Heuristics, Optimizations, and Parallelism for Protein Structure Prediction in CLP($\mathcal{FD}$)

Alessandro Dal Palù, Agostino Dovier
Dipartimento di Matematica e Informatica
Università di Udine
{dalpalu,dovier}@dimi.uniud.it

Enrico Pontelli
Department of Computer Science
New Mexico State University
epontell@cs.nmsu.edu

ABSTRACT

The paper describes a constraint-based solution to the protein folding problem on *face-centered cubic lattices*—a biologically meaningful approximation of the general protein folding problem. The paper improves the results presented in [15] and introduces new ideas for improving efficiency: *(i)* proper reorganization of the constraint structure; *(ii)* development of novel, both general and problem-specific, heuristics; *(iii)* exploitation of parallelism. Globally, we obtain a speed up in the order of 60 w.r.t. [15]. We show how these results can be employed to solve the folding problem for large proteins containing subsequences whose conformation is already known.

Categories and Subject Descriptors

J.3 [**Computer Applications**]: LIFE AND MEDICAL SCIENCES

General Terms

Algorithms,Experimentations

Keywords

Constraint logic programming, parallelism, bioinformatics

1. INTRODUCTION

Proteins are responsible for nearly every function required for life. The sequence of elements (amino acids) identifying a protein is known as the primary (1D) structure. A functional protein can be thought of as a properly folded chain of amino acids in 3-dimensional (3D) space. The 3D structure of a protein characterizes its function. A folded protein interacts three-dimensionally with other proteins (e.g., lock and key arrangements) and this interaction determines the functions of the organism. In fact, an organism is essentially determined by the three-dimensional interactions between proteins and substrates. Thus, without knowing the 3D structure of the proteins coded in a genome, we cannot completely understand the phenotype and functioning of living organisms. Understanding how protein folds has profound implications—e.g., towards the theoretical design of exact drugs, the improvement of proteins functionality, and the precise modeling of cells.

In recent decades, most scientists have agreed that the answer to the folding problem lies in the concept of the *energy state* of a protein. The predominant strategy in solving the protein folding problem has been to determine a state of the amino acid sequence in the 3D space with minimum energy. According to this theory, the 3D conformation that yields the lowest energy state represents the protein's natural shape (a.k.a. the *native conformation*). The energy of a conformation can be modeled using *energy functions*, that determine the energy level based on the interactions between any pairs of amino acids [7]. Thus, we can reduce the protein folding problem to an optimization problem, where the energy function has to be minimized under a collection of constraints (e.g., derived from known chemical and physical properties) [13].

We employ *Constraint Logic Programming (CLP)*, in particular, constraint logic programming over *finite domains* (CLP($\mathcal{FD}$)), to model and solve a tractable representation of the protein folding problem—i.e., protein folding in the context of face-centered cubic lattices [31]. The choice of CLP is natural for a variety of reasons (see, e.g., [2]). CLP is a paradigm that is highly suitable to address optimization problems; it provides declarative and high-level modeling, combined with effective built-in search and resolution strategies. Furthermore, the high-level modeling offered by CLP allows us to easily *add* new constraints and to plug-and-play different search strategies and heuristics.

A preliminary approach to the use of CLP($\mathcal{FD}$) for the protein folding problem has been presented in [15]. In this paper, we elaborate such proposal to consider the issue of *efficiency* and *scalability*. The ultimate objective of this paper is to demonstrate that

- the modeling and optimization capabilities of CLP($\mathcal{FD}$) are highly suitable to tackle the protein folding problem;
- the high-level modeling capabilities allow us to easily add or modify constraints as they become available, as well as to explore the use of different heuristics and search strategies;
- efficiency and scalability can be achieved for realistic problems.

