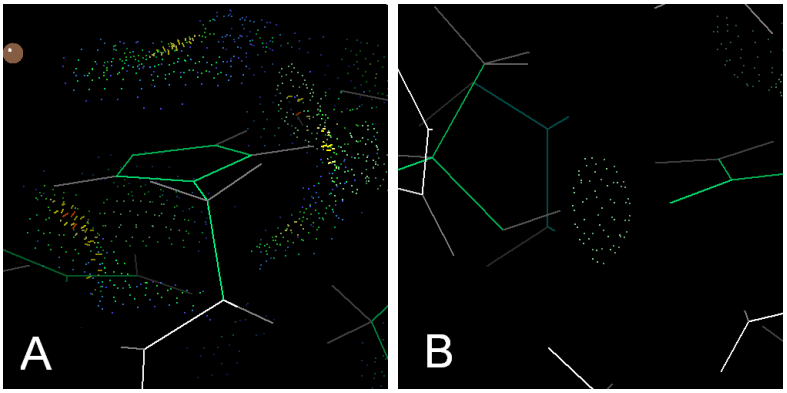


Figure 2.15: *Dot Contact Analysis in PROBE.* A) The contact dots for a histidine side chain in blue and green. B) A hydrogen bond between a threonine and an asparagine.

are replaced by orange and red spikes. However, if the atoms are hydrogen bonding partners (one polar hydrogen, one polar heavy atom), their software leaves turquoise dots on the van der Waals surfaces, creating a what looks like a lens (Figure 2.15b).

They have used their small-probe contact surface to create a scoring function based on the volume of atom collisions, V_{coll}, the volume of hydrogen bonds V_{hbond} and a function of the amount of close contact between non-bonded atoms:

$$\text{score} = -10 * V_{\text{coll}} + 4 * V_{\text{hbond}} + \sum_{d \in \text{dots}} w(gap(d))$$

where the set of dots are the point samples on the van der Waals spheres that are 1) not contained inside the van der Waals sphere of any atom within four chemical bonds and 2) where a tangentially placed probe a) makes contact with some nearby atom not within four chemical bonds and b) the van der Waals surface that the probe's center is closest to is not from an atom within four chemical bonds. The *gap* for a dot, $gap(d)$ is the distance between the center of the tangentially place probe and the closest van der Waals surface. The function $w(gap(d))$ using a probe of radius r is given by the Gaussian approximation

$$w(gap(d)) = e^{-(\frac{gap(d)}{r})^2}$$

The volume of overlap V_{coll} and V_{hbond} could be approximated by measuring for each dot d on the surface of atom i that is inside the van der Waals sphere of a non-bonded atom j the distance from d to the plane of intersection between i and j. The Richardsons' simplify their calculation by instead measuring the *penetration distance*

for d – the distance between d and the surface of the van der Waals sphere of j. Their approximation does not converge to the volume of overlap as the dot sampling density increases; however, it is close (Leaver-Fay et al., 2005a).

For a pair of atoms approaching each other *in vacuo*, the gap score and the collision penalty together fit the gas-phase empirical data for single atom-pair interactions. The crucial difference between the Richardsons' scoring function and the Lennard-Jones 6-12 function is that their function is not decomposable into the sum of atom-pair interactions.

The Richardsons claim that the pairwise decomposable Lennard-Jones energy function is fundamentally flawed. They argue that the dipole interactions that the Lennard-Jones function is meant to capture are subject to shielding: an intervening atom will block the interaction between a pair. Because dots contained inside the spheres of bonded atoms are discarded, and because the score for a dot is a function of its distance to the closest surface (as opposed to all nearby surfaces), the Richardsons' function captures the idea of shielding.

2.5 Protein Design

Protein designers seek new amino acid sequences which fold so that they position chemical groups in a desired manner. Naturally occurring proteins adopt a compact geometric configuration when placed in water. However, a random amino acid sequence is unlikely to form a stable fold (Desjarlais and Handle, 1995). The first challenge of protein design is to create a stable protein. That challenge is magnified if the designer also seeks to require chemical functionality of their new protein.

Current techniques in computational protein design involve solving instances (often many instances) of the side-chain-placement problem. This thesis focuses on the side-chain-placement problem. This section states the problem before it describes protein design's brief history. Section 2.6 reviews the history of the algorithms used to solve the side-chain-placement problem.

Problem 1 *The Side-Chain-Placement Problem. Given*

- *a fixed backbone (the* scaffold*) with a set of fixed residues,*
- *a set of residues to change (the* molten residues*),*
- *a list for each molten residue of the amino acids accessible to it,*

- *a rotamer library, and*
- *an energy function,*

find the assignment of rotamers to the molten residues that minimizes the energy function.

The side chain placement problem does not require that the energy function be pairwise decomposable; however, biochemists have imposed the pairwise decomposability requirement on the energy functions they have used to design proteins. Following side chain placement, biochemists filter their designs using non-pairwise-decomposable energy functions since such functions provide keen insight into the quality of the structures; however, they do not optimize their side-chain placements according to these non-pairwise-decomposable energy functions.

For pairwise-decomposable energy functions, the energy terms allowed fall into two classes: those that depend on pairs of rotamers assigned to neighboring molten residues and those that depend on a single rotamer assigned to a residue. The first class of terms are described as pairwise decomposable if the interaction between two residues is independent of the rest of the protein. The terms that contribute to residue pair energies include the atom pair electrostatic ($\mathcal{E}_{electrostatics}$) and van der Waals interactions ($\mathcal{E}_{vdW}$) from expressions 2.1 and 2.2. They also include solvation terms, such as the pair-burial terms of Street and Mayo or Zhang, Zeng and Wingreen, or the Lazaridis-Karplus solvation model. Call the sum of these individual *pair-energy terms* E_{pair}.

Note that in the context of design, *pairwise decomposable* refers to the ability to decompose the energy function into a sum of residue-pair interactions, and not necessarily into the sum of atom-pair interactions. Certainly any energy function that can be expressed as the sum of atom-pair interactions can be expressed as the sum of residue-pair interactions. The converse is not necessarily true. Representing the interaction energies in terms of residue-pair interactions lets designers store all of the interaction energies between a pair of rotamers on a pair of residues as a single floating point number. This allows designers to precompute and store rotamer-pair energies before optimization begins, so that they need not repeatedly compute rotamer-pair energies during optimization.

The second class of terms includes the internal strain for a rotamer ($\mathcal{E}_{twist}$), the propensity of the rotamer's amino acid to form the particular type of secondary structure, and the pairwise energy terms that describe the interaction of the rotamer with

the backbone and the background residues. Call the sum of the rotamer internal strain and the rotamer/background energies E_{self}

If S_v represents the rotamer assigned to residue v, the energy of the system is

$$\sum_{v} \mathcal{E}_{self}(S_v) + \sum_{v_1<v_2} \mathcal{E}_{pair}(S_{v_1}, S_{v_2}) \tag{2.7}$$

In later sections, I refer to the rotamer set that can be placed at a particular residue, v as $\mathcal{S}(v)$. The assignment of rotamers to the molten residues which minimizes Equation 2.7 is commonly referred to as the *global minimum energy conformation* (GMEC).

2.5.1 The History of Protein Design

The history of protein design teaches us that protein design is difficult and that it must be performed with models at detailed resolution. Attempts to design proteins at low resolution – attempts that lacked a 3-Dimensional representation of side chain geometry and attempts that treated atomic collisions with less of an energetic penalty than they require – have consistently failed to produce proteins that resemble natural proteins. Successes is protein design have resulted from strategies that search through wide regions of protein conformation space and that treat atomic collisions with appropriate energetic penalties. The purpose of this section's review of the protein design literature is to hone in on the challenges present in protein design and particularly on those challenges that continue to frustrate current work in protein design. These challenges are discussed in greater detail in Chapter 3.

Designers did not attempt to solve the side-chain-placement problem for the early decades in design's history. The prevailing philosophy in early protein design was to model designed proteins with only as much detail as necessary (Beasley and Hecht, 1997). To a certain extent, this philosophy remains; for instance, modern designers do not use quantum mechanical calculations while they solve the side-chain-placement problem. Early protein design did not include solving the side-chain-placement problem because designers did not think they needed to. First generation designs (1979–1984) were constructed from physical (plastic or brass) models, and specified the packing arrangement of side chains in the core, but did not rely on computers to search through amino acid sequence space. Designed proteins from this generation were often unstable and none could be shown to adopt their intended structures. Many second generation designs (1985–1995) did not *design-in* side-chain packing at all; designers left it to the

proteins they synthesized to figure out for themselves how best to pack (Kamtekar et al., 1993). The second generation designs that did design-in side chain packing continued to rely on the human designer to pack side chains by hand using either physical models and computational models; these designs none the less avoided the side-chain-placement problem. Third generation designs included solving the side-chain-placement problem, but still treated design at a somewhat low resolution: the designs tolerated slight atomic collisions. Designed proteins from this generation have proven energetically stable (Dahiyat and Mayo, 1997b) and even functional (Looger et al., 2003; Dwyer et al., 2004); however none of these proteins could be crystallized. This inability to characterize designed proteins structurally using the same technique to characterize naturally occurring proteins implies that these designed proteins are still lacking a crucial element. Fourth generation designs, the current generation, involve solving thousands of instances of the side-chain-placement problem so that a tight, collision-free packing of side chains can be found (Harbury et al., 1998; Kuhlman et al., 2003). This fourth generation has finally resulted in crystal structures that match the design; a substantial accomplishment. In order to appreciate the series of decisions that led to the current generation of design strategies, I begin with the earliest mentions of protein design in the biochemistry literature and work forward through time.

There are two papers commonly cited as inspirations by the protein design community. In 1981, Drexler described the future of molecular engineering where the machines of tomorrow would be built one molecule or one atom at a time by other microscopic machines (Drexler, 1981). He pointed out that the principles of DNA manipulation provided the means to engineer novel proteins. By engineering proteins, we would be able to create molecular sized analogs to the motors, rotors, struts and bolts found in man-sized engineering. The task of designing proteins might be difficult; of the 20^{1000} different amino acid sequences ($\sim 10^{1300}$) for a 1000 residue protein, there might be only a tiny fraction which would function as desired. Drexler projected that only 10^9 amino acid sequences might perform the desired task. While these functioning sequences compose a vanishingly small percentage of all available sequences, there may still be an enormous number of them. Drexler projected possible limitations to protein design. For instance, designed proteins would likely fail to perform at certain tasks or at high temperatures; however, proteins could be designed to create new molecular tools that then would function as desired or operate at high temperatures. The future is bright.

In 1983, Pabo suggested that selecting the appropriate amino acid sequence for a

task start with selecting a folded structure for the designed protein, and then seeking the amino acid sequence that would fold to that structure (Pabo, 1983). This avoids the problem of predicting the folded structure for a designed protein and removes protein folding from the list of problems to be solved before protein design can proceed. Instead, designers would solve an *inverse folding problem*: that of predicting the set of sequences that would adopt a given fold.

Pabo credited the inverse folding problem to the approach used by Gutte *et al.* (Gutte et al., 1979) from four years earlier where they had selected a folded structure for the peptide they were designing as it interacted with RNA. They designed the sequence of their 34-residue protein so that it could adopt a structure with two β-strands and an α-helix. In this possible conformation, the protein could interact favorably with yeast tRNA^{Phe}. Their peptide did show some enzymatic activity, an impressive feat. However, it was unstable; in the absence of RNA, the protein was unfolded. The instability made it impossible to characterize their protein structurally, preventing them from knowing whether their protein behaved as it was designed to.

Moser *et al.* (Moser et al., 1983) thought the instability of the designed protein was due to the difficulty in designing a mixed α-β protein. They instead designed a four-stranded β-sheet protein intended to bind the insecticide DDT. Their protein aggregated in water. Proteins aggregate when inter-molecular interactions stabilize large amorphous and insoluble structures. Peptide aggregates precipitate out of solution. In an aqueous 50% ethanol solution, they were able to dissolve their peptide and observe that it was monomeric. It bound DDT with a moderately strong affinity; they measured the dissociation constant at 2×10^{-5} M. They did not discuss their strategy for designing their DDT binding protein, though they mentioned model building, which I take to mean physical models (plastic, space-filling CPK models or brass Kendrew models).

Kullmann followed in the same vein in building a poly-peptide designed to bind the opiate, [Leu]-enkaphalin (Kullmann, 1984). He designed a small trough formed between four beta sheets to use as binding pocket for the opiate. He created a space-filling CPK model of the structure he wanted to build and designed the amino acid sequence so that the ligand would pack well inside the trough. While Kullmann was able to observe binding between his peptide and the ligand, a structural characterization by circular dichroism (CD) spectroscopy did not suggest that the peptide actually adopted the structure he had designed. Kullmann could not make a structural argument for the ligand/peptide affinity.

These early attempts at protein design were too ambitious. By heading straight toward creating new function, designers missed the critical first challenge which was to design stable protein structures. The next decade was spent in a cautious step back from designing function: the immediate challenge was to design a stable protein.

Designers took two approaches to designing stable proteins: they either attempted α-helical design or β-sheet design. Alpha helical design proved much easier.

Lau *et al.* (Lau et al., 1984) designed and synthesized several coiled coils. Coiled coils are composed of α-helices that are wrapped around each other – helices wound up in super-helices. As mentioned before, the most common α-helix forms when the backbone carbonyl oxygen on residue i forms a hydrogen bond with the backbone nitrogen on residue $i + 3$ and forms a right-handed helix. These atoms are roughly three and a half residues apart; seven residues make just over two turns along the helix. The seven-residue repeat is called a *heptad* (Figure 2.16). If seven residues made exactly two turns, then a pair of helices aligned in parallel in the plane would point the first and fourth residues (the a and d positions) in the seven residue repeat directly at each other. Since seven residues make up just a little more than two turns, a pair of helices has to form a left-handed super-helical twist so that the first and fourth residues remain pointed towards the center. The design Lau *et al.* created was based on a placing hydrophobic amino acids at the first and fourth positions in a heptad repeat and hydrophilic amino acids at the five other positions so that two proteins would dimerize to bury the hydrophobic residues away from solvent. Both hydrophobic and hydrophilic amino acids were chosen for their high helix propensities. Additionally, they placed a negatively charged amino acid, GLU, at position five (the e position) and a positively charged amino acid, LYS, at position seven (the g position), so that the parallel arrangement of helices could create stabilizing salt bridges.

They synthesized five proteins of varying with sequences given by the formula (lys-leu-glu-ala-leu-glu-gly)$_n$-lys with n ranging from 1 to 5 creating proteins with 8, 15, 22, 29, and 36 residues. They found that the 29 and 36 residue proteins formed stable dimers.

DeGrado and colleagues *et al.* (Eisenberg et al., 1986) designed a helix intended to form a tetramer, which they called α-1. Their 12-residue helix, composed of GLY-LYS-LEU-GLU-GLU-LEU-LEU-LYS-LYS-LEU-LEU-GLU-GLU-LEU-LYS-GLY was minimal in the choice of amino acids. They chose amino acids that have a high helical preference. They designed their helices to align in an anti-parallel arrangement, an arrangment commonly observed in natural four-helix bundles. The positively and neg-

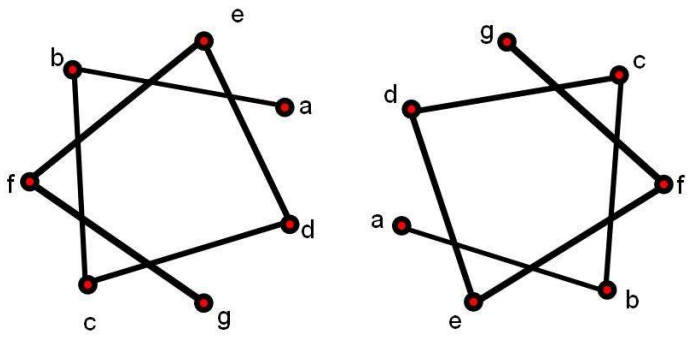


Figure 2.16: *The Coiled-Coil Heptad Repeat.* Seven residues in a pair of helices. This view is down the center of two helices, represented by the location of their Cα atoms, so that the residues rise out of the plane of the picture in the N to C direction. Every seven residues, the helix makes slightly more than two turns. In order to preserve the pattern that a and d positions point towards the space between the two coils (the core), the coils must wrap around each other forming a left-handed coiled-coil. The pattern that e and g residues are in close proximity also persists along the length of the coiled-coil. Many coiled-coil designs place opposite charges at these positions so that favorable salt-bridge interactions may be formed.

atively charged residues aligned to form salt bridges.

They synthesized their protein and measured its CD spectra at low and high protein concentrations. At low concentrations, the protein was monomeric and disordered. At high concentrations, the protein formed helical aggregates. They were unable to determine the molecular weight of the aggregates to determine whether they tetrameric. However, they were able to crystallize their protein . The resulting crystal showed 2,2,2 symmetry, consistent with the anti-parallel tetramer arrangement they had designed.

They did not present a crystal structure until four years later (Hill et al., 1990); with the crystal structure, they discovered that their 12 residue fragments formed hexamers instead of tetramers as designed. Pairs of helices formed dimers and three dimers together formed a favorable hexamer. The crystal packing was such that adjacent hexamers packed as a tetramer, however, the two dimers were much further apart than usual four-helix bundles, and the two hydrophobic amino acids pointed towards the core of their source hexamers and not into the interior of the tetramer. Their structure did not match their design.

They were able to crystallize one more minimalist helical design, but disappointingly

found that the protein did not adopt the desired dimer configuration but instead formed a trimer (Lovejoy et al., 1993).

DeGrado continued work with minimal, helical designs (Ho and DeGrado, 1987; Regan and DeGrado, 1988; Lear et al., 1998; DeGrado et al., 1989; Osterhout et al., 1992; Raleigh et al., 1995) with the explicit intent of not *designing in* side-chain packing. Helix propensity and a heptad repeat to favor hydrophobic burial brought him quite far; the final design of their four-helix bundle protein α-4 had a stability of -21 kcal/mol, making it more stable than most native proteins. They never succeeded, however, in crystallizing α-4.

The minimalist approach hit its peak in 1997 with Beasley and Hecht's review of protein design (Beasley and Hecht, 1997). They ask:

> *Is good packing essential? Must it be explicitly designed a priori? We suggest the answers to these questions are (respectively): most certainly yes; and probably no.*

The evidence for requisite specificity in hydrophobic packing was somewhat contradictory in the run-up to Beasley and Hecht's declaration. The major evidence for their side was the observation in two studies that randomly packed protein cores formed stable protein structures. The conclusion drawn from these two studies was that designing protein cores with random hydrophobic amino acids would produce well packed proteins; nature would figure out how to pack cores, so designers do not have to.

In the first of these two studies, Lim and Sauer (1991) attempted to exhaustively mutated three of the core residues in bacteriophage T4 lysozyme to one of five hydrophobic amino acids (LIVMF). These mutations give rise to a possible 125 possible mutants. They were able to synthesize DNA for 78 of these sequences, and found that 55 (70%) of the mutants maintained catalytic activity. Random mutations did not completely disrupt catalytic activity and therefore did not completely disrupt the folded structure.

In a similar experiment, Hecht and colleagues constructed random four-helix bundles (Kamtekar et al., 1993) dividing the amino acids simply into two classes – those that are hydrophobic (H) and those that are polar (P). This simplistic representation of a protein is called the "HP Model." They specified a protein generically using the HP model and created DNA to describe these proteins relying on degeneracy within the genetic code. They used the degenerate NAN codon (where N is any nucleic acid, and A is as adenine) to specify any of the hydrophobic amino acids among PHE, LEU, ILE,

MET, and VAL and the NTN codon (where T is thyiomine) to specify any polar amino acid among GLU, ASP, LYS, ASN, GLN, and HIS. The sequence they design connected four helices each with a heptad repeat. They were able to generate 40 sequences and incorporate them into bacteria. Of these 40, they were able to isolate protein product from 28 (60%). Now, cells have a natural defensive mechanism against improperly folded proteins, which could be potentially deadly, whereby they detect and degrade such proteins. Because 28 of the 40 sequences could be purified, the majority of the HP proteins were sufficiently stable to survive the bacterial protein degradation mechanisms. Although random amino acid sequences are not likely to form stable structures, those proteins who are capable of orienting their hydrophobic and hydrophilic amino acids in the right directions seem sufficiently well formed. They selected three of these proteins (by filtering for low-pH stability) and found that two had near-native stabilities between 3 and 4 kcal/mol and that each of the three proteins displayed α-helical structure as measured by circular dichroism, consistent with the idea that the native structure matched the design. However, Hecht and colleagues never published a crystal or solution structure for any of 28 proteins they designed.

Countering the idea that designed proteins with core residues chosen randomly would form stable folded structures, and supporting the idea that explicit design of hydrophobic core packing is necessary for protein design, Harbury *et al.* considered three variants on a simple helical sequence (Harbury et al., 1993). They kept fixed the amino acid sequence for all but the first and fourth residues (the a and d residues) in the heptad repeat of a thirty one residue protein. At the first and fourth positions, they tested IL, II, and LI which formed dimeric, trimeric and tetrameric proteins. Because isoleucine and leucine occupy the same total volume, the sole explanation for the differences between the three structures is the packing geometry. To not explicitly model core packing in protein design runs the risk that the structure of the designed protein will not adopt the desired fold.

Further weakening the HP model, Baldwin *et al.* crystallized and solved several of the bacteriophage T4 lysozyme mutants that Lim and Sauer generated (Baldwin et al., 1993). These variants consistently displayed backbone movement that accommodated the mutations. While this result can be interpreted as suggesting that hydrophobic packing does not need to be designed in, it can also be interpreted to suggest that random mutations will not fit on the original backbone. The bacteriophage T4 lysozyme backbone seems especially flexible as all attempted redesigns of it have resulted in (undesired) backbone motion (Hurley et al., 1992; Mooers et al., 2003).

The evidence needed to end the debate over the role of packing specificity in protein design was the crystal structure of a designed protein that matched the design. If either side produced such a crystal structure, they would have possessed the strongest confirmation of their hypothesis; naturally, much work focused on crystallizing designed proteins. The two crystal structures presented by DeGrado and colleagues in 1990 and 1993, though they did contain the intended secondary structures, did not display the intended tertiary structure. Before I tell who first designed a protein with a matching a crystal structure, I rewind to the 1980's to discuss the second track of protein design: the design of β-sheet proteins.

While DeGrado and colleagues pursued α-helical designs, the Richardsons' and Erickson and colleagues pursued the design of β-sheet proteins (Richardson et al., 1984; Richardson and Richardson, 1987; Richardson et al., 1992; Yan and Erickson, 1994; Quinn et al., 1994) though they also dabbled in α-helical protein design (Hecht et al., 1990; Engle et al., 1991). The design of β-sheet proteins proved more difficult than the design of α-helices; the crucial difference between the two kinds of secondary structure comes from the nature of their backbone hydrogen bonds.

Alpha helices and beta sheets are similar in that in the middle of a helix and in the middle of a sheet both backbone carbonyl and nitrogen groups are satisfied by intra-molecular hydrogen bonds. In a helix the carbonyl oxygen residue i hydrogen bonds to the nitrogen on residue $i+4$, while the nitrogen on residue i hydrogen bonds to the carbonyl oxygen on residue $i-4$. In an anti-parallel beta sheet the carbonyl oxygen and the nitrogen on residue i hydrogen bond to the nitrogen and carbonyl oxygen on residue j. In a parallel beta-sheet, the nitrogen on residue i forms a hydrogen bond with the carbonyl oxygen on residue j so that the carbonyl oxygen on residue i can form a hydrogen bond on residue $j+1$.

While the residues in the middle of each helix and the middle of each sheet form satisfying intra-molecular hydrogen bonds to other backbone elements, the residues at the ends of the helix and the ends of the sheet do not. These hydrogen bonding groups must either form hydrogen bonds with side chains or remain solvent accessible. For a helix, there are only two residues that are unsatisfied, the N- and C-cap residues mentioned earlier (Richardson and Richardson, 1988). For a β-sheet, however, every other residue along the length of both edge strands are left out of the backbone hydrogen bonding. These "unsatisfied" hydrogen bonding groups present a danger; they may allow the protein to polymerize. The unsatisfied groups from the terminal strand on one protein can align with the unsatisfied groups from the terminal strand on another

protein to form a stable structure, an aggregate (Figure 2.17). Proteins that aggregate precipitate out of solution. Alzheimer's disease is believed to be caused by β-sheet aggregation. The design of β-sheet proteins requires explicit consideration of potential aggregate structures and the selection of sequences that could not favorably adopt these structures. The Richardsons call this process *negative design* (Richardson et al., 1992). To design against a structure, designers place like charges next to each other or ensure a steric clash.

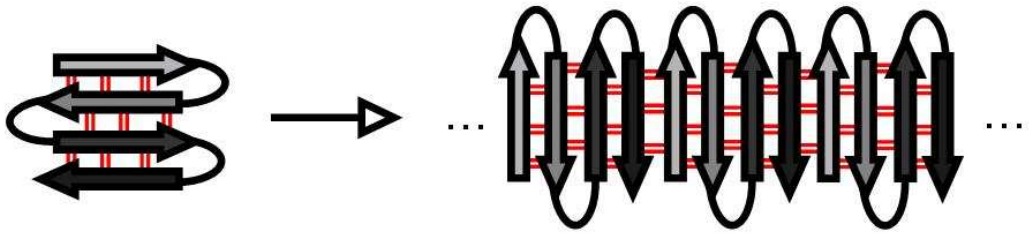


Figure 2.17: A gray-scale illustration of β-sheet aggregation. The anti-parallel hydrogen bonding pattern is illustrated with red lines connecting adjacent β-strands. The light gray and dark gray strands both point half of their backbone residues towards solvent in the monomeric form of the protein (left). These hydrogen bonding groups are primed to pair up into a dimer or a multimeric aggregate (right). Neurons from Alzheimer patients are killed by the formation of protein "plaques" which pierce the cell membrane like shards of glass. While the structure of these plaques is not yet known, they are thought to be β-sheet aggregates.

The Richardsons' β-sheet design included designed-in hydrophobic core packing. They used brass Kendrew models to physically position side chains in the interior, noting that molecular graphics and computer-aided design had not advanced far enough to significantly aid design (Richardson et al., 1992). They also used plastic, space filling CPK models to design their structure. The space filling models are very good at preventing steric overlap; however, the interior of a CPK model is visually inaccessible. Molecular graphics with opaque spheres were able to produce structures that looked like the CPK models. They preserved the opacity of the cores for CPK models while removing the useful steric information of a physical model; the worst of both worlds. The Richardsons' frustration with molecular graphics motivated them to design their dot-based visualization tools (Richardson and Richardson, 2001).

Although the series of β-sheet designs the Richardsons produced did achieve expressibility and stability, they were not crystallizable. The strategy that lead to the

first crystal structure was, as hinted, based on the explicit packing of hydrophobic cores by solving the side-chain-placement problem. The initial designs based on this strategy were not crysallizable, but none the less represented a significant breakthrough.

Mayo and colleagues achieved notable success in designing a pair of proteins that matched their NMR solution structures to a resolution of 1.4Å and 1.08Å (Dahiyat and Mayo, 1997b; Dahiyat et al., 1997). They redesigned a small α-β-protein by preserving the initial backbone structure of a zinc-finger protein and explicitly solving the side-chain-placement problem. The energy function they used included a Lennard-Jones potential and a solvation model based on surface-area burial between rotamer pairs (Dahiyat and Mayo, 1996; Street and Mayo, 1998) which is decomposable into the sum of rotamer-pair interactions.

The first crystal structure arrived shortly thereafter as Harbury *et al.* incorporated Crick's parametric equations for the coiled-coil super-helical twist (Harbury et al., 1995) to sample many different backbone conformations. Harbury *et al.* solved an instance of the side-chain-placement problem for each backbone sample (Harbury et al., 1998). Harbury designed three coiled coils: a dimer, a trimer, and a tetramer. The tetramer displayed high stability, refusing to melt below 100° C. Its crystal structure matched the structure of the design to within a remarkable 0.20 Å root-mean-square deviation (RMSD).

Mayo's and Harbury's successes suggest the conclusion that design, if it aims to achieve native-like protein stability, requires designed-in core packing, contrary to the beliefs of Hecht and DeGrado. It also suggests that computers are required to perform side-chain conformational optimization, contrary to the attempts of the Richardsons to design core packing manually.

All recent successes in protein design have involved computationally optimized side-chain placements. Kortemme *et al.* designed a 20 residue β-sheet protein with marginal stability and cooperative unfolding (Kortemme et al., 1998). Kuhlman *et al.* iteratively combined backbone motion and side-chain placement to redesign the turn between two β-strands, increasing the number of residues in the protein and improving the protein's stability (Kuhlman et al., 2002). Dantas *et al.* redesigned the cores for seven proteins using the same software (Dantas et al., 2003). Kuhlman *et al.* designed and crystallized a protein with a completely novel backbone (Kuhlman et al., 2003). Hellinga's group redesigned the active site residues in ribose binding protein to bind the explosive trinitrotoluene (TNT) by sampling ligand binding orientations in the binding pocket and optimizing side-chain placement around each of the ligand samples (Looger et al.,

2003). Hellinga later designed the same ribose binding protein backbone to catalyze the isomerization of trios phosphate. This protein marked the first successful enzyme design with near-natural rate enhancements (Dwyer et al., 2004). Ambroggio and Kuhlman created a "switch" protein that formed a monomeric zinc finger in the presence of zinc and adopted a trimeric coiled-coil in zinc's absence (Ambroggio and Kuhlman, 2006). In this variation of the design problem, they sought the sequence optimal for both structures simultaneously.

Computational optimization of side-chain placement has won out as the dominant technique in protein design. The next section describes the computational approaches for solving the side-chain-placement problem.

2.6 Techniques for Side-Chain Placement

Like all important computational problems, the side-chain-placement problem is NP-Hard. In 2002, Peirce and Winfree used the minimization formulation in Equation 2.7 to reduce SAT (Cook, 1971) to the side-chain-placement problem (Pierce and Winfree, 2002). I show in Chapter 4 that formulating the problem to represent only those interactions present in the problem instances provides way of bounding the complexity for problem instances.

Despite the difficulty, designers pursued computational methods for solving or at least attempting a solution to the side-chain-placement problem. The methods they employed can be classified as either exact or inexact.

The exact techniques include brute force enumeration (Ponder and Richards, 1987), branch and bound (Gordon and Mayo, 1999; Canutescu et al., 2003), integer linear programming (Eriksson et al., 2001; Kingsford et al., 2005) and iterative applications of the dead-end elimination (DEE) theorems (Desmet et al., 1992; Lasters and Desmet, 1993; Goldstein, 1994; Lasters et al., 1995; Looger and Hellinga, 2001). Of these, dead-end elimination has proven the most popular. The dead-end elimination theorem (discussed in greater detail in Section 2.6.1) can be applied to prove that a rotamer is not a member of the global minimum energy conformation (GMEC). Once a rotamer has been eliminated, it can be removed from further consideration. Iteratively eliminating rotamers either results in the GMEC – all rotamers but one from each residue have been eliminated – or a smaller problem, an irreducible core at which the theorems provide no insight. If the size of the state space for the irreducible core is small enough ($< 10^8$), then it can be solved simply with brute force enumeration. Though it provides

no guaranteed solution, DEE has none the less proven incredibly useful at solving the side-chain-placement problem.

The inexact techniques are all stochastic on some level. They include simulated annealing (Holm and Sander, 1992; Hellinga and Richards, 1994; Dahiyat and Mayo, 1997b; Saven and Wolynes, 1997; Kuhlman and Baker, 2000), genetic algorithms (Desjarlais and Handle, 1995; Walsh et al., 1999; Johnson et al., 1999), and self-consistent mean fields (Koehl and Delarue, 1994). The inexact techniques offer no performance guarantee; the answers they generate cannot be proven to lie within any range of the GMEC's energy. However, the inexact techniques in practice often converge to the GMEC (Leaver-Fay et al., 2005a) and are orders of magnitude faster than the exact techniques. Both Mayo and Hellinga are now using a variation on simulated annealing, called the FASTER algorithm (Desmet et al., 2002; Allen and Mayo, 2006). Kuhlman and Baker continue to use simulated annealing.

2.6.1 The Dead-End Elimination Theorems

The dead-end elimination theorems are used to rule out certain rotamer assignments. One keeps a list of all the rotamers at all molten residues, and then removes from this list those rotamers that can be proven to not belong in the optimal configuration. The optimal solution remains when all but one rotamer from each molten residue has been removed from this list. The dead-end elimination theorems provide the means by which rotamers may be scratched off the list; they have the general form "If rotamer s is *not as good* as rotamer s', then s is not a member of the GMEC and may be removed from the list" where the *"not as good"* portion of the theorem is an inequality.

Desmet *et al.* originally introduced the dead-end elimination theorem in the context of the side-chain-placement problem in 1992. Their theorem applies to a single residue v, on which a pair of rotamers s and s' have been chosen. It relies on the idea of a *conformational background*. A conformational background for residue v is a rotamer assignment to the rest of the residues in the protein not including v. The theorem says that if the conformational background that favors rotamer s' the least favors s' more than the conformational background that favors s the most favors s, then every conformational background favors s' over s and therefore s is not a member of the GMEC. Using the notation from Section 2.5, the criteria for the elimination of s can

be expressed by the following inequality:

$$\mathcal{E}_{self}(s) + \sum_{u \neq v} \min_{t \in \mathcal{S}(u)} \mathcal{E}_{pair}(t, s) > \mathcal{E}_{self}(s') + \sum_{u \neq v} \max_{t \in \mathcal{S}(u)} \mathcal{E}_{pair}(t, s') \tag{2.8}$$

where each rotamer set $\mathcal{S}(u)$ contains only those rotamers for residue u that have not yet been eliminated. Through iterative applications of the theorem, the conformational backgrounds favoring or penalizing various rotamers change. A rotamer that could not be eliminated in the first round of eliminations might be eliminatable after some other rotamers have been eliminated. An iterative method based on this theorem would repeatedly attempt to eliminate each rotamer until it could eliminate none; at this point either a single rotamer for each residue remains, and the problem is solved, or an irreducible sub problem remains. If this sub-problem is small enough, then it can be treated with brute-force enumeration.

The time it takes to find the conformational background that penalizes s' the most, as well as the time it takes to find the conformational background that favors s the most, scales linearly with the number of rotamers being considered at all other residues in the protein; $\sum_{u\ inV, u \neq v} |S_u|$. Notice that the time does not scale exponentially with the number of other rotamers at all other residues; there are after all $\prod_{u\ inV, u \neq v} |S_u|$ conformational backgrounds available for consideration. The key to the speed is the pairwise decomposability of the energy function; the rotamer assigned to each residue in the conformational background that penalizes s' the most can be computed independently of the rest of the rotamer assignments to the other residues. The same is true, of course, for the rotamers in the conformational background that favors s the most. Pairwise decomposability is required of any energy function to be solved by iterative applications of the dead-end elimination theorems.

Desmet *et al.*'s theorem was expanded to include rotamer pairs (Lasters and Desmet, 1993). The idea is that, not only does one maintain a list of rotamers that have not yet been eliminated, one also maintains a list of rotamer pairs that have not yet been eliminated. As one applies the theorem to a rotamer pair and finds it is not a member of the GMEC, one removes that rotamer pair from the list. The new theorem, called the *fuzzy ended* theorem, considers the rotamer pair s_1 and s_2 on residues u and v in comparison with an alternate rotamer pair s_1' and s_2'. The theorem states that the pair

(s_1, s_2) is not a member of the GMEC if the following inequality holds:

$$\begin{array}{rcl} \mathcal{E}(s_1) + \mathcal{E}(s_2) + \mathcal{E}(s_1, s_2) & > & \mathcal{E}(s_1') + \mathcal{E}(s_2') + \mathcal{E}(s_1', s_2') \\ + \sum_{w \neq u,v} \min_{t \in \mathcal{S}(w)} \mathcal{E}(t, s_1) & & + \sum_{w \neq u,v} \max_{t \in \mathcal{S}(w)} \mathcal{E}(t, s_1') \\ + \sum_{w \neq u,v} \min_{t \in \mathcal{S}(w)} \mathcal{E}(t, s_2) & & + \sum_{w \neq u,v} \max_{t \in \mathcal{S}(w)} \mathcal{E}(t, s_2') \end{array} \quad (2.9)$$

where both the left and right half of the inequality exclude any rotamer t that has been eliminated, and where the left half of the inequality may exclude any pair (t, s_1) or (t, s_2) from the min operations if that pair has already been eliminated; the right half of the inequality may not exclude eliminated pairs. This discrepancy between the left and the right hand sides earns this theorem its *fuzzy-ended* name. The rotamer pairs eliminated by this theorem can also be excluded from the left-hand side of Inequality 2.8 for the elimination of single rotamers.

Goldstein formulated a more lenient elimination criteria for rotamer singles (Goldstein, 1994). He noted that the alternate rotamer s' had only to prove more favorable than s in each individual conformational background. Instead of searching for two conformational backgrounds, the one favoring s the most and the one penalizing s' the most, he searches instead for the single conformational background that favors s' over s the least. If in this background, s' is still preferred, then s may be eliminated from consideration. Specifically, s' eliminates s if

$$\mathcal{E}(s') - \mathcal{E}(s) + \sum_{u \neq v} \max_{t \in \mathcal{S}(u)} (\mathcal{E}(t, s') - \mathcal{E}(t, s)) < 0 \quad (2.10)$$

where again the rotamer set $\mathcal{S}(u)$ contains only un-eliminated rotamers, and any rotamer t may be ignored if the pair (t, s) has been eliminated; whether or not t may be ignored is independent of the elimination status of pair (t, s'). Goldstein's theorem is weaker in the sense that any rotamer pair that satisfies Inequality 2.8 also satisfies his inequality; it is more powerful in the sense that it can eliminate rotamers even after Inequality 2.8 fails. His inequality ends up more expensive to include in an iterative method for rotamer elimination than Inequality 2.8, not because once s and s' have been chosen it takes more time (both Inequalities 2.8 and 2.10 take $\Theta(\sum_{u\ inV, u \neq v} |S_u|)$ time to compute), but rather because the conformational backgrounds for s and s' may not be computed independently. If s is chosen, then the iterative method for rotamer elimination chooses from the rest of $\mathcal{S}_v$ for alternate rotamers s', and for each combination of s and s', check if s and s' satisfy Inequality 2.10.

The elimination of state singles and state pairs lead to the generalized dead-end elimination theorem (Looger and Hellinga, 2001) which eliminates arbitrarily large rotamer *clusters* (rotamer triples, rotamer quadruples, etc.) if it can find a winning replacement cluster. Of course, the complexity of eliminating larger and larger rotamer clusters increases to the complexity of the original problem. Looger *et al.* (Looger and Hellinga, 2001) have reported that the generalized DEE algorithm has succeeded on problems with state spaces as large as 10^{1044}. In their numbers, however, Looger *et al.* include rotamers that collide with the background, which are pruned from consideration before DEE begins; that is, these rotamers are never proven not to be in the GMEC by DEE, so it is somewhat misleading to say that DEE solves problems of this size. It is likely that DEE could have eliminated such rotamers. What is most misleading is the idea that Dezymer ever computes and stores as many rotamer-pair energies for as many rotamers as are implied to exist in a problem that large

2.6.2 Simulated Annealing

In 1953, Metropolis *et al.* introduced a stochastic minimization technique designed to bias sampling in the areas of low energy in such a way that the distribution of sampling probability for states converges to the Boltzmann distribution (Metropolis et al., 1953). The minimization proceeds at some temperature, T. The minimizer walks through the search space upon which the energy function is defined. At its current point p, it computes the difference in the energy function induced if it were to step to a nearby point q, namely, $\Delta E = E_q - E_p$. The minimizer then steps to point q with probability

$$\min(1, e^{\frac{\Delta E}{kT}})$$

where k is the Boltzmann constant. If q is at a lower energy than p, then the minimizer always steps. If the minimizer does not step to q then it stays put for that iteration and considers another point q' in the next iteration.

Simulated annealing uses the Metropolis acceptance criteria to accept or reject alternate locations in the search space while it gradually lowers its operating temperature. At the beginning of the optimization, when temperature is the highest, the annealer accepts almost any move through the state space; as the annealer cools, it biases acceptance towards points at lower energy.

In side-chain placement, a point in the state space corresponds to an assignment of rotamers to all of the residues in the protein. An alternate location in the search

space that the annealer would consider stepping to from the current location is formed from the current location by replacing a single rotamer at a single residue with some other rotamer. The change in energy between the two rotamer assignments, ΔE, can be computed by simply subtracting the energies for the two rotamer assignments.

It is important to note that simulated annealing places no restrictions on the nature of the energy function, unlike DEE, which requires that the energy function be pairwise decomposable. The only thing that simulated annealing requires of the energy function is that given an assignment of rotamers to all residues, the energy function be computable; the change in energy can be computed by evaluating the total energy for the protein for the two rotamer assignments and subtracting. If one knows the nature of the energy function, for instance, if one knows it to be pairwise decomposable, then one can possibly compute ΔE more efficiently than computing the total energy of the two rotamer assignments and subtracting. Since the same simulated annealing algorithm can be used to optimize all sorts of energy functions, and because knowing something about the energy function can produce faster ΔE calculations, the annealer and the representation of the energy function should be separated: I discuss how I have separated the annealer from the energy function in Rosetta in Chapters 5 and 8.

Simulated annealing samples the state space without searching it exhaustively. For this reason, it provides no quality guarantees on the rotamer assignments it produces. What has proven fortunate is that simulated annealing for the side-chain-placement problem often does find the GMEC, and when it does not, it finds another rotamer assignment whose energy is close to that of the GMEC.

Chapter 3

Challenges in Protein Design

I see three main challenges in protein design: the necessity of broad conformational sampling, the difficulty in quantifying stability, and the inherent complexity of the side-chain-placement problem. The first stems from the opposing demands that stable proteins be tightly packed, yet collision-free. The second stems from the non-pairwise decomposable nature of stability; that the stability of a particular structure is not dependent only on its own energy, but the energies of all other structures that it is not. The third stems from the nature of the problem, and the inability to disentangle which rotamer is optimal for a particular residue from the rotamers assigned to every other residue in the protein. Each challenge brings additional technical challenges with it.

Each section in this chapter delves into one of these three challenges and describes how the work presented in this thesis addresses these challenges.

3.1 Tight, Collision-Free Side-Chain Packing

Stable protein structures are tightly packed and collision-free. Since proteins are only marginally stable, they cannot afford the extreme energy penalties induced by atomic collisions. If one were to synthesize a protein from a design that included collisions, the protein would not adopt the structure from the design; it might adopt a similar structure, but the structure will have resolved any collisions. To pack protein cores tightly, designers have two options, both of which they have tried: pack the cores tightly by allowing slight collisions, and hope with crossed fingers that the synthesized protein will adopt a structure that resembles the design; or sample conformation space broadly to find tightly packed, collision-free structures.

The hope in designing side-chain packing that includes slight collisions is that the

actual structure the synthesized protein adopts when placed in water will resemble the structure that was designed. If a pair of atoms overlap by 0.4 Å, they each only have to move by 0.2 Å to resolve the overlap. Designers can permit slight overlap in their designs by shrinking the van der Waals radii. With smaller radii, or "soft repulsion," designers can create a tightly packed structure that almost fits. In Ponder and Richard's landmark paper, they shrank their van der Waals radii by a large amount – 0.375 Å to permit 0.75 Å of overlap (Ponder and Richards, 1987). Hurley, Baase, and Mathews used these same shrunk radii in an early redesign of the core of the bacteriophage T4 lysozyme (Hurley et al., 1992). Through a series of mutagenesis and computational experiments, Dahiyat and Mayo decided to use a van der Waals radius shrinking factor of 0.9 (Dahiyat and Mayo, 1997a). Though I do not know how exactly how the Hellinga lab shrinks their van der Waals radii, the TNT-binding R3 design contains collisions (Looger et al., 2003). The R3 protein binds TNT more tightly than any other designed receptor protein; however, the R3 protein could not be crystallized (Hellinga, personal communication).

The largest problem with soft repulsion is that it is possible to create a set of collisions that cannot be relaxed away, either because the direction of motion that resolves the sphere collision cannot be created through small dihedral flexes or because a sphere collides on all sides and no motion is able to relieve all the collisions. The evidence that shrunk van der Waals radii do not produce viable designs is visible in relief: one has to perceive what is not present in the literature – the crystal structures of successful designs generated with shrunk radii. Since proteins must be well ordered in order to form crystals, the inability to form crystals from a designed protein is evidence that the protein is not as stable and rigid as are naturally occurring proteins.

Disappointingly, the input scaffold and a small rotamer library often does not allow a tight, collision-free packing of side chains. To find well packed proteins, a designs must search wide regions of conformation space. There are two options for allowing greater conformational freedom. The first and easiest is to push more conformational freedom into side chains; the second and much more challenging is to allow backbone conformational freedom.

Side-chain conformational freedom can be expanded by increasing the size of the rotamer libraries used in design. A pair of rotamers that just barely collide might not collide at all if their χ dihedral angles were relaxed by two or three degrees. In addition to dihedral angles chosen from the centers of the χ distributions, designers include in their rotamer libraries extra dihedral samples in the range of $\pm 10^\circ$ of the lowest-energy

values, usually ± one standard deviation (σ) of the dihedral angle's observed values. In Rosetta, a single residue might have 2000 rotamers to choose from; in Dezymer (Homme Hellinga's design software), a residue has upwards of 17,000 rotamers to choose from (Loren Looger, personal communication). These rotamer library sizes greatly exceed the 67 member rotamer library first published (Ponder and Richards, 1987). Of course, the extra dihedral samples produce sub-optimal side chain conformations; the hope is that these extra rotamers with higher internal strain are able to compensate for this strain by forming more favorable interactions.

The effect of large rotamer libraries on protein design can be observed both in theory and in practice. In theory, larger rotamer libraries represent a challenge in that running time of the optimization algorithms scale as a function of the number of rotamers. Even when using simulated annealing, the more rotamers, the larger the search space, and the more time that needs to be committed to rotamer optimization. Large rotamer libraries are especially problematic for dead-end elimination. Although the amount of time required for each attempted rotamer elimination scales as a polynomial of the number of rotamers used in the design (with higher degree polynomials for the increasingly complex theorems), the real problem with using large rotamer libraries is that, the more rotamers there are, the harder it is to prove a rotamer can be eliminated. With large rotamer libraries, the number of conformational backgrounds grows so that a rotamer that looked poor when there were fewer conformational backgrounds looks better with more. In a small rotamer library design, if a rotamer s on residue v collided a little with every rotamer t on a nearby residue u, then dead-end elimination could readily prove s was sub-optimal; almost any other rotamer s' on v could be used to show that s is not in the GMEC. Adding an extra rotamer t' to u where t' does not collide with s makes it more difficult to prove that s is sub-optimal: the conformational background would be less likely to favor s' over s. DEE bogs down in large rotamer library designs because fewer attempted rotamer eliminations succeed. The simple, fast theorems prune few rotamers, and the complicated, time consuming theorems have to be resorted to.

