A Functional Programming Approach to a Computational Biology Problem

Natalio Krasnogor Pablo E. Martínez López Pablo Mocciola David Pelta

LIFIA, Departamento de Informática, Universidad Nacional de La Plata.
C.C.11, Correo Central, 1900, La Plata, Buenos Aires, República Argentina.
Tel/Fax: +54 21 228252
E-mail: {natk,fidel,pablom,davp}@info.unlp.edu.ar
URL: http://www-lifia.info.unlp.edu.ar/

Abstract

Protein Folding is an important open problem in the field of 'Computational Biology'. Due to its combinatorial nature, exact polynomial algorithms to solve it could not exist, and so approximation algorithms and heuristics has to be used.

In this paper, a new heuristic is studied, based on the approach that considers that the folding process is coded into the protein. One important aspect of this work is that the algorithm was implemented using functional programming, resulting in advantages for the understanding of the problem. The results obtained are comparable with the ones obtained for classical algorithms.

Keywords: Protein Folding, Functional Programming, Combinatorial Optimization

1 Introduction

The Protein Folding problem is one of the most important open problems in Biochemistry due to its theoretical and pragmatic implications. In the field of Computer Science it is positioned in a branch called "Computational Biology", that tries to solve problems raised from Biosciences using mathematical and computational tools. Computational Biology had received a lot of attention and fundings after the Genoma 2000 Proyect.

Proteins are amino acid polymers. There are 20 aminoacids, each composed of a carbon backbone and a residue that determines its identity and its chemical properties. There are amino acids wich are neutral and hydrophobics, other neutral and polar, basic or acidic. While in its native state, each protein presents a charateristic three dimensional shape. This 3D shape is strongly related to the biological properties of the given protein and the role it plays in living beings.

The Protein Folding problem can be stated as follows: given an unfolded aminoacid sequence, find the 'right' folding of that sequence. The unfolded state is just the linear arrangement of amino acids. In nature, the proteins fold to their 'native' state, which determines its functionality. Also, the common believe is that this native state is minimizing (maximizing) some yet unknown criteria.

Some lattice-based computational models of the Protein Folding were shown *NP-Complete*, others remain *NP-hard* [Fra93, UM93a], but some approximation algorithms exist [HI95].

However its theoretical and practical relevance [SSK94, UM93a] makes worthwhile spending resources and time in modeling the folding process. Usually, strong emphasis is put in the results obtained, rather that in the way they are generated, enlarging the gap between researchers from Computer Science and Biology. The claim of this paper is that, using the right tools, both communities can colaborate much closer, enhancing the results at the same time.

Historically, 'Functional Programming' [BW88] has been associated with a small scope of applications, mainly academic. Computer Science community did not pay enough attention to its potential, perhaps due to the lack of efficiency of functional languages. Now, new theoretical developments in the field of Functional Programming [JM95] are emerging, and better languages (e.g. Haskell [PH$^+$96], Concurrent Haskell [JGF96]) have been defined and implemented. Also, the gap between theory and practice is smaller in this paradigm than that of other paradigms, making Functional Programming a good choice for developing simulation and optimization programs [Wad95]. Traditionally, all programs for optimization problems were written in C, C++ or Ada. This fact builds a firewall between developers and end-users. Protein Folding is suitable to be modeled with a lazy concurrent functional language for many reasons:

- non-computer-science people can think in a very high abstraction level and map their ideas, almost directly, to functional code;
- the learning curve of a Functional Programming language is smoother than that of an imperative one, bridging the gap between developers and users;
- functional code is concise;

- the folding process is intrinsecally parallel and Functional Programming is specially adequate for managing parallelism;
- concurrent processes on the string to be folded can be simulated using easy-to-use features of concurrent functional languages;
- the use of lazy languages avoids the construction of protein configurations until they are needed (if ever);
- using Functional Programming, it is straightforward to associate folding algorithms to folding patterns [KLMP97].

The proposal is to use Functional Programming as a bridge between researchers of Computer Science and Biosciences. Computer-science-researchers have their benefit because of rapid prototyping, while bioscience-researchers have it because of the high abstraction that Functional Programming provides. Also, it will be showed that the functional programming paradigm is, at the current time, capable of affording combinatorial optimization tasks.

In this paper a new functional heuristic is presented. It is deterministic, and linear in the size of the input protein, but the quality of its output has is comparable with Monte Carlo method (that is non-deterministic and takes much more time). The concept of grammar was employed to parse the protein; that is, when a word in the language of the grammar is recognized, a fold of that word in a certain way is performed. The parsed proteins can be used as initial population for evolving methods, with the properties that they are factible (self avoiding, not two aminoacids are mapped to the same location in space) and that they are local optimum (because so are the folding patterns).

2 The problem

A protein can be understood as a linear sequence of components, called aminoacids, that under certain physical circumstances, is folded in a unique functional structure called its native state, or terciary structure. To find the native state given only the linear sequence of aminoacids is the Protein Folding Problem.

