Computer Science Honours
Dissertation

Creation of an Extensible Environment for the Investigation of Genetic Algorithmic Solutions to 3D Cubic Lattice Protein Folding

Kieran O'Neill
School of Mathematics, Statistics and Computer Science
University of KwaZulu Natal

November 29, 2004

Dedicated to Stewart Adcock, Naofumi Yasufuku, and the people who are *Open Source software.*
Oh, and everyone working on the Wikipedia.

Contents

List of Figures

List of Tables

Abstract

An environment for running and exploring a genetic algorithmic solution to the folding of HP lattice proteins on the 3D cubic lattice was developed. The genetic algorithm created did not perform competively with comparable algorithms cited in the literature. However, the environment proved useful for exploring the functioning of genetic algorithms, as applied to this problem. The program is also highly extensible, and can be used in the future as a basis for algorithm development and testing.

Chapter 1

Introduction

Contents

1.1 The need for protein structural prediction software

Proteins are the workhorses of biological systems, responsible for most of the functionality therein. Studies have shown a statistical correlation between proteins' three dimensional structure and their function. Knowledge of the three dimensional structures of proteins enables molecular biologists to predict their functions. Apart from increasing the overall scientific knowledge about biological systems in general, this enables scientists to design medicinal drugs targeting specific proteins, thereby having specific effects and lower side effects.

Unfortunately, the cost of determining three dimensional protein structures is relatively high, for a number of reasons, including the cost of the equipment required (either a nuclear resonance spectrometer, or an x-ray crystallographer, both of which take up an entire room), the computational complexity of resolving a three dimensional image, and the difficulty of crystallising proteins into an appropriate form. By contrast, the cost of determining the amino acid sequences of proteins is much lower, and has been heavily automated. As an illustration, in the SWISS-PROT/TrEMBL protein database, available on-line, there are currently 1070786 protein sequences (*TrEMBL/SWISS-PROT statistics* 2004). By contrast, in the PDB (Research Collaboratory for Structural Bioinformatics Protein Data Bank), there are only 22348 protein structures available (*Protein Data Bank (Research Collaboratory for Structural Bioinformatics) holdings* 2004). As this information implies, computational techniques for predicting three dimensional protein structures from their sequences are extremely useful to scientists, as they provide a cheap alternative to in vitro determination. This problem has received increasing interest over the last decade or so.

1.2 The hydrophylic-polar (HP) model of protein folding for protein structure prediction

One set of solutions to predicting protein structure is to model the process of their folding into their native conformation. The relative complexity of proteins as molecules makes greedy approach modelling of their dynamics computationally impractical. Thus, a number of simplifications have been developed for the modelling of protein structures. A popular simplification is to embed the protein on a lattice, with one amino acid residue at each lattice node. Another is to simplify the interactions between residues to one or two important ones. Of particular note is the partitioning of the amino acids into those which have an affinity for water, and those that are repelled by it. This model then uses the empirical observation that hydrophobic amino acids tend to interact with each other, and turn inward in a protein to avoid the surrounding water medium, which has been found to be the most significant force in globular protein folding. Amino acids not adjacent to one another on the chain, but occupying adjacent nodes on the lattice, are regarded to be interacting. The likeliest structure is taken to be that one which maximises the interactions between hydrophobic amino acids. Even with such simplifications, finding the best structure has been shown to be NP complete (Berger & Leighton 1998), at least for some of the simpler lattices.

Consequently, a number of heuristic techniques have been applied to the problem (Duan & Kollman 2001). These include Tabu Search, Monte Carlo algorithms, branch and bound, genetic algorithms, simulated annealing and combinations of these (Jiang Tianzi & Songde 2003). Algorithms have also been developed which provide a solution not necessarily the best, but guaranteed to be within within a certain bound thereof (Hart & Istrail 1996). This continues to be an area of research.

Genetic algorithms (GA's) have proved popular in the folding of lattice proteins. Genetic algorithms use observations from evolution theory to rapidly find optimal solutions to an optimisation problem without getting stuck in local sub-optima. Adaptation of GA's to protein folding has been carried out in a number of ways, using a range of representations and genetic operators.

1.3 The goal of this project

The goal of this project was to produce an extensible environment for experimenting with genetic algorithmic solutions to three dimensional lattice protein folding. No such environment for the problem exists as yet, and none of the previous programs mentioned in the literature have been made generally available to the public.

Chapter 2

Literature Review

Contents

2.1 Proteins and their structure

Proteins are biopolymers of amino acids. This means that they are chains, of varying length and sequence, of amino acids linked together. There are 20 amino acids that occur naturally in proteins, and the way in which they link together is common. The portion of amino acids that allows them to polymerise is sometimes referred to as the backbone of a protein. Protruding from each amino acid is a short chain, which is different for each amino acid.

It has been determined that there is a strong correlation between the conformation a protein adapts (its structure) and the sequence of amino acids it is composed of. This is due to chemical interactions between the side chains of the amino acids making up proteins, and between these side chains and the solvent in which the protein exists.

Proteins can be divided into two main classes: globular and fibrous. Fibrous proteins tend to be large, fibrous and form support structures in living organisms. Globular proteins, however, tend to be roughly spherical, water soluble, and tend to be active components in living organisms, such as catalysing chemical reactions, assisting more complex operations, such as protein folding, and regulating the expression of genes by binding to DNA. Generally, this functionality depends on the structure of globular proteins being complimentary to that of the molecules on which they act, such that proteins are extremely specific. For instance, gene regulatory proteins bind to highly specific DNA sequences, while enzymes (proteins catalysing chemical reactions) tend to have an active site, at which the reaction they catalyse takes place, and around which are structural features which align the reagents in the reaction (called substrates) so that the reaction is more likely to take place.

2.2 Practical significance and applications of protein structure prediction

2.2.1 Finding drug targets

Many medicinal drugs act by interacting with the active (or other) site on a protein, usually inhibiting the protein's ability to carry out its function. Finding candidate proteins for medicines can be aided significantly when protein structures are known, since this aids in guessing their function. *In silico* prediction of protein structure offers to speed up the determination of protein structures, and hence to facilitate the drug design process. Furthermore, techniques now exist for computational design of molecules that are stereo-specific for a known protein structure. If protein structures can be predicted computationally, it is conceivable that this could be used to sterically design drug molecules without determining protein structures experimentally.

2.2.2 Designing de novo proteins

Another use for the prediction of protein structure from their amino acid sequence is the reverse process: designing proteins by structure, then determining an amino acid sequence that will code for the desired structures. The applications of this process include the design of proteins which catalyse chemical reactions not found in living organisms, which has applications for the chemical industry, since enzymes are much more efficient catalysts than inorganic (eg: platinum-based) ones. This process could also have applications in nanotechnology.

2.3 The problem of protein folding

2.3.1 Levinthal's paradox

It was pointed out, in 1968, that the number of possible configurations a protein could adopt is astronomical (Levinthal 1969). Using an early computer simulation for protein folding, Levinthal postulated that this number exceeded 10^{300}. However, experiments in which proteins were denatured (caused to enter a low-folding, high-energy state), showed that the proteins refolded into their native structures within seconds. Levinthal pointed out that, to fold this quickly, a protein could only have attempted on the order of 10^{8} conformations, a figure significantly less than the problem's entire state space. This problem has been named Levinthal's paradox. The broad solution is that some heuristic is employed in natural protein folding. Levinthal proposed that protein folding was "speeded and guided by the rapid formation of local interactions which then determine the further folding of the peptide". This corresponds to a divide-and-conquer approach.

Levinthal also conjectured that the state attained by natural protein folding was not necessarily the global minimum energy state, but that proteins can fold to a local minimum, which is not necessarily the same for each instance of folding. This has since been refuted, and proteins have been shown to have "funnel shaped" folding kinetics, with depth coordinate representing the interaction free energy of the chain, and the width coordinate representing the chain entropy (Schonbrun & Dill 2003). In other words, proteins fold to a single, or small group of, native states, and, in folding to them, occupy fewer possible states with similar interaction energy and entropy as they near the native state. This is significant to the modelling of protein folding, as it implies that a global minimum should be sought by an algorithm attempting to find the native state of a protein.

The same paper presents arguments supporting Levinthal's divide-and-conquer solution to the rapidity of protein folding, suggesting that protein folding does occur as a series of parallel folding events, rather than as a linear sequence, as had been suggested elsewhere. This suggests that protein folding heuristics employing some form of divide-and-conquer approach might meet with some success in finding the global minimum free energy structure.

2.3.2 The energetics of protein folding

A review important to lattice protein folding, and regarding protein folding in general, was published by Dill in 1990 (Dill 1990). A somewhat updated (but not essentially different) view can be obtained in any recent protein structure textbook, such as (Petsko & Ringe 2004).

Initially, it was believed that forces between amino acids with positively and negatively charged side chains were mainly responsible for protein folding. Although these forces do play a role, it has been since found that it is not dominant. It was also, at one point, believed that hydrogen-bonded structures within proteins (which occur very frequently) were the dominant force. It has been shown, however, that, at least for globular proteins, it is the tendency for some amino acids, with hydrophobic (water-repellant) side chains to turn inward and associate with each other, rather than the surrounding aqueous medium, that is the major driving force in protein's folding into their native structures. Thus, folding simulations emulating only hydrophobic effects still have a tendency to some degree of accuracy, and can give some insight into the process of protein folding itself.

2.4 Lattice proteins

2.4.1 The first proposal

Lattice proteins are a simplification of protein chains down to a self-avoiding walk on a lattice. This was first proposed by Dill (Dill 1985), who used the model to analyse the forces involved in protein folding, and accessibility of conformations to proteins in their globular vs their denatured (amorphous) states. Since then, this model has been used to

2.4.2 Complexity of the problem

As mentioned in the section describing the Levinthal paradox, the number of possible configurations of a protein is astronomical.

To date, even with the simplifications provided by the lattice protein model, the problem counting the number of configurations a lattice protein has yet to be solved exactly (Schuster & Stadler 2000). The problem is equivalent to counting the number of self-avoiding walks of length N. It can be shown that for each lattice, if the number of distinct self-avoiding walks is c_N there exists a constant

$$\mu = \lim_{N \to \infty} \sqrt[N]{c_N} < z - 1 \tag{2.1}$$

However, exact values for μ, even for the simplest lattices, have not yet been found (Schuster & Stadler 2000). Estimates are available, however. For the equation:

$$c_N \approx B N^{\gamma - 1} \mu^N \tag{2.2}$$

which applies to two and three dimensional lattices. For walks on the simple three dimensional cubic lattice, μ, γ and B have been approximated to be 4.6839, 1.161 and 1.39 respectively (Schuster & Stadler 2000). 2.4.2 presents estimations of the complexity of folding a lattice protein on the simple 3D cubic lattice, using the above equation. These give an idea of the rapidity with which the complexity increases with chain length, and suggest the intractability of the problem.