In practice, larger rotamer libraries represent a challenge in that they require more memory and more time. Designers often pre-compute their pairwise-decomposable energy functions between all rotamer pairs and store these *rotamer-pair energies* in large tables. In general the number of rotamer-pair energies scales quadratically with the number of rotamers per residue (and linearly with the number of residues being designed for short-ranged energy functions; quadratically in the number of residues for

long-ranged energy functions). The four gigabyte memory limit on 32-bit processors means that designs with more than 45 thousand rotamers have more rotamer pairs than address space to store their interaction energies. To run larger designs requires either a clever representation of the rotamer-pair energies or a 64-bit processor.

Moreover, if the design software optimizes using simulated annealing, then realistically, all of the rotamer-pair energies must fit in physical memory. Simulated annealing retrieves the rotamer-pair energies in a random fashion; if the energies spill out of physical memory onto the hard drive, the annealing algorithm thrashes as the system continuously swaps pages of memory to and from disk. In Chapter 5, I describe a data structure that I have introduced into Rosetta that reduces the amount of memory used during side-chain placement by 12%.

Finally large rotamer libraries increase the amount of time consumed in rotamer-pair-energy calculation. Rotamer-pair-energy calculation scales quadratically with the number of rotamers per residue. In Rosetta, which relies on a rapid simulated annealing method for optimizing side-chain placement, rotamer-pair-energy computation dominates the running time in any single invocation of the side-chain-placement subroutine. The more rotamers used in design, the more patient the designer must be in waiting for output. In Chapter 7, I describe a rapid algorithm for computing rotamer-pair energies that proceeds 3.5 times faster than the previous technique.

Using larger rotamer libraries does not completely resolve the problem of finding stable, tightly packed protein cores. First, including more rotamer samples on a fixed backbone does not guarantee that a tightly packed collision-free placement of side chains exists. Second, designs that include many rotamers that are far away from their energetically optimal χ angles will not be as stable as designs that include rotamers closer to the lowest energy dihedral angles. Although it is certain that side-chain dihedrals do deviate from their optimal values, it is also certain that backbones move, too. Additional χ_1 samples for amino acids with large side chains take advantage of the lever arm effect; small changes in χ_1 produce large changes in atom location at the ends of side chains. If flexing a χ_1 dihedral can have a profound effect on the location of distal side-chain atoms, then swinging the Cα-Cβ bond vector by moving the backbone will also produce this large effect.

Backbone flexibility has proven an important part of successful (crystallizable) designs. The designs that have resulted in successful crystal structures have included not only hard-repulsion (full van der Waals radii, as opposed to soft repulsion) but also many sampled backbone conformations (Harbury et al., 1998; Kuhlman et al., 2002;

Kuhlman et al., 2003). The techniques for modeling backbone flexibility however have been limited to solving multiple instances of fixed-backbone side-chain placement, each instance deriving from a different backbone conformation.

Adding local backbone flexibility exponentially increases the size of the design space. If a designer wanted to model local backbone flexibility in the eight core β-strands of a TIM barrel (Figure 3.1) while redesigning the protein, then if each strand were allowed 10 different conformations, the designer would have to solve 1 billion (8^{10}) instances of fixed-backbone side-chain-placement. The designer might be able to prune some backbone combinations in a branch-and-bound-like fashion but at the risk of missing good combinations of backbones if not done correctly. However, even if the designer can avoid 99% of the backbone combinations, they are still left with ten million instances of the problem to solve. Side-chain placement would have to proceed very rapidly.

Frustratingly, the pair energy computations (the bottleneck in Rosetta's side-chain placement subroutine), in these million SCPP instances would be repeated between many fixed-backbone-optimization instances. For example, consider two residues i and j in two fixed-backbone instances of SCPP where the backbones for i and j did not move relative to one another, but rather where the instances differ at some other part of the backbone. The rotamer-pair energies for i and j do not differ between the two instances. The frustration in this realization is that there is not yet a good way to preserve these rotamer-pair energies between fixed-backbone problem instances. Even if the rotamer optimization phase were fast enough to handle one million instances, the rotamer-pair-energy-computation phase is not fast enough and includes redundant computations.

Chapter 7 describes the incorporation of local backbone flexibility into the side-chain-placement optimization itself. The incorporation makes efficient use of rotamer-pair-energy calculations so that the 100 million fixed-backbone problem instances described for the TIM-barrel protein could be performed in a single design trajectory. The new software I have written is currently in use in enzyme design for TIM-barrel proteins, as well as for other proteins.

3.2 Expressing Protein Stability

The tradition in design of relying upon pairwise-decomposable energy functions limits the kind of biochemical understanding that can be incorporated into the design process. If, as I suggested in Chapter 1, protein design takes its scientific worth from its insis-

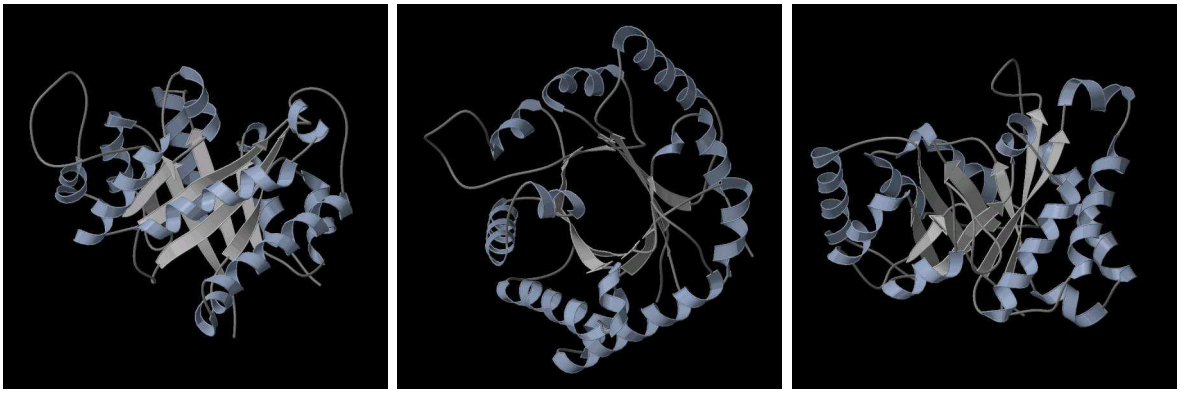


Figure 3.1: *A TIM Barrel Protein.* PDB ID: 1TIM. The core of the TIM barrel is composed of eight parallel β-strands, forming a cylindrical β-sheet. Because the sheet wraps around on itself, it does not leave any backbone hydrogen bonding groups unsatisfied and exposed to solvent to form unwanted aggregates. A helical region connects each β strand to the next. The middle view is through the center of the barrel with the β-strands coming out of the page and the α-helical connections between the strands going into the page. TIM barrels appear frequently; often in enzyme proteins. One of the design goals in the ongoing Protein Design Project is the redesign of a TIM barrel protein to perform novel catalytic activity. Images created with Mage and Render3D (Merritt and Bacon, 1997).

tence that designers explicitly state and test the current biochemical understanding of protein energetics, then the requirement that energy functions be pairwise decomposable detracts from design's worth. Current biochemical understanding of protein stability cannot be expressed in a pairwise decomposable fashion. The *rule* (Fleming and Rose, 2005) that "buried hydrogen bonding groups form intra-molecular hydrogen bonds" cannot be captured as a sum of pair interactions.

For the most part, protein designers have avoided the problems associated with satisfying buried hydrogen bonding groups by packing cores with hydrophobic groups alone. If there are no buried hydrogen bonding groups, then none end up unsatisfied. There are two ways to prevent hydrogen bonding groups from ending up in protein cores: explicitly forbid them by only considering hydrophobic amino acids for buried residues (Dahiyat and Mayo, 1997b), or implicitly forbid hydrophilics from the core by including a solvation model (e.g. the Lazaridis-Karplus solvation model [1999]) that penalizes the approach of any atom towards a hydrophilic group (Kuhlman et al., 2002; Kuhlman et al., 2003) and thereby penalizes the presence of a hydrophilic group in the core. However, some design instances require burial of hydrogen bonding groups: for instance in enzyme design, the ligand that needs to be buried inside the enzyme's active site will likely contain hydrogen bonding groups. If a ligand's buried hydroxyl group is not satisfied by a hydrogen bond to the protein, the stability of the enzyme/ligand complex will be lower, diminishing the effectiveness of the enzyme.

There are two key exceptions to the observation that designers prohibit hydrophilic amino acids from the core, and they both are involved in ensuring hydrogen bond satisfaction for buried ligands (Looger et al., 2003; Dwyer et al., 2004). In both of these cases, the designers filtered the designed structures that resulted from the optimization of a pairwise decomposable energy function in a post-processing step. The filters examined, among other things, the number of buried unsatisfied hydrogen bonding groups. They discarded designs that contained several unsatisfied hydrogen bonding groups. The problem with this approach, as demonstrated in Chapter 8, is that optimizing according to one function and filtering according to a second function produces sub-optimal designs.

The effect of solvent on a protein's stability is non-pairwise decomposable for a second reason. If solvation free energy is understood in Dill's framework as hydrophobic groups ordering water around them so that the waters' hydrogen atoms can hydrogen-bond with the rest of the solvent (Dill, 1990), then the solvation free energy of a protein conformation is a function of the amount of solvent-accessible hydrophobic surface area

and the solvation stability of a protein conformation is a function of the amount of buried hydrophobic surface area. While significant effort has gone into expressing hydrophobic surface area burial in terms of rotamer-pair burials (Street and Mayo, 1998; Zhang et al., 2004), these efforts nonetheless produce artifacts suggesting that hydrophobic surfaces near the protein's surface are not fully buried when they in fact are and suggesting that hydrophilic surfaces are buried when they remain sufficiently solvent accessible.

These artifacts do not show up in core redesigns, when everything is completely solvent inaccessible anyway; however, they have a dramatic effect on the design of protein/protein interfaces. Protein interfaces bury significant amounts of hydrophobic surface areas (Janin and Chothia, 1990; Waksman et al., 1993); they also contain many buried hydrogen bonds. The pairwise decomposability requirement has hindered interface design.

Incorporation of non-pairwise decomposable energy functions into design software has proven difficult. For starters, the dead-end elimination theorems require the energy function they optimize be pairwise decomposable. Moreover, frameworks for optimization by simulated annealing that rely on the precomputation and tabulation of pair energies cannot incorporate non-pairwise decomposable energy functions as simply another term.

This dissertation describes the optimization of one non-pairwise decomposable energy function with dynamic programming in the context of the hydrogen placement problem (Chapter 4), the optimization of a different non-pairwise decomposable energy function into simulated annealing for protein design (Chapter 8), and the groundwork I have laid in the Rosetta molecular modeling program for the optimization of future non-pairwise decomposable energy functions (Chapter 5).

3.3 SCPP's Complexity

Finally, optimizing side-chain placement is NP-hard. I would like to have proven in this thesis that P=NP, and to have built a polynomial time solver for the side-chain-placement problem; however, I have not been able to. Instead, in Chapter 4, I put another nail into the tractability coffin of side-chain placement, by proving that this problem is like other NP-hard problems in admitting a dynamic programming solution with time and space requirements exponential in the treewidth of the graph corresponding to each problem instance. While this result at first sounds encouraging, the

treewidth of most problem instances in protein design is large, making dynamic programming infeasible.

What I have found encouraging is that simulated annealing performs extremely well at the side-chain-placement problem. In implementing both the exact techniques of dynamic programming and dead-end elimination, I have not found any cases where Rosetta's simulated annealing protocol produced poor side-chain placements in comparison to the optimal placement produced by either of the exact techniques.

The complexity of the side-chain-placement problem caries over to the hydrogen-placement problem; however, the problem instances that arise in hydrogen placement have dramatically smaller treewidths. This dissertation includes a description of dynamic programming in the hydrogen placement problem; the dynamic programming algorithm presented here is now part of the Richardson's program for hydrogen placement, REDUCE.

Chapter 4

Dynamic Programming on an Interaction Graph

This chapter[1] defines the interaction graph as a model for the side chain placement problem. As a model, the interaction graph connects the side-chain-placement problem to a class of NP-hard problems for which certain problem instances can be solved in polynomial time (Bodlaender, 1988; Arnborg and Proskurowski, 1989). The complexity for a single problem instance for this class of graph problems depends on a property of the instance's graph, a property called *treewidth*. The treewidth of the problem instance often sits in the exponent of its complexity; those instances with treewidths less than some constant can be solved in polynomial time.

The chapter defines a novel version of dynamic programming specifically tailored to the kinds of problems present in protein design. This version uses less memory and runs in less time at the cost of introducing a guaranteed level of approximation to the energy function. I call it *algorithm adaptive dynamic programming*.

Finally, the chapter provides an interaction graph formulation of the hydrogen-placement problem that includes a non-pairwise decomposable energy function. It shows that dynamic programming similarly solves this problem. The dynamic programming algorithm described here is now in use and being distributed inside software

[1]This chapter represents a combination of three publications. The sections on dynamic programming in the side-chain placement problem and adaptive dynamic programming was published in the Pacific Symposium on Biocomputing (PSB) 2005. This publication was in collaboration with Brian Kuhlman and Jack Snoeyink. The sections on dynamic programming in the hydrogen-placement problems were published initially in the Workshop on Experimental Algorithms (ALENEX) 2004. This publication was in collaboration with Yuanxin Liu and Jack Snoeyink. Following ALENEX'04, the paper was invited for submission into the Journal of Experimental Algorithms, and awaits a second round of review. Xueyi Wang contributed to this expanded publication.

for hydrogen placement (Word et al., 1999b).

4.1 Graphs, Hypergraphs, and Treewidth

A *graph* $G = \{V, E\}$ consists of a set of *vertices* V and a set of *edges* $E \subset V \times V$. Vertices u and v are *adjacent* if the pair $(u, v) \in E$. The *neighbors* of a vertex v are those vertices adjacent to v, and the *degree* of a vertex is the number of neighbors it has. A *hypergraph* is a generalization of a graph in which an edge (sometimes called a *hyperedge*) can contain any number of vertices of V. The *degree* of a hyperedge is the number of vertices it is incident upon. Figure 4.3b contains an example hypergraph with the vertices drawn as points and the hyperedges drawn as curves encircling them.

A *multi-graph* is a graph in which multiple edges may be incident upon the same vertices, and yet be distinct; each edge must be identified by its set of vertices and an index that identifies which edge in the set it is. A *multi-hypergraph* is a hypergraph with repeated hyperedges.

The remainder of the discussion on graphs relies on some set notation. Breifly, for the set S and the set T, the cardinality of S is represented by $|S|$ and the set difference of S and T is represented by $S \setminus T$.

The *treewidth* of a graph can be defined in one of two ways: I present both, as both definitions prove useful in the analysis to come.

The first definition of treewidth arises from the definition of a class of graphs called *partial k-trees*. A *k-tree* is a recursively defined graph: a $(k+1)$-clique is a k-tree, and any graph formed from a k-tree T by connecting a new vertex to each vertex of a k-clique of T is also a k-tree. The 1-trees correspond to ordinary trees (connected graphs with $|V| = |E| + 1$), except that the graph consisting of a single vertex would not be considered a 1-tree. 2-trees look like triangles that have been glued together at their edges (Figure 4.1). 3-trees look like tetrahedra that have been glued together at their faces. A *partial k-tree* is a k-tree from which any number of edges have been dropped. The partial 1-trees are forests. Notice that for a graph G with $|V| > k + 1$, if G is a partial k-tree, it is also a partial $(k+1)$-tree. The treewidth of a graph G is the smallest k such that G is a partial k-tree.

The second definition of treewidth arises from the definition of a *tree-decomposition* of a graph. The *tree decomposition* of a (hyper)graph $G = \{V, E\}$ is a tree T whose vertices $X = \{X_1, X_2, \ldots, X_m\}$ are subsets of V that satisfy the following three properties:

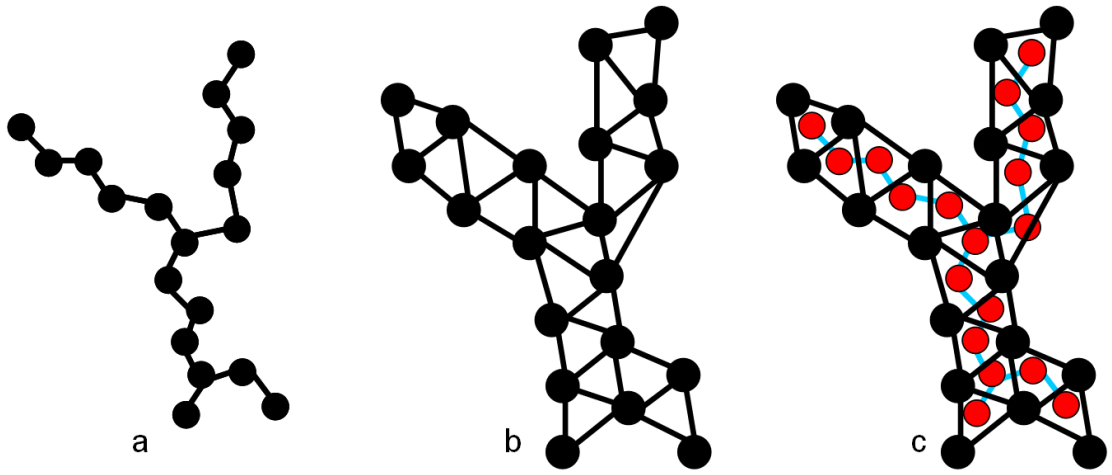


Figure 4.1: *A Treewidth-2 Graph.* The graph in (b) is a 2-tree, constructed by adjoining triangles (3-cliques) at their edges (2-cliques). Graph (c) demonstrates the relationship between the tree in (a) and the 2-tree in (b): each triangle in (b) corresponds to a vertex in (a), and each edge shared by two triangles in (b) corresponds to an edge in (a).

1. The union of sets $\bigcup_{1 \leq i \leq m} X_i = V$.
2. Each edge $e \in E$ is in some set: $e \subseteq X_i$ for some $1 \leq i \leq m$.
3. Each vertex $v \in V$ occupies a connected part of tree T: for any X_j on a path in T from X_i to X_k, the intersection $X_i \cap X_k \subseteq X_j$.

The first two rules say: 1) the tree decomposition must capture all vertices in the graph, and 2) the tree decomposition must capture the edges in the graph. The third rule says that for any vertex $v \in V$ the induced subgraph of T formed by taking all vertices $\{t \in X | v \in X_t\}$ is connected. That is, for any vertex v, if $v \in X_i$ and $v \in X_k$, then $v \in X_j$, for all j on the path from i to k.

The treewidth of a tree decomposition is $\max_i |X_i| - 1$. There are often many ways to create tree decompositions for a single graph. A tree T with only a single vertex $X_1 = V$ is a tree decomposition of G with a treewidth of $|V| - 1$. Such a tree decomposition, however, is uninformative. The treewidth of a graph G is the minimum treewidth over all possible tree decompositions of G. Figure 4.2b illustrates a treewidth-3 tree decomposition of the graph in Fig. 4.2a.

Certain tree decompositions prove especially useful for dynamic programming as they define a "vertex elimination order." I describe vertex elimination orders in greater detail in Section 4.4. These tree decompositions have been called *nice tree decompositions* due to their usefulness (Bodlaender and Kloks, 1993). A *rooted tree decomposition*

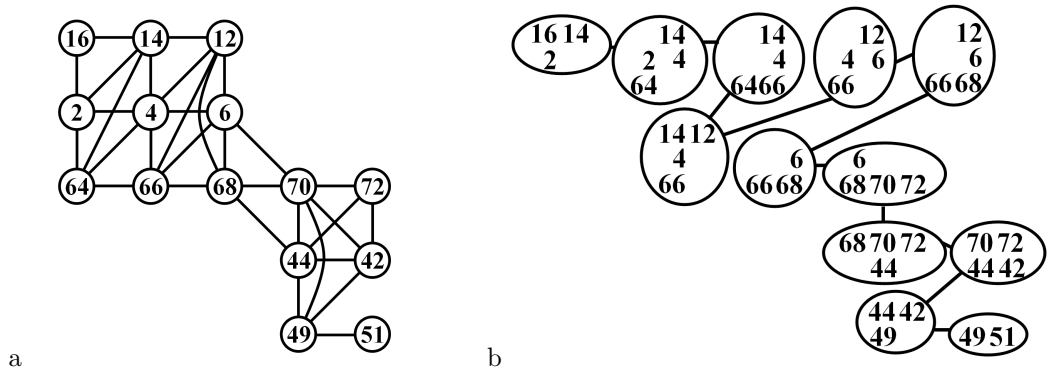


Figure 4.2: *A Tree Decomposition.* The graph in (*a*) represents the interactions between fifteen residues on the surface of ubiquitin's beta sheet, as shown in Figure 4.9; vertices are labeled with their residue numbers. The tree in (*b*) is a tree-decomposition of (*a*). Each tree node contains the vertex label set, X_i. Since $\max_i(X_i) = 4$, this tree-decomposition has a treewidth of 3.

is a tree decomposition where one node $t \in X$ is the *root,* and all edges in T are directed such that they point away from the root. One vertex i is said to be the *parent* of another vertex j if i and j are connected, and j is further from the root than i. A nice tree decomposition is a rooted tree decomposition such that for any child/parent pair i and j, there is exactly one vertex in X_i that is not in X_j, that is the size of the set difference $|X_i \setminus X_j| = 1$. Any tree decomposition can be converted into a nice tree decomposition in linear time without changing the treewidth.

Lemma 2 *A tree decomposition T with treewidth w of the graph $G = \{V, E\}$ can be converted in $O(w(|T| + |V|))$ time into a rooted tree decomposition with treewidth w in which the vertex sets X_i satisfy $|X_i \setminus X_j| = 1$ whenever X_j is the parent of X_i.*

Proof: I give a constructive proof. Begin by choosing an arbitrary node of T as the root. Now, consider any node X_i and parent X_j for which $|X_i \setminus X_j| \neq 1$. If $|X_i \setminus X_j| = 0$, then delete X_i and connect any children of X_i to the parent X_j. Otherwise take one vertex from the set difference, $v \in X_i \setminus X_j$, and create a new node, $Y = (X_i \cap X_j) \cup \{v\}$ between X_i and X_j. Node X_i now has the desired property. Tree T is still a tree decomposition: the first two properties for tree decompositions hold trivially, and the third property holds because $(X_i \cap X_j) = (X_i \cap Y \cap X_j)$. The treewidth of the new decomposition is still w since $|Y| < |X_i| \leq (w + 1)$. Recurse on Y until the desired property holds.

This conversion takes $O(w(|T|+|V|))$ time since each vertex of T must be visited at least once, and each new vertex Y added to T corresponds to exactly one vertex

of G. The cost of visiting a node of T and the cost of inserting a new node Y into T are both linear in the treewidth w. Assuming that nodes of T have their vertex lists sorted, then the set difference operation proceeds in $O(w)$ time. Therefore the time spent visiting nodes of T is $O(w|T|)$. A new node Y is added to T when some vertex v was contained in X_j and not in X_i. By the third property of tree decompositions, since $v \notin X_i$, v cannot be contained in any node of T that is not part of the subtree rooted at X_i. After Y's addition, v is not contained in any other node of T other than Y whose parent does not contain v. Since each vertex in V can induce the creation of one new node Y, the number of nodes added to the tree decomposition is $O(|V|)$. The construction of the new node Y is linear in the number of vertices it contains. and, the number of vertices of V contained by each new node Y is less than $w + 1$. Therefore the cost for vertex addition is $O(w|V|)$. ∎

Nice tree decompositions define a partial order on V called an *elimination order*. The elimination order can be iteratively determined from T as follows: for any leaf node j in T, 1) if j is not the root, then write down the single vertex in $X_j \setminus X_i$ where i is j's parent, and remove j from T; 2) if j is the root, then write down the vertices in X_j in any order.

The elimination order produced by the nice tree decomposition of a graph G provides an intuitive relationship between the two definitions of treewidth: the elimination order for G is the reverse order in which to build a k-tree (with the building rule of adding a vertex and connecting it to a k-clique) from which edges can be dropped to form G as a partial k-tree.

4.2 The Side-Chain-Placement Problem

The side-chain-placement problem (SCPP) arises in many areas of computational structural biology. 1) It arises any time a biochemist treats a backbone separately from its side chains. That is, it arises when a biochemist manipulates a backbone structure (if she translates or rotates part of the backbone as a rigid body, or if she flexes one or more dihedral angles) independently of its side chains, and then then seeks the optimal side-chain placement to complete the structure. 2) It also arises when a biochemist strips some or all of the side chains from the backbone of one protein to use the backbone as a structural template for another protein.

The first situation, in which a backbone is modified independently of its side chains, is common in protein docking, where at each step a biochemist moves one protein

relative to the other and then optimizes the side chains (Gray et al., 2003). The first situation is also common in protein folding, where at each step the biochemist modifies some number of backbone dihedrals and then optimizes side-chain conformation. This second situation, in which a backbone from one sequence serves as a template for another sequence, is common in protein structure prediction by protein threading, wherein the biochemist imposes the amino acid sequence of a protein with an unknown fold onto the backbone of another protein with a known fold; the energy of the optimal placement of amino acids suggests whether that sequence is compatible with that fold. The second situation is also common in protein design, where a biochemist builds from the known folded structure of a protein at a few key residues to form a new amino acid sequence that adopts the original backbone structure and include new functionality.

I restate the side-chain-placement problem from Section 2.5 here:

Problem 1 *The Side-Chain-Placement Problem. Given*

- *a fixed backbone (the* scaffold*) with a set of fixed residues,*
- *a set of residues to change (the* molten residues*),*
- *a list for each molten residue of the amino acids accessible to it,*
- *a rotamer library, and*
- *an energy function,*

find the assignment of rotamers to the molten residues that minimizes the energy function.

As described in section 2.5, designers almost always restrict themselves to pairwise decomposable energy functions. The pairwise decomposability of the energy function means that the minimization may be expressed as minimizing the sum of two types of energies: 1) the sum of the internal rotamer strain and the pair interaction energies of a rotamer with all of the background residues, for a single molten residue, $\mathcal{E}_{self}$, and 2) the pair interaction energies for a pair of molten residues, $\mathcal{E}_{pair}$. The interaction energies for pairs of background residues do not need to be considered in this minimization: the pairwise decomposability means that their interactions are independent of the rotamers assigned to the molten residues, and the interaction energies between pairs of molten residues are independent of the background/background interactions. Given an assignment of rotamers to the molten residues, S_V, the energy of the system

can be expressed as

$$\sum_{v} \mathcal{E}_{self}(S_v) + \sum_{v_1<v_2} \mathcal{E}_{pair}(S_{v_1}, S_{v_2}). \tag{4.1}$$

The side-chain-placement problem is to minimize this sum. There are two natural subproblems: the *rotamer-relaxation problem*, where the amino acid sequence is held fixed, and the *redesign problem* where the amino acid sequence is not held fixed.

In Rosetta, $\mathcal{E}_{pair}$ is the sum of five terms: a Lennard-Jones attractive term, a Lennard-Jones repulsive term, the Lazaridis-Karplus solvation model, the statistically-derived hydrogen bonding term, and a statistically derived potential based on the frequency of observing amino acids with a certain Cβ distances. Importantly, Rosetta does not contain an electrostatic potential based on Coloumb's law (Bradley et al., 2005). Generally, if the energy function includes long range interactions, all atom pairs interact directly; if the function does not contain long range interactions, then distant atom pairs do not interact directly, though they interact indirectly through their interactions with intervening atoms.

4.3 Interaction Graph Formulation of SCPP

I define an *interaction graph*, a multi-hypergraph $G = \{V, E\}$ that captures energy Expression 4.1. Each molten residue is represented by a vertex, $v \in V$, and each vertex carries a state space $\mathcal{S}(v)$, which is the set of rotamers that can be placed at v.

In this hypergraph, there exists a (hyper)edge for each vertex and each vertex pair. Each (hyper)edge $e \in E$ carries a *scoring function* $f_e(S)$ that maps the Cartesian product of the state spaces of its vertices to the reals: $f_e: \prod_{v \in e} \mathcal{S}(v) \rightarrow \Re$. Specifically, each vertex, $v \in V$, has a corresponding degree-1 hyperedge, $\{v\} \in E$, with a hyperedge scoring function $f_{\{v\}}(S_v) = \mathcal{E}_{self}(S_v)$, and each pair of vertices, $v_1, v_2 \in V$, has a corresponding degree-2 hyperedge, $\{v_1, v_2\} \in E$, with a hyperedge scoring function $f_{\{v_1,v_2\}}(S_{v_1}, S_{v_2}) = \mathcal{E}_{pair}(S_{v_1}, S_{v_2})$.

The graph can omit an edge e when f_e is zero under all possible state assignments to the vertices of e. To ignore small contributions to the energy function, the graph can also omit edges for which $|f_e|$ is always less than some chosen *magnitude threshold*, μ. Figure 4.3a shows an interaction graph for ubiquitin design.

I denote a state assignment to all vertices by S_V, and the *induced assignment* to any subset of V (particularly a hyperedge), e, by S_e. Any assignment induces a score for the interaction graph, $\sum_{e \in E} f_e(S_e)$, and the optimal assignment minimizes the score. I call

the problem of assigning states to vertices in the interaction graph the *state assignment problem*.

Problem 2 The State Assignment Problem. Given an interaction graph $G = \{V, E\}$ where vertex $v \in V$ carries a state space $\mathcal{S}(v)$ and each hyperedge $e \in E$ carries a hyperedge scoring function $f_e : \prod_{v \in e} \mathcal{S}(v) \rightarrow \Re$, find the state assignment to each vertex S_V that minimizes $\sum_{e \in E} f_e(S_V)$.

The state assignment problem is useful as a generalization of the side chain placement problem: the task of assigning states to vertices is less concrete than the task of assigning rotamers to residues. The generality is convenient as problems similar to SCPP emerge elsewhere and can be modeled as state assignment problems. The dynamic programming algorithm of the next section extends to these problems as well.

4.4 Dynamic Programming on Interaction Graphs: SCPP

A dynamic programming (DP) algorithm (Bellman, 1957) can optimize an interaction graph by *eliminating* vertex v from the graph, by solving for the optimal state of v for each combination of state assignments to its neighbors, then replacing v with a hyperedge containing these neighbors.

Specifically, let E_v be the set of hyperedges that contain v, and $N_v = \bigcup_{e \in E_v} e \setminus \{v\}$ be the neighbors of v. Create a new hyperedge N_v with an initial scoring function $f_{N_v} = 0$. Next, add to f_{N_v} the scoring functions of E_v with the best assignment to v, and eliminate v and edges E_v from the hypergraph. Specifically, for each $s \in \mathcal{S}(v)$, let $\hat{f}_{e,v=s}$ denote the function from the states of N_v to the reals obtained from f_e by restricting the state of v to be s. That is, compute f_{N_v} as $f_{N_v} = \min_{s \in \mathcal{S}(v)} \sum_{e \in E_v} \hat{f}_{e,v=s}$. The scoring function f_{N_v} represents the simultaneous interaction of the vertices of N_v with the optimal state of v. The minimum score for this interaction graph without v is the same as the score of the original.

Each hyperedge scoring function can be represented as a table whose dimension is the degree of its corresponding hyperedge. Computing f_{N_v} amounts to filling all cells in the table that represents it. At the time dynamic programming computes f_{N_v}, it also records v's optimal state as a function of its neighbors' states for later retrieval.

Repeated eliminations reduce the number of vertices in the graph; Figure 4.3 illustrates the first two vertex eliminations. When a single vertex remains, its optimal state

is simply the state with the lowest energy, found by summing the energies for each of the (possibly many) degree-1 hyperedge scoring functions that are incident upon the vertex. Once the optimal state of the last remaining vertex is known, the optimal states for the eliminated vertices can be determined in a final backtracking step by retrieving their optimal states in the reverse order of their elimination.

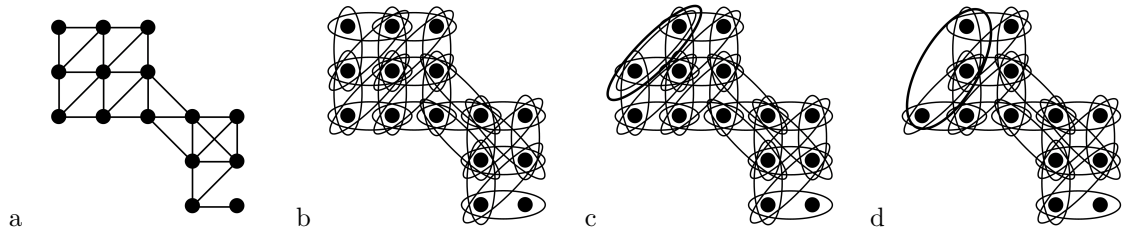


Figure 4.3: *Dynamic Programming on an Interaction Graph.* The graph in (*a*) comes from the redesign of the surface residues of ubiquitin's β-sheet. The edges in (*a*) are drawn as curves encircling the vertices they contain in (*b*). The DP algorithm eliminates the upper-left vertex of (*b*), creates a new degree-2 hyperedge, and computes the values of this hyperedge's scoring function (*c*). It creates a degree-3 hyperedge as it eliminates the next vertex (*d*).

The remainder of this section gives upper bounds on the running time and memory for an interaction graph with n vertices. If s bounds from above the number of states for any vertex, and w bounds from above the number of neighbors any vertex has at the time of its elimination, then DP runs in $O(ns^{w+1})$ time and uses $O(ns^w)$ space. The parameter w is the graph's treewidth. This dynamic programming algorithm shows that side-chain placement is another instance of an NP-Hard problem with a polynomial-time solution for graphs of bounded treewidth.

Theorem 3 *For an interaction graph G with n vertices with at most s states per vertex and with a tree decomposition T of treewidth w, dynamic programming can compute the optimum state assignment in $O(ns^{w+1} + wn)$ time and $O(ns^w + wn)$ space.*

Proof: *Correctness.* By induction on the number of vertices in G. If G contains 1 vertex, then the optimal state of that vertex is the optimal state for the entire graph. For the inductive step, assume dynamic programming correctly optimizes graphs with k vertices. Consider the graph $G = \{V, E\}$ with $k + 1$ vertices for which the tree decomposition T has been converted into a nice tree decomposition and vertex v is the first vertex to be eliminated. What needs to be shown is that the

elimination of v produces a new graph $G' = \{V', E'\}$, for which the optimal states for its vertices V' are the the same as the optimal states for the corresponding vertices in G. I prove this by demonstrating the following equality:

$$\begin{aligned}
\min_{S_V \in \mathcal{S}_V} \sum_{e \in E} f_e(S_V) &= \min_{S_V \in \mathcal{S}(V)} \Big(\sum_{e \in E' \setminus f_{N_v}} f_e(S_{V,e}) + \sum_{e \in E_v} f_e(S_{V,e}) \Big) \\
&= \min_{S'_{V'} \in \mathcal{S}(V')} \min_{s \in \mathcal{S}(v)} \Big(\sum_{e \in E' \setminus f_{N_v}} f_e(S'_{V',e}) + \sum_{e \in E_v} f_e(S'_{V,e}, s) \Big) \\
&= \min_{S'_{V'} \in \mathcal{S}(V')} \Big(\sum_{e \in E' \setminus f_{N_v}} f_e(S'_{V',e}) + \min_{s \in \mathcal{S}(v)} \sum_{e \in E_v} f_e(S'_{V,e}, s) \Big).
\end{aligned}$$

The sum $\min_{s \in \mathcal{S}(v)} \sum_{e \in E_v} f_e(S'_{V,e}, s)$ is the value produced by the hyperedge scoring function, f_{N_v} when the vertices in N_v are in states S'_{V',N_v}. The last line is thus the optimal score of the graph G'. Therefore for the optimal state assignment S_V for G, the state assignment $S'_{V'}$ produced from S_V by excluding the state assigned to vertex v is optimal for G'. By the inductive hypothesis, dynamic programming can optimize the graph G'.

Running Time. The $O(wn)$ terms reflect the time and space required to construct and use the nice tree decomposition and the time required to perform the backtracking step that concludes dynamic programming. I bound the running time for the vertex eliminations by induction on the number of vertices eliminated so far. Before eliminating vertex v, I assert that G is the interaction graph that results from having eliminated the vertices up to vertex v, and that the nice tree decomposition T that dynamic programming maintains is a tree decomposition of treewidth w for G. In the base case, this is trivially true; I show that this remains true after vertex v is eliminated. Let v be the next vertex in the elimination order. By the definition of a nice tree decomposition, $\{v\} = X_i \setminus X_j$ for some leaf-node X_i and its parent X_j. Therefore v is adjacent to at most w vertices, since $|X_i| \leq w + 1$ and vertex elimination proceeds in $O(s^{w+1})$ time and requires $O(s^w)$ space. The hyperedge produced by v's elimination is a subset of X_j, so we can delete X_i from T to obtain a tree decomposition of G after the elimination of v. Since each vertex is eliminated once, the theorem is established. ∎

4.4.1 Dynamic Programming Pseudocode

I now sketch the pseudocode for dynamic programming. This code is built around three classes: the `Vertex` class, the `Hyperedge` class, and the `BacktrackingTable` class.

Vertices contain lists of the hyperedges that are incident upon them, and lists of their neighbors. Each vertex can be uniquely identified by its index. Vertices store the number of states (rotamers) they have, which are enumerated starting at 0. They also contain pointers to back-tracking tables – tables that store the optimal state of a vertex as a function of the states of its neighbors at the time of its elimination.

The hyperedges store large tables of energies where the table dimension is equal to the degree of the hyperedge. Each edge allocates its table in row-major order by vertex index. That is, if an edge connects vertices 1 and 2, then the states for vertex 1 define the rows in the table, and the states for vertex 2 define the columns in the table. I describe in Section 4.4.2 below how important table allocation is to performance of DP.

The `NSV` class (neighbors' state vector) enumerates all state assignments to the neighbors of the vertex being eliminated – in the pseudocode below, this is the `neighborsStateVector`. This class can increment the current state assignment by one and report whether it is has visited the entirety of the neighbors' state space. It can also be passed to a hyperedge to retrieve the appropriate score for that hyperedge under the current state assignment.

I give the pseudocode for vertex elimination. This code creates a new hyperedge incident upon all the neighbors of the vertex being eliminated, and computes the values for the new hyperedge's scoring function. It also records the optimal state of the vertex as a function of the states assigned to its neighbors for use in the backtracking step that concludes dynamic programming. After the vertex has finished computing the new scoring function and the optimal states for backtracking, it drops all of its incident edges. The edges it drops may be deallocated, along with the tables storing their scoring functions.

4.4.2 Optimizing Dynamic Programming

The running time for my initial implementation of dynamic programming was 180 times longer than the running time of the final implementation. The same treewidth-3, small-rotamer-library optimization that ran on my laptop (1.2 GHz Pentium IV Mobile with 576 MB RAM) in 90 minutes the first time, ran in 30 seconds in the months that followed. There were two speedups that were especially important

```
Vertex::eliminate()
{
   newHyperedge = new Hyperedge( neighbors )
   backtrackingTable = new BacktrackingTable( neighbors )
   NSV neighborsStateVector( neighbors )
   neighborsStateVector.startAtZero()
   while ( ! neighborsStateVector.visitedAllStates() )
    bestscore = 0
    beststate = 0
      for (i = 0: numStates - 1)
         scoreThisAssignment = 0
         for ( j = 0: numHyperedges - 1)
            scoreThisAssignment += hyperedges[ j ]->getScore( neighborsStateVector, j )
         if ( scoreThisAssignment < bestscore || i == 0 )
            bestscore = scoreThisAssignment
            beststate = i
      newHyperedge.setScore( neighborsStateVector, bestscore )
      backtrackingTable.setBestState( neighborsStateVector, beststate)
      neighborsStateVector.increment()
   dropAllIncidentEdges()
}
```

Program 1: Psuedocode for vertex elimination in dynamic programming

The first and most dramatic speedup was due to efficient caching. Modern computer memory caches are designed with the concept of locality of memory reference: when data is accessed from one location in memory, the data in surrounding regions is very likely to be accessed as well. The heuristic employed to capitalize on this locality of reference is to move the block that surrounds a piece of memory onto the processor's cache whenever a piece of data is retrieved from memory, so that subsequent data retrievals are faster. Knowing that processor architecture is optimized to handle localized references, programmers can increase the efficiency of their code by organizing data reference to be local.

The first speedup related to the way I laid out memory so that the large hyperedge scoring tables stored next to each other scores that were accessed sequentially during dynamic programming. The vertex elimination subroutine enumerates all state combinations for the states of the neighbors of the vertex being eliminated. Enumeration proceeds in lexicographical order for these neighbors where the vertices are sorted in increasing order by their `index`. That is, if the eliminated vertex had two neighbors and the smaller indexed neighbor had 2 states and the higher indexed neighbor had 3 states, then the enumeration of their states would proceed as follows:

```
0 0 --> 0 1 --> 0 2 -->
1 0 --> 1 1 --> 1 2
```

For each state assignment to the neighbor vertices, dynamic programming then enumerates all state assignments to the vertex being eliminated, finding the optimal state

restricted to the context of states assigned to the neighbor.

The hyperedges allocate their tables in row-major order by increasing vertex index. Therefore, the edge that is incident upon vertices $\{3, 4, 5\}$ stores the score for state assignment $S = (a, b, c)$ next to the state assignment $S' = (a, b, c + 1)$ for $c < |\mathcal{S}_5| - 1$ and next to the state assignment $S' = (a, b + 1, 0)$ for $c = |\mathcal{S}_5| - 1$ and $b < |\mathcal{S}_4| - 1$, etc.

The first optimization technique is to assign vertex indices in reverse elimination order, so that dynamic programming traverses the hyperedge scoring tables in a more cache-efficient manner. At the time that vertex v is eliminated, its dimension in the hyperedge scoring tables for each of its incident edges is the last dimension. Thus retrieving scores while iterating across all the states of v while keeping fixed the states of v's neighbors means retrieving data from several contiguous blocks of memory, one block per hyperedge scoring table.

This optimization works only if the order of vertices for elimination is known before dynamic programming begins; this is a reasonable requirement for this implementation as it takes the vertex elimination ordering as an input. I describe another version of dynamic programming in Section 4.15 where I compute the vertex elimination ordering on the fly and thus do not lay out memory for cache efficiency.

This first optimization alone brought the running time down from 90 minutes to 1 minute; a 90× speedup.

The second optimization relates to computing indices when retrieving values from the hyperedge scoring tables. After the first optimization, profiling the code revealed that half of the time spent in dynamic programming is spent on computing indices into the hyperedge scoring tables; the other half is spent retrieving the values from these tables. Computing the index for an entry in a table of dimension d takes $d - 1$ multiplies and $d - 1$ additions. However, in dynamic programming, nowhere near this much work need actually be performed. The traversal of dynamic programming through the hyperedge scoring tables is incredibly simple: for the most part, dynamic programming retrieves the entry in the table next to the entry it just retrieved, and then at regular intervals, it jumps backward and starts over at a certain part of the table.

I developed a scheme I call *implicit indexing* whereby each hyperedge keeps track of the index for the last score retrieval and then in preparation for the next retrieval either increments that index by one, or decrements that index by some large, easily calculated quantity. The size of the decrement, the jump backward, is a simple function of the number of vertices the edge is incident upon that changed to their first state.

Consider the elimination of a vertex x with degree 4. For notational convenience, lets say these are vertices 1, 2, 3, 4 and 5 and that vertex i has s_i states. Consider also a particular hyperedge e that is incident upon vertices 1, 4 and 5. Let j be the step at which dynamic programming reaches the state assignment $(1, 2, s_3 - 1, s_4 - 1, s_5 - 1)$ so that the state assignment it considers at step $j + 1$ is $(1, 3, 0, 0, 0)$. The index that hyperedge e used retrieved its score at step j is $1 * s_4 * s_5 + (s_4 - 1) * s_5 + s_5 - 1$. The index that hyperedge e uses to retrieve the score at step $j + 1$ is $1 * s_4 * s_5$. The difference between the index of the retrieved score at step j and step $j + 1$ is simply $s_4 * s_5$. Generally, the index is decremented by the product of the number of states of those vertices that are reset back to zero.

The implicit indexing scheme was responsible for a speedup of 2×.

4.5 Results: DP in SCPP with Rosetta

I have implemented this dynamic programming algorithm so that it works within the design module of Rosetta (Kuhlman and Baker, 2000). The implementation takes as input the number of molten residues, the number of rotamers for each molten residue, the precomputed rotamer/background and rotamer-pair energies, and a vertex- (or molten-residue-) elimination order. The input vertex-elimination order could either be computed by some other algorithm, or created by hand.

I tested my implementation at the redesign of the 15 surface-pointing residues of ubiquitin's β-sheet. I allowed all amino acids except arginine and lysine, since their long side chains interact with residues that are far away and produce densely connected graphs with high treewidth. I used a small rotamer library composed of the Dunbrack rotamers (Dunbrack and Cohen, 1997), and removed all rotamers that collided with the background or with the C-β atom on any of the molten residues. I excluded edges between vertices based on an interaction magnitude threshold of 0.2 kcal/mol. Together, these restrictions on the design problem produced the interaction graph pictured in Figure 4.9c. Dynamic programming succeeded at optimizing this design problem in 30 seconds on a Pentium IV Mobile 1.2 GHz Processor with 576 MB of RAM. However, dynamic programming failed on my laptop when run with a large rotamer library: it required more than 4 GB of RAM and crashed as it attempted to exceed that limit.

4.6 Large Rotamer Libraries

The analysis of the DP algorithm shows that it is principally limited by two parameters: the number of states per node, s, which can be in the tens to the thousands, and the treewidth, w, which is fixed by the interaction graph. I have only tried problems where $3 \leq w \leq 5$. Since the number of vertices n is in the tens for a redesign problem, the linear dependence on dynamic programming's running time and the number of vertices is insignificant. I would like to reduce the impact of increasing s. How and why does s increase?

Large rotamer libraries are typically cousins of the standard rotamer libraries (Dunbrack, 2002; Lovell et al., 2000) obtained by taking extra dihedral samples from the neighborhood in dihedral space surrounding the standard rotamers (Figure 4.4). The typical way to include extra dihedral samples is to consider a box in dihedral space that surround the standard rotamers so that in each dimension (each χ angle) the box extends $\pm\delta$. In Rosetta, the δ for χ_i is the reported standard deviation for that dihedral angle for that standard rotamer, σ_i. In Dezymer, the δ is 10°. The large rotamer library samples this box for additional rotamers to include.