Until now, there were two approachs to the problem:

1. the thermodynamic approch. In this approach, aminoacid conformations are studied in terms of free energy. The fact that the native state is the one that minimizes free energy is assumed. There exist different models of the energy function. Many factors are considered when developping an analitical expresion for this function, i.e., protein's shape, size and polarity of the involved molecules. Furthermore, there is no consensus of the relative weigth of each factor.
2. the dynamic approach. It assumes the existence of "folding tunnels" that guide the protein to a unique and stable state – its native state. It uses some concepts from thermodynamics, but in a different way.

An alternative approach, and the one developed in this paper, assumes that the folding mechanism is 'coded' in the protein by means of an unknown language. The work consists then in finding the language that rules the folding process.

The most simple models used to represent proteins are based on grids (of 2 and 3 dimensions), where each 'position' is filled by at most one aminoacid. The correspondence between aminoacids and positions is called *embedding* of the protein, and when the embedding is injective, it is called *self avoiding*. The problem, under these hypothesis, was shown NP-complete by [Fra93] and [UM93a] between 1992 and 1993. The existence of a polynomial algorithm that solves the problem exactly is then assumed imposible. So, the use of heuristics and approximation algorithms became the most promising chance to solve, at least, some of the problem instances.

In this paper the hydrophobic-hydrophilic model is assumed. This model considers only two kinds of aminoacids: hydrophobic ones, represented as a 'B' (for Black), and hydrophilic ones, represented as a 'W' (for White). The energy function takes into account only the interactions between topological neighbours of type B (P). This can be easily seen in the following table where interactions between hydrophobics and hydrophilics aminoacids are shown,

	W	B
W	0	0
B	0	-1

This table states that, every time a hydrophobic aminoacid is a topological neighbor of another aminoacid of the same type free energy is minimized by a constant factor of -1. The same score table can be used to measure the number of bonds of a given folding (see Chap. 2.1), in this case the optimization task is to maximize the number of bonds given by an B-B interacion.

There exist solutions to the problem under this model, using Simulated Annealing [SSK94], Genetic Algorithms, and the Monte Carlo method [UM93b]. Also [PKM95, PKM96] present a polynomial, deterministic algorithm that follows the research line of [UM93b].

2.1 Generalization of the model

In order to study the *Protein Folding Problem* in an abstract way, a generalization of it due to Paterson and Przytycka [PP] can be used. In their paper they consider the *String Folding Problem*, wich can be stated as follows: given a finite string S, an integer k, and a grid G, is there a fold of S in G with a score at least k? A fold of S in G is defined as an injective mapping F: $[1 \ldots n] \mapsto G$, where $n = \mid S \mid$, and if $1 \leq i, j \leq n$, i=j-1 then $F(i)$ is adjacent to $F(j)$ in G. the score of F is computed counting the number of identical symbol pairs mapped to adjacent nodes of G, calling those pairs *bonds*. Paterson and Przytycka showed that *String-Folding* is *NP-Complete* in the $\mathbb{Z}^2$ and $\mathbb{Z}^3$, while other instances of the problem remain *NP-Hard*. In this paper, the process of string folding is modelled using an extension of *L-system* to generate a family of restricted parallel rewriting grammars. The biologist Aristid Lindenmayer develops what it came to be named 'Lindenmayer Parallel rewriting systems' when he was trying to model development in plants [Lin68]. A basic *L-system* is a grammar G=$\{\Sigma, \Pi, \alpha\}$, where Σ is a finite set of symbols called the alphabet, Π is the set of rewriting rules and $\alpha \in \Sigma^*$ is the starting string that generates the language. The most simply *L-system* is context-independent, taking Π with the structure $\{\pi : \Sigma \mapsto \Sigma^*\}$, wich means that a simple character of a string S maps to a string of Σ^*. One of the most important features of *L-systems* is that the rewriting rules are applied in parallel all over the original string, while in other grammars, rules apply sequentially. *L-systems* can be extended to allow context-sensitivity, if the rewriting rules are of the form: $L\langle P\rangle R \mapsto S$, with $P{\in}\Sigma$ and $L,R{\in}\Sigma^*$. The traditional interpretation to *L-systems* are 'Logo-like' draws.

2.2 Extension to *L-systems*

$G2L$ grammars are a neat extension to *L-systems* that can be used to specify arbitrary graphs (see [Boe95] for a detailed description). With the same spirit as in Boers work, looking for restrictions on *L-systems* that allows for the specification of arbitrary foldings of string is one of the author's goals. The intention is to represent the graph induced by the mapping $F()$ using this new subfamily of grammars. Cycles in the graph must be correctly disabled, wich in turn means that the folding is self avoiding. The restrictions under develop apply to the set of rewriting rules. Only some kind of structures are allowed, and arbitrary left and right contexts are not permitted.