It has, in fact, been proven that folding a HP model lattice protein is NP complete on the two dimensional square (Crescenzi, Goldman, Papadimitriou, Piccolboni & Yannakakis 1998) and three dimensional cubic (Berger & Leighton 1998) lattices, although not for any others (though it can probably be expected that this is so).

These results illustrate the intractability of the problem, and the need for heuristic techniques for solving it.

Table 2.1: Estimates of the number of configurations a lattice protein of length N can adopt on the 3D cubic lattice.

N	C_N
10	10235073
20	5.81611×10^{13}
30	3.15544×10^{20}
40	1.67977×10^{27}
50	8.84972×10^{33}
100	3.35565×10^{67}
200	4.3152×10^{134}

2.4.3 Energy functions

The general form of an energy function for lattice proteins is as follows:

$$E(x) = \sum_{i<j} E(x_i, x_j)C_{ij} \tag{2.3}$$

where x is the whole protein, in a particular fold, C_{ij} is a contact (ie: immediate adjacency) between two amino acid residues, of type x_i and x_j respectively.

The simplest instance of this function, and the one first suggested by Dill (Dill 1985) divides the amino acids between those having hydrophobic side chains, denoted H, and those having polar (hydrophylic) side chains, denoted P, as follows:

$$\mathsf{H} = \{\mathsf{A,C,I,L,M,F,W,Y,V}\} \tag{2.4}$$
$$\mathsf{P} = \{\mathsf{R,N,D,E,Q,G,H,K,P,S,T}\} \tag{2.5}$$

The energy function is then computed from the matrix:

$$\begin{array}{c|cc} & H & P \\ \hline H & -1 & 0 \\ P & 0 & 0 \end{array} \tag{2.6}$$

A problem with this model is that, in many folding experiments carried out thus far, a high degeneracy of optimal energy states have been found (Yue, Fiebig, Thomas, Chan, Shakhnovich & Dill 1995), (Schuster & Stadler 2000). Exhaustive explorations of some small proteins' folding spaces have been carried out, showing extremely high degeneracies of the HP model, compared with degeneracies generally of one or two orders of magnitude lower than those for the HP when an extended alphabet was used for the energy function (Backofen & Will 1998). Degeneracy of optimal states means that such a model does not emulate natural protein folding, which reaches a single, or very small group of lowest-energy states when it folds. It has been proposed that the degeneracy for small alphabet models decreases as the length of the protein chain increases (Yue et al. 1995), but, given the intractability of the problem, this is difficult to investigate.

To better model real protein folding, some alternative alphabets for lattice proteins have been used by some researchers:

Firstly, a modified version of the HP matrix may be used (Bornberg-Bauer 1997), which provides an increased grouping force:

$$\begin{array}{c|cc} & H & P \\ \hline H & -3 & -1 \\ P & -1 & 0 \end{array} \tag{2.7}$$

An extended alphabet may be obtained by accounting for electrostatic interactions between residues with positively and negatively charged side chains (ie: dividing the residues between those with positive and negative side chains, then dividing the remainder between those with hydrophobic and hydrophilic side chains). A naive form (Bornberg-Bauer 1997) of this is the matrix:

$$\begin{array}{c|cccc} & H & P & N & X \\ \hline H & -4 & 0 & 0 & 0 \\ P & 0 & 0 & -1 & 0 \\ N & 0 & -1 & 0 & 0 \\ X & 0 & 0 & 0 & 0 \end{array} \tag{2.8}$$

where X corresponds to hydrophilic residues, while P and N correspond to positive and negatively charged residues respectively.

A better matrix for this alphabet has been presented (Kaffe-Abramovich & Unger 1998), based on experimental observations (Miyazawa & Jernigan 1985):

$$\begin{array}{c|cccc} & H & X & P & N \\ \hline H & -4 & -2 & -1 & -1 \\ X & -2 & -3 & -2 & -2 \\ P & -1 & -2 & 0 & -5 \\ N & -1 & -2 & -5 & 0 \end{array} \tag{2.9}$$

Another energy function (Bornberg-Bauer 1997), adapted from Crippen's empirical potential (Crippen 1991), has been given:

$$\begin{array}{c|cccc} & h & H & Y & X \\ \hline h & -2 & -4 & -1 & 2 \\ H & -4 & -3 & -1 & 0 \\ Y & -1 & -1 & 0 & 2 \\ X & 2 & 0 & 2 & 0 \end{array} \tag{2.10}$$

Use of more complex energy functions has the advantage over use of the HP function, as it increases the modelling of forces contributing to protein structural stability.

2.5 Other protein structure prediction techniques

2.5.1 All atom simulations of protein folding

All atom simulations of protein folding involve modelling the forces involved at the quantum level, on a per atom basis. This approach, while highly sophisticated, and likely to yield results close to experimental observations, is impractical due to the computational time required to model all the quantum events taking place during the folding process (Duan & Kollman 2001). Useful results for the purposes of research have been obtained for small proteins, but this remains impractical as a general method for protein structure prediction.

2.5.2 Protein threading

The technique of protein threading bases itself on known correlations between protein sequence and structure, and on the fact that it is known that structures tend to fall within rough classes, called motifs. Protein threading works by aligning motifs against a protein sequence, in an attempt to find the one which best fits the protein.

2.6 Genetic algorithms in their general form

Genetic algorithms are heuristic algorithms for carrying out state space search, which employ the biological process of natural selection to arrive at an optimal, or near-optimal state rapidly. A widely-cited reference on the topic is a book by David Goldberg (Goldberg 1989).

Genetic algorithms require firstly that a function for assessing the optimality of a state be available. In their most general form, genetic algorithms represent states as strings of bits, with values of either 1 or 0. At any step, a number of states, called the population, are kept in memory. Initially, the population is randomly generated. At each step in the algorithm, pairs of states are selected, via a weighted random selection process (wherein the probability of selecting a given state is proportional to its desirability), and "bred" together. This involves selecting a random point in the state representation, and swapping the values of the two states from that point to the end, to

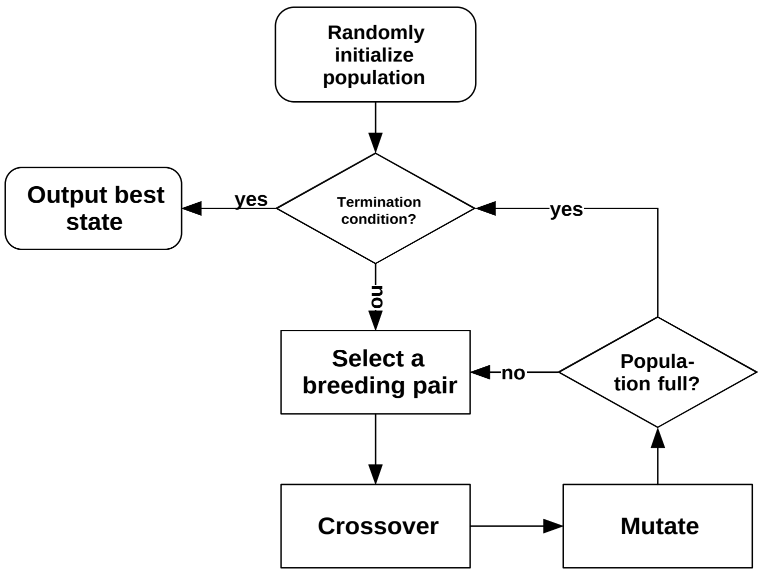

Figure 2.1: A flow diagram illustrating the general form of a genetic algorithm. Blocks contain summaries of the actions of various procedures.

produce two child states. This process is known as crossover. The second step in breeding is to mutate the child states by randomly flipping a random (usually small) number of their bits. Once sufficient children have been produced to create a complete population, the next step is carried out. Once a state has been found that has reached some optimality threshold (or, if the optimum is known, which has attained the optimum), the process is halted. This is summarised in 2.6.

By weighting the selection of parents at each generation by their closeness to solving the problem, genetic algorithms ensure that they gradually approach the solution. This resembles the natural selection process at work in biological evolution. Crossing over of states, along with mutation, both model events taking place in biological organisms's DNA at the molecular level, and serve to help prevent genetic algorithms from converging to a local optimum. A standard technique is to reduce the mutation rate with time, as the algorithm approaches the optimum, so that the space around the optimum can be explored finely. The rate of mutation is a variable which needs to be experimented with, however, since too high a rate can prevent convergence, but too low a rate can cause the algorithm to converge to a suboptimal point. Similarly, when the mutation rate is decreased with time, if it is decreased too rapidly, a suboptimal point may be converged to.

2.7 Genetic algorithms applied to protein folding on lattices

2.7.1 The algorithm presented by Unger and Moult

The first genetic algorithm solution to the problem of lattice protein folding, presented by Ron Unger and John Moult (Unger & Moult 1993), differed in a number of ways from the basic genetic algorithm:

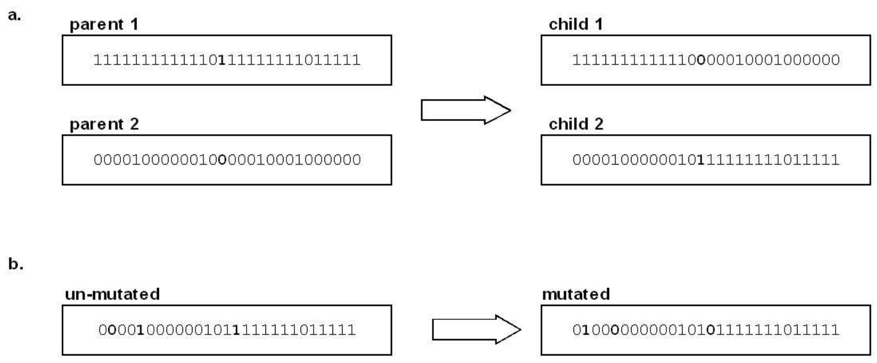

Figure 2.2: An illustration of a sample crossover (a) and mutation (b). Crossover and mutation points are highlighted in bold.