Call those rotamers reported in the standard rotamer libraries the *canonical rotamers*. Call the boxes in dihedral space that surround the canonical rotamers the *base-rotamers*. The base-rotamers represent a low-resolution description of a side-chain conformation so that a an assignment of a base-rotamer to a residue specifies a range of dihedral angles that the residue's side chain can take. Call the dihedral samples taken inside a particular base-rotamer s the *sub-rotamers* of s, denote them with subscripts, $s_1, s_2, \ldots$. The canonical rotamer is also a sub-rotamer. For convenience, let s_1 represent the canonical rotamer for a base rotamer. These three definitions are depicted in Figure 4.4.

Protein designers have found that large rotamer libraries produce better designs (Looger et al., 2003; Kuhlman et al., 2003; Dantas et al., 2003). The improvement can be attributed to the high penalty assigned to colliding atoms. The need for larger rotamer libraries is suggested by the Lennard-Jones 6-12 potential: the $\frac{1}{d^{12}}$ repulsive term is sensitive to small changes when d is small, so that slight flexes of the χ dihedrals can resolve high penalty collisions. At greater distances, the repulsive term drops to 0, but the other terms in the energy function – the attractive term and the solvation term – are still present. These other terms, in contrast to the repulsive term, are much less sensitive to small changes. Thus, for base rotamer pairs that do not position any atoms

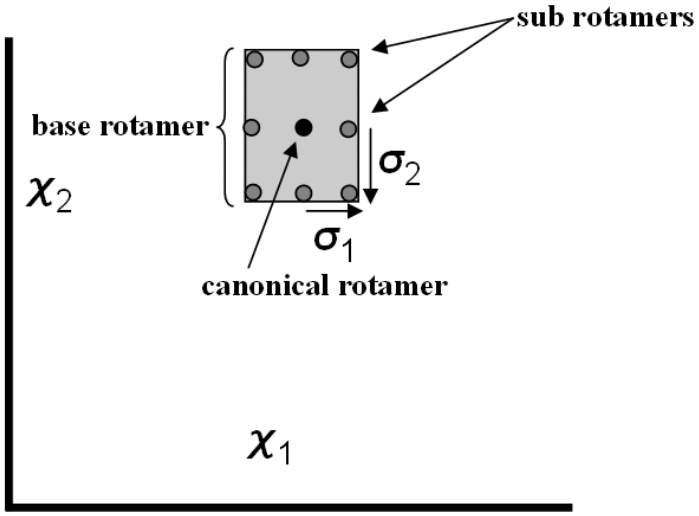


Figure 4.4: *Dihedral Space Definitions for ADP.*

in collision or near collision, I hope to represent the many sub-rotamer-pair interaction energies by a single interaction; the interaction of the canonical rotamers. I formalize this idea by defining the concept of stiff rotamer interactions.

4.7 Stiff Interactions

An assignment of base-rotamers, (s, t), to the adjacent vertices u and v is *stiff in* $\{u, v\} \in E$ if and only if there exist sub-rotamers s_i and t_j where

$$|f_{\{u,v\}}(s_i, t_j) - f_{\{u,v\}}(s_1, t_1)| > \epsilon.$$

For base-rotamer pairs that are not stiff, the energy between all pairs of sub-rotamers can be approximated by the energy between the canonical rotamers at the expense of introducing an error of at most ϵ. Thus, any edge-scoring function $f_{\{u,v\}}$ with no stiff interactions can be approximated by a more compact table $\tilde{f}_{\{u,v\}}$ that stores the canonical rotamer interactions instead of all sub-rotamer interactions – a considerable savings in memory and processing time. I show below how to record any stiffness that exists in the input interaction graph, and how to propagate that stiffness through hyperedges created during dynamic programming, while working with the compact,

approximate scoring functions, $\{\tilde{f}\}$.

The analogy is to stiff ordinary differential equations (ODEs): the stiff interactions are those where the energy is rapidly changing between among the sub-rotamer pairs for a pair of base rotamers. As adaptive schemes for numerical integration increase their temporal resolution when their input ODEs become stiff, and decrease their resolution when the ODEs behave smoothly, adaptive dynamic programming increases its spatial resolution when it encounters stiff base-rotamer combinations, examining all possible sub-rotamer combinations, and decreases its resolution for non-stiff base-rotamer combinations.

4.8 Irresolvable Collisions

The second improvement I make to the DP algorithm is to avoid spending time or allocating space for base-rotamer combinations in an *irresolvable collision*. That is, the user defines a collision tolerance threshold: the user declares any pair of rotamers that interact with energies worse than this threshold uninteresting for the sake of design. The threshold can then be applied to entire sets of rotamer pairs so that if no rotamer pairs are interesting, then all computations involving pairs from this set can be avoided. With a final parameter, τ, measured in kcals/mol, I define a pair of base-rotamers b and c to be in an irresolvable collision if no pair of sub-rotamers is without a collision:

$$\neg\exists_{i,j}\mathcal{E}_{pair}(b_i, c_j) < \tau$$

While some pairs of base-rotamers can resolve their collisions by slight dihedral flexes, others cannot. As long as we have hope that a collision-free placement of side chains exists, we need not examine the colliding pairs.

The parameter τ lets the user specify how much of a collision she is willing to tolerate in the rotamer assignment returned by the optimization. While it is possible for the optimal solution to contain a collision, it is unlikely that it will. If the purpose of the side-chain placement is to design a new protein, then any optimal solution that included a collision would represent a bad design that would not help the designer.

4.9 Adaptive Dynamic Programming

Adaptive dynamic programming stems from an alternate interpretation of rotamer state spaces. The state spaces that vertices carry can be interpreted in two different ways. In the first interpretation (the interpretation used in dynamic programming), each vertex v caries a state space, $\mathcal{S}(v)$ and each state represents an assignment of dihedral angles to a side chain. In the second interpretation (the interpretation that drives adaptive dynamic programming), each vertex v carries a hierarchy of state spaces. At the top level is the (low-resolution) base-state state space, $\mathcal{L}(v)$ where each base state represents a restriction on the range of dihedral angles for a side chain; a box in dihedral space, a base rotamer. At the bottom level, each base-state $b \in \mathcal{L}(v)$ carries a (high resolution) sub-state state space, $\mathcal{H}(b)$ where each sub-state represents an assignment of dihedral angles to side chains and each $s \in \mathcal{H}(b)$ lies inside b in dihedral space.

In the first interpretation of the vertices' state spaces, the hyperedge scoring function f_e would be interpreted as a mapping $f_e : \prod_{v \in e} \mathcal{S}(v) \rightarrow \Re$. In the second interpretation, the hyperedge scoring function f_e would be interpreted hierarchically. The high-resolution scoring function for a particular assignment of base-states B is a mapping from the sub-states to their interaction energies; $f_e^B : \prod_{b \in B} \mathcal{H}(b) \rightarrow \Re$. The low-resolution scoring function would be interpreted as mapping from the vertices low-resolution states into the high-resolution scoring functions for sub-states of those base-states; $f_e : \prod_{v \in e} \mathcal{L}(v) \rightarrow f_e^B$.

With these two interpretations defined, I now give the algorithm for adaptive dynamic programming. Consider a scoring function $f_e \colon \prod_{v \in e} \mathcal{S}(v) \rightarrow \Re$ for a hyperedge e. In adaptive dynamic programming, we will use an approximate function $\tilde{f}_e \colon \prod_{v \in e} \mathcal{S}(v) \rightarrow \Re$. Let s^i denote the sub-state assigned to the i^{th} vertex that e is incident upon. Let the base-state assignment B denote an assignment of base-states to each of the the vertices that e is incident upon. The value of the approximation function for a sub-state assignment, $\tilde{f}_e(s^1, s^2, \ldots, s^k)$, is derived from the value of $f_e(t^1, t^2, \ldots, t^k)$, where each t^i is either the canonical rotamer s_1^i or the original state s_j^i if $b(s_j^i)$ is involved in stiff interactions. For each hyperedge e and each base state assignment B, we maintain a *stiffness descriptor*, C_e^B which is a family of subsets of B. The base-rotamers contained in the stiffness descriptor are those which interact stiffly – if the stiffness descriptor is empty, then that assignment of base-rotamers does not interact stiffly. There are possibly many subsets of B in the stiffness descriptor. Let $C_{e,k}^B \subseteq B$ represent the k^{th} element of C_e^B where $1 \leq k \leq |C_e^B|$. Alongside the stiffness descriptor,

define the set of all stiff base-states in an assignment as $U_e^B = \bigcup_{k=1}^{|C_e^B|} C_{e,k}^B$.

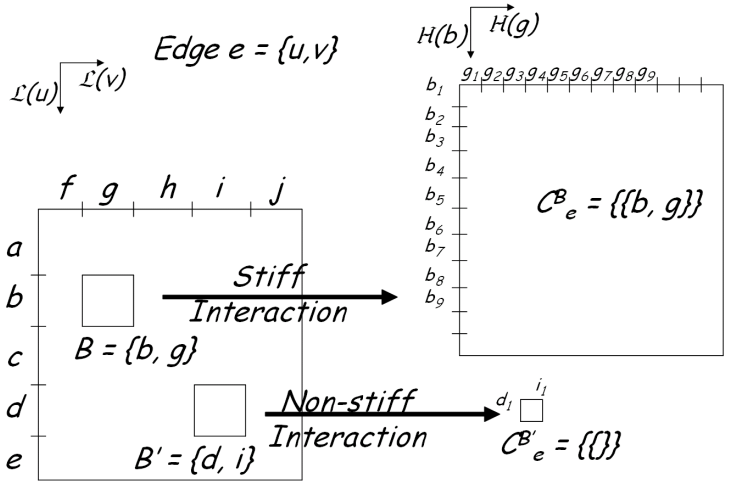


Figure 4.5: *Stiffness Descriptors for Input Edges.* The two-dimensional table of tables for holding the hyperedge scoring functions. Input hyperedges store the interaction energies for base state pairs in one of two possible ways. The first way is to store for stiffly interacting base states a high-resolution table of interaction energies for each sub-state pair, as is pictured for the base state assignment $B = (b, g)$. In this case, the top-level table stores a pointer to a second high-resolution table as well as the stiffness descriptor $C_e^B = \{\{b, g\}\}$. The second way to store base-rotamer interaction energies is to store a single interaction energy for base-rotamer pairs that do not interact stiffly, as is pictured for the base state assignment $B' = (d, i)$. This single energy approximates all sub-state interaction energies. Instead of a pointer to a table, the top level stores a pointer to a single float as well as the stiffness descriptor $C_e^{B'} = \{\{\}\}$.

If e is an input hyperedge $e = \{u, v\}$, then under the base-state assignment $\{a, b\}$, the stiffness descriptor is either $C_{\{u,v\}}^{\{a,b\}} = \{\{a, b\}\}$ if a and b's interaction is stiff and $\{\{\}\}$ otherwise (Figure 4.5). A non-input hyperedge is one created over the course of vertex elimination, and therefore corresponds to an eliminated vertex; the stiffness descriptor for such a hyperedge is determined by the stiffnesses that existed with the optimal states of the eliminated vertex.

The elimination of vertex v with incident edges E_v leaves behind a hyperedge N_v incident upon v's former neighbors. I describe how to compute the stiffness descriptor $C_{N_v}^B$ for a single base-state assignment, B, to the vertices of N_v. Define the stiffly-interacting base-states for the neighboring residues of v for a particular base-state of v, $b \in \mathcal{L}(v)$ to be $T_{E_v}^{(B,b)} = \cup_{e \in E_v} U_e^{(B,b)_e} - b$ where $(B, b)_e$ represents the base-states assigned

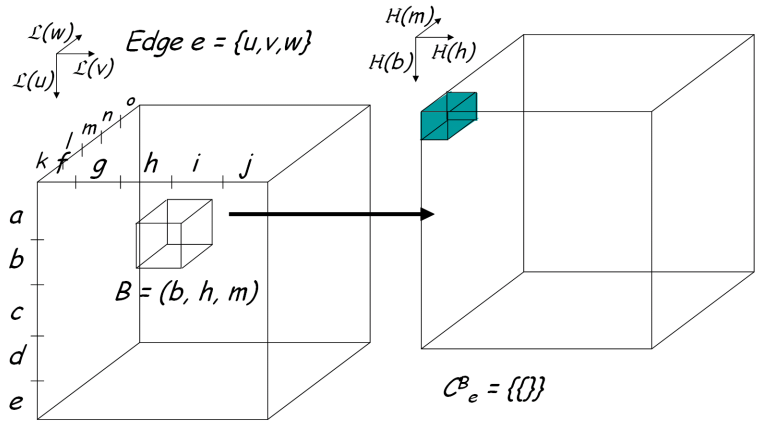


Figure 4.6: *Tables For Hyperedges Produced by ADP:* $C_e^B = \{\emptyset\}$. The elimination of vertex x has left behind a degree-3 hyperedge e incident upon vertices u, v, and w. The base states for x's neighbors under the base-state assignment $B = (b, h, m)$ did not interact stiffly with the optimal state of x, therefore, $C_e^B = \{\emptyset\}$ and the top-level table contains a pointer to a single float (blue region). The memory savings is the difference between the blue region for which memory is allocated, and the empty region that would have been allocated to store all sub-state assignments.

to the vertices e contains. Let P^B represent the power set of B, where if $|N_v| = d$ then $|P^B| = 2^d$, and let P_i^B be the i^{th} member of P_i^B. Let $\mathcal{L}_{P_i^B}(v)$ be $\{b \mid T_{E_v}^{(B,b)} = P_i^B\}$, that is, the set of all base-states of v which have the same stiffly-interacting base-states of P_i^B.

For each i with a non-empty $\mathcal{L}_{P_i^B}(v)$, compute the function $f_{P_i^B} : \prod_{b \in P_i^B} \mathcal{H}_b \rightarrow \Re$ so that

$$f_{P_i^B} = \min_{b \in \mathcal{L}_{P_i^B}(v)} \min_{s \in \mathcal{H}_b(v)} \sum_{e \in E_v} f_e((B,b)_e)_s$$

where $f_e((B,b)_e)_s$ has v fixed in sub-state s. Represent the range of $f_{P_i^B}$ by $[\text{best}_{P_i^B} \ldots \text{worst}_{P_i^B}]$. Define the *best-worst* score as $\min_i \text{worst}_{P_i^B}$ and define the set of competing stiffnesses as $I = \{i \mid \text{best}_{P_i^B} < \textit{best-worst}\}$.

Now, the power set suggests a natural partial-order based on the subset property. Define an *anti-chain* family of sets to be any where no element in the family is a subset of any other. Given a family of sets, P we define anti-chain(P) to be the function that returns the anti-chain family produced by throwing out any set if it is a subset of any other. The stiffness descriptor $C_{N_v}^B$ is anti-chain($\{P_i^B \mid i \in I\}$). Let the set I_k, corresponding to $C_{N_v,k}^B$, represent $\{i \mid P_i^B \subseteq C_{N_v,k}^B\}$. Define a set of functions, $f_{C_{N_v,k}^B} = \min_{i \in I_k} f_{P_i^B}$. Then the hyperedge scoring function with the state assignment

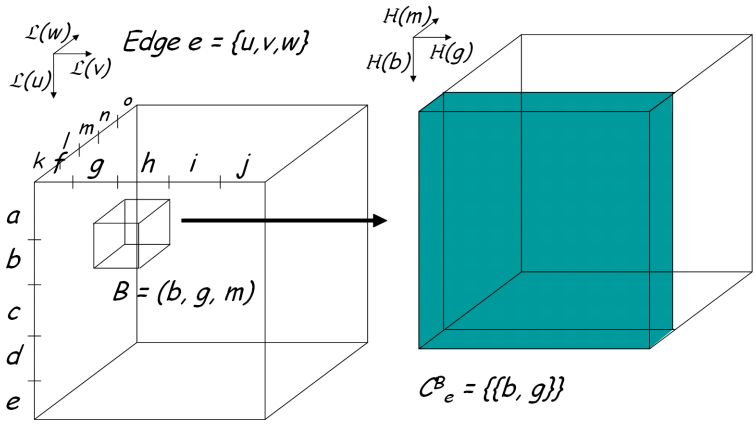


Figure 4.7: *Tables For Hyperedges Produced by ADP:* $|C_e^B| = 1$. The elimination of vertex x has left behind a degree-3 hyperedge e incident upon vertices u, v, and w. The base states for u and v under the base-state assignment $B = (b, g, m)$ interacted stiffly with the optimal states of x, but the base state for w did not, therefore, $C_e^B = \{\{b, g\}\}$ and the top-level table contains a pointer to a 2D table (blue region). The memory savings is the difference between the blue region for which memory is allocated, and the empty region that would have been allocated to store all sub-state assignments.

B is $f_{N_v}(B) = \min_k f_{C^B_{N_v,k}}$.

Where DP represents each hyperedge scoring function as a multi-dimensional table, ADP represents each function as multi-resolution multi-dimensional table: a table of tables. The top-level table has an entry for each base-state assignment to the vertices that the edge contains. Each cell in this top-level table contains the set of tables described by the stiffness descriptor. If the size of the sub-rotamer state space for base rotamer b is $|\mathcal{H}_b| = h_b$, then each $f_{C^B_{N_v,k}}$ requires $\prod_{b \in C^B_{N_v,k}} h_b$ space all $|C^B_{N_v}|$ tables require $\sum_{k \in C^B_{N_v}} \prod_{b \in C^B_{N_v,k}} h_b$ space. ADP requires less space than the original DP algorithm. I have illustrated a few of the possible high-resolution tables stored for different stiffness descriptors in Figures 4.6, 4.7, and 4.8. All of these different stiffness descriptors could be found on the same hyperedge.

There are two positive effects of declaring base-rotamer pairs in irresolvable collision on adaptive dynamic programming (ADP). First, is that ADP may avoid calculating the best state of a vertex for any combination of neighbors' base-states that put them in an irresolvable collision. Secondly, because the $\frac{1}{d^{12}}$ collision term is fluctuating wildly for base rotamer pairs that are in an irresolvable collision, all such base-rotamer pairs would meet the criterion for acting stiffly. Without avoiding all work for these base-rotamer pairs, adaptive dynamic programming would otherwise treat them at high

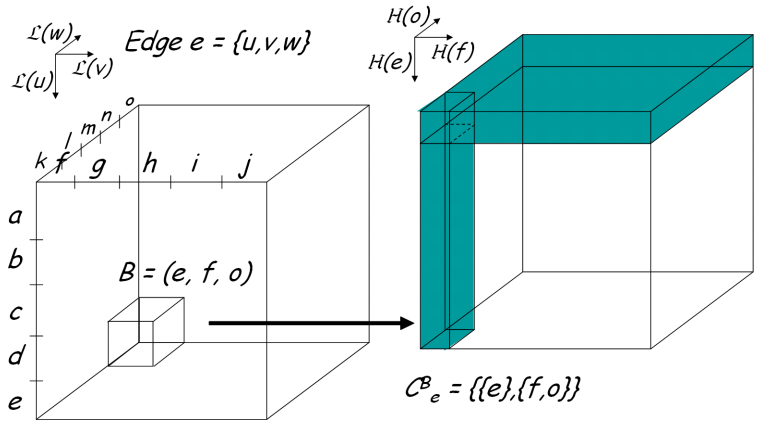


Figure 4.8: *Tables For Hyperedges Produced by ADP:* $|C_e^B| > 1$. The elimination of vertex x has left behind a degree-3 hyperedge e incident upon vertices u, v, and w. The base states for x's neighbors under the base-state assignment $B = (e, f, o)$ interacted stiffly with the optimal states of x, but such that some of the optimal states of x simultaneously interacted stiffly with base states f and o but not with e, and the other optimal states of x interacted stiffly with e but not with base states f and o. The stiffness descriptor $C_e^B = \{\{e\}, \{f, o\}\}$ is an anti-chain family that contains two sets. Therefore, the top-level table contains pointers to two tables (blue regions). The memory savings is the difference between the blue region for which memory is allocated, and the empty region that would have been allocated to store all sub-state assignments.

resolution, performing much more work than they are worth.

4.9.1 Pseudocode of ADP

Below, I sketch the code for vertex elimination. The chief difference between the vertex elimination in dynamic programming and that in adaptive dynamic programming is the hierarchical structure of the loops. Dynamic programming contains two nested loops: a `while` loop to iterate across the all state assignments to the neighbors, and inside of this `while` loop, a `for` loop to iterate across the states of the vertex being eliminated. Adaptive dynamic programming contains four nested loops: the outermost `while` loop iterates over the base-state assignment to the neighbors, and inside of this `while` loop, a `for` loop to iterate across the base-states of the vertex being eliminated. Inside of this `for` loop is another nested `while` loop/`for` loop pair that iterates across the sub-state assignments for the base-rotamers specified in the outer two loops – this final pair of loops does not always iterate across all sub-state assignments, as some base-states do not interact stiffly and their sub-states do not need to be examined. Adaptive dynamic programming saves time by examining fewer sub-state combinations than dynamic programming examines.

The vertex and hyperedge classes are similar to those classes in dynamic programming, the exception of course being the table of tables that represent the hyperedge scoring functions. The `NSV` (neighbors' state vector) class now iterates over either the base-states for the neighbors, or the sub-states for the neighbors. At `NSV` instantiation, the elimination code specifies whether the vector should operate at either low resolution (`LOWRES`) or at high resolution (`HIGHRES`). When initialized at high resolution, the NSV must know exactly which of the neighbors to treat at high resolution, since not all neighbors need to be examined at high resolution. The elimination code specifies exactly which base-states should be treated at high resolution by passing the `stiffness` vector to the `NSV` constructor. The `stiffness` vector is a vector of booleans; if a base-state is stiff on any hyperedge, it must be treated at high resolution in the inner-most while loop/for loop pair.

4.10 Error Analysis for ADP

Theorem 4 *The score of the state assignment S_V computed by adaptive dynamic programming on a graph with $|E_2|$ degree-2 hyperedges is within $2\epsilon|E_2|$ of the GMEC score.*

```
IG_Vertex::eliminate()
{
   newHyperedge = new Hyperedge( neighbors )
   NSV neighborsStateVectBaseStates( neighbors, LOWRES );
   neighborsStateVectBaseStates.startAtZero()

   while( ! neighborsStateVectBaseStates.visitedAllStates())
      if ( any neighbors in irresolvable collision )
         neighborsStateVectBaseStates.increment()
         continue

      for i = 1:numBaseStatesThisNode
         if ( base state i in irresolvable collision with neighbors )
            continue

         stiffness(:) = false
         for k = 1:numHyperdges
            stiffness |= hyperedge[k].getStiffness(stateVectBaseStates)

         if (stiffness != false)
          neighborsStateVectSubStates( neighbors, HIGHRES, stiffness )
            neighborsStateVectSubStates.starAtZero()
            while ( ! neighborsStateVectSubStates.visitedAllSubStates())
              for j = 1:numSubstatesThisNode[i]
                  theScore = 0
                  for k = 1:numHyperedges
                     theScore += hyperedge[k]->getScore(neighborsStateVectSubStates, j)
                  if ( theScore < bestScore[stiffness][stateVectSubStates] )
                     bestState[stiffness][stateVectSubStates] = (i,j)
                     bestScore[stiffness][stateVectSubStates] = theScore
               neighborsStateVectSubStates.increment()
         else
           theScore = 0
            for k = 1:numHyperedges
               theScore += hyperedge[j]->getScore( stateVectBaseStates, i)
            if (better_than(theScore, bestScore[stiffness]))
               bestState[stiffness] = i
               bestScore[stiffness] = theScore
      selectBestStiffnesses(bestScores)
      mergeSubsets(bestScores, bestStates)
      newHyperedge.saveScoresForState( neighborsStateVectBaseStates, bestScores)
      optStateForTraceback.saveStatesForState( neighborsStateVectBaseStates, bestScores, bestStates)
      neighborsStateVectBaseStates.increment()
   return
}
```

Program 2: Psuedocode for vertex elimination in adaptive dynamic programming

Proof: Let σ_{S_V} be the energy of the adaptive algorithm's state assignment, S_V, and σ_{opt} be the energy of the global minimum energy conformation, S_{opt}. The parameter ϵ allows the approximation of the sum of the hyperedge scores to contain an error of at most $\epsilon|E_2|$. Because adaptive dynamic programming chose S_V over S_{opt},

$$\sigma_{S_V} - \epsilon|E_2| \leq \sigma_{opt} + \epsilon|E_2|$$

and thus $\sigma_{S_V} - \sigma_{opt} \leq 2\epsilon|E_2|$. ∎

4.11 Results: ADP in SCPP

I tested adaptive dynamic programming at the rotamer relaxation task (where the amino acid sequence is held fixed) and at the redesign task (where the amino acid sequence is allowed to vary). For both tasks, I selected 15 surface residues from ubiquitin's β-sheet, pictured in Fig. 4.9a. I excluded the following amino acids to keep the treewidth of our interaction graphs low: arginine, lysine, and methionine. I chose this optimization problem for two reasons: DEE has been reported to perform poorly at protein surfaces (Gordon and Mayo, 1998), and the interaction graph defined by this problem has a small treewidth. If dynamic programming or adaptive dynamic programming is to beat dead-end elimination, it will be for problems like this one. Indeed, the implementation of dead-end elimination I have put together is unable to solve this problem. Loren Looger's implementation, which contains theorems I have not implemented (Looger and Hellinga, 2001), solves this problem in 30 minutes (Loren Looger, personal communication).

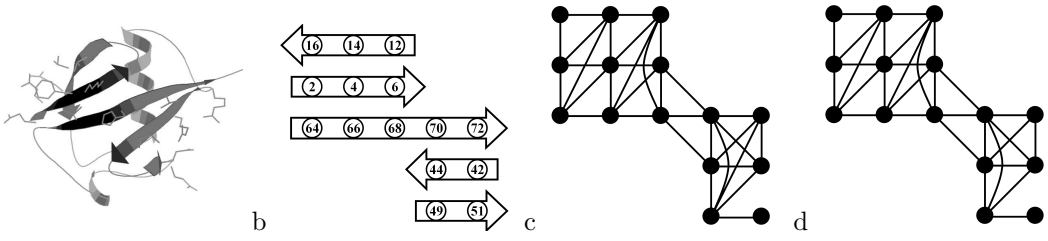


Figure 4.9: Ubiquitin's β-sheet. *The β-sheet in (a) is flattened in (b) with its 15 surface residues shown. Rosetta's energy function defined the treewidth-4 interaction graph in (c) when it included edges between residues for any pair of rotamers that interact with an energy magnitude at least $\mu = 0.2$ kcal/mol. I artificially defined the treewidth-3 interaction graph in* (d) *by dropping an edge from the lower right K_5 clique in* (c).

For the rotamer relaxation task, I first created 100 sequences for the ubiquitin backbone, asking the design module of the Rosetta molecular modeling program (Kuhlman and Baker, 2000) to redesign these 15 surface residues. I then evaluated Rosetta's experimentally validated energy function (Kuhlman et al., 2003) between all pairs of sub-rotamers, and included degree-2 hyperedges that met the interaction magnitude threshold $\mu = 0.2$ kcal/mol. This produced a treewidth-4 interaction graph, shown in Fig. 4.9c. I set the irresolvable collision cutoff to $\tau = 1$ kcal/mol. I compared the standard dynamic programming algorithm against the adaptive algorithm with ϵ values of 0, 10^{-4}, 10^{-3}, 10^{-2}, 0.1, and 1.0 kcal/mol. Against dynamic programming, I compared the time spent in a single simulated annealing trajectory and the score simulated annealing produced.

In the relaxation problem, the average residue had 32 total rotamers, breaking down into 3 base-states and 10 sub-states per base-state. The median state space size was $\sim 10^{18}$. I measured performance on a dual 2 GHz AMD Athlon with 2 GB RAM. In Fig. 4.10, I plot the relative running time of the adaptive and standard dynamic programming algorithms against the actual error observed. In Table 4.1, I present the actual running times. Except for three instances of the 100 rotamer-relaxation tasks, simulated annealing produced the optimal answer when run for as long as standard dynamic programming.

		ADP						
Run Time	DP	$\epsilon = 0$	$\epsilon = 10^{-4}$	$\epsilon = 10^{-3}$	$\epsilon = 10^{-2}$	$\epsilon = 0.1$	$\epsilon = 1.0$	SA
Mean	206.2	63.7	62.9	63.0	61.2	17.5	6.4	65.1
Median	117.3	53.7	53.2	53.7	50.7	11.3	4.8	65.0
Std Dev	399.3	49.1	48.6	48.9	48.8	38.2	7.6	6.5

Table 4.1: *ADP Runtimes at Rotamer Relaxation.* Average running time comparison of dynamic programming (DP), adaptive dynamic programming (ADP), and simulated annealing (SA), in miliseconds, at the rotamer-relaxation task (fixed sequence side-chain placement) for the 15 surface residues of ubiquitin's β-sheet.

For the redesign task, I artificially created a treewidth-3 interaction graph on the problem, pictured in Figure 4.9d. This interaction graph differs by a single edge from the graph in 4.9c. The absence of this edge may decrease the quality of the design. I none the less include the task as it pushes the DP algorithm's limits. (After I made the decision to drop this edge and published these observations, Loren Looger kindly ran dead-end elimination on exactly this problem instance and found the answer produced

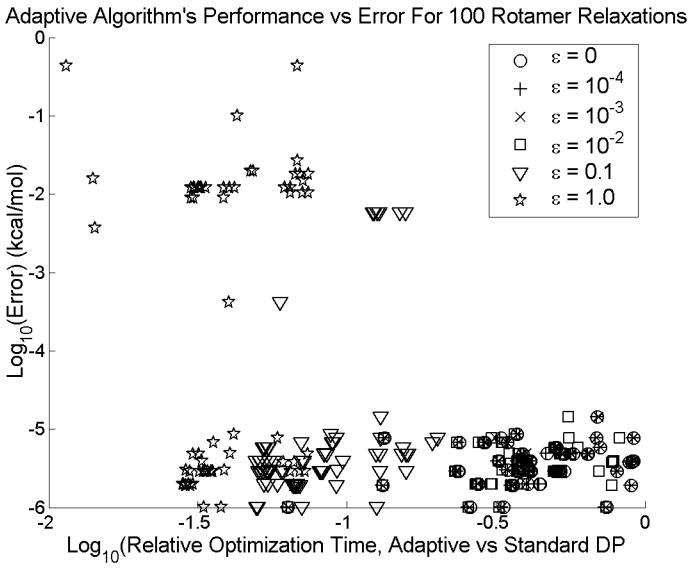


Figure 4.10: *Speedup vs Error for ADP.* Increasing ϵ to as high as 1.0 kcal/mol gives a theoretical error bound of $\pm$ 32 kcal/mol but in actuality greatly preserves accuracy and greatly decreases running time.

by dynamic programming in the absence of this edge was the same as produced by a complete graph – the graph including both this edge and all other dropped edges that failed to meet the 0.2 kcal/mol magnitude threshold [Looger, personal communication]).

Each residue in the design problem had on average 680 rotamers to choose from. This broke down into about 57 base-rotamers per residue and 12 sub-rotamers per base-rotamer. The size of the state space was $\sim 10^{42}$. I measured the performance of both the standard and the adaptive dynamic programming algorithms on a dual 900 MHz 64-bit Itanium-2 with 10 GB of RAM. I compared the running time, the memory use, and the score produced by ADP against simulated annealing, measuring simulated annealing on the same 2 GHz Athlon described above. Because both DP and ADP required so much memory, they had to be run on a 64-bit machine. This state-of-the-art machine is slower than the commodity 32-bit machines available; a fact that a protein designer should take into a consideration when choosing among optimization algorithms. Table 4.2 contains the results of this experiment.

	DP	ADP, $\epsilon = 0.1$	ADP, $\epsilon = 1.0$	SA
Run Time	15.99 hrs.	5.07 hrs.	1.52 hrs.	3.42 seconds
Memory Usage	3.7 GB	3.4 GB	1.5 GB	0.2 GB
Score (kcal / mol)	-42.5893	-42.5893	-42.5579	-42.5692
Error (kcal / mol)		0.0000	0.0314	0.0201

Table 4.2: *Redesign Task Performance Comparison.* A comparison of the running time and memory usage for dynamic programming and adaptive dynamic programming at the redesign task for the 15 surface-pointing residues on Ubiquitin's β-sheet.

4.12 Discussion and Limitations

I have presented a novel application of dynamic programming, proven the membership of the side-chain-placement problem in the class of NP-Complete problems for which certain problem instances admit a polynomial time solution, and designed and implemented an improvement upon dynamic programming that, in terms of speed, begins to make DP competitive with exact optimization techniques, although it makes it no longer exact. Adaptive dynamic programming speeds dynamic programming by one to two orders of magnitude.

Dynamic programming offers an alternative to DEE for finding the global optimal solution in the side-chain-placement problem. DEE is designed to solve a very difficult problem; the interaction graph implied by energy Expression 4.1 is fully connected. However, in the absence of a long range energy function, distant residues have interaction energies of zero and the interaction graphs tend to be sparsely connected.

Dynamic programming in protein design is greatly limited by the treewidth of the interaction graphs it defines. It is further limited by the complexity of finding optimal tree decompositions, but this is a lesser problem. Most importantly, dynamic programming is limited by its memory requirements: designers run out of memory when their memory requirements scale quadratically with the number of rotamers per residue; if memory scales exponentially with the number of rotamers per residue, there is no hope. In the rest of this section, I go into greater detail about these limitations.

Although it is encouraging that interaction graphs from protein design problems are sparse, a sparsely connected graph can still have a high treewidth. For example, the two-dimensional grid-graph is sparsely connected (Figure 4.11); the highest degree vertex has a degree of four. However, the treewidth for the grid graph with n vertices is $\sqrt{n}$. Although the interaction graph from the surface redesign problem had a low treewidth, protein design problems generally do not. The interaction graphs generated

by the complete-protein redesigns of 55 proteins showed treewidths of between 10 and 12 (Bodlaender, personal communication). Optimizing such problems with dynamic programming alone would be prohibitively expensive.

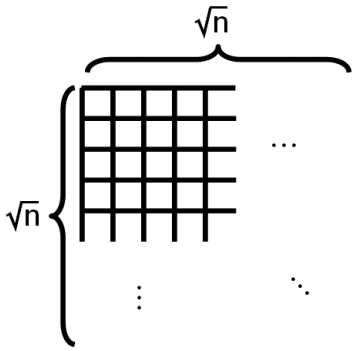


Figure 4.11: *Low Degree Grid Graph with High Treewidth.* If each line crossing in the above graph represented a vertex in a graph with n vertices so that there are $\sqrt{n}$ vertices in each dimension, then this *grid graph* has a treewidth of $\sqrt{n}$.

Dynamic programming also requires a good tree decomposition to specify the vertex-elimination order. Bad vertex-elimination orders produce slower optimizations. Since finding an optimal tree decomposition is NP-hard, we require that the user supply one as input to dynamic programming. Although it is not too difficult to to find a near-optimal tree decomposition for protein graphs by hand, an automated decomposition protocol would be required to integrate dynamic programming into a design protocol like the one that iterates between fixed backbone design and backbone modification (Kuhlman et al., 2003).

In comparison with simulated annealing, even when an optimal tree-decomposition is given, dynamic programming and adaptive dynamic programming both lose. Though a single simulated annealing trajectory may not find the optimal rotamer assignment, when simulated annealing is given as much time as dynamic programming required, at least one of the trajectories found the optimal solution for the rotamer relaxation task in all but 3 of the 100 cases I examined. A single simulated annealing trajectory takes orders-of-magnitude less time, and uses only as much memory as is required to store the precomputed rotamer-pair energies.

Moreover, for dynamic programming the memory requirement is too high. Though ADP generally reduced the memory usage, it did not reduce it as much as I had hoped.

Memory use is critical as designers tend to use as many rotamers as they possibly can, filling all available memory with tables of precomputed rotamer-pair energies. For such problems, dynamic programming would not have any remaining memory to store the hyperedge scoring tables nor the subproblem solution tables.

What dynamic programming and dead-end elimination offer a protein designer is a way to test that their rapid stochastic techniques are genuinely optimizing side-chain placement. That is, designers will mostly rely on simulated annealing after they have used either dynamic programming or dead-end elimination to verify that their simulated annealing implementation is performing as desired.

4.13 The Hydrogen-Placement Problem

Molecular biologists frequently analyze atom contacts within a protein molecule to accurately determine and evaluate protein structure. Michael Word and others from David and Jane Richardsons' 3D Protein Structure laboratory (Lovell et al., 2003; Richardson and Richardson, 2001; Word et al., 2000; Word et al., 1999a; Word et al., 1999b) have developed a set of open-source programs and web tools for atom contact analysis `http://kinemage.biochem.duke.edu`. A key program, named REDUCE, adds hydrogens to a Protein DataBank (PDB) molecular structure based on atom contacts (Word et al., 1999b). It finds the placement of hydrogen atoms that maximizes a scoring function.

REDUCE uses a brute-force search, which is reasonably fast, since most inputs require enumerating only a small number of hydrogen placements. Unfortunately, there are a number of bad input cases that REDUCE cannot handle in an acceptable amount of time. In the remainder of this chapter, I present a graphical formulation of their optimization problem and its dynamic programming solution. Whereas in protein design the treewidths of the observed interaction graphs were high, and the number of states per vertex were higher still, in the hydrogen-placement problem, the treewidths of observed interaction graphs were small, and the number of states per vertex smaller still, making the hydrogen-placement problem ideal for dynamic programming.

I have incorporated dynamic programming into REDUCE by replacing their scoring and search subroutines. I have compared the modified version of REDUCE with the original on these bad input cases. The new program produces identical outputs, and is faster by a factor of seven in general and faster by ten orders of magnitude at a particularly challenging problem instance.

4.13.1 Hydrogens and Structures from X-ray Crystallography

The most popular method for structure determination, X-ray crystallography, is based on observing electron density. Crystallographers thread a protein chain through the observed density to create a structural model. When the density predicted from the threaded model matches the observed density, that threading is said to *fit* the density. Density is easily observed for the heavy atoms of proteins—carbon, nitrogen, and oxygen—since they have about a dozen electrons in the orbitals that surround them. However, electron density is rarely resolved for hydrogen atoms since each hydrogen atom has only a single electron, and moreover, protrudes only slightly from the density of the heavy atoms to which it is chemically bound. Thus, hydrogen atoms are invisible in a standard X-ray diffraction experiment, and crystallographers often do not include them in the structural models they deposit in the PDB.

This is unfortunate since hydrogens represent about half the atoms of a protein, making hydrophobic packing analysis difficult (or dubious) in their absence. Moreover, some hydrogens form hydrogen bonds, which as discussed previously play a crucial role in protein stability. For example, the secondary structural elements of α-helices and β-sheets are characterized by intra-molecular backbone hydrogen bonds. Its hard to analyze the contribution hydrogen bonds make toward protein stability when there are no hydrogens recorded in protein structures.

REDUCE takes a protein model (the coordinates of all non-hydrogen atoms) from a PDB file and finds the best (defined as maximizing a scoring function) placement of hydrogens onto the model. The output of REDUCE can then be used by two other programs created in the Richardson lab, PROBE and KINEMAGE, to analyze the interaction of all of the protein's atoms (Richardson and Richardson, 2001; Word et al., 2000). The broad goal of REDUCE, when searching for the best hydrogen placement, is to minimize overlap between non-bonded atoms and to maximize the amount of hydrogen bonding.

What choices does REDUCE have to add hydrogens? The positions of most hydrogen atoms are fixed once one knows the coordinates of the bound heavy atoms. REDUCE computes these positions with simple vector geometry. Hydrogens in -OH, -SH and $-NH_3^+$ groups have rotational freedom (*i.e.* there is a circle in space on which the hydrogen atom's center lies). REDUCE infers possible positions for these hydrogen atoms from the presence of nearby hydrogen bond acceptors or obstacles.

REDUCE can also resolve another structural ambiguity that arises in X-ray crys-

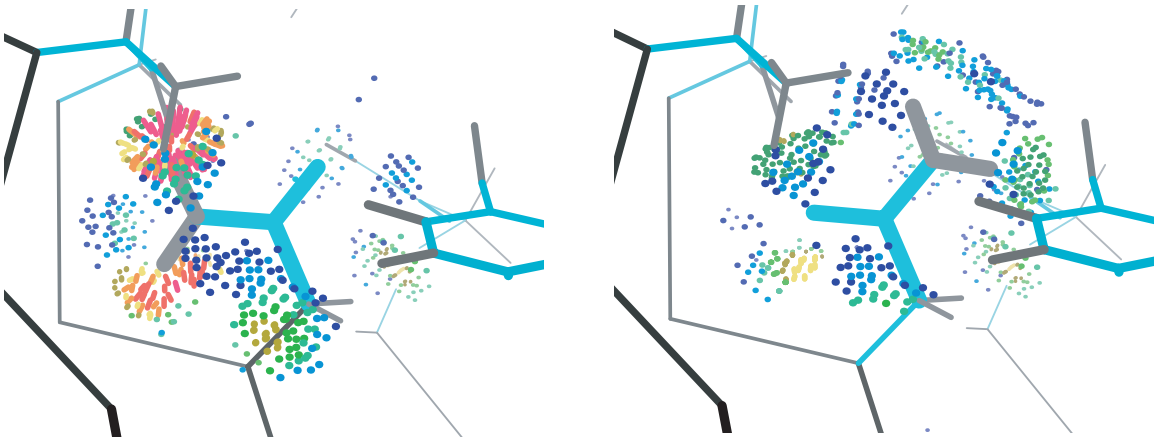


Figure 4.12: *Side Chains with Ambiguous Density.* ASN induces different atom contacts in two "flip" positions. The spikes (red) on the left indicate a bad van der Waals overlap; the dot pillows (green) on the right indicate favorable hydrogen bonds. (Images from KINEMAGE)

tallography. Certain asymmetric side chains—those of ASN, GLN, and HIS, in particular (Branden and Tooze, 1999)—appear symmetric within the X-ray crystallographic data because the observed electron densities for carbon, oxygen, and nitrogen atoms are similar. As 4.13 illustrates, when the amide groups of the ASN and GLN side chains are *flipped*, the nitrogen and oxygen atoms swap positions, and when the ring of a histidine is flipped, two carbon and two nitrogen atoms swap positions. If a configuration of one of these side chains, having been threaded through the electron density by the crystallographer, fit that density well, then the flipped configuration would fit that density equally well. A crystallographer threading these side chains through the electron density, without concern for hydrogen placement, will often make an arbitrary choice between the two viable flip positions. REDUCE resolves the flip states for these ambiguous side chains by looking at their interactions with the surrounding atoms.

On the ASN and GLN terminals, the nitrogen-bound hydrogen atoms are able to act as hydrogen bond donors, and the oxygen atom can act as a hydrogen bond acceptor. If one flip state produces good hydrogen bonding on both sides of the amide, then the other flip state matches two hydrogen bond acceptors and pushes two hydrogen bond donors into impossible collision (4.12). By explicitly modeling the hydrogen atoms for these side chains, REDUCE is able to resolve structural ambiguity for heavy atoms as well.

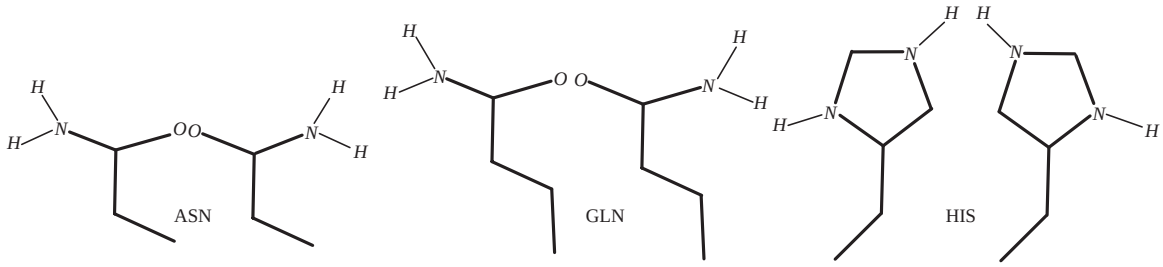


Figure 4.13: *Flip state pairs for ASN, GLN, and HIS.* ASN and GLN can be in one of two flipped states. HIS can be in one of two flipped states, and one of three protonation states for each of flipped state. Either the Nδ1 nitrogen is protonated, the Nϵ2 nitrogen is protonated, or both nitrogens are protonated. The dual protonation state has a +1 formal charge.

4.13.2 Hydrogen Placement as Optimization

REDUCE searches for the best hydrogen placement by considering the assignment of states to a collection of movers. A *mover* is a group of atoms that can be rotated or flipped together, changing their coordinates. Each discrete choice of rotation angle for a rotatable group or each flip state for a flippable group defines the state for that mover. The best hydrogen placement is the configuration of movers that maximizes a scoring function that considers non-bonded overlap and hydrogen bonding. Word *et al.* present their scoring function as

$$\text{Score} = \sum 4\ \mathit{Vol}(\text{ Hbond Overlap }) - 10\ \mathit{Vol}(\text{ Non-Hbond Overlap}).$$

The actual function is a discrete approximation of these volumes. This scoring function is not pairwise decomposable, unlike many of the scoring functions typical in computational chemistry. To accommodate the non-pairwise decomposability, REDUCE's authors chose to optimize using brute force instead of turning to exact optimization techniques, such as dead-end elimination (Desmet et al., 1992).

I decompose the energy function as much as I can, and find that when I am done, the majority of units I have decomposed the function into are singles, pairs, and triples; these units describe hyperedges in a graph, hyperedges with degree 1, 2, and 3. The decomposition is a hypergraph, an *interaction graph.* I apply dynamic programming to the interaction graph.

4.14 Decomposition of a Dot-Based Scoring Function

Word *et al.* (Word et al., 1999a; Word et al., 1999b) define a variation on contact-dot analysis originally developed fifteen years earlier (Richardson and Richardson, 1987), and describe its use in scoring atomic packing. Initially, between 200 and 600 dots are placed on the van der Waals surface of each atom, A, at roughly uniform density and spacing. Dots that fall inside an atom chemically bound to A are discarded. Each remaining dot d on A finds the closest van der Waals sphere B of an atom not bonded to A. This distance is considered negative if d is inside the van der Waals sphere B. Dot d is assigned a favorable *hydrogen bond score* if it is inside B (but not too deeply), and the atoms A and B are a hydrogen bond donor and acceptor. The hydrogen bond score is weighted by dot d's penetration depth inside B. Dot d is assigned an unfavorable *overlap penalty* if d lies inside B and either A/B is not a hydrogen bond donor/acceptor pair, or d is too deeply inside B. The overlap penalty score is also weighted by d's penetration depth. (In PROBE, but not in REDUCE, dot d is assigned a favorable *contact score* if it is outside of B, but within a small distance of 0.5 Å.)