2.2.1 Folding L-systems, Some Definitions:

$\Pi \in G$ will be called LES for $L-$Equation System. A *Folding L-system* with radius r, *Folding L-system*$_r$, is a LES in wich each $\pi \in \Pi$ is of the form $L\langle P\rangle R \mapsto S$, with $P,S,L,R\ \in\Sigma^*$, being $r = Max(k)$ where $k \in (\{\ |L|, L \leftarrow \pi\ \ \in \Pi\ \} \cup \{\ |R|, R \leftarrow \pi\ \ \in \Pi\ \})$, and $|P| = |S|$. That is, k is a number that takes values from the set comprised of all the sizes of the right and lefth context of the rewriten rules. r is the greater of this numbers. Also it was shown in [KT97] that a *Folding L-system*$_r$ can simulate a Cellular Authomata of radius r, and a Cellular Authomata of radius r can simulate an *Folding L-system*$_r$. In that way *Folding L-system*$_r$ is equivalent to a Unidemensional Cellular Authomata of radius r.

3 The heuristic

In the previous section the use of heuristics to solve the Protein Folding Problem is mentioned – Simulated Annealing, Genetic Algorithms and Monte Carlo Method being the most common ones. In this section a new familiy of folding heuristics is developed. The heuristics in the family satisfy two important goals:

- they can give energy values that are competitive with the ones of existing algorithms, and
- they reflect the biological process of folding in a way that is concise and easy to express using grammars.

The goal of these heuristics is to minimize the value of the energy function; this function will be defined as the number of bonds resulting of embedding the protein in a grid. The grid (in 2 or 3 dimensions) is important in the algorithm because the concepts of bonds and patterns are based on the position of the aminoacids in the grid, and the notion of energy is defined in terms of bonds.

A heuristic in the family described, first alignes the string using *folding patterns*, then selects some reference points (the criteria used for this identifies the particular heuristic) and finally folds the reference point that minimizes the energy function.

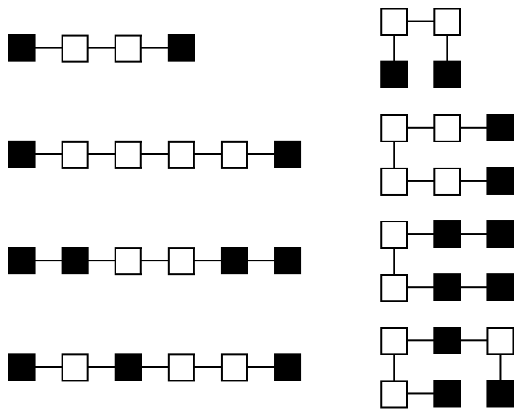

Figure 1: Examples of folding patterns.

A *folding pattern* is a pair of proteins. The meaning of a folding pattern arise in the folding process: the first component of the pattern is replaced with the second component as part of the process. The set of possible folding patterns is give by extension, and it is a regular set. A future generalization can be obtained through the use of grammars in order to express families of folding patterns. One important point about patterns is that they are not necesarily pairwise disjoint, thus reflecting the fact that there is more than one way to do the folding a protein.

The application of folding patterns can be understood as a process of *parsing* the protein looking for the pattern and replacing it. This allows a generalization and abstraction that are ideal for the extension of the model, where the patterns are expressed using grammars. The parsing process returns all the possible foldings according to the given patterns, and a parallel analysis of alternatives can be considered. The combinatorial explosion implied by the parsing process is restrictive when coded in the imperative paradigm, but, in the functional paradigm, the lazy evaluation suspends the computation of all the alternatives until they are needed. If any alternative is never needed, then it will not be computed. Additionally, the functional paradigm is better suited for a future parallelization of the code, where each alternative is considered by a different processor.

The algorithm begins with the aplication of folding patterns, in order to reflect the short range interactions between aminoacids of type B. The output of this step will be called *parsed proteins*. The folding patterns used in the present version are defined by extension for aminoacid sequences with lengths between 4 and 6 (see Fig. 1).

The second step consists in scanning each parsed protein looking for reference points. The choice of these points is done based on the following criteria (note that different criteria determine different heuristics):

1. Energy criterion. Let **El** and **Er** be the energy (number of bonds in this case) between aminoacids 1 and i and between the aminoacid $(i+1)$ and the last one, respectively. The folding points resulting from this criterion is $P=\{i/abs(\mathbf{El}-\mathbf{Er}) \leq \varepsilon\}$

2. Hydrophobicity criterion. Let **Bl** and **Br** be the number of aminoacids B between aminoacids 1 and i and between the aminoacid $(i+1)$ and the last one, respectively. The folding points resulting from this criterion is $P=\{i/abs(\mathbf{Bl}-\mathbf{Br}) \leq \varepsilon\}$

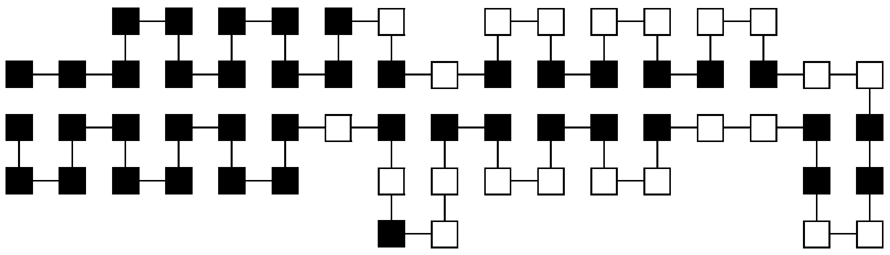

Figure 2: An example of a U-fold for UM8 with energy -29.