- States are represented by the coordinates of residues on the lattice, rather than by bit strings.
- A starting population of straight chains was used (rather than a random population of starting states).
- Mutation was carried out prior to crossover, as a rigid change in rotation at one point of the chain.
- Mutation was carried out as several iterations of a step in a Monte Carlo algorithm developed by the same authors, wherein the chain was randomly rotated about a random point, and the rotation rejected if it violated self-avoidance, or at random if it produced a structure with higher free energy than its parents' average.
- Crossover involved translation of the subsequences at each cut point to the cut point on the other parent, and testing of different angles of rotation around that point until a valid self-avoiding configuration was obtained, or rejecting the crossover if one was unobtainable.
- Following a successful crossover, a Monte Carlo acceptance procedure was again used, with a higher chance of rejecting higher-energy conformations than that for mutation.

For computing the energy function, Unger and Moult used the simple HP matrix, counting only non-local (non adjacent on the protein chain) interactions.

Effectively, the algorithm was a hybrid Monte Carlo/genetic algorithm. It was also only implemented on the 2D matrix.

2.7.2 The algorithm presented by Patton et al

In response to this, a better algorithm was designed, that followed the original genetic algorithm format more closely (Patton et al. 1995). Their solution was formulated so that it could be implemented in a standard genetic algorithm package (GALOPPS) developed by the research group Patton was a member of. Patton pointed out that shortest paths to the lowest-energy self avoiding walk sometimes passed through illegal (non-self-avoiding) states, and that, since Unger's algorithm avoided these states completely, it would sometimes avoid shortest paths to the optimum. Thus,

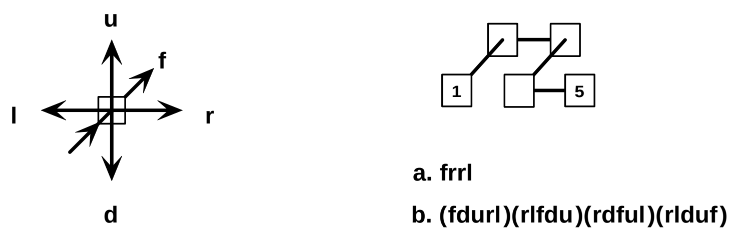

Figure 2.3: An illustration of the relative encoding of lattice chains, as implemented by Patton et al (Patton et al. 1995). On the left are the five possible directions a lattice chain can go from a given point. On the right is an encoding of a short, five-residue chain, beginning at residue 1, and ending at residue 5. The basic encoding is labelled a, while an example of a priority list encoding is labelled b, with each move's list in brackets. Note that at the last move, the first choice in the priority list, a right move, would produce a collision, so the next choice, a left move, is selected.

in his group's algorithm, illegal states were allowed, but penalised in the computation of the energy function.

Furthermore, they used two forms of representation for the problem, both differing from Unger's: One in which the next move was encoded relative to the current direction, as illustrated in 2.7.1, and one in which moves were encoded as preference lists (ordered lists of the five possible moves). In the second representation, if the first move in the list produced a collision, the next was tried, down to the last, which was accepted even if it collided. This provided an additional built-in avoidance of collisions, since the rate of collisions in the simple relative-move encoding was found to tie the algorithm up somewhat (Patton et al. 1995). It should be noted that this representation also represents an implicit chain-growth heuristic.

The energy function was computed by plotting the course encoded by a genotypic movement list on a 3d lattice, then testing each occupied cell for the presence of a hydrophobic residue, and summing the number of contacts with hydrophobic residues in adjacent cells. Each contact was given a positive value, while a negative value was added to the total if a collision was detected at the cell being tested. The function did not discount neighbours on the chain, in contrast to Unger and Moult, probably as a minor optimisation.

This algorithm substantially outperformed Unger and Moult's, showing an average 29.22 fold speedup for 27 length sequences, and an average 8.86 fold speedup for 64 length sequences. The algorithm also always found solutions of energy at least equal to those of Unger and Moult's, and occasionally better. Furthermore, Patton et al developed their algorithm to work on both 2D and 3D matrices. Curiously, despite this obvious improvement, much of the more recent research in Genetic algorithms applied to lattice protein folding cites Unger and Moult's algorithm, not Patton et al's.

2.7.3 More recent implementations

Another hybrid Monte Carlo/GA algorithm for solving 2D HP lattice proteins was presented by Liang and Wong (Liang & Wong 2001). This was essentially a more sophisticated version of Unger and Moult's algorithm, and performed slightly better than it, finding lower energy structures in less time in most cases. It was not tested against Patton et al's algorithm, and may have performed significantly worse by comparison.

A genetic algorithm employing a tabu search in its selection process was developed by Jiang et al (Jiang Tianzi & Songde 2003), also based on Unger and Moult's algorithm, and evaluated against that, and Liang and Wong's. This algorithm showed comparable performance to Liang and Wong's algorithm, and is interesting simply in the novelty of its approach.

Both of the above introduced two new mutation operators: crankshaft moves, and three-bead flips. These moves add to the variety of mutation, but can only be applied to specific subsequences.

An genetic algorithm that used known or predicted protein secondary structure as part of its fitness criterion was presented by Dandekar (Dandekar 1997). This algorithm experimented with additional strategies, such as ensuring, at every tenth generation, that all children are different from all parents. The algorithm predicted the structure of a simple protein with a RMSD error of around 6Å.

Chapter 3

System Design

This chapter describes the design of the system and its interface. It details the technologies, algorithms and data structures used.

Contents

3.1 The Design in Overview

3.1.1 Broad structure of the program

The program consists of a windowed interface, with an OpenGL panel included. Widgets are provided for the entry of a sequence for structure prediction, the configuration of parameters for the genetic algorithm, and running of the genetic algorithm. When the button to run the genetic algorithm is pressed, control passes to the genetic algorithm main loop until it terminates. The OpenGL panel displays intermediate results from the running of the genetic algorithm, and displays the final result in a manipulable form. Additional output is written to the terminal.

3.1.2 Technologies Used

Operating system

The program was developed to run in Linux. Linux has the advantage over Windows that it is free, runs on many platforms besides the PC, and numerous development tools are available for free which run in it. Xemacs was used as the development environment for the program (and for this dissertation) (*Xemacs, a free open source text editor and application development environment* 2004).

Programming language

The program was developed in c, and compiled using the Gnu c compiler (gcc) (*The Gnu c compiler, a free, open source, multiplatform c compiler* 2004). C lacks support for object oriented programming, which reduces the level of abstraction in c programs, and hence the ease of programming. However, c executables are shorter than equivalent OOP executables, and run faster. Faster runtime is an advantage in computationally intensive applications, such as protein structure prediction. A makefile was used for project management.

Genetic algorithm utility library (GAUL)

GAUL was used as a framework for the genetic algorithmic component of the program. GAUL is an open source c library, with functions for setting up, running and evaluating genetic algorithms (Adcock 2004*a*). GAUL uses callback functions for all aspects of the genetic algorithms it sets up. Functions implementing most common genetic algorithm operations are provided. Custom functions may also be written. A custom scoring function was created for this project. A per-generation callback function was also created, to update the OpenGL display, and print intermediate results to the terminal.

The GIMP toolkit (GTK+)

GTK+ was used for the creation of the graphical user interface. The GIMP toolkit is a free windowing toolkit developed for use with X Windows in Linux (*The GIMP toolkit, a free open source windowing API* 2004). It was originally developed for the creation of the Gnu Image Manipulation Package (GIMP). Cross-platform compatibility (to Microsoft Windows) is in the beta testing stage.

An OpenGL Extension to GTK (GtkGLExt)

GtkGLExt was used to create the OpenGL panel in which three dimensional images of solutions were displayed. GtkGLExt is a free extension to GTK+, allowing for an OpenGL panel to be drawn as a widget, to which OpenGL commands can be passed, and which intercepts keyboard and mouse input (Yasufuku 2004).

3.2 The genetic algorithm component

3.2.1 Format of chromosomes

Bitstrings

Chromosomes were formatted as bitstrings, as per Patton's group's solution (Patton et al. 1995). GAUL has built-in operators for bitstring chromosome initialisation, mutation and crossover, in addition to utility functions for bit extraction and manipulation.

Encoding of moves

Moves were encoded relatively, as explained in section 2.7.2 of the Literature Review (page 11). Only the simpler encoding was implemented, in which 5 moves were encoded, requiring $\lceil log_2 5 \rceil = 3$ bits per move. This does mean that 3 possible codewords don't have corresponding moves (since 3 bits encode $2^3 = 8$ possible codewords). The consequences of this, and how it was dealt with, are discussed in the next section. Chromosomes are thus created to be of length $3\times$ no of beads.

3.2.2 Genetic operators

Initialisation

Patton's group initialised their chromosomes to represent a straight line. In the encoding scheme, 0 (of 8) was used to represent a forward move. Thus, a bitstring of 0's would encode a sequence of 3-bit 0's, representing forward moves, and hence a straight line. The GAUL reference manual mentions a function for initialising bitstrings to a string of 0's, (ga_seed_bitstring_zero) (Adcock 2004*b*). However, this had not been included in the version of GAUL available at the time of development, so this function was implemented and included in gafoldcore.h.

Selection

Standard roulette-wheel selection was used. The GAUL functions for this are ga_select_one_roulette and ga_select_two_roulette.

Mutation

Multiple point mutation was used. The GAUL function for this is ga_mutate_bitstring_multipoint, which mutates 20% of the bits in a chromosome selected for mutation. The rate of mutation can be set by the user before a run of the algorithm.

Crossover

Two-point crossover was used. The GAUL function for this is ga_crossover_bitstring_doublepoints. The rate of crossover can be set by the user before a run of the algorithm.

3.2.3 Translating from bitstrings to 3D

Translation from bitstring to absolute three dimensional coordinates is a two-step process, as decoding to relative coordinates is necessary as an intermediate step.

From bitstring to relative coordinates

Decoding from bitstring to relative coordinates is relatively straightforward, as the number represented by each 3 bit code is taken to represent the number (from 0 to 4) of a direction. The number is taken modulo 5, so that the numbers 5 to 7 represent directions 0 to 2. This does mean some directions are favoured, but is necessary, as the chromosomes are mutated bitwise, potentially producing numbers greater than 4.