The first step in this process lies in a remodeling of the constraint problem, aim at making the modelization more suitable to the capabilities of current CLP($\mathcal{FD}$) solvers. We also introduce new high-level heuristics for guiding the exploration of the search space, leading to a more effective pruning and enhanced scalability. In particular, we introduce a heuristic called *Bounded Block Fails*. Where the built-in strategies are insufficient to achieve acceptable levels of performance, we introduce the use of *parallelism*, easily exploitable from the high-level search structure generated by the CLP execution. Using a the novel structure, heuristics, and a 14 processors parallel machine, we obtain a speed-up in the order of 60 w.r.t. the performance of the code presented in [15]—running on a single processor of the same parallel machine. The investigation proposed here pushes the built-in capabilities of CLP solvers to, what we believe, are the limits for the problem at hand. The proposed results indicate also that *better performance could be accomplished*, but only at the price of building some of the proposed heuristics at a lower level—i.e., as an ad-hoc constraint solver, and bypassing the built-in strategies of CLP($\mathcal{FD}$). Finally, we demonstrate what, we believe, is one of the greatest application areas for our technology: folding large proteins containing subsequences whose native conformation is already known (e.g., by homology)—e.g., macro blocks linked by a neutral coil. This type of situation is very common; in our framework, the known structures can be directly added as constraints (with *no modifications* to the rest of the constraint model), allowing us to tackle large problems with excellent performance.

1.1 Related Works

The bibliography on the protein folding problem is extensive [8, 28]; the problem has been recognized as a fundamental challenge [23], and it has been addressed with a variety of approaches (e.g., comparative modeling through homology, fold recognition through threading, ab initio fold prediction).

An abstraction of the problem, that has been recently investigated, is the protein folding problem in the *HP* model, where amino acids are separated into two classes (H, hydrophobic, and P, hydrophilic). The goal is to search for a conformation produced by an *HP* sequence, such that most HH pairs are neighboring in a predefined lattice. The problem has been studied on 2D square lattices [14, 22], 2D triangular lattices [1], 3D square models [20], and face-centered cubic lattices (fcc) [25]. Backofen and Will have extensively studied this last problem [3, 4, 5]. The approach is suited for globular proteins, since the main force driving the folding process is the electrical potential generated by Hs and Ps, and the fcc lattices are one of the best and simplest approximation of the 3D space (Sect. 2.2). Compared to the work of Backofen and Will, our approach refines the energy contribution model, extending the interactions between classes H and P to interactions between each pair of amino acids [7]. Moreover, we introduce the possibility to model secondary structure elements, that cannot be reproduced correctly using only a simple energy model as the one adopted by other researchers.

The use of constraint programming technology in the context of the protein folding problem has been fairly limited. Backofen and Will have made use of constraints over finite domains in the context of the *HP* problem [5]. Clark et al. employed Prolog to implement heuristics in pruning a exhaustive search for predicting α-helix and β-sheet topology from secondary structure and topological folding rules [12]. Distributed search and continuous optimization have been used in ab initio structure prediction, based on selection of discrete torsion angles for combinatorial search of the space of possible protein foldings [18].

2. PROBLEM MODELING

2.1 The Protein Folding Problem

The *Primary* structure of a protein is a sequence of linked units (*amino acids* or *residues*) of a given length. The amino acids can be identified by an alphabet $\mathcal{A}$ of 20 different symbols, associated to specific chemical-physical properties. A protein has a high degree of freedom, and its 3D conformation is named *Tertiary* structure.

From the *energy* point of view, the molecule tends to reach a conformation with a the minimal value of free energy (*Native* conformation). Native conformations are largely built from *Secondary Structure elements* (i.e., helices and sheets) often arranged in well-defined motifs (see Fig. 1, α-helices in red—dark grey and β-sheets in yellow—light grey). α-helices are constituted by 5 to 40 contiguous residues arranged in a regular right-handed helix with 3.6 residues per turn. β-sheets are constituted by extended strands of 5 to 10 residues. Each strand is made of contiguous residues, but strands participating in the same sheet are not necessarily contiguous in sequence. Algorithms, e.g., based on neural networks, have been developed that are capable to predict the secondary structure of a protein with high accuracy (75% [8]).