I now decompose this scoring function into hyperedges. Let the mover set V of a protein have for each mover $v \in V$ an associated set of states, $\mathcal{S}(v)$. Then

- For a set of movers V', let $\mathcal{S}(V') = \prod_{v \in V'} \mathcal{S}(v)$ denote the state space of the movers in V', and let $S_{V'} \in \mathcal{S}(V')$ denote a particular state vector in the state space of V'.
- Given a set of movers $V' \subseteq V$ and their state vector $S_{V'}$, let $S_{V'' \subseteq V'}$ denote the partial state vector for a subset of V', V''.
- Let $v(s)$ denote that v is in state $s \in \mathcal{S}(v)$.

A particular state assignment $v(s)$ fixes the atoms of v in space. Denote the dots placed on the surface of the atoms of $v(s)$ by $\text{dots}(v(s))$ and $\text{dots}(v) = \cup_{s \in \mathcal{S}(v)} \text{dots}(v(s))$. A dot originates from some fixed atom of a mover and has a coordinate. The scores of the dots in $\text{dots}(v(s))$ depend on the positions of the atoms around them, which means they depend on the states of the movers around them. Consider a second mover v'. A dot $d \in \text{dots}(v(s))$ is inside $v'(s')$, for $s' \in \mathcal{S}(v')$ and $v' \neq v$, if one of the atoms of $v'(s')$ contains d. More generally d is inside v' if $\exists_{s' \in \mathcal{S}(v')} d$ inside $v'(s')$. Define the set

of movers that influence a dot d as $V_d = \{v : d \in \text{dots}(v) \text{ or } d \text{ inside } v\}$. The dot score for a dot, $score(d)$, is a function of the states of V_d.

By grouping dots together that are influenced by the same movers, I decompose the scoring function. Denote the set of dots that associate with a set of movers $e \subseteq V$, as $D(e) = \{d : V_d = e\}$. The scores for the dots in D_e are determined by the states of e. Defining $D(e \mid S_e) = \{d : d \in D(e) \text{ and } \exists_{v \in e} d \in v(S_{v \subseteq e})\}$, then the scoring function is ready to be decomposed. Specifically, define a mapping $f_e \colon \prod_{v \in e} \mathcal{S}(v) \rightarrow \Re$, where

$$f_e(S_e) = \sum_{d \in D(e|S_e)} score(d)$$

Define a family of subsets of V, $E = \{e : D(e) \text{ non-empty}\}$. Then the protein's score for a particular assignment of states S_V is

$$\sum_{v \in V} \Big(\sum_{d \in \text{dots}(v(S_{v \subseteq V}))} score(d) \Big) = \sum_{e \in E} f_e(S_{e \subseteq V})$$

This decomposition produces an interaction graph. The vertices in this graph are the movers; the vertex state spaces are the flip states for flippable movers and the set of possible hydrogen locations for the rotatable moves. Each set of movers that have dots in simultaneous overlap defines a hyperedge, and the scores for these dots define the scoring function for that hyperedge. The key difference between the interaction graph defined by this decomposition and the interaction graph from the side-chain-placement problem introduced in Section 4.3 is that the input hyperedges – the hyperedges defined by the problem itself, as opposed to those hyperedges created during the course of dynamic programming – may be of arbitrary degree instead of always being either degree 1 or degree 2.

4.15 Dynamic Programming on Interaction Graphs: HPP

The original REDUCE algorithm found the connected components of the interaction graph and optimized them individually by brute-force. I also optimize the connected components individually, and will henceforth assume that any interaction graph is connected.

I give a dynamic programming solution to the hydrogen-placement problem. I use

the interaction graph both to guide the dynamic programming process and to store the subproblem solutions. In this dynamic programming solution, instead of creating a new hyperedge for each vertex during its elimination (which sometimes produces repeated edges) I sometimes redefine the hyperedge scoring function for an existing edge.

Dynamic programming begins with the interaction graph defined by the particular problem instance. At each step, dynamic programming selects a vertex x for *elimination*. Let E_x denote the set of hyperedges incident upon x and V_x denote the neighboring vertices of x. Dynamic programming updates interaction graph as follows:

1. If V_x is not already a hyperedge, DP inserts V_x into E' and sets f_{V_x} be zero for all $\mathcal{S}(V_x)$.

2. For each state $S_{V_x} \in \mathcal{S}(V_x)$, DP computes the new score $f'_{V_x}(S_{V_x}) =$

$$f_{V_x}(S_{V_x}) + \max_{S_x \in \mathcal{S}(x)} (\sum_{e \in E_x} f_e(S_x \times S_{(e-x) \subseteq V_x}))$$

3. DP deletes the vertex x.

Dynamic programming stops eliminating vertices when only a single vertex remains with a single degree-1 hyperedge. DP then reconstructs the optimal state assignment in a final backtracking step.

Each step of dynamic programming reduces the original problem by eliminating x and redefining the scoring function for hyperedge V_x, which may need to be created. To see that the new problem instance is equivalent to the original one, I demonstrate an equality. For notational convenience, when the states of vertices in V are fixed, let the resulting set of scores F be partitioned into two: F_x, the scores for hyperedges containing x, and F_{V-x}, the rest of the scores. Then,

$$\begin{aligned} \max_{S_v \in \mathcal{S}(V)} \sum_{i \in F} i &= \max_{S_{V-x} \in \mathcal{S}(V-x)} (\max_{S_x \in \mathcal{S}(x)} (\sum_{i \in F_{V-x}} i + \sum_{i \in F_x} i)) \\ &= \max_{S_{V-x} \in \mathcal{S}(V-x)} (\sum_{i \in F_{V-x}} i + \max_{S_x \in \mathcal{S}(x)} \sum_{i \in F_x} i) \end{aligned}$$

The left-hand side of the equality is the optimal solution to the original problem. I can easily rewrite the right-hand side as the optimal solution to the reduced problem. Given a set of states S_{V-x}, let j denote the score for the hyperedge V_x if V_x exists and 0 otherwise. Then the optimal score for the original hypergraph, which is the right-hand

side above, is equal to

$$\max_{S_{V-x} \in \mathcal{S}(V-x)} \{(\sum_{i \in F_{V-x}} i - j) + (j + \max_{S_x \in \mathcal{S}(x)} \sum_{i \in F_x} i)\}$$

This is the optimal score for the modified hypergraph, since the first of the two parenthesized terms is the sum of the unmodified hyperedge scores, and the second is, by definition, the new hyperedge score for V_x.

4.16 Vertex Elimination Order

When eliminating a vertex x, dynamic programming creates a hyperedge V_x if the hyperedge V_x does not already exist in the graph. Define the *degree* of x to be $|V_x|$, the number of neighbors of x (note that this may be different than the number of incident hyperedges). The created hyperedge needs a table of scores whose size is the product of the number of states of the neighbors, $\prod_{v \in V_x} |\mathcal{S}(v)|$, and computing this table takes $\Theta(\prod_{v \in V_x \cup \{x\}} |\mathcal{S}(v)|)$ time. Since both time and space are exponential in the degree of x, it is fastest and cheapest to eliminate vertices with low degree.

The class of graphs that can be reduced to a single vertex by eliminating vertices of degree at most k has been characterized: they are the *partial k-trees* (Arnborg and Proskurowski, 1989). To refresh the definitions given in section 4.1, a *k-tree* is defined recursively: a $(k+1)$-clique is a k-tree, and any graph formed from a k-tree T by connecting a new vertex to each vertex of a k-clique of T is also a k-tree. A partial k-tree is a k-tree from which any number of edges have been removed. The treewidth of a graph, G, is the smallest k such that G is a partial k-tree.

For partial 3-trees there exists an elimination order in which vertices have degree at most three. In fact, Arnborg and Proskurowski (1986) gave a simple greedy algorithm for recognizing partial 3-trees that computes such an elimination order in $O(n \log n)$ time. Their algorithm maintains three queues for vertices of degree 1, 2, and 3, where a degree-3 vertex x is placed in the third queue only if local connectivity tests show that elimination of x would not increase the treewidth of the graph. The algorithm then eliminates any vertex from the first non-empty queue, and updates queues as degrees change. They prove that this algorithm finds an elimination sequence if and only if the graph is a partial 3-tree. The connectivity tests take logarithmic time, so determining an optimal vertex ordering takes $O(n \log n)$ time. Moreover, if the graph has treewidth $w \leq 3$, then each eliminated vertex has degree at most w.

Now, for an interaction graph in which the maximum number of states for a vertex is $s = \max_v |\mathcal{S}(v)|$, if dynamic programming is given an elimination sequence in which vertex degrees are at most w, it finds the optimum score in $O(ns^{w+1})$ time. This algorithm shows that optimizing hydrogen placement falls into the class of NP-Hard problems whose instances admit polynomial-time solutions when their underlying graphs have constant treewidth (Bodlaender, 1988; Arnborg and Proskurowski, 1989; Arnborg et al., 1991). For our special case of partial 3-trees, we can therefore observe:

Theorem 5 *Dynamic programming can compute the optimum score of an n-vertex interaction graph with treewidth $w \leq 3$ in $O(ns^{w+1}+n\log(n))$ time, where $s = \max_v |\mathcal{S}(v)|$ is the maximum number of states for any vertex.*

In practice, this algorithm can be extended to reduce graphs with treewidth greater than three to manageable size; simply place the degree-3 vertices rejected by the connectivity tests into a fourth queue, and eliminate vertices from this queue if the other queues have been exhausted. When all queues are empty, then resort to brute force enumeration of the states for vertices that remain in the graph. As a heuristic, I sort those vertices in the fourth queue by decreasing number of states so that dynamic programming eliminates vertices with the most states first, with the aim of maximally reducing the size of the state space that will remain when brute force enumeration begins.

4.17 Implementation

I have implemented both the decomposition of the dot-based scoring function into a hypergraph and the dynamic programming algorithm on a hypergraph in `C++`. I use one set of classes to represent a hypergraph for scoring, and another set of classes to represent a hypergraph for performing dynamic programming. For each problem instance, I first compute the scores in one phase and then optimize the scores in a second phase.

The dynamic programming hypergraph is designed to for partial 3-trees. I therefore represent hyperedges of degree at most four. That is, the implementation decomposes the scoring function into one-body, two-body, three-body, and four-body interactions and then into a catch-all set of higher-body interactions. For interactions that depend on four or fewer vertices, the implementation scores all involved dots in the initial scoring phase and stores those scores. If the implementation encounters dots that

participate in a five-body interaction, then it "punts." It will not score those dots in 5-way overlap as part of the scoring hypergraph, but rather, wait to score them until the brute force enumeration phase that follows dynamic programming. The implementation induces a 5-clique on vertices involved in the five-way interaction, so that dynamic programming will not remove any of these five vertices before brute force enumeration begins.

I separate the scoring and the optimization into two phases so that I can score the hypergraph once and save these scores for multiple optimization runs. Unlike brute force, which needed be run only a single time, dynamic programming must be run multiple times. The brute-force search of the original REDUCE reported not only the optimal network configuration, but also reported for each flippable group, i, the optimal network configuration with i fixed in its sub-optimal state. The difference between these scores is the score for flipping; it is not deemed worth flipping a group if the flipped state is only marginally better than the original. I run a second round of dynamic programming optimization to find this difference for residues that do flip. It would be possible to re-use some sub-problem solutions for these subsequent rounds of dynamic programming, thereby performing less work and saving time; however, dynamic programming runs so very fast that I have not explored this approach.

The new version of REDUCE scores the network once and records those scores in a hypergraph. It then solves the optimization problem using a second hypergraph built specifically for dynamic programming. After finding the optimal assignment of states, REDUCE iterates over the flippable groups, and instantiates a new dynamic programming hypergraph to solve another optimization problem, one smaller than the original: a problem where one of the vertices has half as many states. ASN and GLN each have a single state when restricted to the sub-optimal flip state; HIS has three states (protonation at ND1, NE2, or both) when restricted to the sub-optimal flip state. After each optimization, REDUCE deallocates the second graph.

The new version of REDUCE that includes the dot decomposition scheme and dynamic programming is faster, on one problem, by ten orders of magnitude, generally slicing through problems in seconds that previously would have taken years. The orders-of-magnitude performance gain achieved by the new implementation of REDUCE comes from the combination of the scheme to decompose the scoring function and dynamic programming. It is always gratifying when an algorithm means that previously infeasible problems are now readily solved.

Version 3.02 of REDUCE includes both the dot score decomposition algorithm and

dynamic programming, is now available from `http://kinemage.biochem.duke.edu/software/`.

4.18 Results: DP in HPP with REDUCE

I validated the dynamic programming implementation on a set of 2002 PDB files, containing 34,248 subproblems with non-trivial graphs. There were 86 subproblems whose state spaces had over 100,000 states. The previous version of REDUCE gave up on these 86 problems and took 6 hours 47 minutes to complete optimization of the remaining 34,162 subproblems in this test set. The dynamic programming / brute force hybrid took 56 minutes to optimize all subproblems, including the omitted 86. All runtime experiments were performed on a 2.5 GHz Mac G5 with 4 GB of RAM.

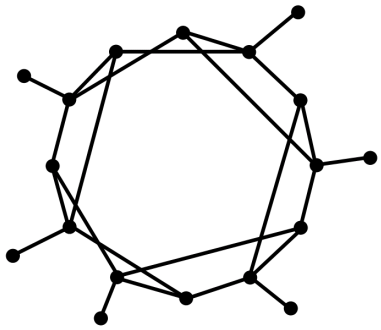


Figure 4.14: *Sole Treewidth-4 Interaction Graph.* The protein 4-oxalocrotonate tautomerase (1OTF) contained a large interaction graph with a treewidth greater than 3. Dynamic programming eliminated all leaf nodes, and then for the treewidth-4 core that remained, eliminated as many degree-3 vertices as it could. The six vertices left in the graph had a state space size of 64.

In the test set, I encountered one protein, 4-oxalocrotonate tautomerase (1OTF), with a subproblem whose graph had a treewidth greater than three. This subproblem was also the one with the largest state space size, $4.8\text{x}10^{12}$. I have illustrated the problem instance's interaction graph in figure 4.14. This problem did not contain any 3-way mover overlap. The strategy of eliminating as many degree-3 vertices as possible resulted in an irreducible core of six vertices with a state space size of 64. Brute force optimization of this irreducible core proceeded rapidly.

Of the 34,247 other graphs in this test set, two were of treewidth 3, and the rest were of treewidth 1 or 2. There was no mover overlap of higher order than three. There

PDB	$\|V\|$	$\|State\ Space\|$	*TW*	$\|H_3\|$	*Effort*	*Time (sec)*
2NLR	9	1.016×10^{6}	2	1	390	0.06
1HTP	7	1.143×10^{6}	2	0	675	0.09
1JDB	9	1.369×10^{6}	2	0	252	0.10
1A9X	9	1.493×10^{6}	2	1	252	0.12
1XVA	7	1.693×10^{6}	2	1	2246	0.09
1HVB	9	2.246×10^{6}	1	0	350	0.08
1EK6	8	2.281×10^{6}	2	2	1078	0.09
3PTE	9	2.546×10^{6}	1	0	370	0.08
1A8R	8	2.765×10^{6}	2	1	1844	0.13
1E4M	11	8.709×10^{6}	2	2	794	0.12
1H72	9	1.008×10^{7}	2	1	918	0.06
1JIL	12	1.739×10^{7}	1	0	242	0.04
1GM7	11	2.013×10^{8}	2	0	508	0.09
1A8R	11	2.395×10^{8}	2	2	1337	0.12
1PNK	11	3.137×10^{8}	2	0	650	0.10
1A8R	11	3.835×10^{8}	2	2	1198	0.12
1A8R	11	5.552×10^{8}	2	2	2262	0.47
1A8R	11	8.983×10^{8}	2	1	2290	0.17
1A8R	11	1.344×10^{9}	2	2	2191	0.19
1A8R	11	1.670×10^{9}	2	2	2758	0.19
1A8R	11	1.858×10^{9}	2	2	2216	0.18
1A8R	11	2.312×10^{9}	2	2	1963	0.15
1A8R	11	2.961×10^{9}	2	2	2704	0.48
1A8R	11	3.064×10^{9}	2	2	1974	0.16
1QRR	11	2.008×10^{10}	2	6	7317	0.98
1OTF	18	4.812×10^{12}	4	0	786	0.08

Table 4.3: *Large Interaction Graphs.* Interaction graphs in my test set that have more than 1 million possible state combinations. The column $|V|$ reports the number of vertices in each interaction graph. There were few cases of degree 3 overlap that required representation with hyperedges of degree 3 ($|H_3|$). Only one graph had a treewidth (*TW*) greater than 3. After dynamic programming eliminated all but 6 of the vertices from this graph, enumeration of their states completed the optimization. The *Effort*, reported in the second to last column, represents the number of state combinations examined during dynamic programming (and, for 1OTF, in the round of brute-force enumeration that followed). The optimization itself would be described by the user as instantaneous; the running time is dominated by the precomputation of dot scores. GTP Cyclohydrase I (1A8R) contained several large connected components, and so it appears in several places in this table

was no four-way overlap, so whereas the degree-4 hyperedges proved unnecessary, the strategy of decomposing the scoring function into interactions of order 4 or less was sufficient to handle all observed input cases. I present a short list of the largest problems we encountered in this test set in Table 4.3. The majority of these problems were treewidth 2, and could be solved without resorting to brute force enumeration on a reduced problem. Our dynamic programming / brute force hybrid is able to solve each subproblem in less than a second. The speed-up that our improved algorithm represents can be measured by comparing for these large problems their state space sizes with the effort (in the second-to rightmost column of Table 4.3)that dynamic programming required to optimize them. In the case of 1OTF, that speedup is almost ten orders of magnitude.

4.19 Discussion of DP in HPP and SCPP

The difference between dynamic programming's success at the hydrogen-placement problem in REDUCE and its relative failure at the side-chain-placement problem in Rosetta is due to the differences in treewidth of the interaction graphs most commonly observed. The interaction graphs in the hydrogen-placement problem have a lower treewidths than those from the side-chain-placement problem for three reasons: the short range of the energy/scoring function, the short spatial reach for vertices in the graph, and the lower density of vertices in the protein. Consider each in turn.

The shorter the range on the energy function, the fewer the things that interact. In the hydrogen-placement problem, the range of the scoring function, if put in terms of the surface-to-surface range is zero. Two atoms must overlap to interact. On the other hand, in Rosetta, the Lennard-Jones attractive term is non-zero at a distance of 5.5 Å between two atom centers – for a pair of oxygen atoms with a van der Waals radius of 1.4 Å, the equivalent surface-to-surface range is 2.7 Å.

Although this difference in energy-function range is part of the difference in resulting treewidths, it is slight in comparison to difference in the *spatial reach* of each vertex – the size of the region in space that a vertex may position its atoms in. In the hydrogen-placement problem the spatial reach of a vertex is very small. The spatial options for a hydroxyl hydrogen lie on the perimeter of a circle in space. The diameter of this circle is $\sqrt{3} \approx 1.73$ Å, since the ideal C-O-H bond angle is 120° and the O-H bond length is 1.0 Å. Similarly, the spatial options for an ASN or GLN vertex is between one of two flip states. When the vertex flips the group, the oxygen and nitrogen swap coordinate

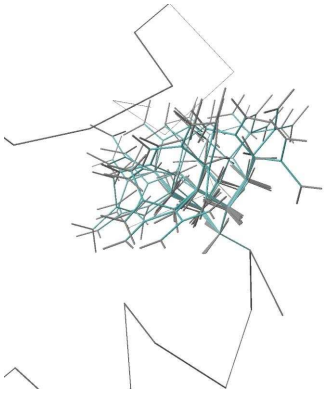


Figure 4.15: *Spatial Reach for Side-Chain Placement.* Twenty-seven rotamers for arginine attached to a single residue. The rest of the protein is shown as a *Cα trace* – straight lines drawn between successive Cα atoms. The distal end of one rotamer is 7 Å from its Cβ atom, and two rotamers have atoms that are 10 Å from each other. This figure was generated with Render3D (Merritt and Bacon, 1997)

centers; what differs between the two flip states is where the hydrogens protrude. The hydrogen sticks out from the nitrogen of the amide by 1 Å, and because its van der Waals sphere has a radius of 1 Å, it reaches out only 0.45 Å past the van der Waals sphere of the nitrogen, which has a radius of 1.55 Å. The region of space the vertex could possibly occupy but does not necessarily occupy is small. In contrast, in the side-chain-placement problem, the spatial reach for each vertex is large. For example, the tips of two arginine rotamers from the same residue can be 10 Å apart (Figure 4.15). Because vertices in the graph can reach out further, the number of other vertices that a single vertex may interact with is far greater. The more residue interactions, the higher the connectivity of an interaction graph. The higher connectivity roughly corresponds to a higher treewidth.

The final reason that treewidths are low for the hydrogen-placement problem is the relative low frequency of ambiguous groups. In REDUCE, the hydrogen-placement is ambiguous for ASN, GLN, HIS, SER, THR, and TYR residues only: at a rough approximation, only 30% of residues need to be optimized. Complete-protein redesign optimizes 100% of the residues. Whereas the low frequency of ambiguous side chains in the hydrogen-placement problem means that the protein's interaction graph consists of dozens of small connected components (with each component having one or two vertices and very rarely more than six), the high frequency of ambiguous side chains in the side-chain-placement problem means that the protein's interaction graph consists

of a single large connected graph.

The treewidth can be understood as the size of the set of *vertex cuts* used to create the tree decomposition – a vertex cut is a set of vertices that, if removed, disconnect the graph. Vertex cuts for spherical graphs are large; the cuts to divide the graph in two equal sections pass through the center of the sphere. If a protein's interaction graph is a single connected component, the graph is roughly spherical and has a high treewidth.

In March of 2005, not long after my January of 2005 publication of the adaptive dynamic programming algorithm, Xu published a dynamic programming algorithm on interaction graphs of low treewidth for the side-chain-placement problem (Xu, 2005) and implemented the algorithm specifically for the rotamer-relaxation problem, which is the variation on the side-chain-placement problem that keeps the amino acid sequence fixed. The original software that Xu built upon, SQWRL, created by Roland Dunbrack's group (Bower et al., 1997; Canutescu et al., 2003), included only two terms in its energy function: a one-body internal rotamer strain term, and a two-body collision term. Xu included edges in the interaction graphs only if there existed a pair of rotamers for those residues that collided with at least a 10 kcal/mol energy (a dramatically higher interaction threshold than the 0.2 kcal/mol threshold I used). His threshold means that residues must not only collide to interact, but they must also collide badly. The upshot is that, with this edge inclusion threshold, Xu mostly observed treewidth-3 and treewidth-4 interaction graphs and dynamic programming performed well. He later noted that it performed better on average than integer linear programming (Xu et al., 2005).

Though the rotamer strain and collision energy function works well enough at predicting side-chain conformations (Bower et al., 1997), it is insufficient to design proteins (Kuhlman, personal communication). The main conclusion of this chapter is that although dynamic programming provides a nice theoretical result for the complexity of protein design, it is practical only to help calibrate the stochastic optimization techniques that designers employ in the majority of their design simulations; time and memory are far more precious to designers than absolute certainty of globally optimal side chain placements, which is why all of the designers I have spoken with rely on stochastic techniques (Kuhlman, Baker, Mayo, and Hellinga, personal communication). The interaction graph formulation, on the other hand, has more to offer than the dynamic programming algorithm. I describe its other offerings in Chapters 5, 7, and 8.

Chapter 5

An Interaction Graph As A Data Structure

This chapter demonstrates how an interaction graph can serve as a data structure for the storage of rotamer-pair energies in protein design software. I have refactored the design module of Rosetta, *the packer*, by incorporating the interaction graph. The previous sparse matrix was a monolithic data structure – a single contiguous block of memory – which made it very difficult to work with. The interaction graph replaces it with a set of smaller structures that are easier to work with (e.g. they allowed a trivial fix for a long-standing bug in the packer). The interaction graph reduces memory usage by 12% and maintains the speed for retrieving energies during simulated annealing.

I define an interface between the interaction graph and the simulated annealing algorithm that allows the incorporation of new energy functions. On one side of the interface sits the simulated annealing algorithm (the *annealer*), which makes decisions about which rotamer substitutions to accept and which to reject. On the other side of the interface sits the interaction graph, which is responsible for informing the annealer of the change in energy produced by the rotamer substitutions that the annealer considers. Because the interaction graph reports the change in energy for a rotamer substitution, it is responsible for representing the energy function. The interface can thus be viewed as between the annealer and the energy function. The interface means that various interaction graphs are interchangeable. The same annealer that optimizes Rosetta's pairwise decomposable energy function as represented by one kind of interaction graph can optimize novel non-pairwise decomposable energy functions as represented by another kind of interaction graph. The interface provides important extensibility for protein design software.

Although the idea of an interface between the annealer and the energy function does not require that the energy function be represented with an interaction graph, I show an interaction graph is nonetheless a good way to represent an energy function. The interaction graph eases the implementation of new energy functions, and leads to clear programming. Graphs generally make intuitive the storage of information about entities and their relationships. Vertices represent entities and edges represent their relationships. If information needs to be stored about an entity, then that information becomes a data member of the vertex class. If information needs to be stored about the relationships between entities, then that information becomes a data member of the edge class. The interaction graph thus makes intuitive the storage of information for residues and their interactions. For this reason, I have used an interaction graph in each of the extensions I have made to the packer.

To provide the context for the incorporation of the interaction graph into Rosetta, I begin with a description of the packer which provides a generic side-chain-placement subroutine (Kuhlman and Baker, 2000; Kuhlman et al., 2002; Kuhlman et al., 2003). This subroutine attempts to solve both the fixed-sequence side-chain-placement problem and the design problem. The Rosetta community relies on this subroutine for a variety of tasks, including protein design (Sood and Baker, 2006; Palmer et al., 2006; James J. Havranek and Baker, 2004; Ashworth et al., 2006; Joachimiak et al., 2006), high resolution protein structure prediction (Bradley et al., 2003; Grana et al., 2005; Bradley et al., 2005; Schueler-Furman et al., 2005), and protein/protein docking (Gray et al., 2003; Wang et al., 2005). Because Rosetta is used so broadly, the improvements I have made to the packer have reached a broad user base and are thus able to benefit a wide range of problems in computational structural biology.

5.1 The Packer

The packer solves the side-chain-placement problem. It takes as input a protein or several proteins, and a description of which residues it should seek an optimal side-chain placement for, the *molten residues*. Its output is one or several protein structures (one structure handed back to the calling function, or several structures written to output files) whose side chains have been optimized according to the input instructions. The packer operates in three stages, which I expand upon in the subsequent sections.

- *Rotamer Creation.* The packer identifies the molten residues and the set of which amino acids to allow at each position. It then builds rotamers for each of the

residues, and tests to see if they are in serious collision with the background (the backbone and the fixed side chains). It discards any rotamers that collide with the background.

- *Rotamer-Pair-Energy Calculation.* Next the packer iterates across all residue pairs, and evaluates the energy for each rotamer pair. It stores these rotamer-pair energies in, `energy2b,` a large table of two-body energies. It also calculates the one-body energies, the interactions of each rotamer with the background atoms and stores these energies in another table, `energy1b.` This is the packer's most time consuming stage.

- *Optimization by Simulated Annealing.* The packer optimizes side-chain placement by making random substitutions to a working rotamer assignment, rejecting or accepting those substitutions according to the Metropolis criterion. During the course of the minimization, the packer slowly lowers its operating temperature so that over time it biases itself towards lower and lower energy conformations.

5.1.1 Rotamer Creation

The rotamer creation stage begins with the identification of the molten residues, and for each molten residue, which of the amino acids to allow. By default, the packer treats each residue as a molten residue. In design mode, the packer optionally reads from an input file that identifies the molten residues and the background residues, and for each molten residue, identifies which amino acids to consider. If no such input file is given, the packer defaults to considering all amino acids at all positions. In docking mode (interface design mode), the packer determines which residues make up the interface and repacks (redesigns) only those residues. In fixed-sequence mode (docking or folding), the packer preserves the original amino acid sequence present in the input structure.

The packer then iterates across the molten residues and builds rotamers for each – it computes coordinates for rotamer atoms. In building rotamers, the packer first selects the appropriate χ dihedral angles to sample. It interpolates the χ dihedrals reported in Dunbrack's backbone dependent rotamer library: this library reports preferred χ angles at 10° bins for ϕ and ψ. For residue i, the packer interpolates the χ angles contained in the four ϕ and ψ bins that surround the ϕ_i and ψ_i. Having computed the χ dihedrals, the packer computes rotamer coordinates using idealized bond geometries, vector geometry, and the computed χ angles.

After the packer builds rotamer r, it computes the interaction energy between r and the background in a stage called *bump-check*. If r's energy exceeds some positive threshold (*e.g.* the rotamer collides with the background and is thus at a high energy), then the packer discards it. It is possible for the packer to build a set of rotamers for a residue, find that each rotamer in the set collides with the background and discard each rotamer. In that case, it defaults to preserving the rotamer present on the input structure.

5.1.2 Rotamer-Pair-Energy Calculation

The rotamer-pair-energy calculation stage follows rotamer creation. The packer iterates across all interacting residue pairs and iterates across all rotamer pairs on those two residues, evaluating the rotamer-pair energies for each and storing their interaction energies in a large table. I go into great detail on the calculation of these rotamer-pair energies in Chapter 6. Rotamer-pair-energy calculation dominates the running time.

The short range of Rosetta's energy function lets Rosetta save memory. If the energy function were long-ranged, then an interaction energy would have to be stored for each rotamer pair; a design that used more than 45 thousand rotamers would have to store more than 1 billion rotamer-pair energies, using more than 4 GB of memory. Rosetta's short-ranged energy function lets the packer get away with a sparse matrix representation of the rotamer-pair energies. This means that designers can use more than 45 thousand rotamers in their designs – I have been able to run complete protein redesigns that included 93 thousand rotamers, while keeping the memory usage under 4 GB.

5.1.3 Simulated Annealing

Simulated annealing works by assigning each of the molten residues a rotamer and then making substitutions to that assignment that bias the assignment toward lower energies. The annealer has to know what energy any one of the rotamer assignments have so that it can properly bias the substitutions.

With a pairwise decomposable energy function, the energy of the protein under a particular rotamer assignment is the sum of the rotamer-pair energies, the rotamer one-body energies, and the sum of the background/background interaction energies. Since the background/background interaction energies do not change for different rotamer assignments, they need not be considered as part of the optimization. All the packer

needs to compute are the rotamer pair energies and the rotamer one-body energies.

As mentioned before, in the rotamer substitution step, simulated annealing considers replacing one rotamer in the rotamer assignment (the *current rotamer*) with some other rotamer (the *alternate rotamer*). The annealer calculates the change in energy induced by this rotamer substitution by looking at two energies: the sum of the interaction energies the current rotamer has with all of its neighboring molten residues and with the background and the sum of the interaction energies the alternate rotamer has with its neighboring molten residues and the background. The difference between these two energies is the difference in the energy for the entire protein.

The packer can exploit the pairwise decomposability to gain speed in two ways. First, the packer can precompute rotamer-pair energies, so that energies need not be computed during simulated annealing. Energy calculation is expensive, and the annealer often requires knowing one interaction energy repeatedly. (However, the annealer rarely looks at all of the rotamer-pair energies in the energy calculation stage. An strategy alternate to precomputing all rotamer-pair energies would be to compute only those rotamer-pair energies that actually get used; that is, procrastinate each energy computation until the annealer requires it, and then store each energy after it has been computed. Such a strategy is also benefited by the pairwise decomposability of the energy function.)

Second, pairwise decomposability means that a rotamer substitution at one residue has no effect on the interactions between other residues. In computing the ΔE from a substitution, the packer needs only examine interaction energies between the residue undergoing substitution and its neighbors. This cuts the number of energy retrievals for an N residue protein by a factor of N: for a long-ranged energy function, only $O(N)$ interactions with the alternate rotamer need to be examined, as opposed to the $O(N^2)$ interactions present in the protein, for a short-ranged energy function, only $O(1)$ interactions with the alternate rotamer need to be examined, as opposed to the $O(N)$ interactions present in the protein.

Although a typical call to the packer spends very little time in simulated annealing, some calls to the packer run simulated annealing many times, so that simulated annealing takes over as the most time consuming stage. For this reason, I have been careful to make sure that my refactoring has not slowed simulated annealing. When Rosetta is folding or docking proteins, the packer runs a single simulated annealing trajectory and spends about 1% of its time in simulated annealing. In protein design, however, the packer often runs hundreds of simulated annealing trajectories after a

single rotamer-pair-energy computation – after all, simulated annealing is a stochastic optimization technique offering no quality guarantee and the expense of computing rotamer-pair energies has already been paid; running several trajectories gives a better chance of finding lower energy designs. Furthermore, in Harrison and Gray's pKa calculations (Harrison and Gray, 2005), they ask the packer to run thousands of simulated annealing trajectories and measure the number of annealing steps that each chemical group spends in its protonated and deprotonated states. Because simulated annealing converges to the Boltzmann distribution, the proportion of time in simulated annealing that the chemical groups spends in these two states reveals its pKa. (Briefly, pKa is the pH at which an acid spends half of its time protonated and half deprotonated. For example, aspartic and glutamic acid side chains both have low a pKa ($\sim$4) and are usually deprotonated at physiological pH ($\sim$7)).

5.2 Previous Rotamer-Pair-Energy Storage

The previous representation of the rotamer-pair-energy table was as one contiguous sparse matrix, a table `energy2b` (energy, 2-body) which relied on three offset arrays that were used to index into it. This table represents only a small subset of the rotamer-pair energies that would be stored for n rotamers in a hypothetical $n \times n$ that held an entry for each rotamer pair. Call this hypothetical $n \times n$ table T. The structure of `energy2b` is easiest to describe by noting the regions of T that `energy2b` does not allocate memory for.

First, T is symmetric. Therefore `energy2b` does not allocate space for any part of the lower triangle of T (Figure 5.1a).

Next, pairs of rotamers that originate from the same residue can never be assigned simultaneously, so there is no way they can interact. `energy2b` allocates blocks of T. The rotamers for all residues are sorted by the residue that they originated from, so that all rotamers on the same residue are next to each other. The blocks of T that represent the interactions between pairs of rotamers that originate from the same residue look like steps along the diagonal (Figure 5.1b). `energy2b` does not allocate these blocks.

Finally, for short-ranged energy functions only, not all rotamers interact. `energy2b` uses the technique of allocating blocks to store subsections of T; it uses this technique twice. First, `energy2b` groups rotamers together by the residue they originated from; for example, all rotamers from residue i are grouped together and all rotamers from residue j are grouped together. Then if there are no rotamers from i and j that interact,

`energy2b` does not allocate space for any rotamer pairs in this block (Figure 5.1c). Second, `energy2b` groups rotamers together from the same residue by their amino acid type: for example, all of the serine rotamers on residue i are grouped together, and all the arginine rotamers on residue j are grouped together. Then if there are no serine rotamers from residue i that interact with any arginine rotamers from residue j, then `energy2b` does not allocate space for any rotamer pairs in the block between the serines and arginines on this residue pair (Figure 5.1d). Since pairs of serine side chains are less likely to interact at long range than a serine/arginine pair or a pair of arginines,

This second of the two sparse-matrix-by-blocks techniques represents a significant space savings. I have refactored the original code that supports this technique into its own class, the `AminoAcidNeighborSparseMatrix` class. I like to think of this technique as keeping track of "amino acid neighbors:" some amino acid pairs may neighbor one another for one residue pair (e.g. a pair of arginines) while other amino acid pairs for that same residue pair might not (e.g. a pair of serines). In instances of protein design, I have observed this technique to yield a 40% reduction in memory over the picture painted in Figure 5.1c.

Because `energy2b` is allocated as a single contiguous block of memory, it must be allocated before rotamer-pair-energy calculation begins. Therefore, the packer must predict without actually calculating any interaction energies which pairs of residues interact, and for the interacting residue pairs, must predict which pairs of amino acids interact. First, the packer examines the Cβ distances for all residue pairs and compares them against a threshold of 16 Å; any residue pair with a Cβ distance less than this threshold is presumed to interact. Second, for each interacting residue pair, the packer compares their Cβ distance against a set of amino-acid-pair-specific cutoffs. For a pair of alanines, their Cβ atoms must be within 8 Å to interact; for a pair of tryptophans, their Cβ atoms must be within 15 Å to interact. After predicting which blocks of T contain interactions, Rosetta allocates the `energy2b` array, providing exactly as much room as can store those energies it has predicted could be non-zero.

Given that `energy2b` is restricted to not represent both halves of the symmetric energy table, it is laid out for optimal cache efficiency for energy retrieval in simulated annealing. When simulated annealing considers substituting rotamer r with rotamer r' on residue i, it has to retrieve all energies that rotamer r' has with each rotamer assigned to the neighboring residues. These neighboring residues can be divided into those with a higher residue number and those with a lower residue number. `energy2b` lays out its memory so that all of the interactions that rotamer r' has with the residues

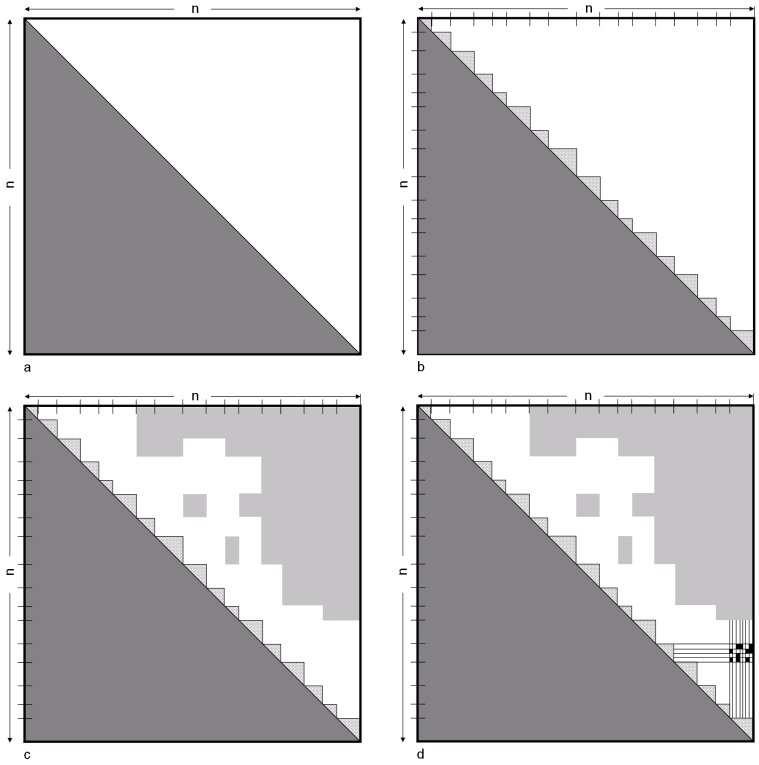


Figure 5.1: *Previous Two-body Energy Table* An illustration of what regions of the complete $n \times n$ table that `energy2b` allocates space for; darkened regions are not allocated. `energy2b` does not allocates space to store rotamer-pair energies for a) the lower diagonal, b) rotamers from the same residue, c) residue pairs that have no interacting rotamers, and d) amino acid pairs with no interacting rotamers. The picture in d) only illustrates one residue pair for which the amino-acid neighbor sparse matrix representation has carved away space from the remaining $n \times n$ table, but representation applies to the entire table.

of higher index are in a contiguous block of memory. That is, the location in `energy2b` of the energies that r' has with the larger-indexed residues can be visualized as a row going across the upper triangle in Figure 5.1d. In this row, the interaction energy that rotamer r' has with the assigned rotamer s on residue j is separated from the interaction energy that rotamer r' has with the assigned rotamer t on residue $j+1$ by only the set of interaction energies that rotamer r' has with the other rotamers of residues j and $j+1$ – a small enough number that for the most part, r''s interactions with s and t are in the same line of cache. Thus, the energy retrievals for substituting r' for r on rotamer i incur one cache miss per residue for those residues with a smaller index than i, and one cache miss for the entire set of residues with larger indices than i.

On average, if each residue has n neighbors, then the average cost of retrieving the interaction energies of a rotamer is $n/2 + 1$ cache misses.

I postulate that the most time-efficient way to lay out rotamer-pair energies for retrieval in simulated annealing would cost a single cache miss per rotamer substitution. If memory were laid out so that all of the interaction energies that rotamer r' had with the rest of the rotamers in the protein were stored next to each other, and if these interactions all fit in a single line of cache (they might not), then a retrieval for a rotamer substitution would cost a single cache miss. However, this scheme would use twice as much memory.

5.3 Interaction Graph Data Structure

The interaction graph classes do three things: they stores rotamer-pair energies, they encapsulates complex bookkeeping for the support of the most time consuming step of the simulated annealing – the computation of ΔE for a single rotamer substitution – and they provide a framework for extending the packer.

An interaction graph can model a pairwise decomposable energy function as follows. An interaction graph is a graph $G = \{V, E\}$, where each vertex $v \in V$ carries a state space, $\mathcal{S}(v)$. Each vertex v carries a scoring function F_v that maps from the state space of v into the reals: $F_v = \mathcal{S}(v) \rightarrow \Re$. Additionally, each edge $e \in V \times V$ carries a scoring function, F_e defined as a mapping from the Cartesian product of the state spaces of its vertices to the reals: $F_e = \prod_{v \in e} \mathcal{S}(v) \rightarrow \Re$.

In this graph, molten residues are represented as vertices. The set of allowable rotamers for a single residue are represented as the vertices' state spaces. The rotamer/background energies are captured in the vertex scoring functions F_v and the rotamer-pair energies are captured in the edge scoring functions F_e.

This model is readily turned into an implementation. In object oriented terms, there are three classes: `class Vertex`, `class Edge`, and `class Graph`. Vertex v keeps a one-dimensional table of energies as a data member to store the vertex scoring function, F_v. Edge e keeps a two-dimensional table of energies as a data member to store the edge scoring function, F_e. The graph G maintains an array of vertices and a list of edges.

The annealer uses two methods to interface with the interaction graph:

- `float deltaE_for_substitution( node, alt_state)`
- `void commit_considered_substitution()`

Through the first of these two methods, `deltaE_for_substitution(node,alt_state)`, the annealer asks the interaction graph for the change in energy, ΔE, induced by substituting an alternate state on a particular node in the graph for which ever state is currently assigned to that node. The interaction graph returns this answer to the annealer, which then decides whether or not to commit that substitution. If the annealer decides to commit the substitution, then it informs the graph of its decision through the second of the two methods, `commit_considered_substitution()`. If the annealer does not invoke this second method before the next call to `deltaE_for_substitution( node, alt_state)`, the graph assumes the annealer has rejected that rotamer substitution.

In order for the graph to compute the change in energy induced by a state substitution, it must know the current state assignment. When a node considers a change of state, it retrieves the interaction energies on its incident edges that the alternate state has with the currently assigned rotamers on the other states. The edges have to know what states are currently assigned to the neighbors of the changing node: either the annealer could provide this information to the graph in the form of an additional vector parameter to the `deltaE_for_substitution` method, where the vector holds the current state assignment for all vertices in the graph, or the graph could track this information by keeping track of each time the annealer commits a rotamer substitution. For the sake of speed, I have chosen to have the graph track this information; by storing the current state assignment, the interaction graph is able to perform time-saving bookkeeping. The consequence of this decision is that the annealer must inform the interaction graph about the rotamer substitutions it commits.

5.3.1 Extensions Through Abstractions

By generalizing away from residues and rotamers into vertices and states, the same abstraction for side-chain packing can store energies to perform other similar optimizations. For instance, an interaction graph can model ligand flexibility during side-chain placement. Suppose we were interested in designing a receptor protein for 1,3,5-N-butyl benzene (Figure 5.2). In one design approach, we would consider one fixed position of the benzene ring in the binding pocket of the target protein at a time. For each fixed position, we would simultaneously optimize the conformation and amino acid identity of the binding pocket residues and the conformation of the butyl "side chains" of the ligand. We could then repeat the optimization for a large number of ring positions.

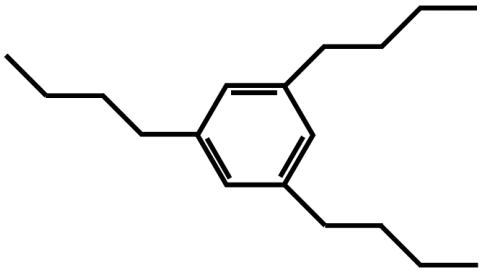


Figure 5.2: *A Flexible Ligand* One approach to designing a protein that can bind the ligand above, 1,3,5,-N-butyl benzene would be to sample many different rigid orientations of the benzene ring inside the binding pocket of a perspective protein, and for each sampled ring position, to simultaneously optimize the butyl "side chains" of the ligand and the sequence of the protein. Modeling this problem with an interaction graph, the ligand's three butyl groups each become vertices in the graph.

This approach could be modeled with an interaction graph where in addition to the vertices corresponding to the molten residues of the redesigned binding pocket, the graph include three extra vertices, one for each butyl group of the ligand. The same algorithm that optimizes an interaction graphs created to model simple protein-side-chain placement is also able to optimize an interaction graph created to model protein-side-chain placement with ligand flexibility.

5.3.2 Class Responsibilities

The hierarchy divides the data stored in the interaction graph into two parts: the data responsible for maintaining a graph, which is reused among all graphs in the hierarchy, and the data responsible for representing the energy function, which differs among all graphs. The base classes, the `InteractionGraphBase` class, the `NodeBase` class, and the `EdgeBase` class, contain data and methods required to maintain a graph. These three classes are abstract – their purpose is not to be instantiated but to be inherited from. The derived classes are responsible for representing the energy function. For example, the `PDInteractionGraph` class is a concrete class that inherits from the `InteractionGraphBase` classes. This class, along with the `PDNode` class and `PDEdge` class, replaces the `energy2b` matrix responsible for representing a pairwise decomposable (PD) energy function. These concrete classes are responsible for holding the tables to store the rotamer-pair energies.

The rest of this chapter describes the functionality provided by the base classes, it

describes the memory usage and the performance issues in simulated annealing of the derived `PDInteractionGraph` class, and it concludes with a graphical model and a set of abstract classes for representing non-pairwise decomposable energy functions.

5.3.3 Base Classes

The base classes provide structure for the derived classes and provide various methods to those derived classes with certain performance guarantees.

Each `NodeBase` object is given a unique index, from 1 to $|V|$ (Rosetta was originally written in `FORTRAN`, and still indexes from 1 after the port to `C++`). The `NodeBase` class holds a list of pointers to `EdgeBase` objects, an edge list, such that edges connecting the node to nodes of smaller index are in the first half of the list, and edges connecting the node to nodes of larger index are in the second half. Each `NodeBase` object keeps an integer `numStates_`, representing the number of states available to it.