Lookup tables for determining absolute coordinates

Decoding from relative to absolute coordinates is slightly more complex, and makes use of two lookup tables. The first is a $six \times 5$ table of relative directions against current facing direction, giving the absolute direction produced by combinations of these. For instance, if the current direction were positively along the y axis (one number of six possible directions), and the move were directly ahead

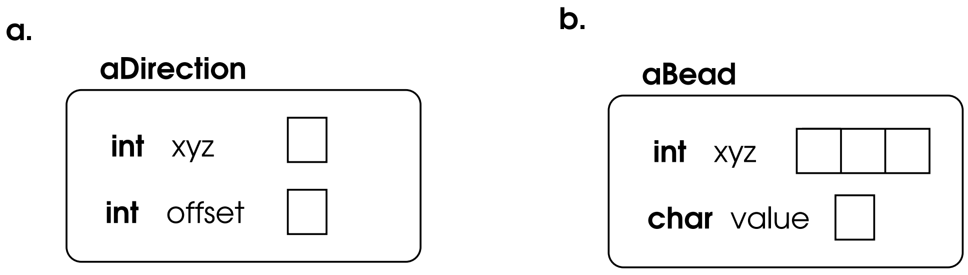

Figure 3.1: ADT diagram of the data structures for translating to and storing 3D solutions. The structure aDirection (a) contains two integers, one for representing the axis of a direction, and one to represent the direction along that axis (positive or negative). The structure aBead (b) contains an array of three integers, for storing the x, y and z coordinates of a bead. The value of the bead (H or P) can be stored in the char value.

(one of five possible relative moves), the resulting absolute direction would be positively along the y axis. Table 3.2.3 shows this.

The second lookup table consists of 6 cells, of type aDirection (shown in figure 3.2.3). Each cell contains a direction (x, y or z), in the range 0-2, and an offset, either +1 or -1. The direction indicates the cell in a 3-cell xyz coordinate array, as found in aBead, also shown in figure 3.2.3, and the offset indicates the number to be added to that cell to achieve the corresponding change in absolute position. The directions stored in the first lookup table are represented as numbers from 0 to 5, so that they are indices into the second lookup table. Table 3.2.3 shows this.

The actual decoding process

A three dimensional candidate solution is built from a bitstring by extracting 3 bits at a time, using their numerical value (modulo 5) as an index into the first lookup table (with the previous direction as the other index), then using the result of that as an index into the second lookup table, to obtain the index in the array containing the current coordinates, and the offset to add to it. By starting facing down the x axis, at set coordinates, a complete solution is built up, and stored in the global variable containing the current three dimensional solution, which is an array of aBeads (see figure 3.2.3).

3.2.4 Scoring: the use of an octree for rapid interaction detection

The fitness criterion for this genetic algorithm is the number of adjacencies between hydrophobic (H) beads in candidate three dimensional folds of the protein. This presents complexity issues in its determination.

Table 3.1: The first lookup table used by the direction decoding algorithm, for obtaining an absolute direction from a past absolute direction and a new relative direction.

From Direction ↓	**0**	**1**	**2**	**3**	**4**
0	0	5	4	3	2
1	1	4	5	2	3
2	2	4	5	0	1
3	3	5	4	1	0
4	4	0	1	2	3
5	5	1	0	3	2

Table 3.2: The second lookup table used by the direction decoding algorithm, for obtaining an index into a 3-cell xyz array, and a number to add to it.

Abs direction	**xyz**	**Offset**
0	0	1
1	0	-1
2	1	1
3	1	-1
4	2	1
5	2	-1

Problems with using an array to represent lattice space

Scoring, in theory, is a process of checking, for each bead, whether any adjacent beads are hydrophobic. This would mean checking the six possible adjacency positions in the lattice around the bead being examined. However, for this lookup to be an array dereferencing operation, an array of dimensions $(2n)^3$ is required, where n is the number of beads for the protein being worked on. This is because, at worst, the protein can be a straight line from the centre of the lattice space, n lattice ticks along any of the axes, in either direction. Such an array, for a protein of 100 beads, would contain 8×10^6 cells, or 8MB of storage, at one byte per cell. It was decided that this was impractical, as 8MB of consecutive storage space may not always be available to the program in memory.

A possible solution: search the current solution array

One possible solution is, for every bead, to search through the entire array of bead coordinates, checking each bead for adjacency to the current bead. This has the disadvantage that it is an $O(n^2)$ operation, where direct array lookup would be $O(n)$.

A compromise between space and time complexity: the octree

An octree is a variant of a binary search tree, for three dimensional lattice space. Each node in an octree represents a cubic region of space, and can be split in half along all three axes to form eight possible children. Since only the nodes necessary to store the elements contained in the octree are created, in the case of a sparsely occupied lattice space, such as the space in HP protein folding, an octree is extremely space efficient. Octrees also have $O(log_2 n)$ insertion, deletion and search time complexity. Al Globus provides a discussion of octree optimisation (Globus 1991).

Actual time complexity

For each candidate solution, the entire solution's coordinates are stored in a new octree. For each solution, the six adjacent positions in the lattice are searched for in the octree, and scored accordingly. The memory used by the octree must then be deleted. All three of these processes are $O(log_2 n)$, and are carried out n times, making each, and the overall process, $O(nlog_2 n)$.

3.3 The windowed interface

3.3.1 Overall layout

Details of the overall layout are shown in figure 3.3. The OpenGL panel takes up most of the window. The GA parameter input widgets are grouped together at the right, in a frame. The sequence entry

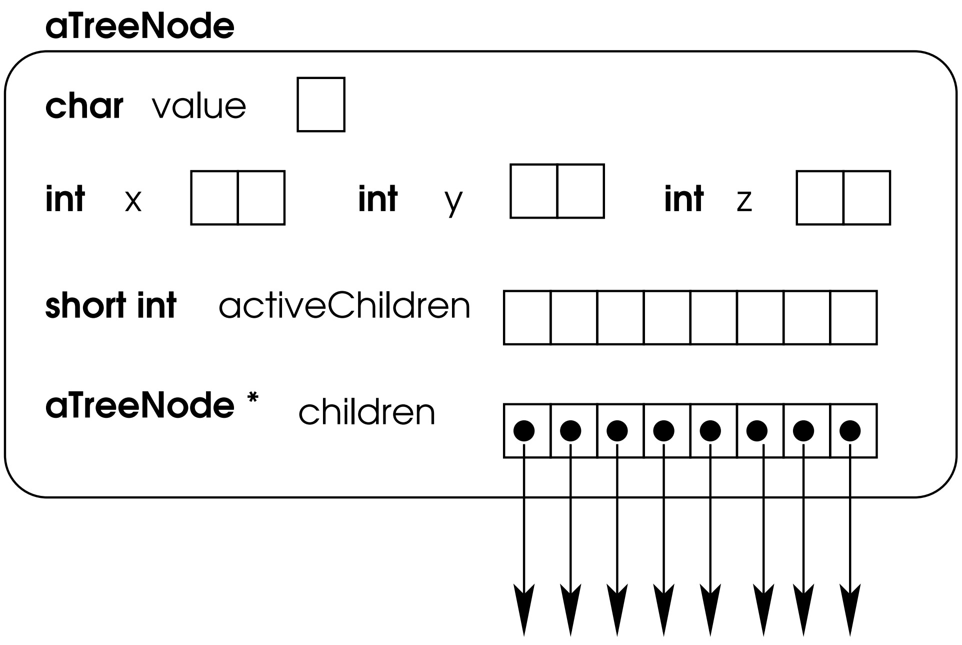

Figure 3.2: ADT diagram of the data structures for storing octree nodes. A char, value, provides storage for the value of the bead located at the node. Three integer arrays, x, y and z, provide storage for the x, y and z bounds of the node. The array, activeChildren provides storage for flags to indicate whether a given child node exists. The array children provides storage for pointers to child nodes.

boxes, being longer, occupy the lower stretch of the window. The sequence load button is placed next to the amino acid sequence input box, by association. The buttons to quit, and to run the algorithm, being the most important functions, are large and obvious, and placed on their own, at the lower right. Most widgets have tooltips and labels, so that their use is as self-explanatory as possible.

3.3.2 Spin buttons for GA parameter tuning

Four spin buttons were placed in a table-laid-out frame, to allow the user to tune the genetic algorithm before running it. They allow the user to change the rates of mutation and crossover, as well as the size of the population to work with, and the number of generations to run the algorithm over. Spin buttons were chosen, as they allow both mouse and keyboard input of values, and values are clearly represented to the user. Reasonable default values have been assigned to the spin buttons, and the buttons limit input to a reasonable range (and disallow negative values), preventing users from causing program crashes by entering invalid values. Values outside the range limits for the spin buttons are truncated to the nearest limit of the range.

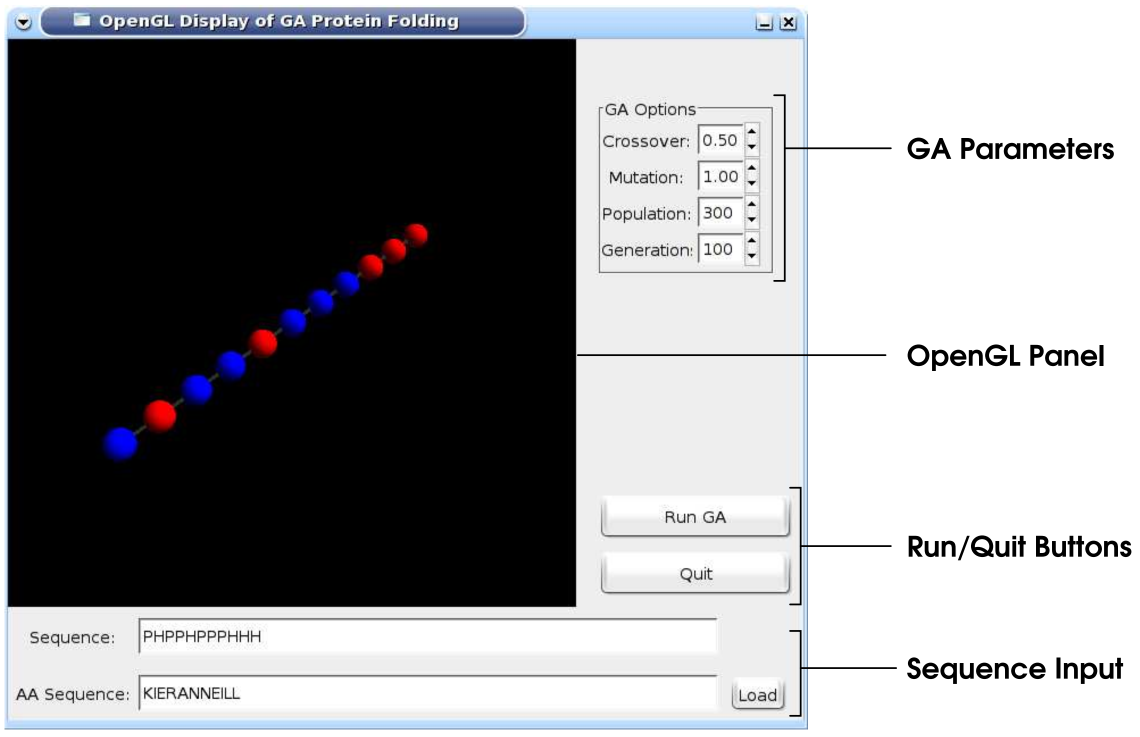

Figure 3.3: An annotated screenshot of the windowed interface of the program. The main widget groups are indicated.