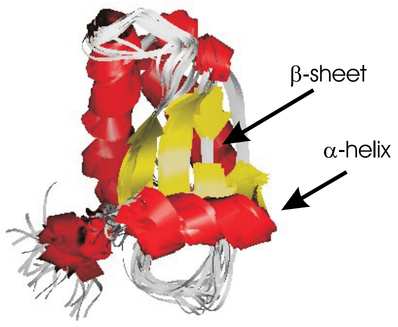

Figure 1: Secondary Structure Elements (protein 1D6T)

Another important structural feature of proteins is the capability of cysteine residues of covalently bind through their sulphur atoms, thus forming disulfide bridges, which impose important constraints on the structure (also known as `ssbonds`). This kind of information is often available, either through experiments or predictions.

Several models have been proposed for reasoning about the 3D properties of proteins—i.e., dealing with the *Tertiary Structure*. Given a primary sequence $S = s_1 \cdots s_n$, with $s_i \in \mathcal{A}$, let us represent with $\omega(i)$ the *position* of a point representing the amino acid s_i in space; $\omega(i)$ is a vector $\langle x_i, y_i, z_i \rangle \in \mathcal{D}$, for a given space domain $\mathcal{D}$. The values x_i, y_i, and z_i can be real numbers—in models in which proteins are free of taking any positions in space—or integer numbers—in models where amino acids can assume only a finite number of positions within a suitable lattice structure. We call $\mathcal{D}$ the set of admissible points.

Given two points $\omega_1, \omega_2 \in \mathcal{D}$, we indicate with $\mathtt{next}(\omega_1, \omega_2)$ the fact that the two points are admissible positions for two amino acids that are contiguous in the primary sequence. It is assumed that two consecutive amino acids are always separated by a fixed distance.

We also employ the binary predicate `contact`, which is used to describe the fact that two amino acids are sufficiently close to be able to interact, and thus they contribute to the energy function: two non-consecutive amino acids s_i and s_j in the positions $\omega(i)$ and $\omega(j)$ are in contact (denoted by $\mathtt{contact}(\omega(i), \omega(j))$) when their distance is less than a given threshold. Lattice models (defined in the next subsection) simplify the definitions of `next` and `contact`.

Given a primary sequence $S = s_1 \cdots s_n$, with $s_i \in \mathcal{A}$, a *folding* of S is a function $\omega : \{1, \dots, n\} \to \mathcal{D}$ such that:

1. $\mathtt{next}(\omega(i), \omega(i+1))$ for $i = 1, \dots, n-1$, and
2. $\omega(i) \neq \omega(j)$ for $i \neq j$ (namely, ω introduces no loops).

A simplified evaluation of the energy of a folding can be obtained by observing the *contacts* present in the folding. In particular, every time a contact between a pair of amino acids is detected, a specific energy contribution is applied towards the global energy. These contributions can be obtained from tables developed using statistical methods applied to structures obtained from X-Rays and Nuclear Magnetic Resonance experiments [21, 7]; these tables associates an energy measure to each pair of non-consecutive amino acids when they are in contact. We denote with $\texttt{Pot}(s_i, s_j)$ the energy contribution associated to the amino acids s_i and s_j (the order does not matter).

The *protein structure prediction problem* can be modeled as the problem of finding the folding ω of S such that the following energy cost function is minimized:

$$E(\omega, S) = \sum_{1\leq i<n} \sum_{i+2\leq j\leq n} \texttt{contact}(\omega(i), \omega(j)) \cdot \texttt{Pot}(s_i, s_j).$$

With a slight abuse of notation predicate `contact` is here used as a Boolean function. This definition is sufficiently general to cover the case of several spatial models $\mathcal{D}$, such as the fcc lattice and the cubic lattice [13].

2.2 Lattice Models

Lattice models have long been used for protein structure prediction (see [29] for a survey). In [25] it is shown that the *Face-Centered Cubic Lattice* (fcc) model is a well-suited, realistic model for 3D conformations of proteins. The model is based on cubes of size 2, where the central point of each face is also admitted. The domain $\mathcal{D}$ consists of the set of triples $\langle x, y, z\rangle$, where $x, y, z \in \mathbb{N}$ such that $x+y+z$ is even (see Fig. 2). Points at Euclidean distance $\sqrt{2}$ are linked; their distance is called *lattice unit*. Observe that, for linked points i and j, it holds that $|x_i-x_j|+|y_i-y_j|+|z_i-z_j| = 2$.