UM1: BWBWWBBWBWWBWBBWWBWB

UM2: BBWWBWWBWWBWWBWWBWWBWWBB

UM3: WWBWWBBWWWWBBWWWWBBWWWWBB

UM4: WWWBBWWBBWWWWWBBBBBBBWWBBWWWWBBWWBWW

UM5: WWBWWBBWWBBWWWWWBBBBBBBBBBWWWWWWBBWWBBWWBWWBBBBB

UM6: BBWBWBWBWBBBBWBWWWBWWWBWWWWBWWWBWWWBWBBBBWBWBWBWBB

UM7: WWBBBWBBBBBBBBWWWBBBBBBBBBBWBWWWBBBBBBBBBBBBWWWWBBBBBBWBBWBW

UM8: BBBBBBBBBBBBWBWBWWBBWWBBWWBWWBBWWBBWWBWWBBWWBBWWBWBWBBBBBBBBBBBB

Figure 3: Proteins used for testing.

3. Aminoacid distribution criterion. Let **Bl** and **Br** be as in criterion 2. Let **Wl** and **Wr** be the number of aminoacids W between aminoacids 1 and i and between the aminoacid $(i{+}1)$ and the last one, respectively. The folding points resulting from this criterion is $P{=}\{i/abs((\mathbf{Bl}/\mathbf{Wl}) - (\mathbf{Br}/\mathbf{Wr})) \leq \varepsilon\}$

In all the criteria, ε is the accuracy parameter, and satisfies $0 \leq \varepsilon \leq 2$.

The last step is to perform a folding of the parsed protein with the shape of a 'lied down U' (called a U-fold, see Fig. 2), in a reference point that minimizes the energy. This fold allows long range interactions between aminoacids of type B.

Tests were done using instances of proteins taken from [UM93b], because the global optima are known for that sequences, and then the results can be checked against the ones obtained with simulated annealing and genetic algorithms. The instances choosed correspond to the Dill's Model [Dil85], and are presented in Fig. 3.

There are some important points to take into account, and to analyze for future works:

- the distribution of aminoacids may be relevant. In the present version an even distribution of them is assumed, but it could not be the case in real situations.

- the spatial form of the folded protein may be relevant. For example, more 'compact' foldings could be preferable to the opposite.

- biological factibility of proteins may be defined, but it is not known the best way to do it.

4 The implementations

In this section two implementations of the algorithm are presented. The first one was done in ANSI C, and several reasons guide this selection: the code is portable, a future parallelization using PVM (Parallel Virtual Machine) is possible, and the C language is very efficient. However, the resulting code was too confuse, and many implementation details blurred the reading and understanding.

Functional programming has been designed with the idea that algorithms coded with it are easy to develop and read, but, historically, it was only confined to applications that came from the academic environment. Nowadays, functional programming is widening its horizons, mainly because new techniques, compilers and tools are developed, and they are competitive in efficiency with their clasical relatives [Wad95]. So, the functional language Haskell [PH+96] was chosen to do a second implementation, looking for an easy to understand but efficient code.

In Sects. 4.1 and 4.2 the C and Haskell implementations are presented respectively and a comparison between implementations is done.

4.1 The C implementation

The first thing to have into account is the data structure that represents proteins. For that, linked lists implemented with pointers was choosed – each element in the list represent one aminoacid. Proteins will also be folded (in a two dimensional grid in this version), and so, information about the position of each aminoacid is provided in two different ways: absolute and relative. The absolute position is given as a coordinate pair indicating a cell in the grid, and the relative one as an element of the set $\{U, D, R, L\}$, indicating that the position of the next aminoacid will be Up, Down, Rigth or Left of the present one. Both representations have to be updated with every change of the embedding.

```
/* Data Structures to represent Proteins */
 struct Amino1 { char type;
                 int x,y;
               };
 struct Chain { int size;            /* Number of aminoacids */
                char *directions;    /* {U,D,L,R}+  */
                struct Amino1 *sec;  /* Chain of aminoacids */
              };
```

The application of a folding pattern can be implemented as substring substitution in the string .