3.3.3 Sequence entry boxes

Two sequence entry boxes are provided to allow the user to enter custom sequences, both in the scoring alphabet (ie: H and P), and in the standard amino acid alphabet. Input to the HP alphabet box causes the amino acid box to fill with ? characters, since the HP alphabet is a simplification of the amino acid alphabet, and back translation is therefore impossible. If a character is entered into the HP alphabet box which does not belong to the scoring alphabet, an error message is written to the terminal. Input to the amino acid sequence box causes the program to translate this sequence into the scoring alphabet, and to update the HP input box accordingly. Pressing of the load button invokes a file load dialogue window, by which the user may select a FastA format amino acid sequence file, from which to load an amino acid sequence. On a successful file load, the sequence loaded from the file is filled into the amino acid input box, and its translation filled into the scoring alphabet input box. The details of the translation are explained in the following paragraph. Both entry boxes automatically convert sequences to upper case, so that both upper an lower case input is acceptable.

3.3.4 Protein sequence translation and FastA file loading

Protein sequence translation is accomplished by looking the amino acid characters up in a translation table, contained within an anEnergyMatrix data structure. This data structure serves as storage both for a translation table, and for a scoring matrix for a given scoring alphabet (hence the name). The ADT for anEnergyMatrix is shown in figure 3.3.4. If a non-standard amino acid character is

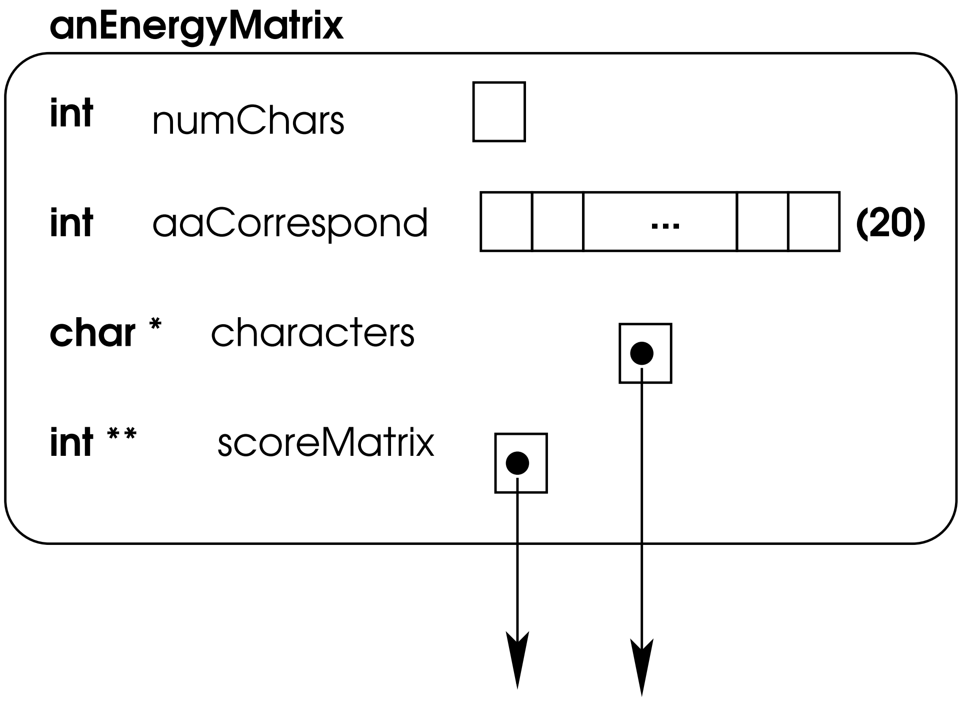

Figure 3.4: ADT diagram of the data structures for storing scoring matrices and alphabets. The integer, numChars, provides storage for the number of different characters in the scoring alphabet. The pointer, characters, provides storage for a pointer to an array containing the scoring alphabet. The array, aaCorrespond, provides storage for the indices into the scoring alphabet corresponding to the 20 amino acids. The double pointer, scoreMatrix, provides storage for a pointer to a two dimensional score matrix for interactions between scoring characters.

encountered, an error message is written to the terminal, and the sequence entry boxes are not updated.

The FastA file loader loads the first amino acid sequence from a FastA file containing unaligned amino acid sequence data. It does not intelligently detect whether the file stores a nucleic acid sequence, or an aligned amino acid sequence, but the protein sequence translator picks up if one of these files are loaded, as characters therein will be recognised as being outside the standard amino acid alphabet.

3.4 The OpenGL interface

The OpenGL interface uses spheres to represent amino acids (or beads), and lines to represent adjacency on the amino acid chain. The colours of the spheres represent their hydrophobicity (red beads) or hydrophilicity (blue beads). Rotation and scaling of solutions is allowed, by pressing and holding mouse buttons over the panel. A black background was chosen for contrast, and the OpenGL smooth shading lighting model is used to aid in visualising the depth of the beads in a candidate solution.

3.4.1 Drawing of solutions

When the program runs, display lists for a red and a blue sphere are created. During running, solutions are drawn by parsing through the global variable containing the current three dimensional solution (an array of aBeads, shown in figure 3.2.3), translating to the coordinates at each position in that array, and calling the display list for the corresponding coloured sphere. Prior to this, the display is translated to the coordinates of the middle bead in the sequence, as a crude means of centring the display. The display is also rotated 45 degrees about each axis, so that, in its default rotation, the user can see as much of the solution as possible. If the solution were aligned along any of the axes, coplanar beads would obscure one another.

3.4.2 Zooming and rotation

These operations are carried out by use of the glscale and glrotate functions respectively. In both cases, the position of the mouse cursor on initial click, and on each subsequence call to the mouse motion function, is stored in a global variable. The change in mouse position is then calculated, and used to determine the degree of scaling or rotation to apply to the scene. Rotation is looped around when it becomes greater than 360 degrees, or less than -360 degrees. Scaling is truncated at a maximum and minimum value, predetermined by global constants.

3.5 Terminal output

Terminal output is used to present large quantities of data to the user, and for debugging purposes. As the genetic algorithm is running, every fifth generation, the score of the best solution is written to the terminal, as it is displayed in the OpenGL panel. The final solution's coordinates, score, and Unger and Moult converted score are displayed when the algorithm completes its execution. A number of debugging `printf` calls can also be found within the code, commented out, which could be uncommented to provide more information to the user (though some are contained in portions of code which are called many thousands of times per run, and may overwhelm the user).

3.6 Source files and functions contained therein

3.6.1 Functions contained in pfoldgl.c

This is the main program source file, containing only the `main` function. The interface definition source files, `pfinterface.h` and `glinterface.h` are included from here.

main

The main function sets up the window for the program, and calls the functions instantiating the interface widgets and linking them to callback functions.

3.6.2 Functions contained in pfinterface.h

This is the main interface definition file. It contains functions which set up and position all the GTK widgets in the interface, with the exception of the OpenGL panel. It also contains the callback functions for user input of genetic algorithm parameters and protein sequence. The main genetic algorithm source file, `gafoldcore.h`, and the source file for handling sequence input and conversion, `seqfile.h`, are included from here.

setSequence This function sets the string storing the current, scoring matrix (ie: HP) form of the protein sequence to work with. The counter of the sequence length is also set. This function is used as a callback for the HP sequence entry box. Some error-checking, and automatic case-conversion are built in.

setSequenceFromAA This function sets the current HP string, taking input in the form of an amino acid sequence. The amino acid sequence is converted using the `convertSequence` function in `seqfile.h`. `setSequenceFromAA` is used as a callback to the amino acid sequence entry box, and uses `setSequence`.

setSequenceFromFile

This function is used to load a sequence from a FastA file. It calls `loadDialogue` to instantiate a file load dialogue box, and passes the file name from the dialogue box to the `loadSequence` function in `seqFile.h`. `setSequenceFromAA` is called to set the HP sequence string from the file input. `setSequenceFromFile` is used as a callback function to the button next to the amino acid sequence entry box.

loadDialogue

This function instantiates a file load dialogue window to get a FastA filename from the user to load an amino acid sequence from.

setPop

This function sets the population size global variable, used by the genetic algorithm. It is used as a callback to the population count spin button.

setMut

This function sets the mutation rate global variable, used by the genetic algorithm. It is used as a callback to the mutation rate spin button.

setCrs

This function sets the crossover rate global variable, used by the genetic algorithm. It is used as a callback to the crossover rate spin button.

setGen

This function sets the generate count global variable, used by the genetic algorithm. It is used as a callback to the generation count spin button.

drawButtons

This function draws the buttons for running the genetic algorithm, and quitting the program. The run GA button is linked to the callback function `runGA` in `gafoldcore.h`.

drawSeqInput

This function draws the HP and amino acid sequence entry boxes, and the button for loading a sequence from a FastA file. These boxes are linked to their respective callback functions mentioned above.

drawSliders

This function draws the spin buttons used to set the genetic algorithm parameters (mutation rate, crossover rate, population size and number of generations to run for). These spin buttons are linked to their respective callback functions, mentioned above.