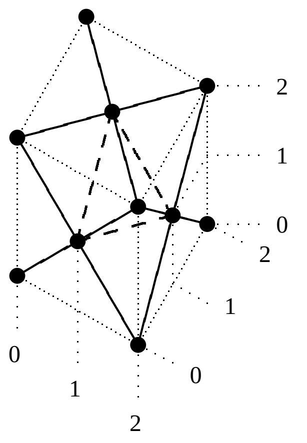

Figure 2: A cube of the fcc lattice. Thick lines link connected points. Dashed lines represent connections inside the cube.

Each point is adjacent to 12 neighboring points. Thus, we define the predicate `next` as follows: $\texttt{next}(\omega(i), \omega(i+1))$ holds iff

- $|x_i-x_{i+1}| \in \{0,1\}, |y_i-y_{i+1}| \in \{0,1\}, |z_i-z_{i+1}| \in \{0,1\}$,
- $|x_i - x_{i+1}| + |y_i - y_{i+1}| + |z_i - z_{i+1}| = 2$.

In fcc lattices, the angle between three adjacent residues may assume values 60°, 90°, 120°, and 180°. Volumetric constraints and energetic restraints in proteins make values 60° and 180° infeasible. Therefore, in our model, we retain only the 90° and 120° angles [31, 17]. No similar restriction exists on torsional angles among four adjacent residues. In detail, let $\vec{v}_{i-1,i} = \omega(i)-\omega(i-1)$ and $\vec{v}_{i,i+1} = \omega(i+1) - \omega(i)$. To impose that the angle between them can only be of 90° and 120°, we use the scalar product between these two vectors: $\vec{v}_{i-1,i} \cdot \vec{v}_{i,i+1} = |\vec{v}_{i-1,i}||\vec{v}_{i,i+1}| \cos(\theta)$. Thus, since $|\vec{v}_{i-1,i}| = |\vec{v}_{i,i+1}| = \sqrt{2}$ we only need to impose that: $\vec{v}_{i-1,i} \cdot \vec{v}_{i,i+1} \in \{1, 0\}$.

A *contact* between two non-adjacent residues in fcc occurs when their separation is two lattice units—i.e., viewing the lattice as a graph whose edges connect adjacent points in the lattice, the positions of the residues are connected by a path of length 2. Physically, two amino acids in contact cannot be at the distance of a single lattice unit, because their volumes would overlap. Consequently, we impose the constraint that two non-consecutive residues s_i and s_j must be separated by more than one lattice units. This is achieved by adding, for the pair i and j, the constraint (called `non_next`):

$$(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 > 2$$

With these additional constraints, we can define:

$$\texttt{contact}(\omega(i), \omega(j)) \text{ iff } |x_i - x_j| + |y_i - y_j| + |z_i - z_j| = 2$$

3. A CLP($\mathcal{FD}$) IMPLEMENTATION

The formalization of the protein structure prediction problem in fcc has been instantiated in a declarative program in CLP over finite domains—and it follows the basic structure outlined in [16, 15]. In this section, we describe the main predicates employed in such implementation. We make use of the library CLP($\mathcal{FD}$) of SICStus PROLOG 3.12.0 [10] and the library `ic` of ECLiPSe 5.8 [11]. Complete code and other related material can be found in: `www.dimi.uniud.it/dovier/PF`.

3.1 Basic Constraints

The program has the classical *Constrain & Generate* [2] structure:

```
constrain(Primary,Secondary,Matrix,Tertiary,...),
labeling(Primary,Secondary,Matrix,Tertiary,...)
```

The `constrain` predicate deterministically introduces the constraints for the variables involved, while `labeling` searches for the solution in the search space (through chronological backtracking). The inputs are `Primary` (a list of amino acids), `Secondary` (a list of known secondary structure components), and `Matrix` (the matrix of energy contributions, see Section 3.2). `Tertiary` is the output list of positions (a flat list of triples of integers) of the conformation and `Energy` is the output value of energy associated to such conformation. We do not discuss here the other, more technical, parameters. The predicate `constrain` is defined in Fig. 3. Given the input list of amino acids $\texttt{Primary} = [s_1, \ldots, s_N]$, the predicate `generate_tertiary` creates the list

$$\texttt{Tertiary} = [X_1, Y_1, Z_1, \ldots, X_N, Y_N, Z_N]$$

of $3N$ variables. The predicate `domain_bounds` specifies the domains for the X_i, Y_i, Z_i variables (the range $0 \ldots 2 * N$), and adds the constraints forcing $X_i + Y_i + Z_i$ to be even.