Three procedures are used extensively in the algorithm: rotation of a protein, energy evaluation, and correctness detection (self-avoidness). These three procedures are simple when implemented over the structure presented.

```
 /* Rotates the chain in the 'start' aminocid 'angle' degrees */
void RotateAm(struct Chain *pr,int start,int angle)
```

```
{int i,CurrX,CurrY; char move;
 for(i=start;i<=pr->size;i++)
   {pr->directions[i-1] = NextPosition((pr->directions[i-1]),angle);}
 UpdateNumericalPositions(pr);
}
```

In the present version, the energy evaluation and the correctness detection are calculated separately, having each of them an order of N^2, and they also have to calculate potentiations, because they involve euclidean distances. If efficiency becomes a real problem, there exist the possibility of evaluating the two functions at the same time with order N.

```
/* Pseudo-code for an energy function that works in O(N) */
Start with an empty matrix M
energy = 0
Put the first amino AM1 in M
If Type(AM1)='H' Then
  For every neighbour position except the next that will be ocuppied
     mark it with P  (possible place of contact)
EndIf
For i=2 To ChainSize
  if the position for AMi is occupied
     then 'you have a colision'; EXIT
  if (Type(AMi) = 'H')
    if the position for AMi is marked with P
       energy += 1    (you make one contact)
    Put AMi in M
    For every FREE neighbour position except the next that will be ocuppied
      mark it with P
  else
    Put AMi in M
Next
```

Even when the results are excellent, and the expectatives are very promising, most of the development time was spent in some problems with pointer arithmetic and the explicit memory management. Also the resulting code is far from readable, and thus the understanding of the solution from the code is not very easy.

4.2 The Haskell implementation

Functional programming was choosed having in mind that programs coded with this paradigm are very easy to read, and that the learning time for non-computer-scientist is short. Also, testing the suitability of functional languages for real applications was one of the goals of this work.

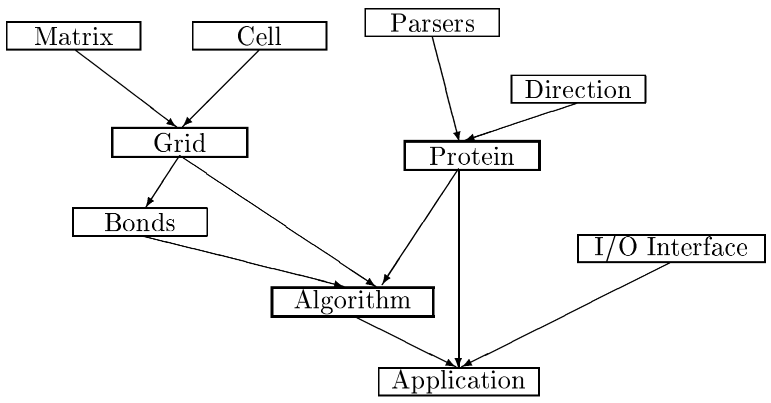

Figure 4: Module hierarchy.

The language used is Haskell in its version 1.3 [PH+96], and Hugs [Jon96] was used for testing and prototipation. The former is a high level, pure language, based on functions that provides modularization. The latter is an interactive system that accepts Haskell code and allows to run any function of it, displaying the results. Both languages have a strong (static) type system.

It is important to mention that the Haskell version of the algorithm is not a simple translation of the C version. The code was designed from scratch using functions, with the goal that different methods can be represented, not only the heuristic presented in this paper.

The functional code is designed as a set of interrelated modules, each of which encapsulate an abstract datatype that represent one entity or concept relevant to the solution, hiding its internal representation. The hierarchy of modules appears in Fig. 4.

There are three important modules: the Protein module, where the structure and basic functions for proteins are defined, the Grid module, where grids, the energy function and correctness detection are defined, and the Algorithm module, where the heuristic is implemented.

The proteins are represented as a list of pairs (aminoacid, direction), providing information about relative positions of aminoacids. The basic functions for proteins are:

```
type Protein = [(Aminoacid, Direction)]
rotateP :: Int -> Way -> Protein -> Protein
uFold :: Int -> Protein -> Protein
```

which rotates a protein in a given point and way, and makes a U-fold in a given reference point, respectively.

Folding patterns are also defined in this module. They are represented as parsers.

```
type Pattern = Parser (Aminoacid, Direction) Protein
```

The type `Parser` takes two arguments: the type of the components of proteins and the type of the result. It can be defined as

```
type Parser comp res = [comp] -> (res, [comp])
```