3.6.3 Functions contained in glinterface.h

This is the file for initialising and working with the OpenGL panel. It contains functions for creating the OpenGL panel widget, and for setting up the OpenGL environment on that widget's creation. Functions for updating the display to reflect the current 3d solution, and to accept mouse input for zooming and rotation, are also contained here. Global variables from `gafoldcore.h` are `externed` from here.

realize

This function sets up the OpenGL scene for the OpenGL panel. Lighting and the camera point are set up. Two coloured `gluSpheres` are created, and stored in display lists. This function is called as the OpenGL panel appears.

configure_event

This function sets up the OpenGL viewport for the OpenGL panel. This is called whenever the widget is resized, and could be used to dynamically resize the OpenGL scene itself to suit the window size, if the window were resized. As the program stands, this function serves to set up a viewport when the panel is created.

expose_event

This function draws the current best solution from the current genetic algorithm population, in a three dimensional form. The coordinates are extracted from a global variable set up in `gafoldcore.h`. The drawing is centred on the middle bead in the sequence, as a rudimentary form of centring. This function is called whenever the OpenGL panel is clicked on, and is called by the generational callback function, `generationHook` in the file `gafoldcore.h`.

button_press_event

This function sets the mouse motion tracking variables used by the rotation and zoom function, `motion_notify_event`. It is called when a mouse button is pressed over the OpenGL panel.

motion_notify_event

This function carries out rotation or zoom when a mouse button is held over the OpenGL panel. The scene is rotated if the left mouse button is held, and zoomed if button 2 (middle button on a 3 button mouse, right button on a two-button mouse) is held. A redraw of the window is then forced.

drawGL

This function starts OpenGL, links OpenGL to the OpenGL panel, and draws the panel. All the other functions in `glinterface.h` are linked to the OpenGL panel as callbacks.

3.6.4 Functions contained in gafoldcore.h

This file contains the core functions for running the genetic algorithm component of the program. This includes scoring functions, functions to initialise and clean up lookup tables, a per-generation callback function to output intermediate results, and the function to set up and start the actual genetic algorithm run. The file for handling octrees, `octree.h`, is included from here. A gtk signal is also sent to the OpenGL redraw function.

ga_seed_bitstring_zero

This function sets every bit in a bitstring chromosome to 0. It was written because it was missing from the current GAUL implementation. It is used as a callback from RunGA, when the genetic algorithm is run, to initialise chromosomes.

directionsToCoords

This function translates a bitstring chromosome to absolute three dimensional chromosomes, updating the current three dimensional solution global variable as it does so. It is used by the scoring functions to assist scoring, and by the generation callback function to draw the current best solution.

scoreOneInteraction

This function is used by the deprecated score function, `foldScore`, to return the score between two beads on the protein chain. It is called by `foldScore`, extensively.

foldScore

This function scores a chromosome. The chromosome is translated to a three dimensional candidate solution by verb1directionsToCoords1. Each bead in this candidate solution is then scored against every other bead, by calls to `scoreOneInteraction`. Collisions are also penalised here. This function is used as a callback by `runGA`, in the genetic algorithm, to assess fitness of chromosomes.

foldScoreOctree

This function scores a chromosome, using an octree. The chromosome is translated to a three dimensional candidate solution by verb1directionsToCoords1. This is stored in a new octree. Each bead is then assessed against the six possible surrounding scoring positions, using the octree for fast lookup. Collisions are also detected by searching the octree, and penalised. The octree is freed, and a score returned. This function is used as a callback by `runGA`, in the genetic algorithm, to assess fitness of chromosomes.

scoreUngerMoult

This function takes a Patton style score for a candidate solution, and converts it to a score in the form Unger and Moult used. It is called by `runGA`, to display an Unger and Moult form score of the best solution found.

generationHook

This function updates the current 3d solution to be the best solution from the population, every fifth generation, and refreshes the OpenGL display to reflect this. It is called every generation, as a callback from `runGA`, but only takes action every fifth generation.

runGA

This function sets up the population and parameters for the genetic algorithm, runs the genetic algorithm, and prints output to the terminal. It uses callbacks to `foldScoreOctree` and `generationHook`, and calls `scoreUngerMoul` and `directionsToCoords` in printing output for the final solution.

performInit

This function sets up lookup tables for direction decoding, allocating memory for them. It is called by the `main` function.

cleanUp

This function frees the memory allocated by `performInit`. It is called by the `main` function.

3.6.5 Functions contained in octree.h

This file contains functions for creating, displaying and manipulating octrees, and the definition of the octree data structure. The functions in this file are used by `foldScoreOctree` in `gafoldcore.h`.

nextPowerOf2

This function returns the next highest power of 2 above the number passed to it. It is used by `makeNewTree`.

makeNewTree

This function creates a new, single-node octree, with dimensions equal to the next highest power of 2 above the number passed to it, multiplied by 2.

printNodeInfo

This is a function for debugging octree-related functions. It prints information about an octree node to the terminal.

addToTree

This function recurses down an octree to find a coordinate, adding leaf nodes where they don't exist yet, then inserts the value passed to it at that coordinate.

recSeekPos

This function recurses through an octree to find the value at a given triple of coordinates, if that location is occupied. If it is empty, as soon as this is discovered, -1 is returned to indicate this.

recSetVal

This function recurses through an existing octree, to find a given location, and change the value at that location to the value given. If the location is unoccupied, the function returns 0.

recDelTree

This function recurses through an octree, freeing memory used by nodes as it goes, to completely free the memory allocated to that octree.

3.6.6 Functions contained in seqfile.h

This file contains functions for working with scoring matrices, and for loading sequence data from a FastA file.

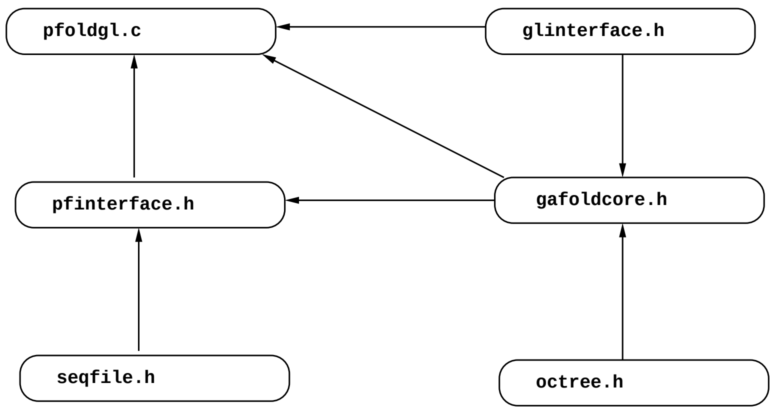

Figure 3.5: A diagrammatic representation of the source file dependencies of the program. File names are inside the boxes. Arrows point to the file doing the including (the parent file). All the arrows therefore eventually lead to the master source file, pfoldgl.c.

HPMatrix

This function generates, and allocates memory for, a standard HP scoring matrix (and translation table). It is passed to `convertSequence` to be used in translating from amino acid to HP sequence.

printMatrix

This function is an internal debugging function, which prints information about an energy matrix.

convertSequence

This function converts a sequence from amino acid sequence to the alphabet described in the scoring matrix passed to it. It is called by `setSequenceFromAA` in `pfinterface.h`.

loadSequence

This function loads the first amino acid sequence from a FastA file, and returns the sequence as a string. It is used by `setSequenceFromFile` in `pfinterface.h`.

scoreChars

This function returns the score incurred by the interaction of two characters, by looking them up in the scoring matrix passed to it. At this point, it is unused, but could be used in future for scoring using custom score matrices.

Table 3.3: A table of functions used in the program, by source file. Internal functions used by the .c files, not included in the .h file, have been shown.

pfoldgl.c	**gafoldcore.h**	**pfinterface.h**	**glinterface.h**	**octree.h**	**seqfile.h**
main	runGA	setSequence	realize	nextPowerOf2	exitClean
	performInit	setSequenceFromAA	configure_event	makeNewTree	HPMatrix
	cleanUp	setSequenceFromFile	expose_event	printNodeInfo	printMatrix
	ga_seed_bitstring_zero	loadDialogue	button_press_event	addToTree	convertSequence
	directionsToCoords	setPop	motion_notify_event	recSeekPos	loadSequence
	scoreOneInteraction	setGen	drawGL	recSetVal	scoreChars
	foldScore	setMut		recDelTree	deleteMatrix
	foldScoreOctree	setCrs			
	scoreUngerMoult	drawButtons			
	generationHook	drawSeqInput			
		drawSliders			

Chapter 4

Evaluation and Results

This chapter describes the qualitative tests that were applied to the program, and the results of these tests, in addition to qualitative observations on the operation of the program.

Contents

4.1 Quantitative tests

These tests were carried out for a small range of values, to get a general idea of the effects of varying some parameters to the genetic algorithm.

4.1.1 Variation of population size

Experimental conditions

Population size was varied, with constant parameters: crossover rate = 0.5, mutation rate = 1.0, number of generations = 100. The sequence used was that of myoglobin, pdb accession number 1A6M. Time was measured by wall clock, and the number of evaluations measured internally by the program. The test was carried out on an Athlon XP 1800+, with 512MB RAM, running SuSE Linux 9.1.

Results

These are illustrated in table 4.3.3, and figures 4.3.3 and 4.3.3. Linear relationships were observed between evaluation time and population size, and between evaluation time and the number of evaluations.

4.1.2 Variation of mutation rate

Experimental conditions

Mutation rate was varied, with constant parameters: crossover rate = 0.5, population size = 100, number of generations = 100. The sequence used was that of myoglobin, pdb accession number 1A6M. Time was measured by wall clock, and the number of evaluations measured internally by the program. The test was carried out on an Athlon XP 1800+, with 512MB RAM, running SuSE Linux 9.1.

Results

These are illustrated in table 4.3.3, and figure 4.3.3. A linear relationship was observed between execution time (and hence number of evaluations) and mutation rate.

4.2 Benchmarking against a solution with a known best score

Although it is impractical to know that a solution is optimal in this problem (due to the size of the state space), known high-scoring solutions from the literature have been collected (Hart & Istrail 2004). The program was tested against a protein of sequence hphhpphhhhphhhpphhpphph-hhphphhpphhppphppppppphh. The literature best solution had an Unger and Moult score of -32, but the best solution produced by the program had a score of -26. The solution is shown in figure 4.3.3.