`avoid_self_loops` forces all triples to be distinct (we employ the built-in predicate `all_different` on the list $[B_1, .., B_n]$ where

$$B_i \texttt{ \#= } (X_i * P * P) + (Y_i * P) + Z_i$$

for a suitable value P). `next_constraints` imposes that the points $[X_i, Y_i, Z_i]$ and $[X_{i+1}, Y_{i+1}, Z_{i+1}]$ are adjacent in the lattice. `distance_constraints` defines that two non-consecutive points are at a lattice distance greater than 1. `angles` defines the admissible angles formed by three consecutive amino acids. `compact_constraints` introduces a user-defined maximal distance between amino acids (called *compact factor*).

`secondary_info` encodes the Secondary Structure information as constraints in the program. The `Secondary` structure is described by a list of elements of the type:

$\texttt{helix}(i, j)$: $s_i, s_{i+1}, \ldots, s_j$ form an α-helix.

```
constrain(Primary, Secondary, Energy, Tertiary, Matrix, Compact, OrdTertiary):-
     length(Primary,N),
     generate_tertiary(N, Tertiary), domain_bounds(Tertiary,N),
     avoid_self_loops(Tertiary,N), next_constraints(Tertiary),
     distance_constraints(Tertiary), angles(Tertiary),
     compact_constraints(Tertiary,N,Compact), secondary_info(Secondary, Tertiary),
     avoid_symmetries(Secondary, Tertiary, SSP),
     define_variables_order(Secondary,Tertiary, SSP, OrdTertiary),
     energy_constraints(Primary, Tertiary, Secondary, Energy, Matrix).
```

Figure 3: constrain predicate

strand(i,j): $s_i, s_{i+1}, \ldots, s_j$ are in a β-strand.
ssbond(i,j): presence of a disulfide bridge between s_i and s_j.
In our tests, we retrieved these information from the Protein Data Bank [6]; it is easy to modify the code to obtain such information from secondary structure prediction programs (e.g., [26]).

avoid_symmetries removes admissible conformations that are equivalent to others modulo symmetries and/or rotations. The predicate selects three particular consecutive amino acids. We set the position of the first of them removing all foldings equivalent modulo translations. We fix also the positions of the second and the third point in order to remove all equivalences modulo rotations.

define_variables_order makes use of the distribution of secondary structure elements to sort the variables in Tertiary for labeling purposes (see Section 4.1).

3.2 Contacts and Energy Constraints

As described above, two amino acids s_i, s_j are in contact if a path of length 2 lattice units links them. The contact information is maintained in a symmetric matrix M, such that $M[i,j]$ is the energy contribution provided by s_i and s_j. The following code constrains the contact contribution to the matrix element:

```
table(si,sj,Pot), M[i,j] in {0,Pot},
2 #= abs(Xi-Xj) + abs(Yi-Yj) + abs(Zi-Zj)
       #<=> M[i,j] #= Pot.
```

where table reports the value Pot(s_i, s_j) as computed in [7]. Values of Pot have been scaled w.r.t. [7] to have only integer values. The global energy is the sum of the elements in M. The optimal folding is reached when the global energy is minimal. During the *labeling* phase, the information stored in M is used to control the minimization process and to prune the search tree.

Various heuristics have been developed, aimed at simplifying the computation of the contact matrix M—see also [15, 16]. In particular, we restrict our attention to contact contributions coming from amino acids s_i, s_j that are *not* included in the same secondary structure element. This is justified by the fact that contact contributions from amino acids within the same secondary structure element is constant in each folding.

To further reduce the search space, it is possible to ignore the contact contributions for all pairs s_i, s_j such that $j < i + k$, where k is a parameter. In those cases, the constraint is not applied and $M[i,j] = 0$. When we increase the value k, we generate a simpler global constraint and, at the same time, we consider only contributions from distant amino acids (considered, from the biological perspective, more relevant).