The `EdgeBase` class contains a two-element array of pointers to the two vertices it is incident upon. Edges are undirected, therefore they order these pointers according to the nodes' indices. Edges keep the indexes of their incident vertices, in a two-element array `nodeIndices_`, again ordered by node index. The edge also stores the number of states for each of the two vertices to keep on hand.

The base classes provide two important performance guarantees. First, edges provide constant time self-deletion: to do so, they keep iterators to their positions in the edge lists of the nodes they are incident upon. Each edge upon construction hands a pointer to itself to the two nodes it is incident upon; the nodes then insert the edge into their edge list and return an iterator to the newly inserted list element. The `EdgeBase` destructor hands back this iterator to the `NodeBase` objects so that they may remove the list member in constant time.

Second, in addition to an edge list, each `NodeBase` object also maintains an edge vector, which it updates immediately before simulated annealing begins. This vector ensures that a vertex can iterate across all of its incident edges at the expense of a single cache miss (though it should be noted that dereferencing an edge pointer may cost a cache miss as well) and that it may access any particular edge in constant time.

The construction of the edge vector effectively divides the graph behavior into two phases: the phase during which the edge vector is not up-to-date, and the phase in which it is. Performance guarantees are provided for the phase in which it is up-to-date, moreover, correctness guarantees for operations on this vector are only provided

after an update operation is performed. The derived classes must know when they want to use the edge vector that they must perform the update operation first. A class derived from `NodeBase` may access one of its incident edges in constant time through the protected method `EdgeBase* getEdge( int edgeIndex );`

Once the edges are put into a vector on a vertex, they have effectively been enumerated. This is useful if an edge is given the responsibility of passing information to a vertex – either information about itself or about the other vertex that the edge is incident upon. If edge e had to give a vertex v a piece of information specific to e for v to hold, e must tell v which edge it is. This could happen in one of two ways. e could pass v a pointer to itself along with the information it was passing so that then v could compare that pointer against all the edge pointers in its edge vector, taking $|E|/|V|$ time on average. Or e could tell v exactly which edge of v's it is, *e.g.* "I'm your third edge," taking constant time. After a vertex converts the edge-list representation into an edge-vector representation, it iterates across its edges and informs them what position they are in its edge vector; each edge keeps track of its position so the edge can later give this information back to the vertex. The `EdgeBase` class keeps this data in the two-element array `posInNodesEdgeVect_`, again ordered by node index. The `posInNodesEdgeVect_` array provides a performance guarantee: an edge can communicate information about itself to a vertex in constant time.

The `InteractionGraphBase` itself manages the nodes and edges of the graph and provides a clean interface to the rest of Rosetta; any section of code that wishes to interact with some node or edge of the graph may keep a pointer to the graph and access data on the nodes and edges through member functions provided by the graph – the section of code need not keep pointers to all of the individual nodes and edges. The `InteractionGraphBase` object maintains a vector of node pointers, taking the number of nodes in the graph as an input parameter in its constructor. It maintains a list of edge pointers (and `EdgeBase` objects maintain iterators to their positions in this edge list, to ensure constant-time edge deletion). Both `NodeBase` and `EdgeBase` objects maintain pointers to the graph that owns them; both classes take a pointer to their owner in their constructor.

The `InteractionGraphBase` relies on two abstract polymorphic methods for node and edge creation.

- `virtual NodeBase * createNode( int nodeIndex, int numStates ) = 0;`
- `virtual EdgeBase * createEdge( int node1, int node2) = 0;`

The concrete derived graph classes must provide the actual implementation. These methods follow the *factory method* design pattern (Gamma et al., 1995). The factory method pattern solves the design problem of needing a base class to initialize objects of unknown type; derived classes specify exactly which object to instantiate but leave the rest of object initialization to the base class. In code, the factory method is incredibly simple. For instance, for the `PDInteractionGraph`, the `createNode` method is as follows:

```
NodeBase *
PDInteractionGraph::createNode( int nodeIndex, int numStates )
{
   return new PDNode( this, nodeIndex, numStates );
}
```

Derived classes simply specify which Node and Edge classes they work with. The `InteractionGraphBase` adds edges as follows

```
void
InteractionGraphBase::addEdge( int node1, int node2 )
{
   EdgeBase * newedge = createEdge( node1, node2 );
   ++numEdges_;
   edgeList_.push_front( newedge );
   newEdge->setPositionInOwnersEdgeList( edgeList_.begin() );
}
```

The `InteractionGraphBase` destructor deletes all edges from the graph and then deletes all nodes from the graph. The deallocation must take place in this order so that nodes and their edge lists still exist at the time edges try to remove the list elements that point to themselves from their list. In general, the assumption is that vertices outlive the edges that are incident upon them; the destructor enforces this assumption.

The data members of the three base classes are all private. Read access is granted to all derived classes through a set of protected methods. The base classes would be unable to guarantee proper performance if outside classes were able to manipulate their internal data; making data private and not simply protected ensures that derived classes do not disrupt data in base classes either. (This is a well known rule for object oriented design, but I would like to acknowledge Scott Meyers for making the necessity

of this rule exceptionally clear [Meyers, 2005].) While the base classes should not have their data written to by derived classes, they most certainly should have their data read by derived classes, or the purpose of creating a class hierarchy would be lost. Not only should the derived classes have read access to the data in the base classes, they should be able to read that data as rapidly as if it were there own. For this reason, the implementation of each of the protected read methods is included in the header file of the base classes, and is therefore implicitly inlined during compilation.

5.3.4 The `PDInteractionGraph` Derived Classes

The interaction graph's first task was to represent the pairwise decomposable energy function already present in Rosetta. The `PDInteractionGraph` and its two associated classes, the `PDNode` and the `PDEdge`, together handle the representation of this energy function.

Each `PDNode` keeps one integer `currState_` that represents its currently assigned state an another integer `altState_` that represents the alternate state it was most recently asked to consider. Each `PDEdge` keeps a two-element array `nodeCurrStates_` representing the states currently assigned to its two vertices.

The `PDNode` class maintains an array of `float`s, `one_body_energies_`, that stores the one-body energies for each of its rotamers. The `PDEdge` class maintains a table of `float`s, `two_body_energies_`, that stores the rotamer-pair energies for each pair of rotamers for the two vertices it is incident upon. The graph includes edges only for those residue pairs for which there exists at least one rotamer pair that interacts with non-zero energy. If each edge allocated a table to hold as many pair energies as there were rotamers on the two vertices, then the graph would allocate as much memory as pictured in Figure 5.1c – it would allocate rotamer-pair energies for some residue pairs but not all. The first three aspects of the energy savings in `energy2b` come with little effort in the interaction graph setup. As mentioned before, the memory use pictured in Figure 5.1d is 40% less than that pictured in Figure 5.1c. Since the amount of memory that goes into rotamer-pair-energy storage is on the order of gigabytes, this difference is unacceptable.

Memory Efficiency

To capture the efficiency of the scheme pictured in Figure 5.1d, I have refactored into a class the previous sparse matrix technique present in `energy2b` that kept track

of which amino acids neighbored each other on a residue pair. I call this class the `AminoAcidNeighborSparseMatrix` class. This class allocates a single contiguous block of memory for a residue pair and allocates space to hold energies for those amino acid pairs with rotamers that interact with non-zero energy.

The class needs to know 1) how many different amino acid types there are (usually twenty), 2) how many rotamers each residue has, and 3) how many rotamers of each amino acid type each residue has. In its constructor, the `AminoAcidNeighborSparse-Matrix` takes two references to arrays of integers, one for each edge, each with as many entries as there are different amino acid types, and each entry represents the number of rotamers each residue has of that amino acid type. It keeps references to these arrays instead of making copies of these arrays – the implicit assumption is that the lifetime of these input arrays is longer than the lifetime of the `AminoAcidNeighborSparseMatrix` class. The input arrays are, in all cases I've created, data members of node classes, while the `AminoAcidNeighborSparseMatrix` objects are data members of the Edge classes. As previously mentioned, the `InteractionGraphBase` class's destructor enforces that the lifetime of the nodes is longer than that of the edges.

The `AminoAcidNeighborSparseMatrix` allocates two tables; one table holds the rotamer-pair energies, the second table holds offsets into the first table. The rotamer-pair-energy table is allocated as a single block of memory divided into several sub-blocks. Each sub-block stores the rotamer-pair energies for a set of rotamer pairs of the same amino acid pair type (*e.g.* all of the rotamer-pair energies for those serine rotamers on residue 1 and those glutamic acid rotamers on residue 2). Energy retrieval requires first looking up the offset into the energy table for the starting index for a pair of amino acids. If two amino acids do not neighbor, then the index stored in this offset array is -1. If an energy retrieval operation for a particular rotamer pair encounters an offset of -1, then the table is implicitly representing the interaction energy for that rotamer pair as 0.

The sub-blocks are laid out in row-major order by node index. Consider rotamers i on residue 1 and j on residue 2 of amino acid types aa_i and aa_j with n_{aa_i} and n_{aa_j} rotamers of that amino acid type on each residue where i is the k^{th} rotamer of amino acid type aa_i and j is the m^{th} rotamer of amino acid type aa_{r_2}. The location in the sub-block for amino acids aa_i and aa_j that stores the rotamer-pair energy between i and j is simply $n_{aa_j} * (k-1) + m$. Again, I index by 1 and not by 0 to be consistent with the rest of Rosetta.

In a large protein design problem in Rosetta, where each residue has 2000 rotamers,

each AminoAcidNeighborSparseMatrix might allocate a 16 MB table, though in practice most tables are 1 or 2 MB in size. Collectively, the AminoAcidNeighborSparseMatrix tables occupy the same amount of memory as energy2b occupied before them. Individually, these tables are dramatically smaller than energy2b. Because the individual tables are so much smaller it is possible to do things with them that could not be done with energy2b before them: in particular, to make a copy of them.

After energy calculation has completed, some of the amino acid pairs that were *predicted* to neighbor can turn out to not actually have any rotamer pairs that interact with non-zero energy. Therefore some sub-tables in an AminoAcidNeighborSparseMatrix object contain nothing but zeros. Because each individual AminoAcidNeighborSparseMatrix is small, it is possible to allocate a second energy table that does not allocate any space for the sub-tables that contain only zeros, and to then copy the non-zero values out of the first energy table into the second. After copying completes, the first table may be deallocated. This strategy could have worked for energy2b if energy2b were not both monolithic and so large.

By reallocating smaller energy tables in AminoAcidNeighborSparseMatrix classes following energy calculation, the interaction graph uses ∼12% less memory to represent rotamer-pair energies in protein design applications than energy2b used before it.

Speed

The interaction graph achieves the same performance inside simulated annealing as when reading from the energy2b sparse matrix. Part of the interaction graph's speed is due to good engineering of the graph, part of it is due to an inefficiency of the old code.

The project_deltaE_for_substitution() operation is the most time-critical operation. Simulated annealing performs this operation millions of times during the course of a single trajectory. I sketch the code below:

```
float
PDInteractionGraph::project_deltaE_for_substitution(
   int nodeIndex,
   int alternateState
)
{
   return getPDNode( nodeIndex )-> ...
      consider_substitution( alternateState );
}
```

```
float
PDNode::consider_substitution( int alternateState )
{
   altState_ = alternateState;
   altEnergyTotal_ = one_body_energies_[ altState_ ];
   for (int ii = 0; ii < getNumEdges(); ++ii)
   {
      altEnergyTotal_ += getPDEdge( ii )-> ...
         getEnergyForSubstitutedState( getIndex(), altState_ );
   }

   return altEnergyTotal_ - currEnergyTotal_;
}
```

The `PDNode` traverses of all of its incident edges, asking for the energy stored on that edge when the node is in the alternate state. For an edge to retrieve the appropriate energy, it needs to know 1) which of its two vertices is considering the state substitution (`getIndex()`), and 2) the alternate state (`altState_`). With this simple traversal algorithm, one can compare the new running time against that of the old running time.

The previous version of simulated annealing was slightly less efficient than it could have been. To compute ΔE, simulated annealing would retrieve the sum of the rotamer-pair energies for the current rotamer from `energy2b,` and would then retrieve the sum of the rotamer-pair energies for the alternate rotamer. Since the previous layout of `energy2b` incurred an average cost of $n/2 + 1$ cache misses per rotamer energy lookup, where n is the average number of neighbors for a residue, a single rotamer substitution incurred an average of $n + 2$ cache misses. If simulated annealing had included bookkeeping to store the sum of the rotamer-pair energies for the current rotamer, then each rotamer substitution would have incurred half as many cache misses. (In fact, I wrote and tested the bookkeeping code that worked alongside `energy2b` and observed a 1.4x speedup. I couldn't quite cut the running time in half since the bookkeeping itself incurred a cost.)

The interaction graph stores each rotamer-pair energy needed for a single rotamer substitution in a separate table. This means that the cost to compute the sum of the rotamer-pair energies for the alternate rotamer will be at least n cache misses, with n being the average number of neighbors, as opposed to the `energy2b` setup which cost $n/2 + 1$ cache misses. Indeed, if a vertex were to iterate across each of its edges, and at each edge retrieve 1) the offset into the `AminoAcidNeighborSparseMatrix` energy table and then 2) retrieve the energy stored in the energy table, it would incur $2n$ cache misses per lookup. Performing one lookup for the current rotamer and then one lookup for the alternate rotamer would cost $4n$ cache misses. (I'm assuming here that because

the vertex and edge objects themselves are so few and so small that they will all residue in cache, but that the objects the vertex and edge objects point to, which are much larger, will not be in cache.)

The interaction graph has an estimated 4 times as many cache misses per ΔE calculation, as described. To bring this down, I do two things: 1) I perform the requisite bookkeeping so that the energy for the current rotamer need not be looked up and 2) I move much of the edge data onto the vertices to avoid the cache misses incurred by going to several different edge objects for the same vertex. Each of these optimizations reduced the number of cache misses by half.

The bookkeeping to for a `PDNode` to keep track of the two body energies for its incident `PDEdges` is easy. When a neighbor u of node v changes its state (the annealer commits a substitution it was considering for node u), then u informs v of their new interaction energy through the `PDEdge` that connects them. To communicate exactly which of the pair energies v must update, the `PDEdge` informs v which edge it is; that information is stored in the `posInNodesEdgeVect_` variable described previously. When the annealer is later ready to consider a state substitution at vertex v, v already knows the interaction energies for its currently assigned rotamer. This bookkeeping cuts the number of cache misses in half.

It's also easy to move data onto vertices in a cache-friendly order. Since a single rotamer substitution requires looking at all the edges incident upon a single node, cache efficiency is improved by moving the edge data to the vertices. In particular, the two tables associated with the `AminoAcidNeighborSparseMatrix` are moved onto the vertices; for the offset table, the contents of the tables are copied to the vertices; for the energy table, a pointer to the table is stored on the vertices. To copy the offset tables, I allocate a large, contiguous table to store all of the `AminoAcidNeighborSparseMatrix` offsets from the edges incident on that vertex right next to each other. The offset table for a single sparse matrix is a two-dimensional table of integers; the table I allocate on each vertex is a three-dimensional table – an array of two dimensional tables. The `PDNode` can then traverse all of these offsets during the rotamer substitution stage and incur only a single cache miss. The movement of data from edges to vertices cuts the number of cache misses by half.

The result of these two optimization techniques is that simulated annealing with the `PDInteractionGraph` proceeds as rapidly as simulated annealing with the `energy2b` sparse matrix.

5.4 A Model for Non-Pairwise Decomposable Energetics

The interface between the annealer and the interaction graph allows the rapid integration of new energy functions. If a new energy function is represented as an interaction graph, inherits from the set of base classes, and answers the two methods for state substitution, then it will seamlessly fit into the existing simulated annealing code. In particular, I have been interested in the challenge of incorporating non-pairwise decomposable energy functions. The interaction graph serves as a worthy framework for the incorporation of such functions. In the remainder of this chapter, I describe a graphical model for non-pairwise decomposable energy functions and a set of classes that extend the interaction graph base classes.

A non-pairwise decomposable energy function F may be partially decomposable, even if not all the way down to pairs. Suppose that a set of functions $\{f_1, f_2, \ldots\}$ can be defined such that each f_i depends on a subset of the atoms in the protein and $F = \sum f_i$. Such a decomposition makes it easier to update the energy for the entire protein after a small part of it changes; the only functions f_i that need to be reevaluated following a change are those which include the changing part of the protein in their domains. In the context of the side-chain-placement problem, the domain of each function f_i is the Cartesian product of the state spaces for a subset of the molten residues.

In Chapter 4, I introduced a model for non-pairwise decomposable energy functions that decomposed the function into a set of hyperedges in a graph. Each hyperedge represented a single function f_i. For dynamic programming, the representation of each scoring function was as a high-dimensional table, with the dimension of the table equal to the degree of the hyperedge. However, this storage scheme is ill suited for the non-pairwise decomposable energy function presented in Chapter 8. The degree of the hyperedges this energy function defines are so high the memory requirement for tabulating the interaction energies would be prohibitive. Fortunately, these non-pairwise decomposable functions need not be tabulated to be optimized with simulated annealing; they may be computed on the fly.

Indeed, in the non-pairwise decomposable energy function presented in Chapter 8, each scoring function f_i can be identified without knowing which molten residues are in its domain. Each hyperedge can then represent the molten residues in its domain as a list; after the hyperedges are allocated, then it examines each molten residue to

determine if it belongs in its domain. If it does belong in the domain, then it adds the molten residue to the list of molten residues in its domain. With this representation, each hyperedge is effectively a vertex in a graph and its list of the molten residues in its domain is effectively an edge list.

Equivalent to the definition of an interaction graph with non-pairwise decomposable energy functions represented by hyperedges, consider an interaction graph $G = \{V, E\}$ that contains two sets of vertices, $V = V_1 \cup V_2$: each vertex in V_1 represents a molten residue, and each vertex in V_2 represents one of the non-pairwise decomposable functions, f_i. Edges are divided into two classes, $E = E_1 \cup E_2$, those that represent the pairwise decomposable interactions between the molten residues $E_1 \subseteq V_1 \times V_1$, and those that represent the relationship between the non-pairwise decomposable scoring functions $f_i \in V_2$ and the molten residues of whose states they are a function, $E_2 \subseteq V_1 \times V_2$. In this graph, there are no edges that connect pairs of vertices from V_2.

This model fits well in the context of simulated annealing. When the annealer considers a state substitution at a single vertex $v \in V_1$, the vertex v must compute the change in energy for the entire system, ΔE, that its state change would produce. To compute ΔE, vertex v iterates across the edges to its neighbors in V_1 to collect the change in energy for the pairwise decomposable portion of the energy function, ΔE_{pd}, and then iterates across its neighbors in V_2, informing them of its potential change of state, and collecting their change in energy ΔE_{npd}.

This model is especially apt if the small functions f_i can be easily identified. For instance, if the function were defined on a per-residue basis, than each of the background residues in the protein would be represented by a vertex in V_2. The domain for each of these vertices could be later determined by a proximity test, for instance, and edges added to the graph. The amount of buried hydrophobic surface area, for instance, can be expressed on a per residue basis: each residue is responsible for reporting how much hydrophobic surface area of its hydrophobic atoms have been buried from solvent.

5.5 Current Class Hierarchy

The next subsection describes the details of the class hierarchy as it exists in Rosetta today. The interaction graph class hierarchy contains six groups of classes providing three sets of concrete classes for use in protein design (Figure 5.3). Two of these concrete classes I describe in greater detail in Chapters 7 and 8. I described the concrete `PDInteractionGraph` classes in Section 5.3.4 above.

Between the `InteractionGraphBase` and the `PDInteractionGraph` classes lie the `PrecomputedPairEnergiesInteractionGraph` classes. These classes define an interface between the rotamer-pair-energy evaluation code and the graphs that store the energies. The non-pairwise decomposable energy function I describe in Chapter 8 still includes the pairwise decomposable portion of Rosetta's energy function. The graph I use to represent the non-pairwise decomposable function stores the pairwise decomposable energies in edges in the same way as the `PDInteractionGraph` class. Both graphs inherit from the `PrecomputedPairEnergiesInteractionGraph` classes.

The `AdditionalBackgroundNodesInteractionGraph` classes provide general functionality for the incorporation of non-pairwise decomposable energy functions into the packer. This group of classes add two new classes to the parallel inheritance hierarchy: the `SecondClassNode` and the `SecondClassEdge` classes. The `SecondClassNode` class defines the set of additional scoring functions that contribute to a non-pairwise decomposable energy function. They are *second class* in that they do not carry state spaces and remain invisible to the annealer. The `SecondClassEdge` class defines relationships between the second class nodes (the non-pairwise decomposable functions) and the first class nodes (those with state spaces) that affect those functions.

The `FirstClassNode` adds to the `NodeBase` class an edge list data member (and an edge vector data member) to maintain the second class edges that connect first class nodes to second class nodes. Like the `NodeBase` and `EdgeBase` classes, the `FirstClassNode`, `SecondClassNode`, and `SecondClassEdge` classes maintain all of their data privately and provide read access to derived classes through protected, inlined methods. The classes used to incorporate the non-pairwise decomposable energy function described in Chapter 8 derive from the `AdditionalBackgroundNodesInteractionGraph` and its associated classes.

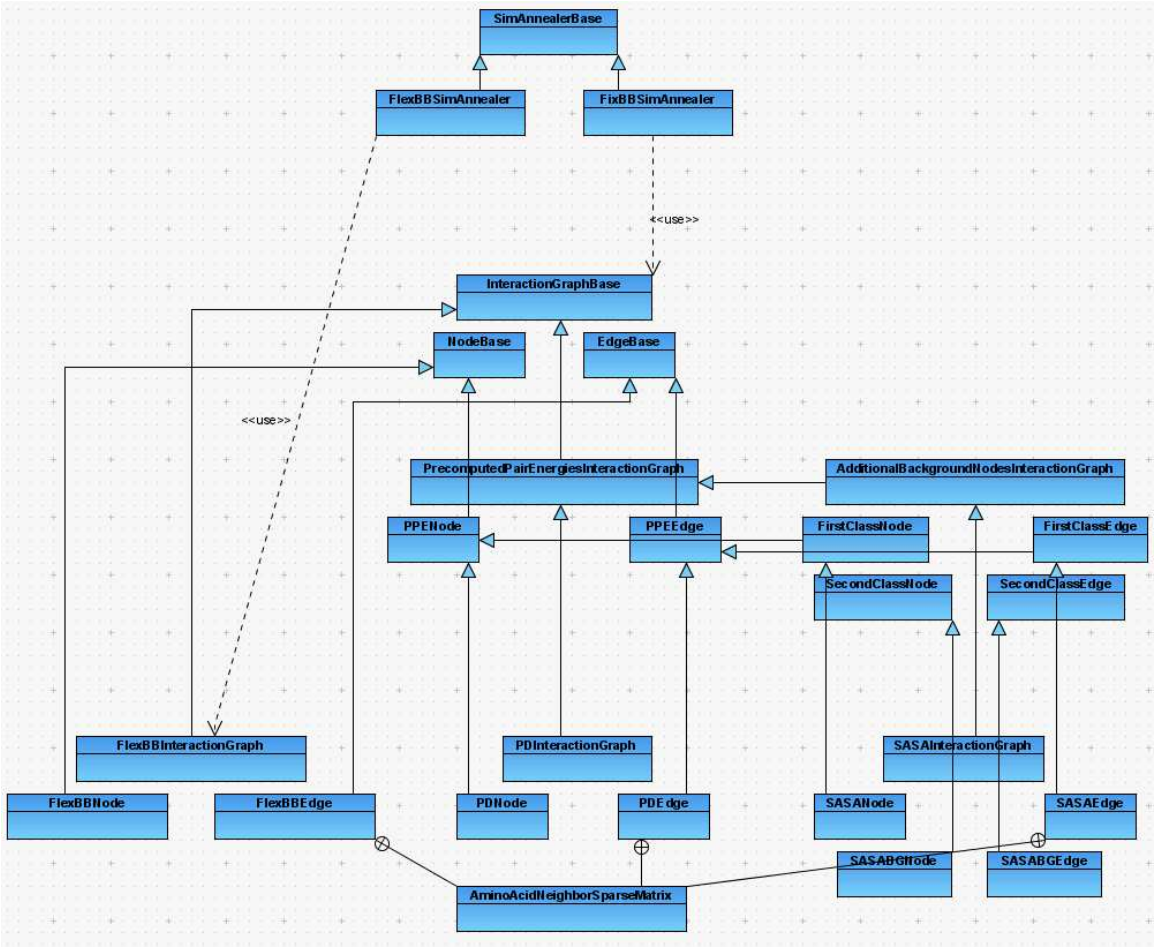


Figure 5.3: *Interaction Graph Class Diagram.* The six sets of interaction graph classes, composed of three concrete sets (the *FlexBB* graph, the *PD* graph, and the *SASA* graph) and three abstract sets. With Jenny Hu, I have refactored the simulated annealing code into the `SimAnnealerBase` class using the strategy design pattern to facilitate later incorporation of variants on simulated annealing. The `FixBBSimAnnealer` interfaces the interaction graph as described in this chapter; I describe the interface between the `FlexBBInteractionGraph` and the `FlexBBSimAnnealer` in Chapter 7. All three concrete edge classes use the `AminoAcidNeighborSparseMatrix` class to store the pairwise decomposable portions of the energy function; memory is always an issue in protein design. Refactoring the efficient sparse matrix representation into its own class makes it easy for new interaction graphs to take advantage of the amino-acid-neighbor-memory-saving technique. Image generated with VirtualParadigm.

Chapter 6

Fast Rotamer-Pair-Energy Calculation

This chapter[1] describes a technique for rapid rotamer-pair-energy computation between pairs of residues. The rotamer-pair-energy-computation phase dominates the running time in a typical call to the packer; I have sped the computations in this phase by a factor of $\sim 3.5\times$.

6.1 Introduction

Designers divide the side-chain-placement problem into two phases. In phase 1, they precompute all possible rotamer-pair interaction energies for their rotamer library, and in phase 2, they search for the (globally) optimal side-chain placement. Significant work has gone into exact algorithms for the side-chain-placement problem (Desmet et al., 1992; Goldstein, 1994; Looger and Hellinga, 2001; Gordon and Mayo, 1999; Leaver-Fay et al., 2005a). Still, the problem is NP-Complete (Pierce and Winfree, 2002), and many researchers choose fast stochastic optimization techniques (Bradley et al., 2003; Dahiyat and Mayo, 1997b; Holm and Sander, 1992; Saven and Wolynes, 1997; Desjarlais and Handle, 1995). The rotamer-pair-energy computation of phase 1 can be a significant fraction of the running time for both techniques, and usually dominates the running time of stochastic techniques.

The interaction energy between two rotamers, A and B, is the sum of the atom/atom interaction energies over all atoms of A with all atoms of B. When a pair of rotamers

[1]Portions of this chapter were previously published in the Workshop on Algorithms in Bioinformatics (WABI) 2005, held in Mallorca, Spain. This work represents a collaboration with Brian Kuhlman and Jack Snoeyink.

on the same residue share torsional angles, they share atoms. Repeated atoms imply repeated atom/atom energy evaluations when computing all rotamer-pair energies.

The obvious way to avoid repeating atom/atom energy computations is to store in a table the result of atom/atom energy computations for unique atom pairs. When a unique atom pair is encountered for the first time, calculate the pair's interaction energy and store it. When a unique atom pair is encountered any subsequent time, simply look up the old result. However, with a moderately large rotamer library of 2K rotamers, which we use as our running example, a single residue can generate ∼10K unique atoms. A unique-atom by unique-atom table with 10K x 10K entries would occupy 400 MB. This table does not fit in a processor's cache (∼512 KB). Although storing energies avoids repeated computation, retrieving the table entries incurs cache misses, eroding any savings in running time.

We use a trie to represent all the rotamers on a single residue. With a pair of these "rotamer tries" we can rapidly compute the rotamer-pair energies, while reducing our memory usage. We have implemented our algorithm within the Rosetta molecular modeling software (Bradley et al., 2003). Because our memory use is minimal, and because we reuse atom/atom energy computations, our algorithm runs nearly 4 times faster than Rosetta's existing method.

6.2 Rotamers and Tries

As mentioned in Chapter 2, rotamer libraries are usually built by sampling certain torsional (or dihedral) angles; these are denoted by $\chi_1, \chi_2, \ldots$ in order from the protein backbone. The most flexible amino acids, lysine and arginine, have four χ dihedrals. Rotamers of the same amino acid on the same residue that share a prefix of χ dihedrals place many of their atoms in the same position. For instance, two leucine rotamers that share a χ_1 dihedral place their C_β, $1H_\beta$, $2H_\beta$ and C_γ atoms identically. If we order atoms by distance from the backbone as well, then the shared atoms are also a prefix. Trie data structures are perfect for capturing shared prefixes.

A *trie* is a rooted tree with two special properties: 1) each node in the trie represents an object and 2) each root-to-leaf path in the trie represents a string of objects. Tries are commonly used to represent dictionaries for spell checkers in word processors. In a dictionary trie, each node represents a letter and each root-to-leaf path represents a word. For example, consider a dictionary trie containing just two words: 'apple' and 'apply.' The root would be the letter 'a'. The shared prefix 'appl' would lie along an

unbranched path of the trie. The 'l' node would have two children, 'e' and 'y'. The path to the leaf node 'y' from the root spells out the word 'apply'.

In a rotamer trie, each node represents an atom. Each root-to-leaf path represents a rotamer. We depict a few rotamer tries in Figure 6.1. Note that the trie connectivity does not reflect the amino acids' chemical structure. Note also that the three threonine rotamers depicted differ only in their hydrogen position: their substantial shared prefix lets us save space. The trie for arginine in Figure 6.1(b&c) shows the trie branching produced by its four χ dihedrals. A trie for a complete rotamer set would be too large to display.

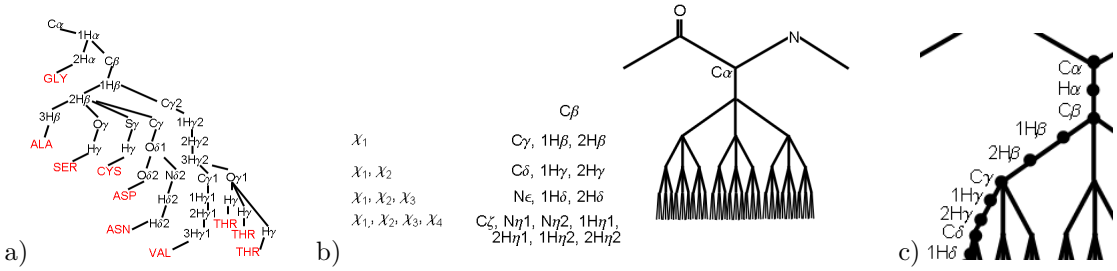


Figure 6.1: *Two Example Tries.* a) One rotamer from each of the small amino acids and three for threonine. b) A set of arginine rotamers showing the branching pattern for arginine's four χ dihedrals. c) Angle χ_1 determines the coordinates of $1H_\beta$, $2H_\beta$ and C_γ, so they lie together along a path.

Tries have proven useful in other string problems in computational biology (Weiner, 1973; McCreight, 1976; Ukkonen, 1995). Homme Hellinga previously introduced representing rotamer sets in a trie-like structure to weed out rotamers colliding with the background (Hellinga and Richards, 1991), but does not use tries to compute energies.

We compute rotamer-pair energies with a pair of rotamer tries. The implementation of this algorithm is compatible with the energy function from Rosetta.

6.3 Rosetta's Energy Function

Rosetta has four terms that apply on an atom-by-atom basis. Between heavy atom pairs, Rosetta includes three terms: a van der Waal's attractive term, a van der Waal's repulsive term, and a Lazaridis-Karplus implicit solvation (Lazaridis and Karplus, 1999) term. Each of these terms depend on the types of the two heavy atoms, and their

distance. For speed, Rosetta uses a maximum distance threshold of 5.5 Å: if two heavy atoms are further than 5.5 Å apart, then their interaction energy is zero.

Between hydrogen/other atom pairs two terms apply: a van der Waal's repulsive term, and a statistically derived hydrogen-bonding term (Kortemme et al., 2003). The hydrogen-bonding term is usually described by four atoms acting simultaneously: the donor hydrogen, the donor heavy atom, the acceptor and the acceptor-base (Figure 2.12). The term depends on one distance and the cosine of two angles: the hydrogen-acceptor distance, the cosine of the donor heavy atom–hydrogen–acceptor angle and the cosine of the hydrogen–acceptor–acceptor-base angle. We reformulate the hydrogen bond function to depend on only two atoms, the hydrogen and the acceptor, by including orientation vectors with each atom. The orientation vectors allow us to compute the two cosines needed.

Because Rosetta's terms involving hydrogens are so short-ranged, Rosetta's developers use a distance threshold between two heavy atoms to determine if their attached hydrogen atoms could be close enough to interact. If two heavy atoms are further than 4.6 Å apart, then all hydrogen/other atom pairs for the attached hydrogens have zero interaction energy.

Rosetta's existing rotamer-pair-energy function, `get_energies()`, takes two rotamers sets, R and S, and outputs their rotamer-pair energies into a rotamer/rotamer energy table (rot_rot_E). We describe `get_energies()` with the following pseudocode:

```
get_energies(R, S)
  for i = 1 : R.num_rotamers
    for j = 1 : S.num_rotamers
      if cbeta_dis( i, j) > threshold(amino_acid(i), amino_acid(j) )
        continue;
      energy_sum = 0
      for k = 1 : num_heavy_atoms(i)
        for l = 1 : num_heavy_atoms(j)
          energy_sum += atom_atom_energy( R.atom(i,k), S.atom(j,l) );
          if dis( R.atom(i,k), S.atom(j,l) ) < 4.6
            energy_sum += ...
               calc_attached_h_energies( R.atom(i,k), S.atom(j,l) );
      rot_rot_E[ i, j ] = energy_sum;
  return;
```

where `calc_attached_h_energies(k, l)` iterates over the hydrogen/heavy atom pairs and hydrogen/hydrogen atom pairs calling `atom_atom_energy()` for the heavy atoms k and l and their attached hydrogen atoms. Because hydrogens make up roughly half of the atoms in a rotamer, it would be roughly 4 times more expensive to evaluate all atom/atom energies as it would be to evaluate all heavy atom/heavy atom energies, descending into the hydrogens only as needed.

Rosetta does not compute rotamer-pair energies between rotamers if their C_β atoms are so distant that it is impossible for any pair of rotamers of those two amino acid types to interact. Rosetta uses its 5.5 Å heavy atom distance cutoff to calculate these thresholds.

6.4 Trie Node

The trie data structure stores everything we need for evaluating Rosetta's energy function. It also stores a number of variables needed to prune energy computations, which we describe after the algorithm.

We represent our trie as an array. We store the nodes in their preorder traversal order. In the recursive description of the algorithm below (Section 6.5.1), we refer to child pointers as if they were explicit. However, we store the depth of each node in the tree instead of explicit child pointers. The preorder/depth representation is sufficient to completely describe the trie structure.

We store the atom type for each atom, its xyz coordinate, and its orientation vector. In a redundant, but time-saving extension of the atom type, we keep several boolean flags: `is_backbone`, `is_heavy_atom`, `is_acceptor`, *is_donor_h*, etc. Each of these flags is stored as a single bit. The logic is somewhat complex to convert between atom type and these boolean values. Instead of evaluating the conversion functions during the trie traversal $O(n^2)$ times, we evaluate these flags outside of the main loop and store them compactly in the trie node.

The implementation that includes pruning uses 40 bytes per node. The last three variables in the `trie_node` are needed only for pruning. The "no pruning" implementation does not allocate space for these three variables, and so the cost per `trie_node` drops to 32 bytes. We have found it especially important to make sure our trie nodes align with the 32-bit memory boundaries.

```
struct trie_node
   float[3]       xyz;                 //12 bytes
   float[3]       o_vector;            //12 bytes
   unsigned char  atom_type;           // 4 bytes
   unsigned char  depth;
   unsigned char  hv_depth;
   unsigned char  flags;
   unsigned short flags2;              // 4 bytes
   unsigned short hybridization;
   unsigned short rotamers_in_subtree; // 4 bytes
   unsigned short sibling;
   float          subtree_radius;      // 4 bytes
```

6.5 Interaction energy between two rotamer tries

We now give the algorithm to calculate the rotamer-pair energies between two tries, R and S. The idea is simple, we perform a preorder traversal of R, and for each atom $r \in R$, we perform a preorder traversal of S. We evaluate `atom_atom_energy(r, s)` for each pair of atoms we encounter ($s \in S$). (We refer to the preorder traversal order when we use the words *before*, *precede* and *after* below.)

To calculate the rotamer-pair energies, we define two recursive functions: `atom_vs_trie()` and `trie_vs_trie()`. For clarity we describe these functions recursively; for speed we implement them iteratively.

- `atom_vs_trie(r, s, ancestral_E)` recursively computes the interaction energy between atom r and all the rotamers in the subtree of S rooted at node s. It stores these energies in a global variable, `AREnergies`, a stack of arrays. `atom_vs_trie()` calls `atom_atom_energy()` and is called by `trie_vs_trie(r, S)`.

- `trie_vs_trie(r, S)` recursively computes the interaction energy between the rotamers in the subtree of R rooted at node r and the rotamers in the trie S. It stores these energies in a global variable, `RREnergies`, the table of rotamer/rotamer energies. An invocation of `trie_vs_trie(R.root, S)` calculates all rotamer-pair energies. `trie_vs_trie()` invokes `atom_vs_trie()`.

6.5.1 Functions in detail

Global variables

We use three global variables in these recursive functions:

- `RREnergies`. Rotamer/Rotamer Energies. This table has (R.`num_rotamers` × S.`num_rotamers`) entries, one for each rotamer pair.

- `AREnergies`. Atom/Rotamer Energies. This is a stack of arrays. Each array contains S.`num_rotamers` entries and holds r's ancestors' interaction energies with rotamers of S. The stack height is limited to the maximum number of ancestors with siblings of any leaf in a rotamer tree.

- `ARStackTop`. Top of stack pointer for `AREnergies`.

atom_vs_trie(r, s, ancestral_E) in detail

Precondition: r is an atom of R, s is an atom of S. `ancestral_E` holds the sum of the interaction energies r has with all ancestors of s. `AREnergies[ARStackTop]` contains the sum of the interaction energies of all of r's ancestors with the rotamers of S that terminate at or after s, and contains the sum of r's ancestors' and r's interaction energies for all rotamers of S that terminate before s.

Postcondition: `AREnergies[ARStackTop]` contains the sum of interaction energies of r and its ancestors with the rotamers of S that terminate before s or terminate in s's subtree. `AREnergies[ARStackTop]` contains the sum of the interaction energies of all other rotamers in S with r's ancestors only.

Pseudocode:

```
atom_vs_trie(r, s, ancestral_E)
   ancestral_E += atom_atom_energy(r, s);
   if (s.terminal_rotamer_id != -1)
      AREnergies[ARStackTop][s.terminal_rotamer_id] += ancestral_E;
   for (int i = 0; i < s.num_children; i++)
      atom_vs_trie(r, s.child[i], ancestral_E);
   return;
```

trie_vs_trie(r, S) in detail

Precondition: r is an atom of R. `AREnergies[ ARStackTop ]` contains the sum of the interaction energies of r's ancestors with the rotamers of S. If r is the root, then ARStackTop must be zero and each entry in `AREnergies[ 0 ]` is zero.

Postcondition: `RREnergies` contains the interaction energies for all rotamers of S and the rotamers of R that terminate in the subtree of R rooted at r.

Pseudocode:

```
trie_vs_trie(r, S)
  atom_vs_trie(r, S.root, 0);
  if (r.terminal_rotamer_id != -1)
    //copy entire AREnergies row
    RREnergies[r.terminal_rotamer_id] = AREnergies[ARStackTop];
  if (r.num_children > 0)
    ARStackTop++;
    for (int i = 0; i < r.num_children-1, i++)
      //copy stack top for children with siblings
      AREnergies[ARStackTop] = AREnergies[ARStackTop - 1];
      trie_vs_trie(r.child[i], S);
    ARStackTop--;
    //last child doesn't need its own stack copy
    trie_vs_trie(r.child[r.num_children - 1], S);
  return;
```

Because we traverse S repeatedly, it is critical that S fit inside the processor's cache. The size of each node in the trie is 40 bytes. In our example rotamer set with 10K unique atoms, S would occupy 400KB. Since most cache sizes are 512KB, S fits comfortably. `AREnergies`'s size would be 4 rows $\times$ 2K rotamers/row $\times$ 4 bytes/float = 32KB. There are only 4 rows in `AREnergies` since the most flexible amino acids have only 4 χ dihedrals.

6.5.2 Pruning Computations

We can use the tree structure of the two tries R and S to avoid performing many of the atom/atom energy computations. Suppose we are somewhere in the middle of the trie traversals, examining atoms $r \in R$ and $s \in S$. Beneath r is a subtree containing some

number of atoms, beneath s is another subtree containing some number of atoms. If r is a heavy atom, then there are some number of hydrogen atoms bound to r in the subtree beneath r. We have three conceptual entities: atoms, heavy atoms (including their associated hydrogen atoms), and subtrees. We may prune calculations for any combination of entities.

We decide how to prune based on r and s's distance. The following sections describe the additional data structures maintained. Briefly, here is a sketch of the pruning options.

1. atom/atom: If the distance between r and s exceeds a threshold, we can assign their interaction energy to zero. After this prune, we still must continue calculating interactions between the atoms in r's and s's subtrees. Rosetta already includes this prune within its `atom_atom_energy()` function. Because we use this function as well, we get this prune for free.

2. atom/subtree: If the distance between r and s is so great that r must be too far to interact with any atom in s's subtree, then we can make an atom/subtree prune.

3. subtree/subtree: If the distance between r and s is so great that all atoms in r's subtree must be too far to interact with any atom in s's subtree, then we can make a subtree/subtree prune. Rosetta makes a similar prune using C_β atoms (see Section 6.3 above).

4. heavy atom/heavy atom: If the distance between heavy atoms r and s exceeds 4.6 Å, we can skip calculating interactions among their bound hydrogens. Rosetta already employs this prune, so we must too, as we want to improve upon the running time. After this prune, we still must continue calculating interactions between the atoms in r's and s's subtrees.

5. heavy atom/subtree: As in the atom/subtree pruning, we may see that a heavy atom r and all of its attached hydrogens are too far to interact with all the atoms in s's subtree and then perform this skip.

Heavy Atom/Heavy Atom Pruning

As we described above, if a pair of heavy atoms are further apart than $\lambda = 4.6$ Å, then all the interactions between the hydrogen/heavy atom pairs and the hydrogen/hydrogen

pairs is zero. We prune computations based on this cutoff by 1) restricting the atom ordering within the trie and 2) including another global variable in our algorithm, `skipH`.

We order the atoms in our trie so that the closest ancestral heavy atom for a hydrogen is the heavy atom to which it is chemically bound. For instance, the ordering of atoms for alanine would be: C O N H CA HA CB 1HB 2HB 3HB. (We include backbone atoms as part of the trie. This leaves room for later incorporating backbone flexibility as part of a protein redesign task.) We store each atom's *heavy atom depth* (`hv_depth`). The heavy atom depth for a heavy atom is its position in a list of an amino acid's heavy atoms. For example, alanine's CA heavy atom depth is 4. The heavy atom depth for a hydrogen atom is the depth of its parent heavy atom.

Our new global variable, `skipH`, is a stack of booleans, represented as a table. It has `MAX_HEAVY` rows (the largest number of heavy atoms for a single amino acid, which in tryptophan is 14). Each row has S.`num_heavyatoms` entries. This table is a stack in that its contents describes properties for ancestor atoms of our currently focused r atom. In essence, the top-of-stack pointer is stored within each atom of R by its heavy atom depth.

Now we'll describe how we use `skipH`. If a heavy atom r at heavy atom depth d is at least λ away from heavy atom s of S, then we set `skipH[`d`][`s`]` to 'true.' Later, if we want to know if the parent heavy atom for a hydrogen atom of R at heavy atom depth, d, and the heavy atom s of S are greater than λ apart, then `skipH[`d`][`s`]` tells us. To capture this formally, we revise the `atom_vs_trie(r, s, ancestral_E)` pre- and postconditions.

Additional Precondition: For all heavy atom ancestors r' of r, `skipH[`r'`.hv_depth][`s'`]` holds "true" iff s' is further than λ from r' for all heavy atoms $s' \in S$. Additionally if r is a heavy atom, then for all s'' that precede s, `skipH[`r`.hv_depth][`s''`]` holds "true" iff s'' is further than λ from r.

Additional Postcondition: For all heavy atom ancestors r' of r, (including r if r is a heavy atom), `skipH[`r'`.hv_depth][`s'`]` holds "true" iff s' is further than λ from r' for all heavy atoms $s' \in S$.

`skipH` scales in size with the number of heavy atoms in S, unlike some of our other global variables that scale with the number of rotamers in S. It is a rather large data structure. In our example rotamer trie with 10K atoms, roughly 5K would be heavy atoms. In this case, `skipH` would occupy 70KB.

Subtree/Subtree Pruning

In a subtree/subtree prune, we avoid calculating atom/atom energies for all pairs of atoms in the subtrees of r and s. We prune based on a sphere overlap test. The *interaction sphere* of heavy atom r is the sphere centered at r that has a radius of one half of the threshold distance for heavy-atom/heavy-atom interaction; in our case, one half of 5.5 Å. For two heavy atoms to interact, their interaction spheres must overlap. The *subtree-interaction sphere* of heavy atom r is centered at r, and is large enough that, for any atom s to interact with an atom in r's subtree, s's interaction sphere must overlap with r's subtree-interaction sphere. Equivalently, the radius of the subtree-interaction sphere is the greatest distance between r and all heavy atoms in r's subtree + (5.5 Å/2).

When two subtree-interaction spheres do not overlap, we may make a subtree/subtree prune. The non-overlapping condition is met when the squared distance between r and s exceeds the square of the sum of r and s's subtree-interaction-sphere radii. This comparison is very fast and we can afford to make it at each heavy atom pair we encounter.

When we decide to skip computations involving the subtrees rooted at r and s, we immediately add `ancestral_E` to `AREnergies` for all rotamers of S that terminate in the subtree rooted at s. We then skip to the first node of S that is not in s's subtree. To make this jump, we must know the child of s's closest ancestor (which may be s's sibling if s has one).

We also skip over s's subtree for all of r's descendants in later calls to `atom_vs_trie()`. We maintain another global array, `trim_depth`, to record which subtrees should be skipped. This array has one entry for each heavy atom of S. The values stored in each entry are small (< 14) so we can get away with using a single byte per entry. In our example rotramer trie with 5K heavy atoms, `trim_depth` occupies only 5KB.