4.3 Qualitative observations

4.3.1 Final results tended towards globularity

Particularly for smaller proteins, the final results tended to be reasonably close to spherical in shape. Figure 4.3.3 provides an illustration of this, although it should be noted that the solution shown is still somewhat flat across one of its axes. Larger proteins tended to result in two or three sub-globules, connected by polar chains, as shown in figure 4.3.3.

4.3.2 Final solutions for large proteins tended to contain overlaps

It was observed that for proteins of length roughly less than 50, the final solutions produced tended not to contain overlaps (ie: they were valid self-avoiding walks). However, for larger solutions, such as the one shown in figure 4.3.3, overlaps in the final solutions were relatively common.

4.3.3 Final solutions were often obviously suboptimal

It was observed, in many cases, that the final solution would contain features that could obviously be changed, by one or two folds, to produce a better solution.

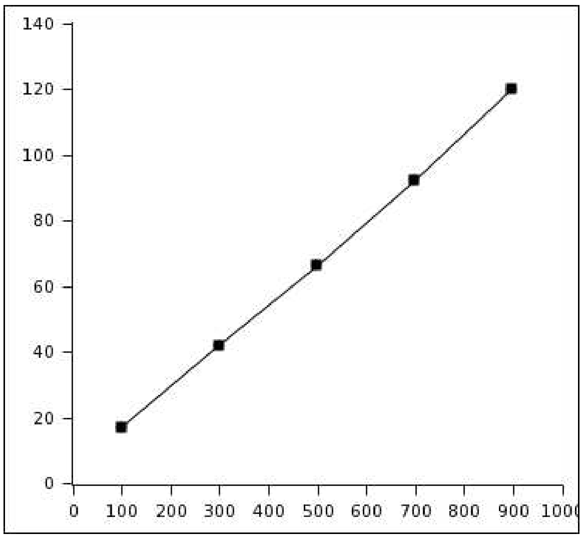

Figure 4.1: Graph of time of execution versus population size for the test varying population size. Time is in seconds. This represents graphically the data presented in table 4.3.3. A clearly linear relation is shown.

Population	Time (s)	Evaluations
100	17	20100
300	42	60300
500	66	100500
700	92	140700
900	120	180900

Table 4.2: Results of the test varying population size, showing time in seconds and the number of evaluations of the score function.

Mutation	Time (s)	Evaluations
0	8	10100
1	15	20100
3	28	40100
5	43	60100
7	58	80100

Table 4.4: Results of the test varying mutation rate, showing time in seconds and the number of evaluations of the score function.

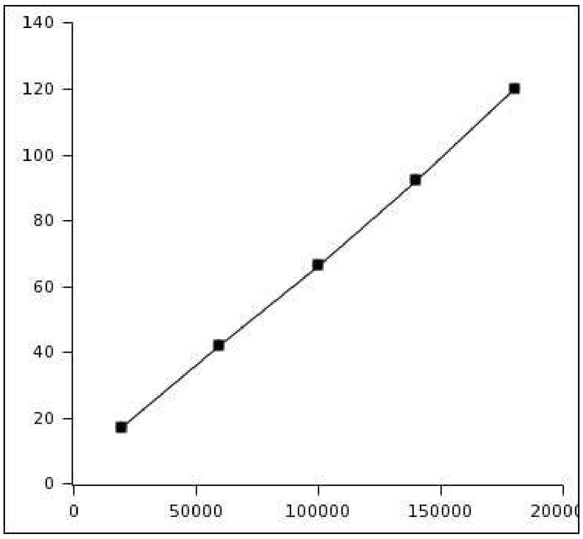

Figure 4.2: Graph of execution time versus number of evaluations of the scoring function, using data shown in table 4.3.3. A clearly linear relationship is shown.

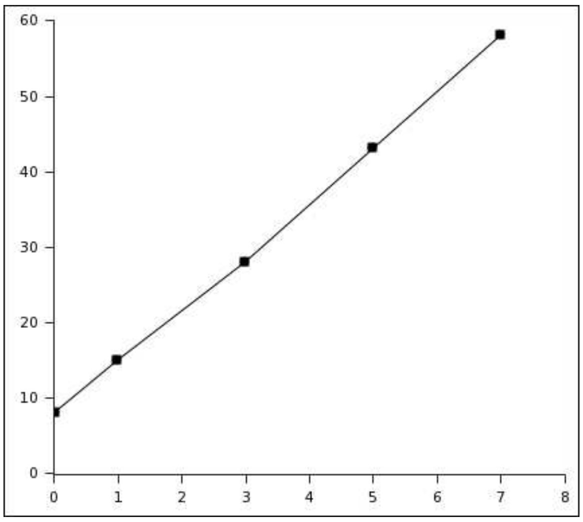

Figure 4.3: Graph of time of execution versus mutation rate for the tests varying mutation rate. A clearly linear relation is shown.

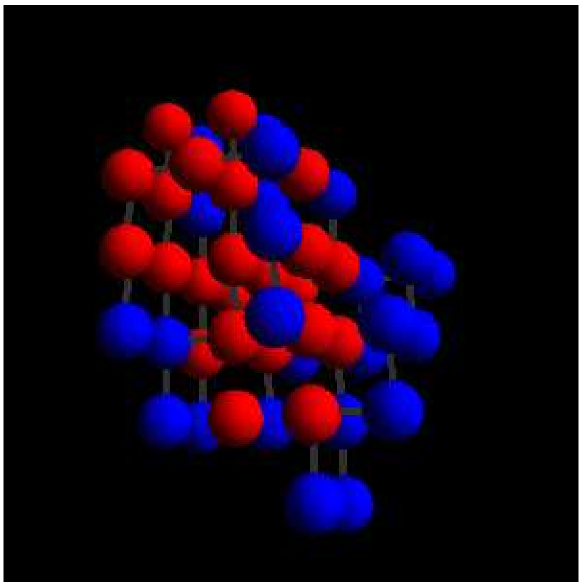

Figure 4.4: A screenshot of the best solution produced by the program for the protein of sequence hphhpphhhhphhhpphhpphphhhphphhpphhppphpppppppphh. The solution had an Unger and Moult score of -26.

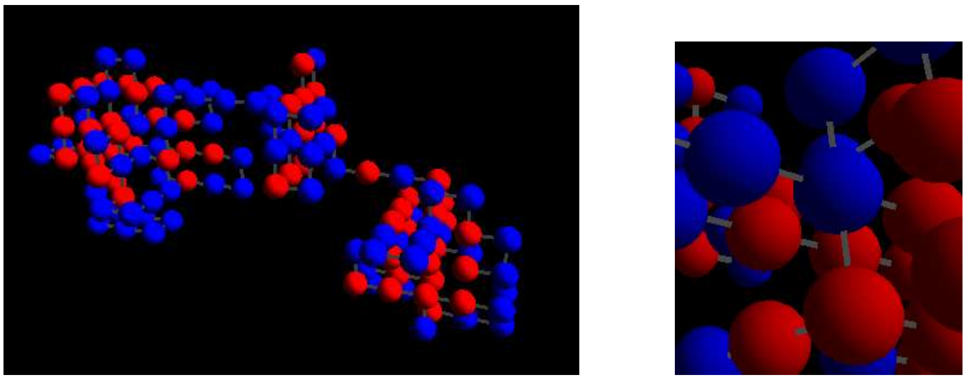

Figure 4.5: A solution for a longer protein, myoglobin (pdb accession number 1A6M). The protein is 150 amino acids long, and the solution has an Unger and Moult score of -43. Three hydrophobic globules can be observed, connected by single sections of the chain. On the right is a closeup of the right globule, showing an overlap in the chain. This solution contained many overlaps, and is not a valid self-avoiding walk.

Chapter 5

Discussion

This chapter provides a discussion of the system, its performance and potential uses. Extensions to the program are also proposed.

Contents

5.1 Performance

5.1.1 Time complexity

Execution time versus number of evaluations

The linear relationship between execution time and the number of evaluations of the scoring function implies that the time taken in executing the scoring function is the limiting factor in the time complexity of the program as a whole. This supports the design decision taken to optimise the scoring function.

Execution time versus size of the population

The linear relationship between execution time and the size of the population implies that the program is scalable, and could most likely handle increasingly larger population sizes for a linear increase in execution time.

Execution time versus mutation rate

The linear relationship between execution time (and number of evaluations) and mutation rate implies that the mutation function is using the scoring function to weight its selection of chromosomes to mutate. It is likely that this is because the mutation function uses the single selection callback function for its selection, and this was set to use roulette wheel selection. A better choice may have been to use a completely random single chromosome selection function, which would not make calls to the scoring function, and make mutation much faster.

5.1.2 (Not) finding optimal solutions

Results of the benchmarking

The program did not manage to find a solution of equal, or even similar score to the best known solution. This implies that it does not measure up to other published algorithms available for solving 3d HP lattice protein folding. The qualitative observations that solutions tend to be clumped, but not globular, that they contain illegal overlaps, and that they are often obviously sub-optimal, also indicate that the program prematurely converges. Patton et al (Patton et al. 1995) did not obtain a single solution with overlaps in it, though they do indicate that overlaps were present in intermediate, but suboptimal states.

Why the program doesn't measure up

One possible reason is that the algorithm implemented does not use refined genetic algorithm techniques, such as deterministic crowding and incest reduction, both used in Patton et al's genetic algorithm. These techniques favour breeding of dissimilar solutions, to prevent premature convergence of the population to a suboptimal solution. Another possible reason is that multiple point mutation was used, which mutated 20 % of the bits in a chromosome: This may cause too much variation in later solutions, so that only very suboptimal solutions are produced besides the best solution, and these are rapidly bred out.

5.2 General outcomes

The goal of this project, however, was not to produce a competitive tertiary protein structure prediction technique, but to explore the application of genetic algorithms to one such technique.

5.2.1 The usefulness of the GUI

The graphical user interface to sequence entry and parameter tuning was invaluable in setting up the experiments laid out in the evaluation chapter. For scientists tuning a genetic algorithm, such an interface could potentially be extremely useful. Learners could also use the program, as an easy-to-use introduction to genetic algorithms.

5.2.2 The usefulness of the OpenGL display

The OpenGL display of the solutions as they emerged allowed for qualitative observations to be made on the progress of the genetic algorithm as it ran, and once it was finished. Such observations would be nearly impossible to make with a solution presented in the form of relative directions (eg: ufffdlr). This could be a very useful tool for scientists creating genetic algorithms for lattice proteins, as it can guide decisions made during the design and tuning of their algorithms.