Let us now describe how the matrix M is used to compute the global energy. As mentioned earlier, the global energy is the sum of the elements of M. To simplify the computation, we first proceed to collect in a list (PotentialList) all the unbound elements present in M (i.e., contributions that are still unknown). The list is used in the heuristics and also to assert the constraint:

```
    sum(PotentialList, #=, Energy).
```

Note that the constraints imposed by the predicates described above, combined to the constraint propagation, have as a consequence that many elements in M receive a binding before the beginning of the labeling phase. The net effect is to further restrict the number of variables that need to be instantiated by the labeling procedure.

In order to further prune the search tree, we introduced in [16] a local labeling heuristic. After each instantiation of t variables,[1] we store the best known ground admissible solution, its energy, and its associated potential matrix. The idea is to compare the current state to saved history, and decide if it is reasonable to cut the search tree. We consider the ground set of elements in the current potential matrix, CurrentGround, and compare them to the corresponding set present in the best solution found so far, BestGround. According to the ratio of ground variables in PotentialList, a coefficient can be computed to control the pruning process:

```
best_solution(BestEnergy),
best_potentialList(BestPL),
sumgrounds( PotentialList, BestPL,
            CurrentGround, BestGround),
term_variables(PotentialList,PlV),
length(PlV,LPotVar),
length(PotentialList,LPotTot),
CoefS1 is (0.5 * (LPotVar) +
          1.1*(LPotTot-LPotVar))/(LTotPot),
S1 is integer( CoefS1 * BestGround),
CurrentGround < S1,
```

where term_variables is a built-in predicate to extract (in constant time) the list of non-ground variables. In this version, if the set of ground elements is small, then we allow the computation to continue even in presence of significant differences in energy from the best known solution (0.5 compared to the best known), with the hope that this choice will pay off in the rest of the matrix to be discovered. When the protein is almost folded, the coefficient becomes higher and, consequently, we prune many solutions that are clearly not improving the local minimum, at that moment.

4. THE NEW IMPLEMENTATION

In this section we present the improvements w.r.t. the code presented in [15], briefly described in the previous section, to seek better performance and scalability to larger proteins.

4.1 Constraints Redesign

In our previous experiments, we observed that one of the causes behind the lack of scalability of the constraint system of Sect. 3 to large proteins is the limited propagation achieved in presence of non-linear constraints (e.g., constraints related to torsion angles, as they require vector scalar products). The first stage of redesign implied removing as many non-linear constraints as possible and enhancing the propagation structure.

[1] t is a parameter of the program.

4.1.1 Angle-free Encoding

The constraint structure proposed in [15] to solve the problem, imposed an alternative local coordinate system, used for the description of the tertiary structure; this coordinate system is composed of a sequence of the torsion angles forming the protein. The secondary structure elements, due to their regular shape, were simply described by a regular sequence of angles. The angles and coordinates descriptions were linked via an additional set of constraints. We noticed that, even though the resulting model was clean and declarative, the resulting constraints caused a bottleneck during the search because of poor propagation.

In the current version, we opted to remove the torsion angles description of secondary structure elements and restrict our description of the tertiary structure only to Cartesian coordinates. We simplify and remodel the description of secondary structure elements.
α-helices: the modeling of α-helices builds on the observation that it is sufficient to constrain the first 5 amino acids of the helix to guarantee its shape—the shape can then be propagated to the rest of the helix via simple vector equalities. More precisely, given the set of the first 5 amino acids $\langle X_1, Y_1, Z_1\rangle, \ldots, \langle X_5, Y_5, Z_5\rangle$ of a helix, the following constraint describes the helical organization in terms of relative distances between the points in the fcc lattice:

```
Dx #=X5-X1, Dy #=Y5-Y1, Dz #=Z5-Z1,
Dx1#=X3-X1, Dy1#=Y3-Y1, Dz1#=Z3-Z1,
Dx2#=X5-X3, Dy2#=Y5-Y3, Dz2#=Z5-Z3,
Dx3#=abs(X4-X1), Dy3#=abs(Y4-Y1), Dz3#=abs(Z4-Z1),
Dx4#=abs(X5-X2), Dy4#=abs(Y5-Y2), Dz4#=abs(Z5-Z2),
count(0,[Dx,Dy,Dz],#=,1),
count(0,[Dx1,Dy1,Dz1],#=,2),
count(0,[Dx2,Dy2,Dz2],#=,2),
Dx3 #=<2, Dy3 #=<2, Dz3 #=<2,
Dx4 #=<2, Dy4 #=<2, Dz4 #=<2.
```