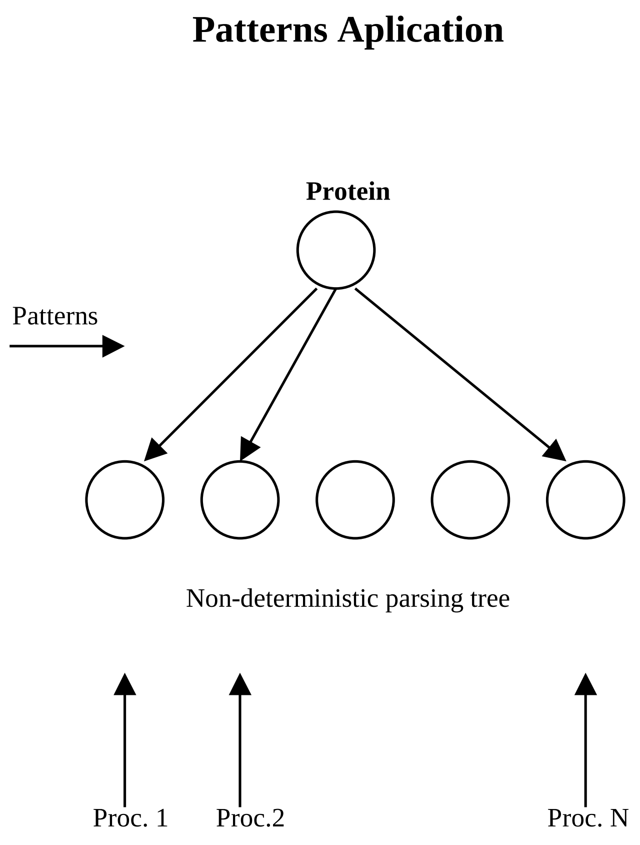

Figure 5: An example of a nondeterministic parsing tree

The type `Parser (Aminoacid,Direction) Protein` is then equivalent to `Protein -> (Protein,Protein)`. Based on the pair of proteins that defines abstractly the folding pattern, the operational behavior of the parser is that when the first protein of the pattern matches an initial subsequence of the input protein, the second one is returned paired with the remaining input.

The first step of the heuristic is the aplication of the folding patterns to a given protein, which is represented by the function `applyPatterns`.

```
applyPatterns :: [Pattern] -> Protein -> [Protein]
```

This function returns a list of all possible ways to apply the given folding patterns to the given protein, see Fig. 5. Thanks to the lazy evaluation of the language, an alternative is not computed unless it is needed.

The Grid module uses matrixes for the representation of grids. The module Matrix used in this version is only a prototype, and it is not efficient at all, but as it is defined as an abstract datatype, its representation can be changed without affecting the rest of the code. In each cell of the grid there are information about the aminoacids that lie in that cell, and about the possible bonds. This representation allows to have non-self-avoiding proteins embedded, and thus a function to test correctness is provided.

```
type Grid = Matrix Cell
isCorrectP :: Protein -> Bool
energyP :: Bonds -> Protein -> Int
energyPpos :: Bonds -> Protein -> Int -> (Int, Int)
```

The function `energyP` calculates the energy of the given protein, and the function `energyPpos` calculates, given a point in the protein, the energies of the first and second segment of the protein.

The different criteria used for determining the reference points are defined in the Algorithm module. The criteria are represented as functions that take the protein and return all the possible reference points.

```
type Criteria = Bonds -> Protein -> [Int]
```

One important characteristic of this representation is that new criteria can be defined as easy as with mathematics – no special attention to representation is needed.

The algorithm that implements the heuristic is then a composition of the different functions already defined:

```
algorithm :: Bonds -> Criteria -> Protein -> Protein
algorithm bonds criteria prot =
   maxEnergy [ fprot |
                  pprot     <- applyPatterns patterns prot,
                  refpoints <- criteria bonds pprot,  i <- refpoints,
                  fprot     <- uFold i pprot,
                  isCorrect fprot
             ]
```

The notation in this definition is called "list comprehension" and is similar to that of set comprehension. The function `maxEnergy` returns the protein of the given list with the maximum energy.

4.3 The results

Two important results have to be mentioned.

The first one is about the use of Functional Programming: the Haskell version is much better than the C version in many aspects. To mention only the most significant:

- the development time was really much less in the functional version;
- the number of bugs and errors in the code was almost nothing when comparing against the C version, and also the most important ones were detected by the type system of the Haskell language;
- the code is easy to read and understand.

The author's expectatives were filled completely.

The second one is about the optimization behaviour of the heuristic: the output of the algorithm is not so far from that of already known solutions, as can be seen in the table of Fig. 6. A U-folded version of the eighth protein of Fig. 3 can be found in Fig. 2.

Instance	Optimum	Monte Carlo	Criterium		
			1	2	3
UM1	-9	-8	-7	-7	-7
UM2	-9	-8	-9	-9	-9
UM3	-8	-7	-7	-7	-7
UM4	-14	-12	-10	-9	-10
UM5	-22	-18	-14	-13	-15
UM6	-21	-19	-17	-17	-17
UM7	-34	-31	-28	-28	-28
UM8	-42	-31	-29	-29	-29

Figure 6: Results of the heuristic.

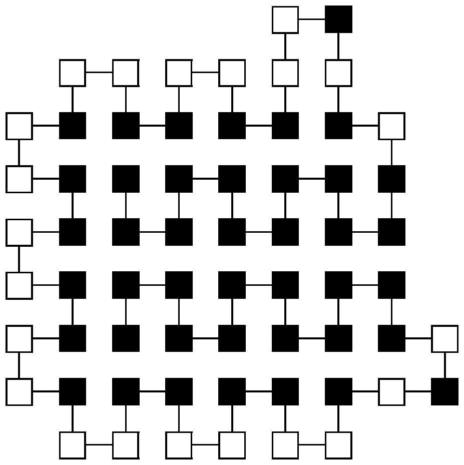

Figure 7: Folded UM8 with energy -39 obtained with a GA feeded with parsed versions of the protein.