We describe the functionality of this variable with another pre- and postcondition pair for `atom_vs_trie(r, s, ancestral_E)`.

Additional Precondition: If r is a heavy atom (hydrogen), then `trim_depth[`s`]` is less than (less than or equal to) r.`hv_depth` if 1) the subtree-interaction sphere of the heavy atom ancestor of r at depth `trim_depth[`s`]` – call this ancestor, r' – does not overlap with s's subtree-interaction sphere, and 2) s is the only atom among it and its ancestors whose subtree-interaction sphere does not overlap with r''s subtree-

interaction sphere. The value of `trim_depth[`s`]` is undefined for those atoms of S for which condition 1, but not condition 2, holds. If s's interaction sphere overlaps with the subtree-interaction spheres for all ancestors of r, then `trim_depth[`s`]` is greater than or equal to r.`hv_depth` when r is a heavy atom, or strictly greater than r.`hv_depth` when r is a hydrogen atom.

Additional Postcondition: If r is a heavy atom (hydrogen) and `trim_depth[`s`]` was less than (less than or equal to) r.`hv_depth`, then `trim_depth[`s`]` remains the same, and the values in `trim_depth` for atoms in the subtree rooted at s are undefined. If r and s are heavy atoms, and r and s's subtree-interaction spheres do not overlap, then `trim_depth[`s`]` is r.`hv_depth`. Otherwise, `trim_depth[`s`]` is `MAX_HEAVY` + 1.

We also make subtree/subtree prunes when we encounter colliding atoms. Collisions reflect physically impossible situations, and an exact representation of a collision's energy is unnecessary. We prune when an atom/atom energy exceeds 20 kcal/mol.

Heavy Atom/Subtree Pruning

If r's interaction sphere and s's subtree-interaction sphere do not overlap, we may skip past s's subtree. In order to repeat this subtree-skip for the hydrogen atoms attached to r, we maintain another global variable, `skipSubtree`. `skipSubtree`, like `skipH`, is a stack of boolean arrays represented as a table. Each array has S.`num_heavyatoms` entries. There are `MAX_HEAVY` rows. We do not describe here the additional pre- and postconditions as they are so similar to those for `skipH`. `skipSubtree` would occupy 70KB in our example 10K atom rotamer trie.

6.6 Results

We wrote our algorithm in `C++` and verified that it generates the same energies as Rosetta's existing rotamer-pair-energy function, `get_energies()`, by compairing the two sets of output energies for an entire protein design task. We compared the running time of our algorithm using six pruning options against `get_energies()` in 57 complete protein redesign tasks (Figure 6.2). All six variants included heavy atom/heavy atom pruning. We measured running times on Intel Xeon 2.8 GHz processors each with 2.5 GB RAM.

The trie-vs-trie algorithm, without any pruning mechanisms, produces a 1.6× speed-up. Adding the pruning mechanisms speeds the trie-vs-trie algorithm to run 3.87×

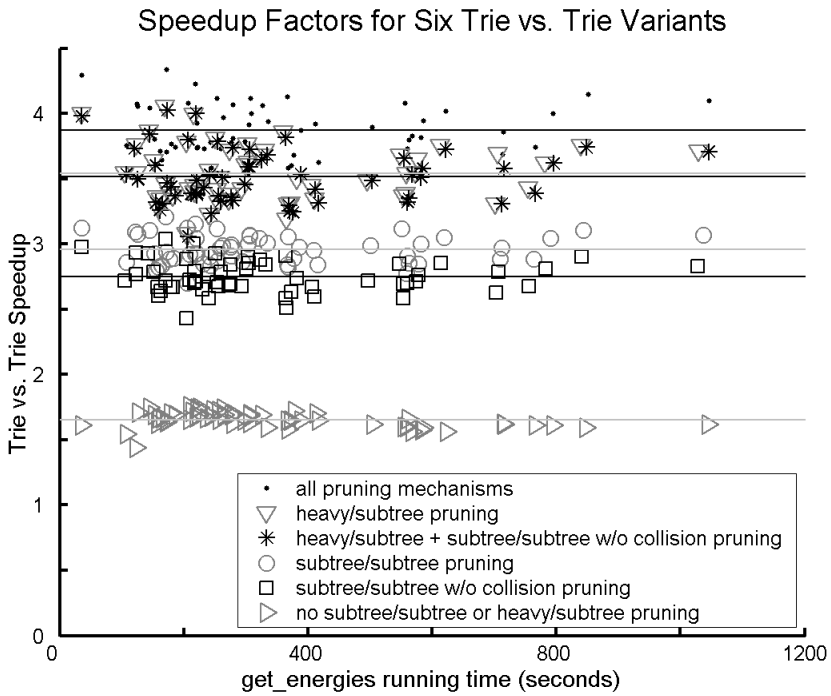


Figure 6.2: *Comparing* `trie_vs_trie()` *and* `get_energies()`. The relative running times for rotamer-pair-energy calculation for 57 entire-protein-redesign simulations. Mean speedup factors for the six pruning combinations were 1.65, 2,75, 2.96, 3.52, 3.54, and 3.87.

faster than `get_energies()`.

The size of the trie depends on the number of shared atoms in the input rotamer set; the number of shared atoms in turn depends on the χ dihedrals in the rotamer library. In Dunbrack's backbone dependent rotamer library (Dunbrack and Cohen, 1997), which Rosetta uses, χ_1 depends on χ_2. That is, for an amino acid with two χ dihedrals, if Dunbrack reports two rotamers with χ_1 is near 60 ° for both and χ_2 is either near 180° or near -60°, then Dunbrack reports these rotamers to have different χ_1 angles – one might be 62.3°, the other might be 64.2°. The Cγ atoms are not shared by rotamers created with these dihedral angles, and the trie that I construct branches at Cβ. The question that needs to be asked is: If Rosetta were to alter its rotamer library so that more dihedral angles were shared between rotamers, would it save more time in rotamer-pair-energy computation?

To answer this question, I modified the input file to Rosetta that contained Dunbrack's rotamer library by rounding each dihedral angle in the library to the nearest

10°. When I constructed rotamer tries for a rotamer set built with this altered rotamer library, they contained 12% fewer atoms. Altering the rotamer library had the desired effect of increasing the number of shared atoms. With 10% fewer atoms in the trie, I expected a 19% reduction in running time. However, there was no performance improvement for the trie-vs-trie algorithm using this altered rotamer library. My guess is that increasing the number of shared atoms increases the size of the subtree interaction spheres for the shared atoms. If several rotamers all had different Cγ atoms when constructed from Dunbrack's library, then the subtree bounding sphere for each Cγ would have been smaller than in the new trie where these rotamers all shared a single Cγ atom. Computations that would have been pruned separately at the several Cγs in the first trie do not get pruned at the joined Cγs of the second trie, but rather at the separate Cδs. If this guess is correct, then the degree of atom sharing between rotamers and the pruning mechanisms are antagonistic: if decreasing the number of atoms in the trie does not improve performance, it is possible that increasing the number of atoms would.

I have also implemented the trie-vs-trie algorithm for a special case in which the trie S contains exactly one rotamer. I have incorporated this algorithm into a subroutine of Rosetta that is used in high-resolution protein structure prediction, `rotamer_trials()`. In this subroutine, Rosetta iterates across a subset of the residues in the protein in a random order so that it focuses on a single residue and finds that residue's best rotamer given that all other residues are held fixed. This subroutine evaluates the rotamer-pair energies for a residue and its neighbors in the same way that `get_energies()` does. Once the sum of the energies is known for each rotamer, picking the best rotamer is trivial. The time-consuming stage in this simply the energy computation.

In my implementation of `rotamer_trials()`, I construct a trie for the set of rotamers at the focused residue and one trie for each other residue in the protein where each of these tries contain exactly one rotamer – the constructed tries are unbranching trees, paths. In this special case, trie-vs-path, I use only the heavy atom/subtree pruning mechanism after observing that both atom/subtree and subtree/subtree pruning mechanisms avoid the same computations and work at the same speed. The heavy atom/subtree prune is easier to implement and requires slightly less memory.

The trie-vs-path algorithm speeds `rotamer_trials()` by a factor of 2.4×. I ran two protein-structure-prediction trajectories through Rosetta with the same starting structure, one trajectory with the original `rotamer_trials()` algorithm and the other with the implemention using the trie. In the first 1225 calls to the `rotamer_trials()`

over the course of a few minutes worth of simulated annealing, the original version of `rotamer_trials()` averaged 0.2460 s per call. The first 1225 calls to the trie-vs-path version of `rotamer_ trials()` averaged 0.1006 s per call.

Both the trie-vs-trie algorithm and the trie-vs-path algorithm have been in Rosetta for over a year. The trie is used by default in both design mode and fullatom-relax mode. Since the original incorporation of the trie, I have expanded its functionality to support the termini residues, where the N-terminus contains extra hydrogen atoms attached to the amino group and the C-terminus contains an extra oxygen atom. I have also expanded its functionality to support local backbone flexibility, which I describe in Chapter 7.

Chapter 7

Local Backbone Flexibility in Design

This chapter describes the construction of software for side-chain placement that simultaneously optimizes both side-chain and backbone conformations. The formulation of the side-chain-placement problem given in Chapter 2 restricts the backbone of the input protein structure. Designers currently overcome the shortcomings of fixed-backbone side-chain-placement by iteratively moving the backbone and solving instances of fixed-backbone side-chain-placement (Harbury et al., 1998; Kuhlman et al., 2003). The goal of the work presented in this chapter is to expand the capacity of Rosetta's existing fixed-backbone side-chain-placement software to simultaneously optimize side-chain and backbone structure. Together, Jenny Hu and I are the first to have achieved this goal.

The purpose of incorporating backbone flexibility into the optimization of protein sequence is to avoid solving an exponential number of fixed-backbone side chain placement problems. For example, and as mentioned in Chapter 3, if the designer were to select eight segments of the backbone for local-backbone flexibility, and at each segment, consider ten different backbone conformations, then the designer would have to solve a billion (8^{10}) instances of fixed-backbone side chain placement in order to examine all possible combinations of backbone conformations. The goal of the research presented in this chapter is to optimize side-chain placement with local-backbone flexibility using a rapid technique similar to the one that optimizes side-chain placement on fixed backbones, thereby avoiding the expense of enumerating all possible backbone conformations and solving instances of fixed-backbone side chain placement on each.

The work in this chapter builds upon Rosetta's existing, experimentally validated

(Kuhlman et al., 2002; Dantas et al., 2003), software for side chain placement. There are four main challenges in expanding the functionality of this system so that it can handle locally-flexible backbones:

1. the rotamers must be created differently,

2. the rotamer-pair energies must be calculated differently,

3. the rotamer-pair energies must be stored differently, and

4. the rotamer assignment must be optimized differently.

Jenny Hu solved the first of these four challenges, I solved both the second and third, and Jenny Hu and I together solved the fourth.

In fixed-backbone design, the backbone can be viewed as part of the background – only side chains change their conformations. For this reason, Rosetta included the side-chain/backbone interactions in the rotamer one-body energies and did not include the backbone/backbone interactions at all. In locally-flexible-backbone design, since the backbone is moving, the backbone's interactions must be included as part of the two-body energies. Whereas rotamer pair energies in fixed backbone design represented the side chain/side chain interactions, the rotamer-pair energies in flexible backbone design must represent the sum of side-chain/side-chain interactions, the side-chain/backbone interactions and the backbone/backbone interactions.

This chapter describes an extension of the trie-vs.-trie algorithm (presented in Chapter 6) for the calculation of those energies needed for design with local-backbone flexibility.

This chapter presents a data structure, an interaction graph, used to store the rotamer-pair energies; the `FlexBBInteractionGraph` is unlike `PDInteractionGraph` presented in Chapter 5 in that the graph need not represent the full complement of rotamer pair energies for a pair of interacting residues due to "backbone continuity" restrictions that prevent many rotamer pairs from being simultaneously assigned.

This chapter describes an interface between the annealer and the flexible backbone interaction graph that differs from the interface between the annealer and a fixed-backbone graph. The backbone continuity restrictions prevent the previous one-rotamer-substitution-at-a-time simulated annealing algorithm from working – several rotamers have to be altered simultaneously to move the backbone. In collaboration with Jenny Hu, I have developed an alternate simulated annealing protocol that is

capable of enforcing the continuity restrictions. I describe the interface between this protocol and the flexible backbone interaction graph, and the algorithm (a depth-first traversal) that the graph uses to compute the energy involved in substituting several residues simultaneously.

Finally, I provide a small proof-of-concept demonstration that the designs produced by altering both side-chain and backbone conformations using local-backbone flexibility have better energies than the designs produced by keeping the backbone fixed.

7.1 Strategy for Local Backbone Flexibility

The strategy for allowing backbone flexibility during the optimization of side-chain placement is to sample backbone conformations (a dozen conformations) for short segments of the backbone (three, four, or five residues) at several regions in the protein and to construct a set of rotamers that hang off each of the conformations. Each residue will therefore have a choice of both which backbone conformation and which rotamer on that backbone conformation to adopt. However, each residue will not be able to make its choice of backbone conformation independently of the other residues in the same segment of the backbone with it. That is, if residues i through $i + 4$ each have five possible backbone conformations, A through E, then if residue i chooses backbone A, residue $i + 1$ must also choose backbone A. If residue $i + 1$ were to chose any other backbone, then it would break the chemical bonds along the backbone – or at least strain the bond lengths and angles in an energetically impossible way. I call the restriction that backbone conformational assignment to residues along a short segment of the protein must be the same the *backbone continuity* constraint.

The backbone continuity constraint is born out of two decisions: first, that segments of the backbone would move *in concert*, and not independently, and second, that residues that move in concert still behave independently with respect to rotamer assignment, once the backbone assignment has been made.

7.1.1 Concerted Fragment Motion

Residues must move in concert as a mathematical necessity: there are not enough degrees of freedom for a single residue to flex independently of its neighbors. Consider the segment of the backbone that lies between the C atom on residue i and the N atom on residue $i + n + 1$ for some integer number of residues, n. This segment of

the backbone, regardless of its length, has 6 degrees of freedom – I describe these degrees of freedom in the paragraphs below. If these residues are restricted to ideal bond geometry, then each residue may flex at its ϕ and ψ dihedrals only. That is each residue contributes two degrees of freedom to the segment. Suppose we are interested in moving a single residue independently of its neighbors: that is, $n = 1$. There are six degrees of freedom between the end of residue i and the beginning of residue $i + 2$, however residue $i + 1$ has only two degrees of freedom. The system of equations for moving residue $i + 1$ independently of its neighbors is over-constrained.

With three residues ($n = 3$), the backbone segment has six degrees of freedom (six ϕ and ψ angles), matching the number of degrees of freedom in the gap between residues i and $i + 4$. Therefore the system of equations to solve for the ϕ and ψ angles for the segment has a finite number of solutions: in fact, it has at most 16 solutions (Raghavan and Roth, 1989). If residue i is held fixed as is residue $i + 4$, then the 16 possible solutions for the dihedral angles of residues $i+1$, $i+2$ and $i+3$ may be computed. If a seventh degree of freedom can be given to the system, then motion can be simulated – the seventh degree of freedom can be sampled, and the values for the other six degrees of freedom can be computed. Small changes to the seventh degree of freedom simulate backbone motion (Figure 7.1).

Raghavan and Roth (1989) provide the robotics formulation of this "loop closure" problem, a robot arm has some number of degrees of freedom (torsional or otherwise), and at the end of the arm it has a hand. The robot must position its hand at a particular point and at a particular orientation. If the base of the arm is at the origin, then it takes three degrees of freedom to describe the point in space for the hand's final position; it takes another three degrees of freedom to describe the hand's orientation (which can be thought of as a point on the unit sphere, which takes two degrees of freedom to specify, and then a twist angle for the hand – the direction to point the thumb). If the robot's arm has six degrees of freedom, then given the desired hand location and orientation, a finite number of solutions for positioning the arm can be found that place the hand correctly.

Raghavan and Roth observed that if the dihedral angles are given for a robot arm with six "revolute" (dihedral) degrees of freedom (often called an R6 arm), then the kinematics computation for the location and orientation of the hand position can be made with a series of linear operations, and in particular, by multiplying a set of Denavit-Hartenburg coordinate frames (Denavit and Hartenberg, 1955). Raghavan and Roth derived a set of linear equations to describe the relationship between the

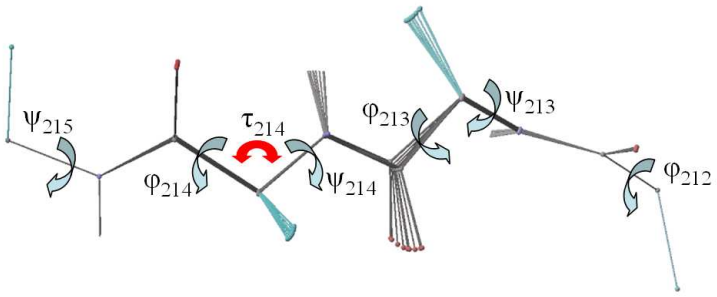


Figure 7.1: *PROBIK: Protein Backbone Motion By Inverse Kinematics.* An illustration of the kinds of "motion" that can be generated by inverse kinematics. These are residues 212 through 215 of ribose binding protein (PDB ID: 1RBP), the scaffold protein on which Homme Hellinga's lab built both a receptor protein (Looger et al., 2003) and an enzyme (Dwyer et al., 2004). By varying the N-Cα-C bond angle, τ, for residue 214 away from ideality, and solving for the other six dihedral angles pictured, inverse kinematics creates a series of conformations. Backbone carbonyl oxygen atoms are in red, Cα-Cβ bonds are in turquoise.

dihedral angles and the position of the robot's hand. They proved that computing the valid dihedral angles given the location of the hand was equivalent to computing the roots of a degree-16 polynomial. There are at most sixteen roots to this polynomial, and thus there are a finite number of solutions for the dihedral angle solutions for the revolute joints. In practice, most of the roots for this polynomial are complex so that the typical number of realizable angular assignments to the revolute angles is 2, 4 or 6 (there are always an even number of realizable solutions since the complex solutions exist as conjugate pairs). Manocha and Canny expanded upon this solution by proving that the task of finding the roots of this degree-16 polynomial, which is numerically unstable, could be performed by finding the eigenvalues of a matrix, which is numerically stable (Manocha and Canny, 1994).

With Manocha's implementation of the the inverse kinematics solution to computing the six revolute angles for an R6 robot arm, Noonan, O'Brien and Snoeyink (Noonan et al., 2004) proposed a method for flexing backbones by sampling a seventh degree of freedom and using inverse kinematics to solve for the other six degrees of freedom. A series of solutions produced by sampling the seventh degree of freedom appears to computer scientists and biochemist alike as smooth backbone motion (Figure 7.1).

7.1.2 Residue Independence

The second design decision, the decision to treat rotamer assignment to residues independently while enforcing backbone continuity was made after another design option proved infeasible. I describe the independent-residue scheme first.

Suppose the locally-flexible backbone segments for a protein each span k residues, and each have b backbone conformations, and suppose that each residue has s states per backbone conformation (or bs states total). In the independent residue scheme, each vertex in the graph represents a single residue. A state assigned to a single vertex represents both the backbone conformation and the side-chain conformation. Vertices that correspond to residues that are part of the same locally-flexible-backbone segment are constrained in their state assignments: they may only take on states such that these states belong to the same backbone conformation.

Each edge connecting a residue pair from different backbone segments would store a pair energy table with b^2s^2 entries, and each edge connecting a residue pair from the same segment would store a pair energy table with bs^2 entries, since intra-segment pair energies are only meaningful for rotamer pairs that originate from the same backbone. The independent-residue scheme would thus store at most $k^2b^2s^2 + k(k-1)bs^2$ pair energies for the segment pair; the scheme would likely store fewer than this number of energies since not all residue pairs for these segments necessarily interact.

This scheme wherein each residue is optimized separately creates opportunities to break the backbone. Suppose residues i and $i+1$ move in concert and are assigned states such that they are both in backbone conformation A – in this state assignment, the backbone is not broken. Then if the annealer tried to perform a single state substitution at residue i that moved it from backbone conformation A to backbone conformation B, the annealer would break the backbone. The way to resolve this break would be for residue $i+1$ to undergo a simultaneous state substitution so that its new state also originates from backbone B. This means that simulated annealing has to change, since the system is no longer one in which single states are substituted at a time.

The alternative to the independent-residue scheme would have been to treat the entire moving backbone segment as a single *super residue* where this super residue would be assigned one of many *super rotamers*. If a four-residue segment had b alternate backbone conformations and s rotamers on each residue in each backbone conformation, then the super residue composed of these four would have bs^4 super rotamers to choose from – each super rotamer specifies an assignment of a backbone conformation to the

segment, and an assignment of rotamers to each of the residues in the segment.

The advantage of the super-rotamer formulation is that simulated annealing would not need to change. Simulated annealing, instead of making a single rotamer substitution at a time, would make a single super-rotamer substitution at a time. Since each super rotamer is built from a single backbone conformation, there is no way to inadvertently break the backbone during a super-rotamer substitution.

The first obvious disadvantage of the super-rotamer scheme is that if the interaction graph to store super-rotamer-pair energies were implemented in the same way the fixed-backbone interaction graph were implemented, then it would require too much memory. In fixed backbone design, an edge connecting a pair of interacting residues with s states each stores s^2 rotamer-pair energies (unless they are quite distant, when the `AminoAcidNeighborSparseMatrix` stores fewer energies). Consider two super residues where each represented a segment of k residues with b backbone conformations and s rotamers per conformation; each super residue has bs^k rotamers. If the edge between these two super-residues were to store a single table, as edges do in the fixed-backbone design interaction graph, then it would store $b^2 s^{2k}$ energies. This memory requirement is dramatically larger than the number of rotamer-pair energies represented in the independent-residue scheme.

The super rotamer scheme should instead represent its pair energies the same way that the independent-residue scheme does. The super residue could still behave as if it were reading from gigantic tables of super-rotamer-pair energies. It would present the same interface to the annealer as before, tell the annealer how many states (super rotamers) each vertex (super residue) had and the annealer would continue to make single state (super-rotamer) substitutions at a time. Behind the scenes, the graph would, for each super-rotamer substitution, retrieve many more energies than if the graph read from a set of tables holding super-rotamer-pair energies.

The true problem with the super-rotamer scheme is that if the segment length becomes long, or the number of rotamers per residue becomes very large, then the number of super-rotamers becomes unmanageably large. If a single segment contained five residues, and allowed four different backbone conformations, and if each residue had 64 rotamers per backbone conformation, then the super-residue would have 4 billion possible super-rotamers. It would not be possible to represent which of the super-rotamers to substitute on a single super residue with a 32-bit integer. Moreover, simulated annealing would take an extraordinarily large amount of time. Currently simulated annealing on a fixed protein backbone considers tens of millions of state

substitutions, considering each rotamer for substitution several hundred times. In order for simulated annealing to visit each super-rotamer once in the example above, it would have to run 1000 times longer than fixed-backbone simulated annealing. The super-rotamer idea has severe practical limitations.

7.2 Pair-Energy Calculation in Flexible-Backbone Design

In fixed-backbone design, the interactions between the side-chain atoms of molten residues and the backbone atoms of other molten residues could be stored as part of the rotamer one-body energies. Because the backbone of another molten residue did not move, the interactions of one molten residue's side chain with that other residue's backbone was independent of the rotamer assigned to the other. The `get_energies` subroutine thus computed the side-chain/backbone interactions in one loop, and the side-chain/side-chain interactions in a separate pair of nested loops.

Because the backbone can move in flexible-backbone design, it can no longer be described as part of the background. The two-body energies for a residue pair must include not only side-chain/side-chain interactions, but also the side-chain/backbone and the backbone/backbone interactions. As in fixed-backbone design, in locally-flexible-backbone design, many side-chain/backbone interaction calculations would be repeated if not handled in a distinct loop from the side-chain/side-chain interaction calculations. One can imagine writing loops to handle the side-chain/backbone interactions separately from the side-chain/side-chain interactions, adding the result of each side-chain/backbone interaction calculation to several rows in the rotamer-pair-energy table. However, the rotamer trie setup provides an efficient alternative.

The rotamer trie includes the side-chain/backbone energies, and the backbone/backbone energies in the rotamer-pair-energy calculations, and does so efficiently: no atom-pair computations involving backbone atoms are unnecessarily repeated. The rotamer trie is ideal for calculating rotamer-pair energies in flexible-backbone design.

For flexible backbone design, I construct one trie for each of the backbone conformations that a residue may adopt. Each `RotamerTrie` object stores all of the tries that correspond to a single residue. For a pair of residues with multiple backbone conformations that come from different backbone segments, I iterate across each combination of backbone pairs and for each combination, compute the rotamer-pair energies for the

rotamers in the corresponding tries. For a pair of residues with multiple backbone conformations that come from the same backbone segment, I iterate across each backbone, restricting myself to examining the same backbone for both residues, and compute the rotamer-pair energies for the rotamers in the appropriate tries (Figure 7.2).

The alternative to creating different rotamer tries for each of the alternate conformations is to create one single trie for the entire set of rotamers. This would, however, have likely produced tries so large that the cache efficiency of the trie-vs-trie algorithm would have been compromised. Since the structure of the main portion of the trie-vs-trie algorithm is of a nested pair of loops, where the outer loop iterates across all of the atoms in trie R and the inner loop iterates across all atoms in trie S, the algorithm will run fastest when the entirety of S can fit in the processor's cache. The algorithm I use for cache-friendly rotamer-pair-energy calculation is not unlike the cache-friendly matrix multiplication algorithms. The other advantage of creating separate tries for each backbone conformation is that it makes it easy to compute for residues on the same segment only those rotamer-pair energies that belong to the same backbone conformation (Figure 7.2b).

7.3 Flexible Backbone Interaction Graph

The interaction graph I have implemented to store the rotamer-pair energies for flexible-backbone design derives from the base classes described in Chapter 5. The `FlexBB-InteractionGraph` derives from the `InteractionGraphBase` class. It works in conjunction with the `FlexBBNode` class, which is derived from the `NodeBase` class, and the `FlexBBEdge` class, which is derived from the `EdgeBase` class. The factory methods that the `FlexBBInteractionGraph` defines for the creation of new vertices are simply:

```
NodeBase *
FlexBBInteractionGraph::create_node( int nodeIndex, int numStates )
{
   return new FlexBBNode( this, nodeIndex, numStates );
}
EdgeBase *
FlexBBInteractionGraph::create_edge( int node1, int node2 )
{
   return new FlexBBEdge( this, node1, node2 );
}
```

What distinguishes the `FlexBBInteractionGraph` from the `PDInteractionGraph` described in Chapter 5 is in the way edges store rotamer-pair energies. The `PDEdge`

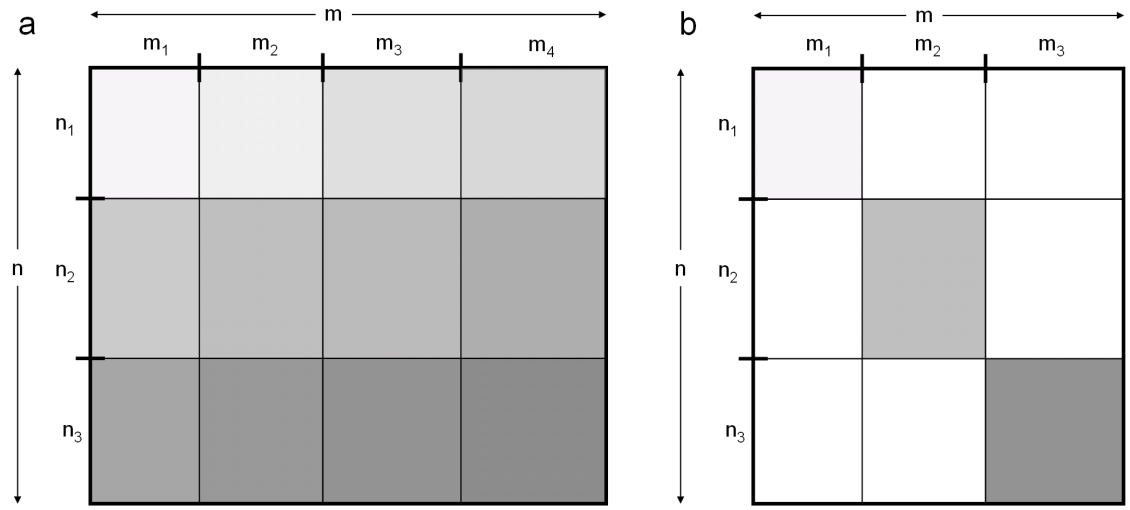


Figure 7.2: *FlexBB Trie-vs-Trie Calculation Order.* Two residues on different segments of the backbone (a) – *i.e.* that do not move in concert – where the first residue has m rotamers, on 4 different backbones, and the second residue has n rotamers on 3 different backbones, the trie-vs-trie algorithm computes the rotamer-pair energies between the first group of m_1 and n_1 rotamers. The order for the rest of the rotamer-pair-energy calculation is shown with increasing gray scale. Two residues on the same segment of the backbone – *i.e.* that move in concert – the trie vs trie algorithm computes rotamer-pair energies for only those rotamer pairs that are on the same backbone conformation.

class kept a single `AminoAcidNeighborSparseMatrix` object to store the interaction energies for all rotamers on one residue with all the rotamers on the other. In contrast, the `FlexBB` edge does not need to store the full complement of energies for certain residue pairs; those rotamer pairs that originate on different backbone conformations from the same flexible backbone segment do not have meaningful interaction energies and thus the `FlexBBEdge` should not allocate space to store their interaction energies.

The `FlexBBEdge` stores rotamer-pair energies by grouping rotamers by the backbone conformation they originated from, and then storing energies for pairs of rotamer groups in separate `AminoAcidNeighborSparseMatrix` objects. Consider the two residues i and j that are both on flexible backbone segments and have b_i and b_j different backbone conformations. If i and j are from different segments (b_i might not equal b_j), then the `FlexBBEdge` $e = \{i, j\}$ will store a $b_i \times b_j$ table of `AminoAcidNeighborSparseMatrix` pointers. The `AminoAcidNeighborSparseMatrix` pointed to by the cell in row r and column c stores the rotamer-pair energies for rotamers on backbone r from residue i and on backbone c from residue j. If i and j are from the same segment ($b_i = b_j$), then the

`FlexBBEdge` $e = \{i, j\}$ will store a $b_i \times b_i$ table of `AminoAcidNeighborSparseMatrix` pointers, but only allocate space for the diagonal elements in the table. The table pointed to by the entry in row r and column r stores the rotamer-pair energies for rotamers on backbone r from residue i and backbone r from residue j. In order to decide whether or not to allocate `AminoAcidNeighborSparseMatrix` objects for its off-diagonal entries, the `FlexBBEdge` object needs to know whether or not the residues it's incident upon are part of the same flexible backbone segment.

Allocating several sparse matrices for a single edge complicates the strategy of off-loading edge data onto vertices for cache efficiency during simulated annealing. Though this strategy could work in principle, I have not attempted it, so the `FlexBB-InteractionGraph` incurs twice as many cache misses per rotamer substitution as does the `PDInteractionGraph`. I feel less guilt for this since the `FlexBBInteractionGraph` is a new addition to Rosetta and not a replacement of an existing data structure. Thus no users could complain that the graph increased running time.

7.4 Simulated Annealing with Backbone Flexibility

As we have seen, the `FlexBBInteractionGraph` cannot perform single rotamer substitutions to move a single residue from one backbone conformation to another backbone conformation without breaking the backbone. Its options are to either restrict the alternate rotamer to belonging to the same backbone conformation as the current rotamer at a particular residue, or to assign multiple rotamers at once so that all of the residues that belong to the same backbone segment end up in the same backbone conformation.

The restriction and the multiple-assignment options define the two basic rotamer substitution operations that make up the interface between the `FlexBBSimAnnealer` and the `FlexBBInteraction- Graph`. The first type of state substitution is one in which the backbone does not move – the current state and the alternate state are rotamers originating from the same backbone conformation. The second type of state substitution is one in which the backbone does move, and all residues that belong to the same flexible-backbone segment will change their state.

7.4.1 Types of State Substitutions

While there are several possible ways in which the annealer could restrict its alternate rotamer choice, the way we have implemented is to have the graph provide a list of all

of those rotamers to the annealer, and then have the annealer pick one from among this list. By forcing the graph to tell the annealer which states are available, the annealer is freed from having to understand the relationships between various rotamers. This simplifies the annealer. This does not complicate the graph, which already has to understand whether various rotamers originate from the same backbone in order to represent the appropriate set of rotamer-pair energies.

The simultaneous assignment of multiple rotamers to several residues could also be handled in several ways. The way we have chosen is to have the annealer choose one new rotamer for a single residue which will bring that residue to a new backbone conformation and to have the graph choose the rotamers for all other residues that belong to the same flexible backbone segment. The graph communicates which rotamers it has chosen for the other residues whose states it has altered to the annealer through an input parameter to the multiple-assignment method call. This input parameter is an array representing the state assignment to each vertex in the graph.

The graph has to choose an alternate rotamer for each of those residues involved in the multiple assignment except the one residue the annealer specifies the alternate state for. The graph selects alternate rotamers with the goal of increasing the odds that the substitution will be accepted: it selects those rotamers that are more likely to fit. The graph selects the alternate rotamer for a residue as the rotamer on the alternate backbone b which has the same amino acid type as the current rotamer, and that minimizes the sum of the atom distances for all side-chain atoms among the set of rotamers on backbone b. That is, in the current rotamer assignment, the rotamer r on backbone b' occupies some region of space. If the currently assigned rotamer is decent, then the rotamer on the alternate backbone b that is most likely to fit into the region that r vacates is the one has its atoms in close to the same space. It might be possible to generate metrics for volumetric proximity that do a better job matching rotamers from different backbones, however, we are content for now with the acceptance rates for backbone motion based on the atom-distance metric.

With a scheme to select alternate rotamers for those residues undergoing a backbone substitution, we are able to define a slight variation on the multiple- assignment-rotamer substitution. This variation makes pure backbone move: instead of specifying an alternate rotamer for one of the residues in a flexible segment, the annealer specifies which of the alternate backbone conformations it would like to try for the segment and then the graph selects an alternate rotamer for each residue. Since the amino acid sequence does not change at any position, and because the new rotamer assignment will be close

to the old rotamer assignment, this rotamer substitution type can be viewed as simply moving the backbone.

7.4.2 Rotamer Substitution Algorithm

The new interface between the simulated annealer consists of four methods that the interaction graph must support and that the annealer invokes:

- `float considerSubstitution(int nodeIndex, int altState );`
- `float considerBackboneMove( int segmentIndex, int altBackbone, std::vector< int > & graphStateAssignment);`
- `float considerSubstitutionWithBackboneMove( int nodeIndex, int altState, std::vector< int > & graphStateAssignment );`
- `void commitConsideredSubstitution();`

The algorithm for the `considerSubstitution` method is the same as for the `PD-InteractionGraph`; the vertex traverses each of its edges, at each edge retrieving the pair energy for the alternate rotamer, and then sums these pair energies. The difference in the sum of the energy for the alternate rotamer and the energy for the current energy (which has been cached) is the ΔE for the substitution.

The algorithm for the `considerBackboneMove` method is more complicated. Since the state assigned to several nodes changes, the graph must retrieve the energies stored in all edges incident upon any of those nodes that change. It does this by starting at one of the vertices in backbone segment that is undergoing a backbone move, and then makes a depth-first traversal of the *intra-segment edges*. The intra-segment edges are those that are incident upon nodes that are part of the same flexible backbone segment, and are marked in the graph with a boolean flag. Each vertex visited in the depth-first traversal makes its own edge traversal, summing the pair energies that its alternate state has with the other vertices in the graph. In order to prevent double-counting of those interaction energies stored in the intra-fragment edges, each vertex counts only the interaction energies stored on intra-fragment edges that connect the vertex to vertices with a higher index.

The algorithm for the `considerSubstitutionWithBackboneMove` method is the same as for the `considerBackboneMove` method, except that the vertex from which

the depth-first traversal begins can be assigned any alternate state, and not just the alternate state that is closest for its backbone to the current state.

The advantage of implementing the data structure for rotamer-pair energies as a graph could not be any clearer than in this case: the rather difficult task of computing the ΔE for a backbone move can be implemented simply with a well known and easy graph operation, a depth-first traversal.

7.5 Proof of Concept

To demonstrate that local backbone flexibility does improve design, I present a small comparison between the designs produced by Rosetta for a fixed backbone and a locally flexible backbone for one of the enzyme design targets currently under investigation by the DARPA Protein Design Processes initiative. In this enzyme design, the ligand is fixed, as are three side chains that are helping stabilize the transition state of the ligand. The geometry of this arrangement comes from quantum mechanics calculations; from my perspective, this geometry is untouchable. The design problem I face is to pack side chains around the ligand conformation.

This particular backbone scaffold is a TIM barrel (Section 3.1). The core of the protein is composed of eight parallel β-strands that wrap around on each other to form a cylindrical β-sheet. α-helices connect adjacent strands. TIM barrel proteins commonly host enzyme active sites, so a TIM barrel is a natural place to try to design an enzyme.

In the design comparison, I have used the PROBIK inverse kinematics software (Noonan et al., 2004) to flex seven regions of the backbone scaffold away from the original crystal structure. I highlight the alternate backbone conformations I allow for each of these regions in red in Figure 7.3.

I ran 200 simulated annealing trajectories for both fixed-backbone and flexible-backbone designs. I used the same starting structures for both fixed and flexible backbone design, (and the same random seed so that the stochastic hydrogen placement step that occurs at Rosetta's launch would produce the same placements). I used a small rotamer library (Dunbrack's original rotamers [1993], em i.e. no additional samples taken at $\pm\sigma$ for any χ dihedral). For fixed-backbone design, Rosetta created 3,666 rotamers for the entire protein; for locally-flexible-backbone design, Rosetta created 18,490 rotamers for the entire protein.

The average Rosetta energies for the two were significantly different (>97.5% confidence with Student t-test and 199 degrees of freedom). For fixed backbone design,

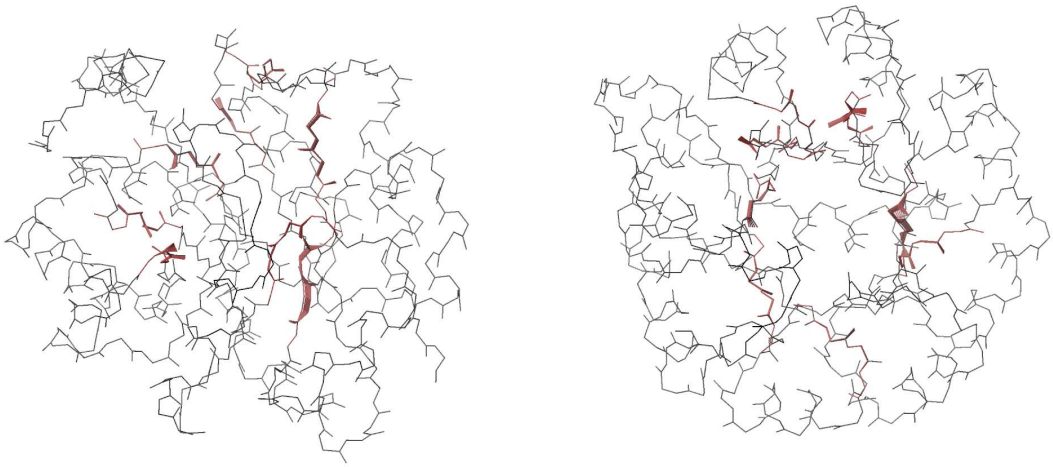


Figure 7.3: *Local Backbone Flexibility in a TIM Barrel Design.* Highlighted in red are the regions of the backbone I have flexed using inverse kinematics. The view on the left is from the side of the barrel, the view on the right is down the center of the barrel.

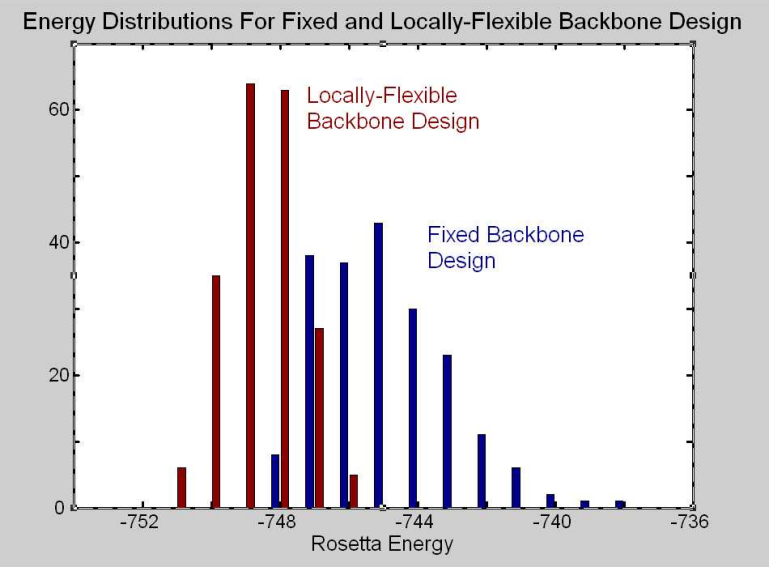


Figure 7.4: *Distribution of Energies From Fixed And Locally-Flexible Backbone Design.* A histogram of the energies produced from 200 simulated annealing trajectories by fixed backbone design (blue) and locally-flexible backbone design (red). The energies produced in locally-flexible backbone design are significantly lower.

the average energy was −744.39 kcal/mol (standard error, 1.851 kcal/mol) whereas for locally-flexible-backbone design, the average energy was −748.11 kcal/mol (standard error, 1.063 kcal/mol). Since the amount of time spent in simulated annealing differs between the two forms of design, with locally flexible backbone design taking dramatically longer (∼1 minute per trajectory as opposed to ∼2 seconds per trajectory), comparing the average energy that results does not prove that one local backbone flexibility is superior. The real question is not what the average trajectory produces but how deep are the energy wells for these designs? That is, does the additional conformational flexibility allow lower energy structures? The lowest energy structure produced with fixed-backbone design had an energy of −747.71 kcal/mol. The lowest energy structure produced through locally-flexible-backbone design had an energy of −750.54 kcal/mol. The structure with this energy moved the backbone away from the original conformation at two locations, with one of these segments moving 0.245 Å, as measured by the distance between Cα atoms. It is also satisfying to see that the average energy for a design with local backbone flexibility was better than the best energy from a design on a fixed backbone.

The difference in energies between fixed-backbone and locally-flexible-backbone design may seem slight in comparison to the total energy; the average improvement of 3.72 kcal/mol represents only 0.5% of the total energy. However, such an improvement is in fact dramatic. Rosetta's energy function, which I've reported in kcal/mol, does not correspond perfectly to the stability of a protein; 3.72 kcal/mol differences in Rosetta's energy function does not reflect a 3.72 kcal/mol increase in the stability of a designed protein. The sensitivity of Rosetta's energy function, however, is most poignantly observed in Rosetta's successes at protein structure prediction. The difference between the energy function of a structure with a 3 Å RMSD and a structure with a 1.1 Å RMSD is often on the order of 1.0 kcal/mol (Bradley et al., 2005). An improvement in the energy of a designed structure of 3.72 kcal/mol is enormous in comparison.

The ability for locally-flexible-backbone design to outperform fixed-backbone design confirms the twin hypotheses 1) that allowing greater conformational freedom to the backbone in design produces lower energy structures, and 2) that these structures can be found through a simulated annealing protocol that optimizes both side-chain and backbone conformation simultaneously. The first hypothesis is both obvious and already confirmed. This hypothesis was confirmed experimentally by Harbury in 1998 and again by Kuhlman in 2003, and was the basis for the attempt to incorporate backbone flexibility into the simultaneous optimization of sequence and structure. My

results further support the second hypothesis, that the simultaneous optimization of both side-chain and backbone conformations through simulated annealing would be fruitful. The annealing protocol for the `FlexBBInteractionGraph` finds lower energy conformations.

Chapter 8

Non-Pairwise-Decomposable Energetics in Design

This chapter[1] describes the incorporation of a non-pairwise-decomposable energy function into the side-chain-placement problem. Unlike when optimizing a pairwise-decomposable energy function wherein all pair energies can be computed before simulated annealing begins, when optimizing a non-pairwise-decomposable energy function, the simulated annealing phase must perform some non-trivial amount of work to decide the ΔE induced by a rotamer substitution. In the case of the non-pairwise-decomposable energy function I describe in this chapter, the work performed in computing ΔE is considerable. With this extra computational work, simulated annealing now dominates the running time spent optimizing rotamer placement. The algorithm I describe for computing this non-pairwise-decomposable-energy function is fast enough, though, that the total time spent in a typical call to Rosetta's packer increases by only a factor 5.

To include this new energy function into the side-chain-placement step, I have extended the interaction-graph base classes. The polymorphic interface I described in Chapter 5 allows the seamless incorporation of this new interaction graph into existing simulated annealing software.

The new energy function is based on the solvent-accessible-surface area (SASA). The algorithm for evaluating this energy function during simulated annealing maintains the solvent-accessible-surface area of the protein; its efficiency comes from its extensive reuse of protein surface data. I follow with a detailed description of its implementation through an extension of the interaction graph framework.

[1]Portions of this chapter have been accepted for publication in the Journal of Computational Chemistry. This work is in collaboration with Glenn Butterfoss, Jack Snoeyink, and Brian Kuhlman

8.1 Introduction

Accurate measures of solvation energy and quality of packing are critical for protein design (Jaramillo and Wodak, 2005; Street and Mayo, 1998). Protein solvation energies are proportional to solvent-accessible surface area, and the quality of packing in proteins can be evaluated by determining the amount of protein surface area that is not in contact with water and is not in contact with neighboring protein atoms (Eisenberg and McLachlan, 1986; Sood and Baker, 2006). Thus, calculation of solvent-accessible-surface area (SASA) is important to many analyses for protein design, and for structural biology in general.

SASA, as defined by Lee and Richards, is the area of the surface traced by the center of a probe sphere whose radius is the nominal radius of the solvent as it rolls over the van der Waals surface of the molecule (Lee and Richards, 1971). Exact evaluation of SASA is often not performed during protein design simulations because of the computational cost. Consider the case of protein design for a fixed backbone in which the goal is to choose side-chain rotamers for selected residues that will minimize an energy function. Design simulations typically calculate the energies of millions of alternate sequences and structures. For energy terms that are pairwise additive, the algorithm can precompute and tabulate the energies for all pairs of rotamers, avoiding the cost of energy computation at runtime. Unfortunately, SASA calculations are not decomposable into pairwise additive terms; the relative positions of atoms A and B influence the SASA of atom C. Therefore, when explicit SASA calculations are performed during a design simulation, they are generally the rate-limiting step.