5.2.3 Extensibility

The fact that the program used GAUL as a framework makes it extremely extensible. A number of ideas for extensions are mentioned in the next section. This extensibility means that scientists wanting to implement hybrid algorithms, or improved genetic algorithmic solutions to lattice protein folding, could use the program as a basis for their own work.

5.3 Potential for extension

5.3.1 Improvements to the genetic algorithm

The failure of the genetic algorithm to obtain optimal (or near-optimal) solutions implies that it requires some refinement. Some possible refinements are mentioned below.

Cooling of mutation and crossover

A technique often employed in genetic algorithms is that of "cooling" the mutation and crossover rates as an optimal solution is approached, so that finer changes can be made to candidate solutions, in the hope of producing a better final solution. Unger and Moult (Unger & Moult 1993) used a similar technique in their algorithm. Since GAUL allows adjustment of mutation and crossover rates during a run, this could be relatively easily implemented.

Refined multipoint mutation

Mutating 20% of a chromosome may aid in producing diverse solutions initially, but is likely to produce un-viable solutions later in the run. A useful extension to the program would be to link multipoint mutation to the mutation rate (which only determines how many chromosomes multipoint mutation is applied to, not how many bits are altered), so that the number of bits changed is dependent on mutation rate. This would work well with cooling of mutation rate.

Use of random selection for mutation operators

The use of roulette wheel selection to choose candidates for mutation is very likely unnecessary, and probably extremely wasteful in terms of computation time, due to repeated calls to the computationally intensive scoring function. A better solution would be to use a completely random selection function, far less complex than the weighted one, for the selection of candidates for mutation.

5.3.2 Use of custom scoring matrices

The scoring (and translation) matrices used in the program were designed to be flexible and extensible. With a few minor modification to the scoring function, the program can be made to work with any amino acid scoring alphabet smaller than or of equal size to the standard amino acid alphabet.

5.3.3 Implementation of additional solutions

Several other solutions to lattice protein folding could be implemented alongside the one used in this program. Such solutions would require extensive coding, but could take advantage of the existing sequence processing and solution display functionality of the program.

Priority list encoding

Patton's group reported that the use of priority list encoding sped up their algorithms execution time significantly (Patton et al. 1995). Implementation of this form of encoding may dramatically increase the performance of the program. Significant changes are necessary to implement this, however.

Unger and Moult's algorithm

Unger and Moult's algorithm, though shown to be inferior to Patton's group's algorithm, continues to be developed on. It may be of interest to some researchers to have an implementation available to build on. This would require extensive coding to implement, however, due to fundamental differences between it and Patton's group's algorithm, which was implemented in this program.

Tabu search and other meta-heuristics

GAUL provides functions to use as a framework for tabu search, simulated annealing, and several other heuristic techniques. Researchers have recently worked with hybrids of genetic algorithms with tabu search (Jiang Tianzi & Songde 2003). Tabu search and other heuristics could be implemented on top of the existing algorithm in this program, though such implementations would require extensive knowledge of the heuristic to be added, and time for coding of it.

5.3.4 Improvements to the GUI

The GUI, as it stands, has

Widgets for choice of genetic operators

Drop-down lists could be used as a front end for the GAUL built-in mutation operators, as these are mostly defined as enumerated data types. This would give the user greater control over the operation of the genetic algorithm, and room to explore its workings.

Input of custom scoring matrices

As mentioned above, use of custom scoring matrices can relatively easily be added to the program. If this is implemented, then an interface for entering these matrices may be useful to users, though its design would require some thought in terms of usability.

Windowed output of results

It may be useful to create a panel, or pop up window, displaying details of the results of genetic algorithm runs. This output could then be removed from the terminal, so that the program could run completely without a terminal. Displaying of more results, such as data on the makeup of the population during a genetic algorithm run, might also give researchers more insight into algorithms they were developing.

File output

Another potentially useful addition would be the option of writing results to a file, so that they can easily be imported into spreadsheets or text editors.

Automatic scaling and centring of the OpenGL display

The OpenGL display, as it stands, uses a standard zoom setting, and a relatively simplistic means of centring the 3D solution in the display pane. Code which computes the bounds centre point of a solution, then centres the drawing of the solution correctly, and sets the zoom to display as much of the solution as possible, could be useful. Care would have to be taken, however, that such code did not begin to slow the program's execution.

More dynamic use of display lists in the OpenGL display

A possible speed optimisation to the display of 3D solutions would be to generate a display list for the current 3D solution whenever it is changed, so that, on rotation and zoom, the solution can be more rapidly redrawn.

5.4 Conclusions

The genetic algorithm, as implemented in this program, does not stand up to comparable programs mentioned in the literature. It does, however, provide useful visual feedback on the progress and final solution of the genetic algorithm used. It is also very extensible, and could be used as the foundation for future research (possibly future masters and honours projects, or even higher level research).

Bibliography

Adcock, S. (2004*a*), 'Gaul, a free open source genetic algorithms utility library', online.
URL: *http://gaul.sourceforge.net*

Adcock, S. (2004*b*), *The GAUL reference manual*.
URL: *http://gaul.sourceforge.net/gaul_reference_guide.html*

Backofen, R. & Will, S. (1998), Structure prediction in an HP-type lattice with an extended alphabet, *in* 'Proceedings of German Conference on Bioinformatics (GCB'98)'.

Berger, B. & Leighton, T. (1998), 'Protein folding in the hydrophobic-hydrophilic (hp) model is np-complete', *Journal of Computational Biology* **5**(1), 27–40.

Bornberg-Bauer, E. (1997), Chain growth algorithms for HP-type lattice proteins, *in* 'RECOMB', pp. 47–55.
URL: *citeseer.ist.psu.edu/bornberg-bauer96chain.html*

Crescenzi, P., Goldman, D., Papadimitriou, C. H., Piccolboni, A. & Yannakakis, M. (1998), 'On the complexity of protein folding', *Journal of Computational Biology* **5**(3), 423–466.
URL: *citeseer.ist.psu.edu/31865.html*

Crippen, G. M. (1991), 'Prediction of protein folding from amino acid sequence over discrete conformation spaces', *Biochemistry* **30**, 4232–4237.

Dandekar, T. (1997), 'Improving protein structure prediction by new strategies: experimental insights and the genetic algorithm', *Journal of Molecular Modelling* **3**(1-5), 1–3.

Dill, K. A. (1985), 'Theory for the folding and stability of globular proteins', *Biochemistry* **24**, 1501.

Dill, K. A. (1990), 'Dominant forces in protein folding', *Biochemistry* **29**(31), 7133 – 7151.

Duan, Y. & Kollman, P. (2001), 'Computational protein folding: From lattice to all-atom', *IBM Systems Journal* **40(2)**, 297–309.

Globus, A. (1991), Octree optimization, Technical report, NASA Ames Research Center.

Goldberg, D. E. (1989), *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley Publishing Company, Inc.

Hart, W. & Istrail, S. (1996), 'Fast protein folding in the hydrophobic-hydrophilic model within three-eighths of optimal', *Journal of Computational Biology* **3**(1), 53–96.

Hart, W. & Istrail, S. (2004), 'Hp benchmarks'.
URL: *http://www.cs.sandia.gov/ wehart/HP Benchmarks.htm*

Jiang Tianzi, Cui Qinghua, S. G. & Songde, M. (2003), 'Protein folding simulations of the hydrophobic-hydrophilic model by combining tabu search with genetic algorithms', *Journal of Chemical Physics* **119**(8), 4592–4596.

Kaffe-Abramovich, T. & Unger, R. (1998), 'A simple model for evolution of proteins towards the global minimum of free energy', *Folding and Design* **8**, 389–399.

Levinthal, C. (1969), How to fold graciously, *in* J. T. P. DeBrunner & E. Munck, eds, 'Mossbauer Spectroscopy in Biological Systems: Proceedings of a meeting held at Allerton House, Monticello, Illinois', University of Illinois Press, pp. 22–24.

Liang, F. & Wong, W. H. (2001), 'Evolutionary monte carlo for protein folding simulations', *Journal of Chemical Physics* **115**(7), 3374–3380.

Miyazawa, S. & Jernigan, R. (1985), 'Estimation of effective interresidue contact energy from protein crystal structures', *Macromolecules* **18**, 534–552.

Patton, A. L., Punch III, W. F. & Goodman, E. D. (1995), A standard GA approach to native protein conformation prediction, *in* L. Eshelman, ed., 'Proceedings of the Sixth International Conference on Genetic Algorithms', Morgan Kaufmann, San Francisco, CA, pp. 574–581.
URL: *citeseer.ist.psu.edu/patton95standard.html*

Petsko, G. A. & Ringe, D. (2004), *Protein structure and function*, New Science Press.

Protein Data Bank (Research Collaboratory for Structural Bioinformatics) holdings (2004), online.
URL: *http://www.rcsb.org/pdb/holdings.html*

Schonbrun, J. & Dill, K. A. (2003), 'Fast protein folding kinetics', *Proceedings of the National Academy of Science USA* **100**(22), 12678–12682.

Schuster, P. & Stadler, P. (2000), 'Discrete models of biopolymers'.
URL: *citeseer.ist.psu.edu/schuster00discrete.html*

The GIMP toolkit, a free open source windowing API (2004), online.
URL: *http://www.gtk.org*

The Gnu c compiler, a free, open source, multiplatform c compiler (2004), online.
URL: *http://gcc.gnu.org*

TrEMBL/SWISS-PROT statistics (2004), online.
URL: *http://www.ebi.ac.uk/swissprot/sptr_stats/*

Unger, R. & Moult, J. (1993), 'Genetic algorithms for protein folding simulation', *Journal of Molecular Biology* **231**, 75–81.

Xemacs, a free open source text editor and application development environment (2004), online.
URL: *http://www.xemacs.org*

Yasufuku, N. (2004), 'Gtkglext,a free open source opengl panel widget for gtk', online.
URL: *http://gtkglext.sourceforge.net*

Yue, K., Fiebig, K., Thomas, P., Chan, H., Shakhnovich, E. & Dill, K. (1995), 'A Test of Lattice Protein Folding Algorithms', *Proceedings of the National Academy of Science, USA* **92**(1), 325–329.
URL: *http://www.pnas.org/cgi/content/abstract/92/1/325*