Also, a genetic algorithm feeded with the parsed proteins gives results better and faster than those of Monte Carlo version of [UM93b]. To ilustrate that, in Fig. 7 is presented a folding obtained for the eighth protein (Fig. 3) that has a better energy value that the best one of [UM93b].

A genetic algorithm can be sketch as follow:

```
[0] Initialize population
[1] While (termination-criteria = FALSE) Do
[2]    Select population
[3]    Cross population
[4]    Mutate population
[5] Output results
```

The selection process assigns a mating probability P_i to each individual i in the population. This probability is computed based in their fitnesses: $P_i = \frac{f_i}{F}$ where f_i is i's fitness and $F = \sum_{i=1}^{i=N} f_i$

During the crossover stage two parents geneterate just one offspring, so the mating pool must be filled with $2 * (N - Z)$ genomes. Here N is the population size and Z the elite set size.

Once the mating pool have been generated, a crossover stage arise. Two parents are selected and they are mated with a probability P_X, where P_X is the crossover probability. The crossover is the same as Unger's crossover. If the mating doesn't happen then the parent with the best fitness is copied to the next generation.

Our GA uses four kinds of macromutation steps which are applied accordingly to a probability P_{MM}. The allowed macromutations begins by choosing two random points an them perfomrs one of the following actions:

1. The genomic substring between this pair of points is changed in order to represent a turn of 0, 90, 180 or 270 grades of this protein segment.

2. Between this points the genome is reflected verticaly or horizontaly.

3. Between them the protein is unfolded.

4. Each peptide in inside the selected point interval is randomly oriented. This is the most structure less macromutation.

The GA preserve the best individual and copies it to the next generation. The original feature of this GA is that the initial population was created in a non-random way. The initial population was set to be the leaves set of the nondeterministic parsing tree of the instance to optimize. When the width of the bottom level of this tree was greater than that of the population it was simple cutdown. The leaves selected as individuals were those with bigger differences in their structure. In that way we asure non premature convergence.

Simulations shown that the performance of the GA initialized with the heuristic population was far better than that of the random initialization. This may seem to be in conflict with the common believe that a random population is a better start place for searching. The reason for that believe is that non-random initialization cause premature convergence. Furthermore, this is not an intrinsic drawback of non-random heuristic. If care is put in the structural scattering of heuristic initialized genomes then premature convergence is avoid, see [NDLlC98, NDP+97].

5 Future work

The most important future work is the discovery of the language used for proteins to code the folding. For that, there are three main possible approaches:

- the first one is to use the 'Computational Mechanical Framework' of Dr. Das [D+95] directly on the proteins, to discover the language;

- the second one is to use the 'Computational Mechanical Framework' on the genetic algorithms, to discover what kind of computation they perform;

- the third one is to use the 'Logical Data Analysis' method of Dr. Hammer [H+96] on the proteins correctly coded, to discover more complex folding patterns.

Further studies on the advantages of using parsed proteins as initial population for evolutives algorithms should be done. The author's found that the designing of folding grammars is very difficult. Research will be done on the use of automatic genereted grammars in two flavors, Cellular Authomata transition rules and *Folding L-system*$_r$ grammars.

The project consist in development a concurrent functional framework to genetic algorithms. It should provide a base to experiment with evolutive algorithms that have arbitrary representations of population and operators. Also it should decrease the amount of work and coding to each new application.

The Radcliffe and Surry's idea in their work called RPL2 is followed. RPL2 stands for The Reproductive Plan Language (1994), and it is a language that was designed to write, perform and modify the evolutive algorithms in an easier way. RPL2 has imperative programming features, and thus the paralellism is explicit, and to extend a program, a new compilation and linking is needed. In this language there are no restrictions to the shape of genomas, and then it is applicable to real world optimization problems.

Finally, the use of Functional Programming for optimization methods will continue.

6 Conclusions

This paper presents a new heuristic to find near optimal solution for the Protein Folding Problem. This heuristic assumes the hypotesis that the folding process is coded in the protein itself, and one of the goals is to discover the language used for that coding. In order to do that, the notion of folding pattern is used.

Two implementations are provided: one that uses the (imperative) C language, and one that uses the (functional) Haskell language. A secondary goal of this work is to compare both languages, and both paradigms.

The resulting code in both versions satisfies largerly the author's expectatives. The testing performed with some known sequences of idealized aminoacids results in values of the energy function that are competitive with already known algorithms. Also the comparison between the two languages gives the expected results: while the C code is faster, the Haskell code is more readable, and it was more easy to design and develop, allowing to concentrate the thinking effort in the problem itself, and not in the coding.