Numerical methods for estimating SASA – as opposed to more approximate neighbor-counting methods (Guvench and Brooks, 2004; Weiser et al., 1999) or exact analytic solutions (Fraczkiewicz and Braun, 1998; Richmond, 1984) – generally discretize the space around the protein according to one of two protocols. The first, an "external grid" protocol, places a grid of points in a 3-D Cartesian space and evaluates whether probes placed at those points intersect the molecule. Using a space grid captures solvent-accessible volume, from which an isosurface computation can derive surface area. Wilson et al. used a cubic grid to calculate solvation free energy when designing mutants of α-lytic protease (Wilson et al., 1991). Bhat and Purisima used a cubic grid coupled with a marching tetrahedra algorithm to estimate solvent-excluded surface area touched by a variable radius probe, where the size and shape of the probe depended on the polarity of the contact atoms (Bhat and Purisima, 2006). Liang and Grishin

used a rectangular grid with 0.6 Å spacing to model a contact-surface term for a new energy function optimized for side-chain-rotamer recovery (Liang and Grishin, 2002). The external grid has disadvantages: the calculated SASA can vary due to rigid body reorientations; and fine grid spacing is needed to make the returned SASA sensitive to small changes in protein structure. The accuracy of the calculation is especially sensitive to grid spacing for probe radii smaller than water.

A second numerical method for estimating SASA, an atom-centered protocol, selects a nearly-evenly distributed set of dots (points) to sample the surface area of a sphere whose radius is the atom radius plus probe radius (Le Grand and Merz, 1993; Shrake and Rupley, 1973). As in the external-grid protocol described above, these dots are evaluated as to whether they are covered by the molecule. This atom-centered protocol allows extensions to add orientation-based terms to the energy function, or to perform rapid calculation of solvent-excluded surface areas (Bystroff, 2003). Le Grand and Merz showed how to make this protocol efficient by representing the coverage of dots as Boolean values and pre-tabulating interaction masks for sphere intersection. One retrieves the appropriate mask for a pair of overlapping atoms based on their radii, their orientation angles, and the distance between their centers, then ORs the masks and counts the number of exposed points. Because of its accuracy, researchers often use Le Grand and Merz as a gold standard for evaluating the actual surface area when developing more approximate methods (Zhang et al., 2004). We will use it as the basis for our extension.

Several laboratories have explored pairwise approximations of buried surface area (SA) for faster SASA computation. Without corrections, pairwise approximations overcount buried SA, since a surface patch on a residue can be buried by more than one neighbor. Steet and Mayo reduce the impact of this overcounting by scaling down the amount of buried SA using empirically determined coefficients that depend on whether the residues are classified as core or non-core (Street and Mayo, 1998). Zhang et al. improved the accuracy of this protocol by including generic neighboring side chains during initial calculation of individual side-chain-buried SAs (Zhang et al., 2004). Pokala and Handel used a similar approach in precalculating Born radii for a solvation term (Pokala and Handel, 2004). Although these methods quite accurately compute the total buried SA, even with the best approximations, local errors of 10 $Å^2$ are not uncommon for a subset of residues in the protein. Errors of this magnitude are significant for important applications of SASA calculations, such as determining which polar groups in a protein cannot hydrogen bond with water.

We use two observations to accelerate the Le Grand and Merz calculation of SASA in protein design. First, we take advantage of the fact that many design programs search through sequence space by making only one or two mutations between energy evaluations. This fact is also exploited by the recently developed FASTER algorithm, and many Monte Carlo optimization protocols (Desmet et al., 2002; Kuhlman et al., 2002; Liang and Grishin, 2004). Second, we note that if we change from Boolean operations to record coverage, which are semigroup operations, to maintaining counts, which are group operations that have inverses, then when we make a sequence substitution, we need only to update the point counts for the new residue and its neighbors. The value of accessible surface area is derived from the number of points that have a count of zero. This approach is much faster than a complete calculation of SASA, because atom overlaps are not recalculated between residues that did not change identity or conformation.

To test our algorithm we have incorporated it into the Rosetta program for protein design, which uses a Monte Carlo algorithm to search sequence space (Kuhlman et al., 2002; Kuhlman et al., 2003). We demonstrate that this approach makes it feasible to calculate a surface-area-based measure of packing, the SASApack score, during protein design (Sood and Baker, 2006). The SASApack score, which we define more precisely in the next section, was developed to identify molecular surface area on a protein that is neither in contact with water nor with other atoms in the protein; alternatively, it can be seen as a calculation of void volume that cannot be occupied by a solvent molecule. Currently, the energy terms in Rosetta do not explicitly penalize voids. Design models produced with a standard version of Rosetta have SASApack scores that are significantly less favorable than those of naturally occurring proteins. We show that by optimizing a SASApack term during sequence design we create models that have favorable SASApack scores and more closely resemble the naturally occurring proteins that were used as the design templates.

8.2 Methods

We briefly specify the protein design algorithm in which we did the implementation, our extension to the algorithm of Le Grand and Merz, and the definition of SASApack scores.

8.2.1 Protein Design Algorithm

We performed all our design simulations with the Rosetta program (Kuhlman et al., 2003). Rosetta uses a Monte Carlo (MC) algorithm with simulated annealing to search sequence space. As in most programs developed for protein design, amino acid side chains are modeled in their most preferred alternate conformations, called *rotamers*. Rosetta uses Dunbrack's backbone-dependent rotamer library with additional rotamers created by varying angles χ_1 and χ_2 by one standard deviation in each direction of their preferred values (Dunbrack and Cohen, 1997). Starting with a random sequence, the energies of single amino acid substitutions or rotamer switches are evaluated and accepted if they pass the Metropolis criterion (Metropolis et al., 1953). The standard Rosetta energy function has been described previously; the most important terms in the potential are a 12-6 Lennard-Jones potential, the Lazaridis-Karplus implicit solvation model, an explicit hydrogen-bonding term and knowledge-based torsion potentials (Kuhlman et al., 2003; Lazaridis and Karplus, 1999; Kortemme et al., 2003).

8.2.2 Extending Le Grand and Merz to Maintain SASA

The solvent-accessible surface (SAS) is the surface traced by the center of a spherical solvent molecule as it rolls across the molecular (van der Waals) surface of the solute molecule. Lee and Richards first introduced the construction of this surface by increasing the van der Waals radii of the solute by the radius of the solvent (Lee and Richards, 1971). Shrake and Rupley introduced a numerical approximation to the computation of the solvent-accessible-surface area (SASA) by taking dot samples on the surfaces of these enlarged spheres (Shrake and Rupley, 1973). Each dot contained within the enlarged sphere of any other atom is inaccessible to solvent; call such a dot *covered*. Each dot not contained within an enlarged sphere is accessible to solvent; call such a dot *exposed*. In this approximation, each exposed dot represents a patch of the solvent-accessible surface.

Le Grand and Merz gave a rapid technique for computing which dots on the surface of one sphere are contained within a nearby sphere (Le Grand and Merz, 1993). This technique uses the same discretization for each sphere so it can precompute Boolean overlap masks recording those dots on a sphere covered by an intersecting sphere with a given radius, orientation, and distance. To determine the set of dots on a sphere that are exposed, determine the intersecting neighbors and OR the overlap masks. This Boolean OR operation is a *semigroup* operation – it is associative, but does not have

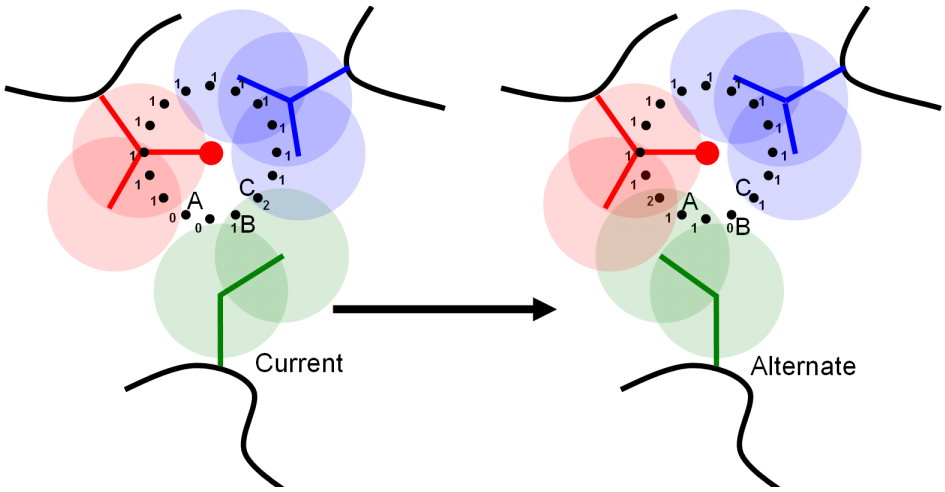


Figure 8.1: *SAS Update Algorithm.* Updating dot-coverage counts for a red atom produced by a rotamer substitution at the green residue. The substitution increases the coverage count for dot A and decreases it for B and C. This covers dot A, and exposes dot B; the count indicates that dot C remains covered by another residue.

an inverse. If one keeps a count of the number of residues whose spheres cover a dot, this is a *group* operation that does have an inverse, and one can return to a previous state by decrementing a count previously incremented. Thus, our algorithm maintains these coverage counts for each dot; a dot with a coverage count of zero is exposed, and a dot with a coverage count of one or more is covered.

Consider a rotamer-substitution step in which a *current structure* is compared against an *alternate structure* created by substituting a *current rotamer* with an *alternate rotamer* at a single residue, as depicted in Figure 8.1. The SASA calculation must determine the dot-coverage counts for both 1) the set of spheres for the alternate rotamer and 2) the set of spheres for the rest of the protein that overlap the spheres from either the current or the alternate rotamer. The dots on spheres that are not in these two sets are unaffected by the rotamer substitution. The spheres in the first set are absent from the current structure; for these spheres we perform Le Grand and Merz sphere-overlap computations. The spheres in the second set are present in the current structure; we decrement the coverage counts for the dots covered by the current rotamer, place the alternate rotamer onto the structure, perform sphere-overlap computations, and increment the coverage counts for those dots covered by the alternate rotamer.

8.2.3 SAS-Update Procrastination

A fruitful algorithmic optimization is to avoid SAS-update computations whenever possible. At low temperature ($kT < 0.5$) in our Monte Carlo (MC) optimization, we avoid SAS computations for collision-inducing rotamer substitutions. The MC optimizer decides whether or not to commit a considered rotamer substitution based on the change in energy $\Delta E = \Delta E_{PD} + \Delta E_{SASA}$. Here, ΔE_{PD} represents the change in the pairwise-decomposable portions of Rosetta's energy function, and ΔE_{SASA} represents the non-pairwise-decomposable-SASA term. If ΔE_{PD} exceeds some positive threshold, t, then we procrastinate SAS-update computations and approximate $\Delta E_{SASA} = 0$. The MC optimizer decides whether to accept the substitution on the basis of ΔE_{PD} alone. If it does, then we perform the procrastinated dot-coverage count update. At low temperatures, 75% of rotamer substitutions are rejected without having to compute ΔE_{SASA}, giving a 4-fold speedup in design simulations on complete proteins (Table 8.1).

8.2.4 Shared Atoms in Rotamers

A further algorithmic optimization is to reuse all possible sphere-overlap computations. When replacing the current rotamer with an alternate rotamer, we count the number of shared atoms between the two and compute sphere overlaps only for those atoms in the alternate rotamer that are not contained in the current rotamer. In the best scenario – replacing a current tyrosine rotamer with an alternate tyrosine rotamer that differs only at χ_3 – our algorithm computes sphere overlaps for the single terminal hydroxyl hydrogen only. A tree (or trie) data structure can be used to find the shared atoms in time proportional to their number (Leaver-Fay et al., 2005b). Shared-atom optimization produces a 1.87-fold speedup in design simulations on complete proteins (Table 8.1).

Together, the SASA-update procrastination and the shared-atom optimizations produce a 6.92× speedup. Since this is slightly less than $4 \times 1.87 = 7.47$, the two optimizations are not completely separable.

8.3 Quality of Packing With a Surface-Area Score

To test our algorithm we use a quality-of-packing score that penalizes voids around a molecule that are too small to accommodate a water molecule. The SASApack score identifies molecular surface area that is accessible to a 0.5 Å probe sphere but not to

a 1.4 Å probe representing water and compares it to the expected area observed in structures in the Protein Data Bank (PDB) (Sood and Baker, 2006). For residue i, the SASApack score is defined as:

$$\text{SASApack\_score}_i = A^i_{0.5} - A^i_{1.4} - E(A^i_{0.5} - A^i_{1.4} \mid A^i_{1.4}, aa^i)$$

where $A^i_{0.5}$ and $A^i_{1.4}$ are the molecular surface areas accessible to the two probes for residue i, and $E(A^i_{0.5} - A^i_{1.4} \mid A^i_{1.4}, aa^i)$ is the expected difference in these two surface areas given the residue's amino acid type (aa^i) and its surface area $A^i_{1.4}$ as measured from a large set of high-resolution protein structures from the PDB. An additional score, the SASAprob score, is a function of a residue's SASApack score and is defined as the fraction of residues in the PDB of the same amino acid type and $A^i_{1.4}$ that have a lower SASApack score. For the SASApack and SASAprob score we use statistics compiled by William Scheffler and Phil Bradley in David Baker's laboratory at the University of Washington (personal communication, Bradley, Scheffler and Baker). To optimize the SASAprob score during design simulations we add the following term to the Rosetta energy function for residue i:

$$-W \times \log_{10}(\max(SASAprob_i, 10^{-5})).$$

The optimal weight W was determined empirically as described in Section 8.4.2. We chose this function for its shape. We found that optimizing SASApack directly lead to swiss-cheese proteins – proteins with giant solvent-filled cavities. On its own, SASApack prefers proteins with little buried surface area and therefore little poorly packed surface areas. The log of the SASAprob score penalizes poor SASAprob scores heavily, but rewards high SASAprob scores modestly; this function avoids the swiss-cheese problem. Of course, the base of the log is irrelevant since the fitted parameter W would nullify the difference between ln and $\log_{10}$.

Two closing notes on these scores. First, observe that the molecular surface accessible to solvent (MSAS) is not the same as the SAS; the areas of these two surfaces differ. The MSAS can readily be computed from the SAS by projecting the exposed dots on the enlarged spheres back down to the molecular van der Waals surface; from there, the area of the molecular surface accessible to solvent (AMSAS) can be computed easily by summing the areas of the surface patches represented by these projected dots. Second, one could parameterize the SASApack term based on unoccupied volumes instead of molecular surface areas by correcting $A_{0.5} - A_{1.4}$ for each atom by a term that depends

on the atom's radius. That is, if a dot for an atom with van der Waals radius r on the the $r + 1.4$ Å surface is covered and its corresponding dot on the $r + 0.5$ Å surface is exposed, then the *stick* that connects these two dots is a discrete representative for an unoccupied volume (a *void volume*) of the protein structure. The volume that this stick represents is

$$v_{stick} = \frac{4\pi}{3}((r + 1.4)^3 - (r + 0.5)^3)/(\#\text{dots}).$$

Since the SASApack term is parameterized in terms of accessible surface areas of the molecular surface, this pair of dots contributes a surface area to the $A_{0.5} - A_{1.4}$ of

$$a_{dot\ pair} = 4\pi r^2/(\#\text{dots})$$

so that the proportional difference in contribution of this dot pair is an affine function of r:

$$\frac{1}{1.4 - 0.5} v_{stick} - a_{dot\ pair} = \frac{1}{1.4 - 0.5} * \frac{4\pi}{3}\Big((1.4^2 - 0.5^2)r + (1.4^3 - 0.5^3)\Big)/(\#\text{dots}).$$

As SASApack is currently defined, it approximates a quantity that correlates to void volume; it would be interesting to examine the variation on SASApack that approximates void volume directly.

8.4 Results

8.4.1 Performance

To assess how much of an advantage our update algorithm represents, we compare it against a straw-man algorithm that computes the SASA of the entire protein *from scratch* with the Le Grand and Merz protocol after each MC rotamer substitution. Although this from-scratch algorithm has not appeared in the literature, it would seem that this algorithm is what researchers have in mind when they claim that exact SASA computations are too slow to be practical (Street and Mayo, 1998; Ferrara et al., 2002).

Complete redesigns (∼420 rotamers per sequence position after backbone-colliding rotamers have been removed) were performed on seven proteins with the SASApack score maintained during rotamer optimization. Because the running times for the from-scratch algorithm are very long, we present running time projections calculated

PDB ID	# residues	# rotamers	# rotamer subs (millions)	From Scratch (projected) $\times 10^5$	SASA Update, + no opt.	SASA, + shared atom opt.	SASA + procrastination	SASA + both opt.	Rosetta PD energy
1PTF	88	42351	3.6	3.6	20361 (21279)	11045 (11986)	4434 (5398)	2565 (3500)	35.9 (942.7)
1BXM	99	33221	2.8	3.2	13529 (14016)	7333 (7819)	4385 (4882)	2503 (2983)	23.2 (490.9)
1J27	102	40403	3.4	3.9	18555 (19245)	9997 (10692)	4388 (5108)	2519 (3216)	33.5 (721.7)
1DDW	120	56199	4.8	6.3	28261 (29519)	14812 (16089)	5792 (7095)	3399 (4671)	50.2 (1295.0)
1DOI	128	61926	5.3	8.0	31321 (32784)	16727 (18185)	6818 (8325)	3939 (5400)	58.5 (1508.6)
1RCB	129	55615	4.7	7.2	26529 (27538)	13821 (14825)	8375 (9406)	4799 (5803)	49.1 (1050.0)
1OSA	148	51015	4.3	7.4	20604 (21202)	11268 (11878)	5918 (6533)	3464 (4067)	40.9 (633.7)

Table 8.1: *Running Times For SASA-Update Algorithm.* Running times in seconds for complete-protein redesigns measured on a 2.5 GHz Macintosh G5 with 4 GB of RAM. The projected running times for the from-scratch algorithm on an N-residue protein are roughly N*1.45 times longer than the measured running times for the SASA update algorithm that includes both optimization techniques. Measured running times are reported both for the Monte Carlo optimization stage of the simulation and for the full simulation (in parentheses). The Rosetta PD column contains times for Rosetta's original pre-computed pairwise-decomposable energy function.

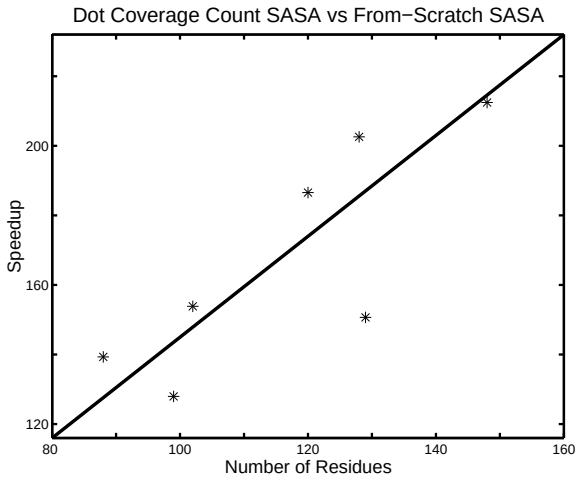


Figure 8.2: *Speedup Against a From-Scratch Algorithm.* Speedup of our dot-coverage count update algorithm relative to computing SASA from scratch. The running times for the from-scratch algorithm were projected for the seven redesigns in our test set and compared against the measured running times of our dot-coverage-count algorithm. The speedup can be approximated by the line Y = 1.45 X.

by measuring the time required to compute the SASApack score of the entire protein and multiplying this by the number of rotamer substitutions made during MC optimization. We also compared our update algorithm against the optimization of Rosetta's standard pairwise-decomposable energy function without the SASA term. Protein PDB identifiers, sizes, and running times are reported in Table 8.1.

On an N-residue protein, our update algorithm runs $N \times (1.45 \pm 0.16)$ times faster than the from-scratch algorithm (Figure 8.2). That is, for a 100-residue protein, the update algorithm is about 145 times faster. This linear speedup is what we had hoped to achieve by making the update proportional to the number of atoms affected by a rotamer substitution, instead of the number of atoms in the protein. For the from-scratch algorithm, the projected per-residue cost for a single rotamer substitution was 1.15±0.03 ms. For the update algorithm, the average rotamer substitution took 0.80±0.11 ms.

As expected, design simulations with our fastest implementation of the SASA-update algorithm are slower than simulations with standard Rosetta. For the seven test proteins the average running time increased by a factor of 4.78±1.22. With standard Rosetta, the rate-limiting step in the simulation is precomputing energies between rotamer pairs. With the dot-coverage-count algorithm, the Monte Carlo rotamer search

SASAprob weight	SASAprob score	Total Rosetta Energy*	% sequence identity to native
native	0.55	-16.5	-
0.0	0.40	-120.62	32
0.01	0.41	-120.60	31
0.05	0.44	-120.05	32
0.15	0.49	-118.29	33
0.20	0.51	-117.00	32
0.50	0.60	-109.43	31
1.0	0.67	-97.23	29

Table 8.2: *SASAprob Improves Sequence Recovery.* Average results from the complete redesign of 102 high resolution protein structures. *Total Rosetta energy is without the SASAprob score.

with simulated annealing becomes the rate-limiting step, and takes on average 82 times longer than with the standard PD energy function. The slow step in optimizing a PD energy function (retrieving precomputed rotamer-pair energies from memory) is much faster than the slow step in our algorithm (computing sphere overlaps).

8.4.2 Optimization of the SASAprob Score

To further test our SAS-update algorithm we examined whether it could be used to optimize the SASAprob score during design simulations. In standard Rosetta simulations with naturally occurring protein backbones, the SASAprob score of the final design model is generally less favorable than that of the starting structure. In complete redesigns of 102 high-resolution protein structures the average SASAprob score per residue was 0.40 while the average score of the wild-type proteins was 0.55. A value below 0.5 indicates that the protein is less well packed than the average protein in the PDB. We see similar results at protein-protein interfaces: 30 protein-protein interfaces

were redesigned by allowing all amino acids in chain A and keeping the sequence of chain B fixed. The average SASAprob score of the redesigned residues was 0.30, while the same residues in the wild-type structures had a SASAprob score of 0.47. These results suggest that Rosetta is systematically creating voids in its designs that are too small to be filled by water.

By including a SASAprob term in sequence optimization, one can design structures with SASAprob scores comparable to those of naturally occurring proteins (Table 8.2). We performed complete-protein redesigns on 102 structures, varying the weight W applied to the SASAprob score in the range between 0.01 and 1.0. As the weight increased, the SASAprob scores of the designed structures become more favorable; however, as more emphasis is placed on the SASAprob term, the other Rosetta energies become less favorable. Weights near 0.20 appear optimal; the SASAprob score is near the native-like value of 0.5 and the other Rosetta energy terms are nearly as favorable as they are in the absence of the SASAprob term. We observe similar results in protein-protein interface redesign.

With a SASAprob weight of 0.15, the identity between the designed sequences and the native sequences is at a maximum. Previous experience has shown that improvements in the Rosetta energy function as evaluated by successful protein designs or structure prediction often corresponds with the ability to recapitulate naturally occurring protein sequences (Dantas et al., 2003). The SASAprob score also increases native sequence recovery at protein-protein interfaces. With the standard Rosetta energy function we obtain 42% sequence identity, whereas by including the SASAprob term with weight 0.25 we obtain 45% sequence identity.

The SASAprob score also somewhat improves accuracy in predicting side-chain geometry, as measured by dihedral angle recovery when repacking native-sequence structures. The chi angles of the predicted structures from our test set of 102 proteins were compared with those of the native structure: 67.4% of χ_1 and χ_2 angles were predicted to occupy the correct rotamer with a SASAprob weight of 0.15, whereas 66.4% were predicted correctly without the SASAprob score.

8.5 Discussion

Our SAS-update algorithm makes it feasible to include SASA-derived energy functions during Monte Carlo sequence design using the more accurate Le Grand and Merz computations instead of pairwise approximations. Here, we have used this capability

to optimize a measure of protein packing based on surface area, but it will also be useful for optimizing other measures of protein energetics that correlate with SASA or void volume near the protein. It should be most useful in cases where the errors associated with pairwise approximations of buried surface area are likely to result in significantly different estimates of protein energy. The desolvation cost for burying a polar atom depends in part on whether the atom can form hydrogen bonds with water. Because hydrogen bonds are discrete, there may be a significant energetic difference between an oxygen atom with 10% of its surface area exposed to water and an oxygen atom with no exposed surface area. Indeed, Liang and Grishen found that a desolvation penalty that depended non-linearly on exposed polar surface area recapitulated naturally occurring sequences more accurately (Liang and Grishin, 2004). In addition, the surface area adjacent to the lone pair electrons on an oxygen should be free in order to form a favorable hydrogen bond with water. The Le Grand and Merz approach, and our extension, make it possible to examine specific regions on the surface of each atom, and therefore, tailor precise orientation-dependent scores based on exposed surface area.

8.6 Implementation of SASApack Optimization With An Interaction Graph

The remainder of this chapter[2] describes the implementation of the `SASAInteractionGraph` and argues that modeling the SASApack energy function with a graph is clean and effective; an interaction graph eases implementation. The data required to implement SASApack efficiently is substantial, and data is often required for specific pairs of residues – but not all pairs. The graph provides a convenient way to store data that might otherwise belong in a sparse matrix. The interaction-graph base classes provide so much functionality and are so well factored that the implementation time for the first `SASAInteractionGraph` prototype was a matter of 5 weeks.

8.6.1 Graphical Model for SASApack

Chapter 5 introduced the idea of decomposing a single non-pairwise-decomposable-protein-energy function F into several non-pairwise-decomposable-energy functions, $\{f_1, f_2 \ldots\}$ such that $F = \sum_i f_i$. This decomposition gives a computational advan-

[2]The remaining sections of this chapter were not part of the paper submitted to the Journal of Computational Chemistry, and are original to this dissertation

tage if the domain of each of these functions is smaller than the domain of F, the whole protein. In the case of the SASApack function, the decomposition into smaller functions is simple, since the SASApack function is defined on a per-residue basis. Each residue i in the protein defines one function f_i, and the domain of f_i is the product of the state spaces of those molten residues that can overlap residue i. Because the enlarged spheres surrounding the atoms on a pair of residues must overlap in order for those residues to affect each other's SASApack score, the same density argument used to prove the sparse connectivity of an interaction graph for Rosetta's pairwise-decomposable energy function applies to the SASApack function: the number of residues that affect the SASApack score for a single residue can be bound by a constant. This means that the size of the domain of each function f_i does not increase with protein size, and for all but the smallest of proteins, will represent a proper subset of all of the molten residues.

Chapter 5 also introduced an extension of the interaction graph to model non-pairwise-decomposable-energy functions. In this extended model the vertex set V consists of two classes of vertices, $V_1 \cup V_2$: those vertices that represent variable parts of protein, which carry state spaces, and to which the annealer assigns states (V_1); and those vertices that represent non-pairwise-decomposable functions and are connected by a set of edges to those variable vertices whose states are in the function's domain (V_2). Call the vertices in V_1 the *first-class vertices*, and the vertices in V_2 the *second-class vertices*.

In implementing the `SASAInteractionGraph`, I represent each function f_i defined on a background (fixed) residue with a second-class vertex; a `SASABackgroundNode` object. I represent each function f_i defined on one of the molten residues as part of that molten residue's corresponding first-class vertex, a `SASANode` object, and not as a separate second-class vertex. Each vertex in the graph maintains dot-coverage information for its corresponding residue. For the first-class vertices, this means maintaining dot-coverage information as its corresponding residue undergoes rotamer substitutions.

I connect a pair of interacting first-class vertices with a first-class edge; first-class edges are represented by the `SASAEdge` class. I connect a first-class vertex that interacts with a second-class vertex with a second-class edge; second-class edges are represented by the `SASABackgroundEdge` class. The first-class edges store the pairwise-decomposable portion of the energy function tables, just as the edges in the `PDInteractionGraph`. Both first-class and second-class edges maintain information relevant for dot-coverage count update.

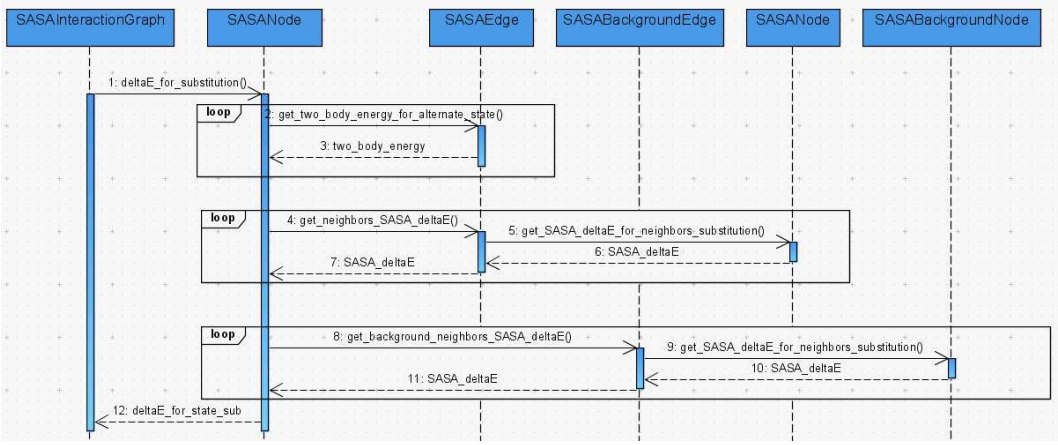


Figure 8.3: *Sequence Diagram of* `SASAInteractionGraph` *Rotamer Substitution Algorithm.* The node undergoing substitution (2nd column) first iterates across its first-class edges to collect the change in the pairwise decomposable portion of the energy function. Next it iterates across the first-class edges again to traverse to its first-class neighbors where it computes the new surfaces for each neighbor, retrieving the change in the SASApack score for each neighbor. Finally, it iterates across the second-class edges to traverse its second-class neighbors where it computes the new surfaces for each of its neighbors, retrieving the change in the SASApack score for each neighbor.

8.6.2 Rotamer Substitution Algorithm

The primary operation that the graph performs is to answer the annealer's call to `deltaE_for_substitution()`, informing the annealer of the energy effect that substituting a state on one vertex v with a new state. When v changes its state, it induces a change in the pairwise-decomposable portion of the protein's energy (ΔE_{PD}) and in the SASApack portion (ΔE_{SASA}). The change in the pairwise-decomposable portion of the energy function can be computed by iterating across the first-class edges incident upon v, and reading from the precomputed pair-energy tables. The change in the non-pairwise-decomposable portion of the energy function induced on the entire protein can be computed by iterating across both the first-class and second-class edges incident on v and updating the the two surfaces (the 1.4 Å surface and the 0.5 Å surface) for each of its neighbors.

The interaction graph updates the surfaces one residue at a time, by iterating across the v's incident edges; both its first- and second-class edges. I sketch the `deltaE_for_substitution()` method in a UML sequence diagram in Figure 8.3 and pictorially in Figure 8.4. When computing ΔE_{SASA}, the vertex undergoing substitu-

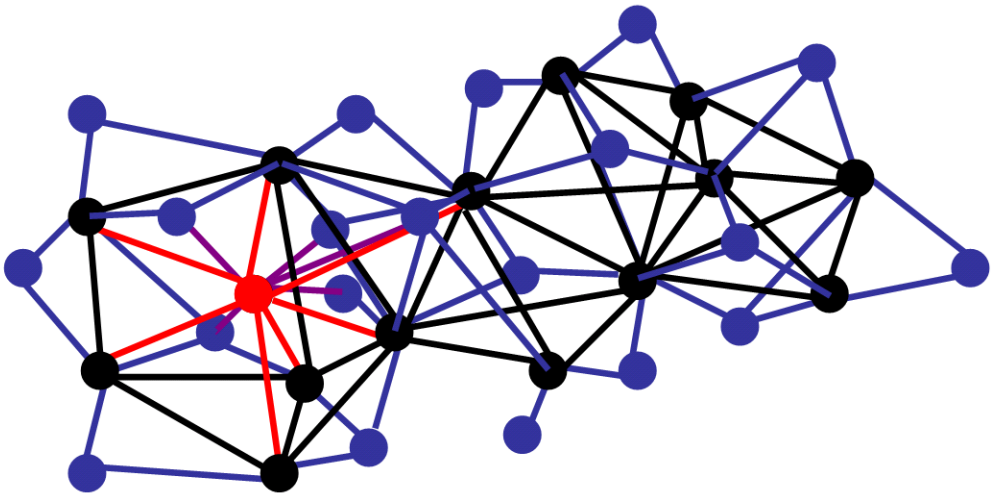


Figure 8.4: *Traversals in the* `SASAInteractionGraph`. A hypothetical interaction graph with first-class vertices (black + red) and second-class vertices (blue) where one of the first-class vertices (red) is undergoing a rotamer substitution. The red vertex must traverse each of its incident edges, those to its first-class neighbors (red edges) and those to its second-class neighbors (purple edges).

tion iterates across each of its incident edges; it iterates across both its first-class and its second-class edges. The operations for the two classes of edges are similar, so I describe the traversal for the first-class edges. The vertex undergoing substitution invokes a method on each first-class edge, `get_neighbors_SASA_deltaE()`, which returns the change in SASAprob score for its neighbor in response to its state substitution. The vertex undergoing substitution passes data about the substitution it is considering. Part of this data includes an object for representing the dot-coverage counts for the alternate rotamer, which gets modified during the course of this function call. Another part of this data is the `index_` of the vertex – knowing the index of the vertex undergoing the substitution, the edge can figure out which of its two vertices is not undergoing the rotamer substitution.

The `SASAEdge` method `get_neighbors_SASA_deltaE()` in turn invokes a method on the `SASANode` that is not undergoing the rotamer substitution. It calls a `SASANode` method `get_SASA_deltaE_for_neighbors_substitution()` which returns the change in SASAprob score for a vertex in response to a rotamer substitution by one of its neighbors. The edge passes data to the `SASANode` that the vertex needs in order to compute its change in SASAprob score: this includes data passed to the edge by the

vertex undergoing the substitution, and also member data from the edge itself. I describe the data passed around in these method invocations in the subsections below.

8.6.3 A Class For Dot-Coverage Counts

Dot-coverage counts for a single residue are stored in the `RotamerDots` class. The `RotamerDots` class contains the coordinate information for each of the atoms in the residue and the dot-coverage counts for the two surfaces. Before giving pseudocode for the `RotamerDots` data members, I should explain the four "magic numbers" that appear in it. 1) There are at most *forty* atoms for any amino acid, and I allocate my arrays so that a `RotamerDots` object can accommodate any amino acid. 2) The coordinate for each atom is represented in *three* dimensions. 3) Each `RotamerDots` object stores coverage counts for the *two* spheres that surround each atom. 4) In Rosetta's implementation of the Le Grand and Merz sphere-overlap-computation algorithm, there are *162* dots on the surface of each sphere. That said, the data in the `RotamerDots` class look like:

```
class RotamerDots
{
  private:
   RotamerCoords coords_;
   DotSphere dots_[2][40];
   //...
};
```

and the two classes it contains, look something like

```
class RotamerCoords
{
  private:
   float coords_[40][3];
   float vdwRadii_[40];   //van der Waals radii
   int aa_type_;          //amino acid type for this rotamer
   int num_atoms_;
   //...
};
```

and

```
class DotSphere
{
  private:
   unsigned char dotCoverageCounts_[162];
   //...
};
```

Each vertex in the graph (first- and second-class) represents the dot-coverage counts for its current state with a `RotamerDots` data member named `currStateDots_`. The graph maintains the following invariant: the two surfaces for the protein's current rotamer assignment are reflected in the dot-coverage counts for the `currStateDots_` objects of the vertices in the graph.

The `RotamerDots` class has a public method `getSASAPackScore()` that returns the SASApack score for the residue using the dot-coverage information already stored in its internal data. The function does not compute sphere overlaps. The responsibility for ensuring that all necessary sphere-overlap computations have been performed before this function is called falls on the graph.

After the neighbor of a vertex undergoing a rotamer substitution finishes updating its surface in response to the substitution, it invokes the `getSASAPackScore()` to compute its new SASApack score, subtracts its old SASApack score from the new one, and returns the difference to the vertex undergoing the substitution.

Each vertex, in addition to representing dot-coverage information for its current state, also represents dot-coverage information for its alternate state in a second `RotamerDots` data member named `altStateDots_`. That is, for the vertex undergoing the rotamer substitution, the `altStateDots_` member variable holds the coordinates for the alternate rotamer and accumulates dot-coverage counts over the course of the sphere-overlap computations it performs with its neighbors. After this vertex completes the traversal of all its incident edges, the accumulation is complete; the SASApack score is ready to be computed for the alternate state. For a neighbor of the vertex undergoing the rotamer substitution, the `altStateDots_` member variable computes and stores the dot-coverage information induced by its neighbor's rotamer substitution. The neighbor copies the data held in its `currStateDots_` member variable into its `altStateDots_` member variable and then decrements, increments, and counts dots for the `altStateDots_` variable, leaving its `currStateDots_` object untouched.

I copy data into the `altStatesDots_` object instead of altering the `currStateDots_` object directly because, when the annealer rejects rotamer substitutions, the graph is

able to rapidly restore the surface maintenance invariant. Since more state substitutions are rejected than are accepted, it is useful to be able to rapidly undo any modifications made to the current state assignment.

When the annealer does commit a state substitution, the vertex copies data from its `altStateDots_` object into the `currStateDots_` object establishing the invariant.

When a neighbor of the vertex undergoing a rotamer substitution updates its dot-coverage counts, it increments the coverage counts for each dot on its surface covered by the alternate rotamer. In the increment step, I perform sphere-overlap computations for this rotamer pair. I save the results of rotamer-pair-sphere-overlap computations for later reuse, as I describe in the next paragraph.

When a neighbor of the vertex undergoing a rotamer substitution updates its dot-coverage counts, it decrements the coverage counts for each dot on its surface covered by the rotamer currently assigned to the vertex undergoing the substitution. This decrement operation requires data from sphere-overlap computations, but does not require that fresh sphere-overlap computations be performed, just that data from old computations be reused. The sphere-overlap computations between the current rotamer assigned to the neighbor and the current rotamer assigned to the vertex undergoing substitution have already been performed; these overlap computations were computed at the time the last one of the two of these vertices underwent a state substitution – the state substitution that left them in their current states. In the course of that state substitution, during the increment operation, I had stored the rotamer-pair-sphere-overlap information for this rotamer pair. That information is ready for re-use in the decrement operation.

The rotamer-pair-sphere-overlap information is simply a 1 or a 0 for each dot on each of the two surfaces of each atom in the two rotamers. To represent the rotamer-pair-sphere-overlap information for a single rotamer, I use an additional class, the `RotamerDotsCache` class. The data for this class look like:

```
class RotamerDotsCache
{
  private:
   DotSphere dotCoverageCounts_[ 2 ][ 40 ];
   //...
};
```

I store one `RotamerDotsCache` object for each of the two rotamers involved in each rotamer sphere-overlap computation. I store these caches as data members on the edges that connect pairs of vertices. Here again, the design decision to implement this energy function using a graph has paid off: data corresponding to pairs of vertices are implemented as data members on edges.

The `RotamerDots` class is responsible for the actual sphere-overlap computations. The class supports a sphere-overlap computation method that computes and caches the sphere overlap for a pair of `RotamerDot` objects:

```
RotamerDots::increment(
   RotamerDots & other,
   RotamerDotsCache & this_covered_by_other,
   RotamerDotsCache & other_covered_by_this )
```

This method computes the sphere overlap between the two `RotamerDots` objects (`this` and `other`), stores this rotamer-pair-overlap data in the two `RotamerDotsCache` objects and increments the dot-coverage counts for both `RotamerDots` objects. Indeed the method increments the coverage counts for both of the `RotamerDots` objects from the pair overlap data stored in the two `RotamerDotsCache` objects. The shared-atom optimization relies on this fact, loading useful data into the two caches before a call to `increment()`.

8.6.4 Shared Atom Optimization

In addition to storing `RotamerDotsCache` objects on edges, I also store the result of sphere-overlap computations on edges. The shared-atom speedup described in Section 8.2.4 relies on previously stored sphere-overlap information for atom pairs in order to avoid sphere-overlap computations for atoms that have not moved. To save the atom-pair-overlap data, I store a single integer for each sphere (four integers for a pair of atoms; two atoms with two spheres each) where that integer represents the index for the precomputed Le Grand and Merz boolean mask describing that atom's covered dots. That is, I store a table of these atom-pair-overlap-mask indices on each edge – this table is passed as an additional parameter to the `RotamerDots::increment()` method. As this method computes atom-pair overlaps it stores the mask indices in this table for later reuse.

The reuse of atom-pair overlap data is tricky. First, there is the problem of avoiding computations the results of which are to be reused. The `increment()` method has to

know exactly which atoms are shared between two rotamers in order to avoid the right computations. With the proper ordering of atoms in a rotamer, I can specify which atoms are shared by indicating a range. That is, I order atoms in the `RotamerDots` class in the same way I order atoms in the rotamer trie (see Chapter 6). Then I can indicate which atoms are shared between two rotamers by stating the size of their shared prefix. In fact, I can compute the number of shared atoms with a linear-time, output-sensitive algorithm: I simply march through the two rotamers, comparing atom i on rotamer 1 with atom i on rotamer 2, and stop when two atoms differ. The number of shared atoms between the alternate rotamer and the current rotamer is given as another input parameter to the `RotamerDots::increment()` method.

Because shared atoms represent a prefix of the alternate rotamer's atoms, the loops that compute the sphere overlaps for the alternate rotamer r's with a neighboring rotamer s need iterate only over the suffix of the r that differs from the suffix of the current rotamer. That is, if `numShared` is an integer input into the `RotamerDots::increment()` method, then the nested loops would look like:

```
for i = numShared + 1 : r.num_atoms_
   for j = 1 : s.num_atoms_
      sphere_overlap( r[ i ], s[ j ] );
      //...
```

The second challenge in the shared-atom optimization is the proper reuse of stored atom-pair overlap data. Before calling `RotamerDots::increment` for the pair of rotamers, I load the `RotamerDotsCache` objects that are passed as input to `increment()` with data from the previously computed sphere overlaps. Since both these `RotamerDots Cache` objects and the tables of atom-pair-overlap-mask indices are data members of edges, I give the task of loading the caches with the overlap data to the edges. The edges load the data into the caches before they call `SASANode::get_SASA_deltaE_for_neighbors_substitution()`. This method in turn calls the `RotamerDots::increment()` method, which computes the appropriate set of sphere overlaps, and then increments the dot-coverage counts for the `RotamerDots` objects based on the data stored in the `RotamerDotsCache` objects.

8.6.5 Precomputing Rotamer-Pair Overlap?

In discussing the strategy of which sphere-overlap computations to perform during simulated annealing, I have left unstated the design decision to not precompute and

store all possible rotamer-pair-sphere-overlap information. My coauthors and I clearly had to make such a decision; the habit of precomputing of rotamer-pair energies for a pairwise-decomposable energy function forced us to consider such precomputation for the SASApack function. Such a strategy could work and save a lot of time: though the increment and decrement operations for `RotamerDots` objects would not disappear, these operations could rely completely on precomputed overlap information and avoid all sphere-overlap computations at runtime.

The reason we decided against such precomputation is because of the memory required to represent rotamer-pair-overlap data. Consider the amount of memory required to represent the pair overlap for an average rotamer. If the average amino acid has twenty atoms, and if the coverage for each of the 162 dots were represented with a single bit (and 6 bits were wasted to fill 21 bytes fully), then it would cost $20*21*2 = 840$ bytes per rotamer, and 1680 bytes per rotamer pair. This is too expensive. Designers use so many rotamers that they run out of memory when storing only 4 bytes (1 float) per rotamer-pair to optimize a pairwise-decomposable-energy function. Increasing the cost per rotamer pair by a factor of 400 would limit designers to unreasonably small rotamer libraries.

Chapter 9

Conclusions

The design of novel proteins continues to be a major challenge in computational structural biology. In this dissertation, I have outlined several specific challenges and described my contributions towards overcoming these challenges; in each one of my contributions, the interaction graph model for side-chain-placement has proven useful. The interaction graph framework I have added to Rosetta invites several extensions that I am eager to implement. This chapter summarizes my results and their limitations, and outlines some future work.

9.1 Summary of Results

I see three principal challenges for protein design: handling the intrinsic complexity of the side-chain-placement problem, sampling conformations at the detail required, and accurately modeling protein energetics. Using an interaction graph, I have addressed (but by no means closed the book on) each of these three challenges.

9.1.1 Side-Chain Placement's Complexity

The interaction graph reveals that the complexity of the side-chain-placement problem depends on a graph parameter: treewidth, which is often bound and sometimes very small. Without the graph, then the most that can be said about side-chain placement is that the problem is NP-Complete.

For very low treewidth graphs, dynamic programming can optimize side-chain placement. I have demonstrated a problem instance from protein design where dynamic programming can optimize side-chain placement, as well as an instance where a novel

form of dynamic programming, which I call *adaptive dynamic programming*, computes a near-optimal side-chain placement in significantly less time than dynamic programming.

I have also demonstrated the utility of dynamic programming at the hydrogen placement problem where the treewidths of the interaction graphs are much smaller than they are in most protein design problems. My implementation of dynamic programming in REDUCE is currently being distributed and is now in use on the Richardson's hydrogen placement server.

9.1.2 Conformational Sampling

Redesign of an existing protein resembles solving a jigsaw puzzle where the boundary of the puzzle has been laid down and the pieces in the middle need filling in. The pieces available don't always fit. Designers have tested three options to fit the middle pieces. First, to accept puzzle-piece placements where the pieces overlap – that is, by using soft repulsion in their energy functions, designers can place side chains to produce designs with atomic collisions that will shift out when the design is realized. Second, to consider more puzzle pieces so that it becomes more likely that collision-free placements can be found. Third, to move those pieces on the boundary so as to allow piece placements that otherwise could not fit.

Harbury and Kuhlman's successes in crystallizing and solving the structure of proteins designed with backbone flexibility (Harbury et al., 1998; Kuhlman et al., 2003), along with the consistent inability to crystallize proteins designed without backbone flexibility (Hill et al., 1990; Kuroda et al., 1994; Dahiyat and Mayo, 1997b; Dahiyat et al., 1997; Fezoui et al., 1997; Walsh et al., 1999; Johnson et al., 1999) suggest that the design of proteins that are sufficiently stable and ordered to be crystallized requires this third design option: that is, a single backbone scaffold is insufficient for the redesign of a protein. Both Harbury and Kuhlman considered many different backbone conformations during their designs, and solved many instances of fixed-backbone side-chain placement. The number of backbone conformations they were able to sample were limited by the computational expense of running many fixed-backbone designs.