The algorithm is deterministic, and linear in the size of the input protein, but its output has energy values comparable with Monte Carlo method (that is non-deterministic and takes much more time). The parsed proteins can be used as initial population for evolving methods, with the properties that they are factible (self avoiding) and that they are local optimum (because so are the folding patterns).

Another conclusion is that many lines of research has emerged as consequence of the development of this work.

Acknowledgements

The authors want to akcnowledge to Gustavo Lobo Acher, Pablo Oña and Cristian Apas for lending the hardware; to Alicia Reija and Fernando Lyardet for visual design guidelines; to Dr. Rajarshi Das for many stimulating ideas; to Lic. Pablo Moscato for bringing attention to the protein folding problem; to the Sloan

Fundation Department of Energy, U.S., for funding support; and finally, to all LIFIA members that help the authors during the development of this work.

References

[Boe95] Egbert J.W. Boers. Using L-systems as graph grammar: G2L systems. Technical Report 95-30, Department of Computer Science, Leiden University, The Netherlands, October 1995.
By FTP, in `ftp://ftp.wi.leidenuniv.nl/pub/CS/TechnicalReports/1995/tr95-30.ps.gz`.

[BW88] Richard S. Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[D$^+$95] Rajarshi Das et al. Evolving globally synchronized cellular automata. In Morgan Kauffman, editor, *Proceedings of the VI International Conference on Genetic Algorithms*, 1995.

[Dil85] Ken A. Dill. *Biochemistry*, 24:1501, 1985.

[Fra93] Aviezri S. Fraenkel. Complexity of protein folding. *Bulletin of Mathematical Biology*, 6, 1993.

[H$^+$96] Peter L. Hammer et al. An implementation of logical analysis of data. RRR 2296, The State University of New Jersey, Rutgers, July 1996.

[HI95] W.E. Hart and S. Istrail. Fast protein folding in the hydrophobic-hydrophilic model within three eighths of optimal. In *Proceedings of the 27th ACM Symposium on Theory of Computation*, pages 157–168, 1995.

[JGF96] Simon L. Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *ACM Symposium on the Principles of Programming Languages (PoPL'96)*, St.Petersburg Beach, Florida, January 1996.

[JM95] Johan Jeuring and Erik Meijer, editors. *Advanced Functional Programming, LNCS 925.* Springer-Verlag, May 1995.

[Jon96] Mark P. Jones. Hugs 1.3 - the Haskell user's Gofer system. User manual. Technical report, Department of Computer Science, University of Nottingham, August 1996.

[KLMP97] N. Krasnogor, P.E. Martínez López, P. Mocciola, and D. Pelta. Modelling string folding with G2L grammars. In *Proceedings of the International Conference on Functional Programming, ACM SIGPLAN, (ICPF'97)*, page 314. Association for Computing Machinery, June 1997. Also published in Recomb'97.

[KT97] Natalio Krasnogor and Germán Terrazas. Reporte interno lifia, unlp. En preparación, 1997.

[Lin68] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[NDLlC98] N.Krasnogor, D.Pelta, P.E.Martínez López, and E.de la Canal. Genetic algorithms for the protein folding problem, a critical view. In *Proceedings of the Engineering of Intelligent Systems (EIS'98)*, 1998.

[NDP$^+$97] N.Krasnogor, D.Pelta, P.Mocciola, P.E.Martínez López, and E.de la Canal. Enhanced evolutionary search of foldings using parsed proteins. In *Anales del Simposio de Investigación Operativa 1997*, 1997.

[PH$^+$96] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell, a non–strict, purely functional language. Version 1.3. Technical report, Yale University, May 1996.

[PKM95] D. Pelta, N. Krasnogor, and P. Moscato. Resultados de la complejidad computacional en el problema de replegado de proteínas. In *II Jornadas de Informática en Investigación Operativa*, Montevideo, Diciembre 1995.

[PKM96] D. Pelta, N. Krasnogor, and P. Moscato. Primeros resultados de un algoritmo determinístico para el problema del protein folding en el modelo H-P. In *IV Jornadas del Grupo Montevideo*, Brasil, 1996.

[PP] Mike Paterson and Teresa Przytycka. On the complexity of string folding. Preprint submitted to Elsevier preprint.

[SSK94] A. Sali, E. Shakhnovich, and M. Karplus. How does a protein fold. *Nature*, 369:248–251, May 1994.

[UM93a] R. Unger and J. Moult. Finding the lowest free energy conformation of a protein is an NP-hard problem: Proof and implications. Technical report, Center for Advanced Research in Biotechnology. University of Maryland, 1993.

[UM93b] R. Unger and J. Moult. Genetic algorithms for protein folding simulations. *Journal of Molecular Biology*, 231:75–81, 1993.

[Wad95] Philip Wadler, editor. *Journal of Functional Programming. Special Issue on State-of-the-art Applications of Pure Functional Programming Languages*, volume 5 (3). Cambridge University Press, July 1995.