With an interaction graph as a data structure, I have extended Rosetta's software for side-chain placement to allow the simultaneous optimization of side-chain and backbone conformations. Though we have not synthesized any proteins designed using this software, I have shown that the additional flexibility allowed to the backbone improves

the energy of designed proteins. The interaction graph for local backbone flexibility and the simulated annealing algorithm together find lower energy conformations by searching through wider regions of conformation space.

9.1.3 Capturing Protein Energetics

Although the stability of a real protein is not expressible through pairwise-decomposable functions, the computational efficiency enabled by pairwise-decomposable functions has motivated many procrustean attempts to fit the non-pairwise-decomposable reality of protein energetics into a pairwise form. As I discussed in Chapters 2 and 8, the solvent-accessible surface (SAS) and the solvent-accessible-surface area (SASA) of a protein depend on the location of all atoms in the protein, and therefore cannot be computed by a sum of pair terms. Therefore, any aspect of protein stability that depends on either the SAS or the SASA cannot be captured exactly by a pairwise-decomposable energy function. Such aspects of protein stability include the hydrophobic effect, which is thought to provide the main source of protein stability (Kauzman, 1959; Dill, 1990); the instability produced by burying from water an unsatisfied hydrogen bonding group (Baker and Hubbard, 1986; Fleming and Rose, 2005); and the quality of packing, which is better expressed in terms of accessible volumes instead of accessible surface areas (Sood and Baker, 2006). If the reality of protein energetics is so firmly non-pairwise decomposable, it stands to reason that to limit ourselves to pairwise-decomposable energy functions hinders our attempts to model and understand protein behavior.

This dissertation describes a general framework for the incorporation of non-pairwise-decomposable functions into the optimization of side-chain placement, and have described one particular application of this framework to penalize the presence of small voids in designed proteins. Because the desired non-pairwise-decomposable function was included during the optimization, the structures Rosetta produced improved according to this function over those structures optimized in its absence. This improvement supports my hypothesis that relying upon non-pairwise-decomposable post-processing filters following the optimization of a pairwise-decomposable energy function cannot produce ideal designs; designs that would look good to the post-processing filters will have already been selected against during the optimization process.

9.2 Limitations

There are several limitations worth noting in the work I have presented. I address the limitations in each area of work in the same order I described their merits above: first, in dynamic programming; second, in design with local backbone flexibility; third, in the incorporation of the SASApack function into design.

Dynamic programming is chiefly limited by the treewidth of the interaction graphs present in protein design. To my disappointment, the interaction graphs for most instances from protein design have a treewidth too large to be solved with dynamic programming (see Section 4.12). As a consequence, dynamic programming in protein design would require too much time and too much memory to serve useful in most design cases. Though dynamic programming can offer a means of verifying the results of the stochastic optimization techniques that designers often use, it is unlikely that designers will ever rely on dynamic programming to perform the majority of their optimizations.

Currently, design with local backbone flexibility is limited by the fact that alternate backbone conformations must be created outside of Rosetta and read as an input file. Design with local backbone flexibility thus cannot be automated. The first step in automation is the integration of PROBIK into Rosetta. Once the integration of PROBIK into Rosetta completes, issues such as the selection of regions of the backbone to flex and the selection of conformational samples to include must be dealt with. In design, because the designer pays a great deal of attention to every scaffold that she uses, it is reasonable to expect her to specify which regions of the backbone to flex. In folding, however, the user encounters thousands of different backbone conformations, and cannot afford to look at them all. If local backbone flexibility is to be useful in folding, then Rosetta will need to automate the decisions of which parts of the backbone to flex. Until both PROBIK and the decision making code are incorporated into Rosetta, the local backbone flexibility functionality cannot be automated and will not likely produce productive designs.

To an extent, design with local backbone flexibility is limited by its speed; this is principally due to the increased number of states per residue. Each residue builds a set of rotamers from each of the backbone conformational samples it has available; if the rotamer library dictates the building of s rotamers for a single backbone conformation, then a residue with b backbone conformational samples will build sb different rotamers and thus have sb states. The time to compute pair energies increases quadratically with the number of states per residue; the pair-energy precomputation phase increases

substantially for design with local backbone flexibility. Moreover, simulated annealing's running time also increases in design with local backbone flexibility over its running time in fixed-backbone design. For one, the number of state substitutions the annealer performs scales linearly with the total number of states for all residues in the protein. Simulated annealing is also slower because the `FlexBBInteractionGraph` incurs twice as many cache misses per state substitution. In the case of the `PDInteractionGraph`, I copied the contents of the offset indices used by the `AminoAcidNeighborSparseMatrix` class into a single table on the vertex – one offset table per edge – and thereby cut the number of cache misses per rotamer substitution in half. Instead of missing cache once to retrieve an offset and missing cache again to retrieve an energy, the `PDInteractionGraph` misses cache only for the energy retrievals. I have not employed this strategy for the `FlexBBInteractionGraph` because each edge maintains several `AminoAcidNeighborSparseMatrix` objects, each sparse matrix with its own table of offsets. For each energy retrieval in a rotamer substitution, the `FlexBBInteractionGraph` incurs two cache misses.

Primarily, design with local backbone flexibility is limited by its memory usage. As is the case in fixed backbone design, the amount of memory required to represent rotamer pair energies increases quadratically with the number of rotamers per residue. Using more than two or three alternate backbone conformations for each of eight segments as well in addition to using a rotamer library with additional rotamer samples at χ_1 and χ_2 requires as much memory as is available on a 32-bit machine. With local backbone flexibility, the designer is now faced with the difficult choice of including more backbone conformations or including more side-chain-dihedral samples.

The time and memory expenses make it difficult to discern whether design with local-backbone flexibility is truly superior to using Rosetta to iterate between backbone motion and fixed-backbone design. Certainly the fact that alternate backbone conformation generation is not fully automated prevents comparing iterative combinations of backbone motion and locally-flexible backbone design with iterative combinations of backbone motion and fixed-backbone design.

The limitation of SASA computations have always been their speed. Our approach makes it possible to maintain the solvent accessible surface during a design trajectory, but is still slow relative to simulated annealing with a pairwise decomposable energy function, and so speed remains a limitation. Although a call to the packer that runs a single design trajectory may take only five times longer from beginning to end than when using a pairwise decomposable energy function, if the designer wants to run so

many trajectories that simulated annealing fully dominates the running time, then it would go 80 times slower. Time is the main enemy when trying to sample backbone conformation space broadly; if each instance of fixed-backbone design takes five times longer when using the SASApack scoring function, then a designer is able to sample only one fifth of backbone conformation space.

The SASApack function is also limited by its incongruity with the rest of Rosetta. The minimizer in Rosetta computes derivatives for the pairwise-decomposable energy function for each of the dihedral angles in the protein (Abe et al., 1984), and using gradient-based minimization, twists these dihedrals to minimize the energy of the structure. In order to compute derivatives for dihedral angles, the function must be decomposable into atom-pair interactions. SASApack is not pairwise decomposable, and therefore SASApack cannot be included as part of the minimizer. The existing design-with-backbone-flexibility protocol iterates between rounds of fixed backbone designs and backbone motion with gradient-based minimization. If SASApack were included in energy function during fixed-backbone design step, but not included in the backbone motion step, it is possible that the minimizer could undo the side chain placements the packer had worked to create. A dissonance between two parts of Rosetta might be problematic, but it is also possible it might not be.

9.3 Future Work

9.3.1 Speed vs Memory Balancing

The extensibility of the interaction graph framework makes it possible to extend the software for use with the existing pairwise-decomposable energy function so that it can trade speed for memory. I envision several sets of interaction graph classes that balance the performance costs of computing rotamer-pair energies *on the fly* and the memory costs of maintaining those rotamer-pair energies.

At one end of the spectrum, I would create an interaction graph that allocates tables to store rotamer-pair energies before simulated annealing begins, but does not precompute those energies. Instead, it computes them on the fly and stores them for later retrieval. This extension would trade memory (12% more) to allocate larger tables than are absolutely necessary for the speed advantage of computing only those energies that get used. Such a system would likely be faster if many precomputed rotamer-pair energies are never accessed during simulated annealing. Indeed, Yi Liu and Brian

Kuhlman have observed that in many design trajectories, the majority of the rotamer-pair energies are never examined (Liu and Kuhlman, personal communication).

At the other end of the spectrum, I envision an interaction graph that allocates no space at all for the storage of rotamer-pair energies, but that instead computes all energies on the fly. Such a graph would permit the use of extremely large rotamer libraries in design, the likes of which cannot be used currently due to space limitations. Even if memory were no object, the scheme to compute energies on the fly would likely outperform the existing framework for designs with extremely large rotamer libraries: the energy-precomputation phase scales quadratically with the number of rotamers per residue, and yet most rotamer-pair energies never get examined. Moreover, the fact that cache misses dominate simulated annealing's running time suggests that at as processor speed improves, it will eventually be faster to compute energies on the fly than to precompute them. With an on-the-fly graph, we will be ready to take advantage of this speed when the changeover occurs.

In between these two ends of the spectrum, I envision a compromise graph, which does not store every computed rotamer-pair energy that the annealer has required, but does store a subset of those energies; energies it thinks it is likely to need again in the near future. Suppose in the late stages of simulated annealing, vertex i has been swapping back and forth between two states, r_1 and r_2. Each time the annealer considers a state substitution at a neighbor j of vertex i, the graph has to know (evaluate or look-up) that state's interaction energy with whichever state is currently assigned to i, which we have already supposed is either state r_1 or r_2. If the graph were to cache the interaction energies between the states r_1 and r_2 of residue i and all states for residue j, then the graph can avoid repeating many rotamer-pair-energy computations.

In this compromise graph, each edge would cache the rotamer-pair energies for a subset of the rotamers on each of its two vertices; *e.g.* the five rotamers most recently committed to each. For example, consider the edge incident upon vertex-1 and vertex-2 where each vertex had s_1 and s_2 states to choose from. This edge would store a pair of tables – one for each vertex – where the size for the table for residue 1 is $5 \times s_2$ and the size for the table for residue 2 is $5 \times s_1$. The memory usage for each edge would scale linearly with the number of rotamers per residue (and not quadratically), and, since the number of edges scales linearly with protein size, the memory requirement for the entire graph would also scale linearly with the number of rotamers. I am excited to explore the implementation of this graph.

9.3.2 Automating Backbone Flexibility in Design

In the very near future, I intend to integrate PROBIK into Rosetta, porting it from its current implementation in Matlab. From there I will work on the automation of flexible backbone segment selection, and backbone conformation filtering. I hope to automate the process so that the designer can specify simply which segments should not ever change, and then let Rosetta iterate between rounds of design with local backbone flexibility and rounds of gradient based minimization.

Additionally, I would like to test the various on-the-fly techniques described in the section above for the instances of design with local backbone flexibility. Since memory use is such a large problem for design with local backbone flexibility, an on-the-fly scheme might prove especially fruitful in permitting the use of a large number of backbone samples, a large number of rotamers per sample, or the use of long backbone segments (10 or more residues).

9.3.3 Integrating Non-Pairwise Decomposable Energy Functions and Local Backbone Flexibility

I would like to incorporate SASApack into a graph that supports local backbone flexibility. SASApack is especially sensitive to slight changes, not simply in avoiding penalties for steric clashes, but also in rewarding good packing. Local backbone flexibility offers a means for incorporating slight changes. The two ideas belong together.

9.3.4 Solvation Models Based on SASA

The most exciting direction for future work is in the extension of the dot coverage count algorithm to model solvation, hopefully replacing the Lazaridis-Karplus solvation model that is currently in Rosetta. To replace the Lazaridis-Karplus solvation model requires the replacement solvation model capture two parts of protein stability: the hydrophobic effect, and the energetic destabilization of buried unsatisfied hydrogen bonding groups. The Lazaridis-Karplus model captures these two aspects in a somewhat rough way: hydrophobic atoms attract other atoms (so they prefer to be buried) and hydrophilic atoms repel other atoms (so they prefer to be near the surface). A buried hydrophilic group can potentially make up for the repulsive terms by forming an intramolecular hydrogen bond.

It would be comparatively easy to incorporate the hydrophobic effect into the energy

function: the stability conferred by the burial of hydrophobic atoms from solvent is a function of the amount of buried surface area. Since the `SASAInteractionGraph` already tracks the solvent-accessible-surface area of the protein, the only additional data it would need is the amount of exposed surface area for the unfolded state; the difference between these areas is the area buried.

The greater challenge in defining a new solvation model is the incorporation of a penalty for the burial of unsatisfied hydrogen bonding groups. There are two parts to this penalty: knowing when a hydrogen bonding group is buried, and knowing when it is satisfied. The `SASAInteractionGraph` stores enough data to identify which hydrogen bonding groups are accessible to solvent and which are buried. The tricky part is the determination of which buried hydrogen bonding groups are satisfied. This determination is tricky on two levels: first, representationally – what data should be stored to represent the presence of hydrogen bonds? – second, chemically – what constitutes a hydrogen bonding group and when is a group satisfied?

The representational challenge is simple enough to state. If we are to separate the storage of hydrogen bond energies for rotamer pairs from the storage of the other energy terms (the Lennard-Jones terms), we have to use extra memory. Hydrogen bonds are rare (the majority of rotamer pairs do not form hydrogen bonds) so some clever sparse matrix representation should be used to store their energies. The challenge is to find the right sparse matrix representation.

The chemical challenge is more difficult to state. The question is, does each donatable hydrogen and each receivable electron pair have to be part of an intramolecular hydrogen bond in order to be satisfied or should hydrogens and electron pairs be grouped together so that only some of them have to form hydrogen bonds to qualify the whole group as satisfied? In a lysine tail, should all three hydrogens form hydrogen bonds in order to prevent a penalty from being applied? If only two of the three hydrogens could form a hydrogen bond, how much of a penalty should the tail incur? What if it can form only one? In an arginine tail, there are five hydrogen atoms that can serve as hydrogen bond donors. Carbonyl oxygen atoms have two lone pair electrons that can participate in hydrogen bonds – these oxygen atoms often form bifurcated hydrogen bonds. Do both lone pairs have to participate in hydrogen bonds in order to be buried without penalty? Probably not: α-helix carbonyl oxygens are often buried when forming only a single hydrogen bond to a backbone amide hydrogen. Hydroxyl groups are even more difficult: they consist of one donatable hydrogen and two lone pairs, needing possibly three hydrogen bond partners each.

9.3.5 Lennard-Jones based on SASA

I would like to see what happens when the attractive term of the Lennard-Jones energy function is removed in favor of an energy function that greatly favors tight packing. The SASAPack term reminds me of the Richardsons' dot-based scoring function used in PROBE. Their scoring function rewards the approach of two atoms when their surfaces are within 0.5 Å of each other, and when there are no other intervening atoms. The scoring function is based on the sum of individual dot scores: each dot finds its distance to the closest atomic surface, and its score is a function of that distance. The SASAPack term is similar to the Richardsons' scoring function in its reliance on dots to represent a tight band of space surrounding an atom – the surface accessible to a probe with a 0.5 Å radius. It is dissimilar in that the score for a dot on this surface does not depend on the exact distance the dot is to the surface of its closest atom – a dot is either covered or uncovered. It would be fairly straightforward to score packing as a linear function of the number of dots covered by intra-molecular contacts for the protein – it would be more difficult to integrate a distance-based metric for dot scores.

Bibliography

Abe, H., Braun, W., Noguti, T., and Go, N. (1984). Rapid calculation of first and second derivatives of conformational energy with respect to dihedral angles for proteins. general recurrent equations. *Computational Chemistry*, 8:239–248.

Allen, B. and Mayo, S. (2006). Dramatic performance enhancements for the faster optimization algorithm. *Journal of Computational Chemistry*, 27:1071–1075.

Ambroggio, X. I. and Kuhlman, B. (2006). Computational design of a single amino acid sequence that can switch between two distinct protein folds. *Journal of the American Chemical Society*, 128:1154–1161.

Arnborg, S., Lagergren, J., and Seese, D. (1991). Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340.

Arnborg, S. and Proskurowski, A. (1986). Characterization and recognition of partial 3-trees. *SIAM Journal of Algorithms and Discrete Methods*, 7:305–314.

Arnborg, S. and Proskurowski, A. (1989). Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23:11–24.

Ashworth, J., Havranek, J., Duarte, C., Sussman, D., Monnat, RJ, J., BL, B. S., and Baker, D. (2006). Computational redesign of endonuclease DNA binding and cleavage specificity. *Nature*, 441:656–659.

Baker, E. and Hubbard, R. (1986). Hydrogen bonding in globular proteins. *Progress in Biophysics and Molecular Biology*, 44:97–179.

Baldwin, E., Hajiseyedjavadi, O., Baase, W., and Matthews, B. (1993). The role of backbone flexibility in the accommodation of variants that replace the core of T4 lysozyme. *Science*, 262:1715–1718.

Beasley, J. and Hecht, M. (1997). Protein design: the choice of *de novo* sequences. *Journal of Biological Chemistry*, 272:2031–2034.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Berne, B., editor (1977). *Statistical Mechanics: part A. Equilibrium Techniques*. Plenum.

Bhat, S. and Purisima, E. O. (2006). Molecular surface generation using a variable-radius solvent probe. *Proteins*, 62:244–261.

Bhat, T., Sasisekharan, V., and Vijayan, M. (1979). An analysis of side-chain conformation in proteins. *International Journal of Peptide and Protein Research*, 13:170–184.

Bodlaender, H. L. (1988). Dynamic programming on graphs with bounded treewidth. In *Proc. 15th Int. Colloq. Automata, Languages and Programming*, pages 105–118, Tampere, Finland. Springer Verlag, Lecture Notes in Computer Science 317.

Bodlaender, H. L. and Kloks, T. (1993). Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21:358–402.

Bower, M., Cohen, F., and Dunbrack, Jr., R. (1997). Prediction of protein side-chain rotamers from a backbone-dependent rotamer library: a new homology modeling tool. *Journal of Molecular Biology*, 267:1268–1282.

Bradley, P., Chivian, D., Meiler, J., Misura, K., Rohl, C., Schief, W., Wedemeyer, W., Schueler-Furman, O., Murphy, P., anc C. Strauss, J. S., and Baker, D. (2003). Rosetta predictions in CASP5: Successes, failures, and prospects for complete automation. *Proteins: Structure Function and Genetics*, 53:457–68.

Bradley, P., K.M.Misura, and Baker, D. (2005). Toward high-resolution *de novo* structure prediction for small proteins. *Science*, 309:1868–71.

Branden, C. and Tooze, J. (1999). *Introduction to protein structure*. Garland Publishing, second edition.

Bromberg, S. and Dill, K. A. (2003). *Molecular Driving Forces: Statistical Thermodynamics in Chemistry & Biology*. Garland, New York.

Brooks, B., Bruccoleri, R., Olafson, B., States, D., Swaminathan, S., and Karplus, M. (1983). CHARMM: A program for macromolecular energy minmimization and dynamics calculations. *Journal of Computational Chemistry*, 4:187–217.

Brooks, 3rd, C. and Karplus, M. (1983). Deformable stochastic boundaries in molecular dynamics. *Journal of chemical physics*, 79:6312–6325.

Brooks, 3rd, C. and Karplus, M. (1989). Solvent effects on protein motion and protein effects on solvent motion. dynamics of the active site region of lysozyme. *Journal of Molecular Biology*, 208:159–181.

Brungler, A., Brooks, 3rd, C., and Karplus, M. (1985). Active site dynamics of ribonuclease. *Proceedings of the National Academy of Sciences, USA*, 82:8458–8462.

Bryant, S. H. and Lawrence, C. E. (1991). The frequency of ion-pair substructures in proteins is quantitatively related to electrostatic potential: a statistical model for nonbonded interactions. *Proteins: Structure Function and Genetics*, 9:108–119.

Butterfoss, G. and Hermans, J. (2003). Boltzmann-type distribution of side-chain conformation in proteins. *Protein Science*, 12:2719–2731.

Bystroff, C. (2003). Masker: improved solvent excluded molecular surface area estimations using Boolean masks. *Protein Engineering*, 15:959–965.

Canutescu, A. A., Shelenkov, A. A., and Dunbrack, Jr., R. (2003). A graph-theory algorithm for rapid protein side-chain prediction. *Protein Science*, 12:2001–2014.

Chou, P. and Fasman, G. (1978). Prediction of the secondary structure of proteins from their amino acid sequence. *Advances in Enzymology*, 47:45–148.

Cohen, F., Sternberg, M., and Taylor, W. (1982). Analysis and prediction of the packing of α-helices against a β-sheets in the tertiary structure of globular proteins. *Journal of Molecular Biology*, 156:821–862.

Connolly, M. (1983). Solvent-accessible surfaces of proteins and nucleic acids. *Science*, 221:709–713.

Cook, S. (1971). The complexity of theorem proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158.

Cornell, W. D., Cieplak, P., Bayly, C. I., Gould, I. R., K.M. Merz, J., Ferguson, D. M. abd Spellmeyer, D. C., Fox, T., Caldwell, J. W., and Kollman, P. A. (1995). A second generation force field for the simulation of proteins, nucleic acids and organic molecules. *Journal of the American Chemical Society*, 117:5179–5197.

Crippen, G. and Havel, T. (1988). *Distance Geometry and Molecular Conformation*. John Wiley and Sons, New York, UK.

Dahiyat, B. and Mayo, S. (1996). Protein design automation. *Protein Science*, 5:895–903.

Dahiyat, B. and Mayo, S. (1997a). Probing the role of packing specificity in protein design. *Proceedings of the National Academy of Sciences, USA*, 94:10172–10177.

Dahiyat, B. I. and Mayo, S. L. (1997b). *De Novo* protein design: fully automated sequence selection. *Science*, 278:82–87.

Dahiyat, B. I., Sarisky, C. A., and Mayo, S. L. (1997). *De Novo* protein design: towards fully automated sequence selection. *Journal of Molecular Biology*, 273:789–796.

Dantas, G., Kuhlman, B., Callender, D., Wong, M., and Baker, D. (2003). A large scale test of computational protein design: folding and stability of nine completely redesigned globular proteins. *Journal of Molecular Biology*, 332:449–460.

De Maeyer, M., Desmet, J., and Lasters, I. (1997). All in one: a highly detailed rotamer library improves both accuracy and speed in the modelling of sidechains by dead-end elimination. *Folding and Design*, 2:53–66.

DeGrado, W., Wasserman, Z., and Lear, J. (1989). Protein design, a minimalist approach. *Science*, 243:622–628.

Denavit, J. and Hartenberg, R. S. (1955). A kinematic notation for lower-pair mechanisms based on matrices. *ASME Journal of Applied Mechanics*, 77:215–221.

Desjarlais, J. and Handle, T. (1995). *De novo* design of hydrophobic cores of proteins. *Protein Science*, 4:2006–2018.

Desmet, J., Maeyer, M. D., Hazes, B., and Lasters, I. (1992). The dead-end elimination theorem and its use in protein side-chain positioning. *Nature*, 356:539–541.

Desmet, J., Spriet, J., and Lasters, I. (2002). Fast and accurate side-chain topology and energy refinement (FASTER) as a new method for protein structure optimization. *Proteins*, 48:31–43.

Dill, K. A. (1990). Dominant forces in protein folding. *Biochemistry*, 29:7133–7155.

Drexler, K. (1981). Molecular engineering: an approach to the development of general capabilities for molecular manipulation. *Proceedings of the National Academy of Sciences, USA*, 78:5275–5278.

Dunbrack, Jr., R. and Cohen, F. (1997). Bayesian statistical analysis of protein side-chain rotamer preferences. *Protein Science*, 6:1661–1681.

Dunbrack, Jr., R. L. (2002). Rotamer libraries in the 21st century. *Curr. Opin. Struct. Biol.*, 12:431–440.

Dunbrack, Jr., R. L. and Karplus, M. (1993). Backbone dependant rotamer library for proteins: application to side chain prediction. *Journal of Molecular Biology*, 230:543–574.

Dwyer, M., Looger, L., and Hellinga, H. (2004). Computational design of a biologically active enzyme. *Science*, 304:1967–1971.

Eisenberg, D. and McLachlan, A. D. (1986). Solvation energy in protein folding and binding. *Nature*, 319:199–203.

Eisenberg, D., Wilcox, W., S.M.Eshita, Pryciak, P., Ho, S., and DeGrado, W. (1986). The design, synthesis and crystallization of an alpha-helical peptide. *Proteins: Structure Function and Genetics*, 1:16–22.

Engh, R. and Huber, R. (1991). Accurate bond and angle parameters for x-ray protein structure refinement. *Acta Crystallographica sect. A*, 47:392–400.

Engle, M., Williams, R., and Erickson, B. (1991). Designed coiled-coil proteins: Synthesis and spectroscopy of two 78-residue alpha-helical dimers. *Biochemistry*, 30:3161–3169.

Eriksson, O., Zhou, Y., and Elofsson, A. (2001). Side chain-positioning as an integer programming problem. In *Workshop on Algorithms in Bioinformatics, WABI2001*, pages 128–141.

Eyal, E. and Halperin, D. (2005). Improved maintenance of molecular surfaces using dynamic graph connectivity. In *Workshop on Algorithms in Bioinformatics, 2005*, pages 401–413, Mallorca, Spain.

Ferrara, P., Apostolakis, J., and Caflisch, A. (2002). Evaluation of a fast implicit solvent model for molecular dynamics. *Proteins: Structure Function and Genetics*, 46:24–33.

Fezoui, Y., Connolly, P., and Osterhout, J. (1997). Solution sturcture of α-t-α, a helical hairpin peptide of *de novo* design. *Protein Science*, 6:1869–1877.

Fleming, P. and Rose, G. (2005). Do all backbone polar groups in proteins form hydrogen bonds? *Protein Science*, 14:1911–1917.

Fraczkiewicz, R. and Braun, W. (1998). Exact and efficient analytical calculation of the accessible surface areas and their gradients for macromolecules. *Journal of Computational Chemistry*, 19:319–333.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston.

Garnier, J., Osguthope, D., and Robson, B. (1978). Analysis of the accuracy and implications of simple methods for predicting the secondary structure of globular proteins. *Journal of Molecular Biology*, 113:97–120.

Goldstein, R. F. (1994). Efficient rotamer elimination applied to protein side-chains and related spin glasses. *Biophysical Journal*, 66:1335–1340.

Gordon, D. and Mayo, S. (1999). Branch-and-terminate: a combinatorial optimization algorithm for protein design. *Structure Fold Des*, 7:1089–98.

Gordon, D. B. and Mayo, S. L. (1998). Radical performance enhancements for combinatorial optimization algorithms based on the dead-end elimination theorem. *Journal of Computational Chemistry*, 19:1505–1514.

Grana, O., Baker, D., MacCallum, R., Meiler, J., Punta, M., Rost, B., Tress, M., and Valencia, A. (2005). Casp6 assessment of contact prediction. *Proteins*, 61 Supplemental:214–224.

Gray, J., Moughon, S., Wang, C., Schueler-Furman, O., Kuhlman, B., Rohl, C., and Baker, D. (2003). Protein–protein docking with simultaneous optimization of rigid-body displacement and side-chain conformations. *Journal of Molecular Biology*, 331:281–299.

Gutte, B., Daumigen, M., and Wittschieber, E. (1979). Design, synthesis and characterisation of a 34-residue polypeptide that interacts with nucleic acids. *Nature*, 281:650–655.

Guvench, O. and Brooks, C. L. r. (2004). Efficient approximate all-atom solvent accessible surface area method parameterized for folded and denatured protein conformations. *Journal of Computational Chemistry*, 25:1005–1014.

Harbury, P., Plecs, J., Tidor, B., Alber, T., and Kim, P. (1998). High-resolution protein design with backbone freedom. *Science*, 282:1462–1467.

Harbury, P., Tidor, B., and Kim, P. (1995). Repacking protein cores with backbone freedom: structure prediction of coiled coils. *Proceedings of the National Academy of Sciences, USA*, 93:8408–8412.

Harbury, P., Zhang, T., Kim, P., and Alber, T. (1993). A switch between two-, three-, and four-stranded coiled coils in GCN4 leucine zipper mutants. *Science*, 262:1401–1407.

Harrison, R. and Gray, J. (2005). Prediction of pKa shifts in proteins using a discrete rotamer search and the Rosetta energy function. In *AIChE 2005 National Meeting*.

Hartsfield, N. and Ringel, G. (1990). *Pearls in Graph Theory: A Comprehensive Introduction*. Academic Press, Boston.

Hecht, M., Richardson, J., Richardson, D., and Ogden, R. (1990). De novo design, expression and characterization of Felix: a four-helix bundle protein of native-like structure. *Science*, 249:884–891.

Hellinga, H. and Richards, F. (1991). Construction of new ligand binding sites in proteins of known structure. I: Computer-aided modeling of sites with pre-defined geometry. *Journal of Molecular Biology*, 222:763–85.

Hellinga, H. and Richards, F. (1994). Optimal sequence selection in proteins of known structure by simulated evolution. *Proceedings of the National Academy of Sciences, USA*, 91:5803–5807.

Hermans, J., Pathiaseril, A., and Anderson, A. (1988). Excess free energies of liquids from molecular dynamics simulations. Applications to water models. *Journal of the American Chemical Society*, 110:5982–5986.

Hill, C., Anderson, D., Wesson, L., DeGrado, W., and Eisenberg, D. (1990). Crystal structure of α-1: implciations for protein design. *Science*, 249:543–546.

Ho, S. and DeGrado, W. (1987). Design of a 4-helix bundle protein: synthesis of peptides which self-associate into a helical protein. *Journal of the American Chemical Society*, 109:6751–6758.

Holm, L. and Sander, C. (1992). Fast and simple Monte Carlo algorithm for side chain optimization in proteins: application to model building by homology. *Proteins*, 14(2):213–23.

Hurley, J., Baase, W., and Matthews, B. (1992). Design and structural analysis of alternative hydrophobic core packing arrangements in bacteriophage T4 lysozyme. *Journal of Molecular Biology*, 224:1143–1159.

James J. Havranek, C. M. D. and Baker, D. (2004). A simple physical model for the prediction and design of proteinDNA interactions. *Journal of Molecular Biology*, 344:59–70.

Janin, J. and Chothia, C. (1990). The structure of protein–protein recognition sites. *Journal of Biological Chemistry*, 265:16027–16030.

Janin, J., Wodak, S., M, M. L., and Maigret, B. (1978). Conformation of amino acid side-chains in proteins. *Journal of Molecular Biology*, 125:357386.

Jaramillo, A. and Wodak, S. J. (2005). Computational protein design is a challenge for implicit solvation models. *Biophysical Journal*, 88:156–171.

Joachimiak, L., Kortemme, T., Stoddard, B., and Baker, D. (2006). Computational design of a new hydrogen bond network and at least a 300-fold specificity switch at a protein-protein interface. *Journal of Molecular Biology*.

Johnson, E., Lazar, G., Desjarlais, J., and Handle, T. (1999). Solution structure and dynamics of a designed hydrophobic core variant of ubiquitin. *Structure*, 7:967–976.

Jones, Jr., M. (1998). *Organic Chemistry*. W W Norton & Co., New Jersey.

Jorgenson, W., Chandrasekhar, J., Madura, J., Impey, R., and Klein, M. (1983). Comparison of simple potential functions for simulating liquid water. *Journal of Chemical Physics*, 79:926–935.

Kamtekar, S., Schiffer, J., Xiong, H., Babik, J., and Hecht, M. (1993). Protein design by binary patterning of polar and nonpolar amino acids. *Science*, 262:1680–1685.

Kauzman, W. (1959). Factors in interpretation of protein denaturation. *Advances in Protein Chemistry*, 14:1–62.

Kingsford, C. L., Chazelle, B., and Singh, M. (2005). Solving and analyzing side-chain positioning problems using linear and integer programming. *Bioinformatics*, 21:1028–1039.

Koehl, P. and Delarue, M. (1994). Application of a self-consistent mean field theory to predict protein side-chains conformation and estimate their conformational entropy. *J Mol Biol*, 239(2):249–275.

Kortemme, T., Morozov, A. V., and Baker, D. (2003). An orientation-dependent hydrogen bonding potential improves prediction of specificity and structure for proteins and protein-protein complexes. *Journal of Molecular Biology*, 326:1239–1259.

Kortemme, T., Ramirez, M., and Serrano, L. (1998). Design of a 20-amino acid three-stranded beta-sheet protein. *Science*, 281:253–256.

Kuhlman, B. and Baker, D. (2000). Native protein sequences are close to optimal for their structures. *Proceedings of the National Academy of Sciences, USA*, 97:10383–8.

Kuhlman, B., Dantas, G., Ireton, G., Varani, G., Stoddard, B., and Baker, D. (2003). Design of a novel globular protein fold with atomic-level accuracy. *Science*, 302:1364–1368.

Kuhlman, B., O'Neill, J. W., Kim, D. E., Zhang, K. Y., and Baker, D. (2002). Accurate computer-based design of a new backbone conformation in the second turn of protein L. *Journal of Molecular Biology*, 315:471–477.

Kullmann, W. (1984). Design, synthesis and binding characteristics of an opiate receptor mimetic peptide. *Journal of Medicinal Chemistry*, 27:106–115.

Kuroda, Y., Nakai, T., and Ohkubo, T. (1994). Solution structure of a *de novo* helical protein by 2D-NMR spectroscopy. *Journal of Molecular Biology*, 236:862–868.

Lasters, I., De Mayer, M., and Desmet, J. (1995). Enhanced dead-end elimination in the search for the global minimum energy conformation of a collection of protein side chains. *Protein Engineering*, 8:815–822.

Lasters, I. and Desmet, J. (1993). The fuzzy-ended elimination theorem: correctly implementing the side chain placement algorithm based on the dead-end elimination theorem. *Protein Engineering*, 6:717–722.

Lau, S., Taneja, A., and Hodges, R. (1984). Synthesis of a model portein of defined secondary and quaternary structure. *Journal of Biological Chemistry*, 259:13253–13261.

Lazaridis, T. and Karplus, M. (1999). Effective energy function for proteins in solution. *Proteins: Structure Function and Genetics*, 35:133–152.

Le Grand, S. M. and Merz, K. M. (1993). Rapid approximation to molecular surface area via the use of boolean logic and look-up tables. *Journal of Computational Chemistry*, 14:349–352.

Lear, J., Wasserman, Z., and DeGrado, W. (1998). Synthetic amphiphilic peptide models for protein ion channels. *Science*, 240:1177–1181.

Leaver-Fay, A., Kuhlman, B., and Snoeyink, J. (2005a). An adaptive dynamic programming algorithm for the side chain placement problem. In *Pacific Symposium on Biocomputing, 2005*, pages 17–28, The Big Island, HI. World Scientific.

Leaver-Fay, A., Kuhlman, B., and Snoeyink, J. (2005b). Rotamer-pair energy calculations using a trie data structure. In *Workshop on Algorithms in Bioinformatics, 2005*, pages 500–511, Mallorca, Spain.

Lee and Richards, F. (1971). The interpretation of protein structures: estimation of static accessibility. *Journal of Molecular Biology*, 55:379–400.

Liang, S. and Grishin, N. V. (2002). Side-chain modeling with an optimized scoring function. *Protein Science*, 11:322–331.

Liang, S. and Grishin, N. V. (2004). Effective scoring function for protein sequence design. *Proteins*, 54:271–281.

Lim, W. and Sauer, R. (1991). The role of internal packing interactions in determining structure and stability of a protein. *Journal of Molecular Biology*, 219:359–376.

Looger, L. L., Dwyer, M. A., Smith, J. J., and Hellinga, H. W. (2003). Computational design of receptor and sensor proteins with novel functions. *Nature*, 423:185–190.

Looger, L. L. and Hellinga, H. W. (2001). Generalized dead-end elimination algorithms make large-scale protein side-chain structure prediction tractable: implications for protein design and structural genomics. *Journal of Molecular Biology*, 307(1):429–45.

Lovejoy, B., Choe, S., Cascio, D., McRorie, D., DeGrado, W., and Eisenberg, D. (1993). Crystal structure of a synthetic triple-stranded α-helical bundle. *Science*, 259:1288–1293.

Lovell, S. C., Davis, I. W., III, W. B. A., de Bakker, P. I. W., Word, J. M., Prisant, M. G., Richardson, J. S., and Richardson., D. C. (2003). Structure validation by c-alpha geometry: phi, psi, and cβ deviation. *Proteins: Structure Function and Genetics*, 50:437–450.

Lovell, S. C., Word, J. M., Richardson, J. S., and Richardson, D. C. (2000). The penultimate rotamer library. *Proteins: Structure Function and Genetics*, 40:389–408.

MacKerell, A. D., Bashford, D., Bellott, M., Dunbrack, Jr., R. L., Eva seck, J. D., Field, M. J., Fischer, S., Gao, J., Guo, H., Ha, S., JosephMcCarthy, D., Kuc nir,

L., Kuczera, K., Lau, F. T. K., Mattos, C., Michnick, S., Ngo, T., Nguyen, D. T., Pro hom, B., Reiher, W. E., Roux, B., Schlenkrich, M., Smith, J. C., Stote, R., Straub, J., W tanabe, M., WiorkiewiczKuczera, J., Yin, D., and Karplus, M. (1998). All-atom empirical potential for molecular modeling and dynamics studies of proteins. *Journal of Physical Chemistry, B*, 102:3586–3617.

Manocha, D. and Canny, J. F. (1994). Efficient inverse kinematics for general R6 manipulators. *IEEE Transactions on Robotics and Automation*, 10:648–657.

McCreight, E. M. (1976). A space-economical suffix tree construction algorithm. *Jrnl. of Algorithms*, 23:262–272.

Merritt and Bacon (1997). Raster3D: Photorealistic molecular graphics. *Meth. Enzymol.*, 277:505–524.

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., and Teller, E. (1953). Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087–1092.

Meyers, S. (2005). *Effective C++, Third edition.* Addison-Wesley, Boston.

Mooers, B., Datta, D., Baase, W., Zollars, E., Mayo, S., and Matthews, B. (2003). Repacking the core of t4 lysozyme by automated design. *Journal of Molecular Biology*, 19:741–756.

Moser, R., Thomas, R., and Gutte, B. (1983). An artificial crystalline DDT-binding polypeptide. *FEBS Letters*, 157:247–251.

Munoz, V. and Serrano, L. (1994). Intrinsic secondary structure propensities of the amino acids, using statistical ϕ-ψ matrices: comparison with experimental studies. *Proteins: Structure Function and Genetics*, 30:301–311.

Murzin, A., Brenner, S. E., Hubbard, T., and Chothia, C. (1995). Scop: a structural classification of proteins database for the investigation of sequences and structures. *Journal of Molecular Biology*, 247:536–540.

Noonan, K., O'Brien, D., and Snoeyink, J. (2004). Probik: Protein backbone motion by inverse kinematics. In *WAFR'04*, Utrecht/Zeist, The Netherlands.

O'Neil, K. and DeGrado, W. (1990). A thermodynamic scale for the helix-forming tendencies of the commonly occurring amino acids. *Science*, 250:246–250.

Osterhout, Jr., J., Handle, T., Na, G., Tourmadje, A., Long, R., Connolly, P., Hoch, J., Johnson, W., Live, D., and DeGrado, W. (1992). Characterization of the structural properties of α_1B, a peptide designed to form a four-helix bundle. *Journal of the American Chemical Society*, 114:331–337.

Pabo, C. (1983). Molecular technology. designing proteins and peptides. *Nature*, 301:200.

Palmer, A., Giacomello, M., Kortemme, T., Hires, S., Lev-Ram, V., Baker, D., and Tsien, R. (2006). Ca2+ indicators based on computationally redesigned calmodulin-peptide pairs. *Chemical Biology*, 13:521–530.

Pardi, A., Hare, D. R., and Wang, C. (1988). Determination of DNA structures by NMR and distance geometry techniques: A computer simulation. *Proceedings of the National Academy of Sciences, USA*, 85:8785–8789.

Pierce, N. and Winfree, E. (2002). Protein design is NP-hard. *Protein Engineering*, 15:779–82.

Pokala, N. and Handel, T. M. (2004). Energy functions for protein design I: Efficient and accurate continuum electrostatics and solvation. *Protein Science*, 13:925–936.

Ponder, J. W. and Richards, F. (1987). Tertiary templates for proteins. Use of packing criteria in the enumeration of allowed sequences for different structural classes. *Journal of Molecular Biology*, 193:775791.

Quinn, T., Tweedy, N., Williams, R., Richardson, J., and Richardson, D. (1994). Beta-doublet: *De novo* design, synthesis, and characterization of a β-sandwich protein. *Proceedings of the National Academy of Sciences, USA*, 41:8747–8751.

Raghavan, M. and Roth, B. (1989). Kinematic analysis of the R6 manipulator. In *International Symposium on Robotics Research*, pages 314–320, Japan.

Raleigh, D., Betz, S., and DeGrado, W. (1995). A *de novo* designed protein mimics the native state of natural proteins. *Journal of the American Chemical Society*, 117:7558–7559.

Regan, L. and DeGrado, W. (1988). Characterization of a helical protein designed from first principles. *Science*, 241:976–978.

Richardson, D. C. and Richardson, J. S. (2001). *Mage, probe, and kinemages.*, volume F. Kluwer Publishers, Dordrecht.

Richardson, J. and Richardson, D. (1988). Amino acid preferences for specific locations at the ends of α-helices. *Science*, 240:1648–1652.

Richardson, J., Richardson, D., and Erickson, B. (1984). *De novo* design and synthesis of a protein. *Biophysical Journal*, 45:25a.

Richardson, J., Richardson, D., Tweedy, N., Gernert, K., Quinn, T., Hecht, M., Erickson, B., Yan, Y., McClain, R., Donlan, M., and Surles, M. (1992). Looking at proteins: Representations, folding, packing, and design. *Biophysical Journal*, 63:1186–1209.

Richardson, J. S. and Richardson, D. (1987). Some design principles: betabellin. In Oxender, D. and Fox, C., editors, *Protein Engineering*, pages 149–163, 340–341. Alan R. Liss, Inc., New York.

Richmond, T. (1984). Solvent accessible surface area and excluded volume in proteins. *Journal of Molecular Biology*, 178:63–89.

Saven, J. G. and Wolynes, P. G. (1997). Statistical mechanics of the combinatorial synthesis and analysis of folding macromolecules. *J. Phys. Chem. B*, 101:8375–8389.

Schueler-Furman, O., Wang, C., Bradley, P., Misura, K., and Baker, D. (2005). Progress in modeling of protein structures and interactions. *Science*, 310:638–642.

Shrake, A. and Rupley, J. (1973). Environment and exposure to solvent of protein atoms. Lysozyme and insulin. *Journal of Molecular Biology*, 79:351–371.

Silverstein, K., Haymet, A., and Dill, K. (1998). A simple model of water and the hydrophobic effect. *Journal of the American Chemical Society*, 120:3166–3175.

Simons, K., Ruczinski, I., Kooperberg, C., Fox, B., Bystroff, C., and D., D. B. (1999a). Improved recognition of native-like protein structures using a combination of sequence-dependent and sequence-independent features of proteins. *Proteins: Structure Function and Genetics*, 34:82–95.

Simons, K. T., Bonneau, R., Ruczinski, I., and Baker, D. (1999b). Ab initio protein structure prediction of CASP III targets using ROSETTA. *Proteins: Structure Function and Genetics*, 37:171–176.

Sood, V. and Baker, D. (2006). Recapitulation and design of protein binding peptide structures and sequences. *Journal of Molecular Biology*, 357:917–927.

Street, A. and Mayo, S. (1998). Pairwise calculation of protein solvent-accessible surface areas. *Folding and Design*, 3:253–258.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14:249–260.

Varshney, A., Brooks, Jr., F., and Wright, W. (1994). Computing smooth molecular surfaces. *IEEE Computer Graphics and Applications*, 14:19–25.

Waksman, G., Shoelson, S., Pant, N., Cowburn, D., and Kuriyan, J. (1993). Binding of a high affinity phosphotyrosyl peptide to the Src SH2 domain: crystal structures of the complexed and peptide free forms. *Cell*, 72:779–790.

Walsh, S., Cheng, H., Bryson, J., Roder, H., and DeGrado, W. (1999). Solution structure and dynamics of a de novo designed three helix bundle. *Proceedings of the National Academy of Sciences, USA*, 96:5486–5491.

Wang, C., Schueler-Furman, O., and Baker, D. (2005). Improved side chain modeling for protein–protein docking. *Protein Science*, 14:1328–1339.

Weiner, P. (1973). Linear pattern matching algorithms. In *Proc. 14th IEEE Annual Symp. on Switching and Automata Theory*, pages 1–11.

Weiner, S. J., Kollman, P. A., Nguyen, D. T., and Case, D. A. (1986). An all atom force field for simulations of proteins and nucleic acids. *Journal of Computational Chemistry*, 7:230–252.

Weiser, J., Shenkin, P. S., and Still, W. C. (1999). Approximate solvent-accessible surface areas from tetrahedrally directed neighbor densities. *Biopolymers*, 50:373–380.

Wilson, C., Mace, J. E., and Agard, D. A. (1991). Computational method for the design of enzymes with altered substrate specificity. *Journal of Molecular Biology*, 220:495–506.

Word, J. M., Jr., R. C. B., Presley, B. K., Lovell, S. C., and Richardson, D. C. (2000). Exploring steric constraints on protein mutations using mage/probe. *Protein Sci*, 9:2251–2259.

Word, J. M., Lovell, S. C., LaBean, T. H., Taylor, H. C., Zalis, M. E., Presley, B. K., Richardson, J. S., and Richardson, D. C. (1999a). Visualizing and quantifying molecular goodness-of-fit: Small-probe contact dots with explicit hydrogen atoms. *Journal of Molecular Biology*, 285:1711–1733.

Word, J. M., Lovell, S. C., Richardson, J. S., and Richardson, D. C. (1999b). Asparagine and glutamine: Using hydrogen atom contacts in the choice of side-chain amide orientation. *Journal of Molecular Biology*, 285(4):1735–1747.

Xu, J. (2005). A tree-decompositon based approach to protein structure prediction. In *RECOMB'05*, pages 423–439.

Xu, J., Jiao, F., and Berger, B. (2005). A tree-decomposition approach to protein structure prediction. In *CSB'05*, pages 247–256.

Xu, Y., Xu, D., Kim, D., Olman, V., Razumovskaya, J., and Jiang, T. (2002). Automated assignment of backbone NMR peaks using constrained bipartite matching. *Computing in Science and Engineering*, 4:50–62.

Yan, Y. and Erickson, B. (1994). Engineering of betabellin 14d: disulfide-induced folding of a beta-sheet protein. *Protein Science*, 3:1069–1073.

Zhang, N., Zeng, C., and Wingreen, N. (2004). Fast accurate evaluation of protein solvent exposure. *Proteins: Structure Function and Genetics*, 57:565–576.