Because of constraints interaction,`[Dx,Dy,Dz]` results to be a vector $(\pm 2, \pm 2, 0)$ in any coordinate permutation, that represents the placement of each next pattern of 4 points. For every amino acid s_i in position X_i, Y_i, Z_i, the position of s_{i+5} will be $X_i + Dx, Y_i + Dy, Z_i + Dz$. Constraints of the form
Dx #= $X_{i+5} - X_i, Dy$ #= $Y_{i+5} - Y_i, Dz$ #= $Z_{i+5} - Z_i$
are added to every s_i, s_{i+5} in the helix to ensure the propagation of the helix shape.
β-strands: the modeling of a β-strand is considerably simpler: given 3 consecutive amino acids ($X_1, Y_1, Z_1, \ldots, X_3, Y_3, Z_3$), in order to form a 90° angle, the first and last point must have two identical coordinates and one coordinate at distance 2:

```
    Dx #= X3-X1, Dy #= Y3-Y1,Dz #= Z3-Z1,
    count(0,[Dx,Dy,Dz],#=,2),
    abs(Dx+ Dy+ Dz) #=2,
```

Once again, the correct shape is propagated to the other amino acids in the strand by vector equalities, using a step of three amino acids.
ssbonds: the original implementation of `ssbond` structures was achieved by imposing a maximal Euclidean distance of 6 between the two residues linked connected by a disulfide bridge. We removed the Euclidean distance of 6, in favor of the following simpler approximation:

```
Dx#=abs(X1-X2), Dy#=abs(Y1-Y2), Dz#=abs(Z1-Z2),
Dx #=< 4, Dy #=< 4, Dz #=< 4.
```

4.1.2 Variable Selection Strategy

The second improvement to the constraint organization lies in the development of an alternative strategy for the selection of variables during labeling. In the original constraint scheme proposed in [15], the variable selection is dynamic. The strategy employed builds on the idea of always selecting, for labeling, a variable which is adjacent to a ground subsequence of the primary sequence. Basically, the method tries to grow the ground part of protein until an acceptable solution is found.

The solution we propose here, instead, relies on precomputing the order of labeling of the variables; the intuition is the desire to avoid the run-time costs of the previous selection strategy. In the constrain phase, we select the longest secondary structure, and we assign to it ground values. The protein is then partitioned in five consecutive parts:

- *Left Tail*—containing no secondary structure elements;
- *Left Body*;
- *Longest Secondary Structure Element*—ground;
- *Right Body*;
- *Right Tail*—containing no secondary structure elements.

The exploration order used during labeling is the following: *(i)* Left Body first (in reverse order), *(ii)* Right Body, *(iii)* Left Tail (in reverse order), and *Right Tail*. The idea is similar to our previous approach, but we save the time to seek for the next variable to select. The tails are free to assume every conformation—they do not contain any superimposed pattern. Since the main energy contribution is given by the protein body, the tails are left to be instantiated at the end.

4.1.3 Contact revisited

In addition to this, we also introduce a modification in the computation of the energy function w.r.t. Sect. 3.2; specifically, we ignore the energy contributions provided by amino acids at distance less or equal than three in the primary sequence. Due to our model, three consecutive amino acid, s_i, s_{i+1}, s_{i+2}, can form an angle of either 90° or 120°. Given the definition of `contact` and the energy contribution, amino acids forming a 90° angle are such that s_i and s_{i+2} are in contact, thus, they provide an energy contribution. On the other hand, when they form a 120° angle, there is no energy contribution, since s_i and s_{i+2} are separated by a distance larger than the minimum contact distance of 2. This fact shows that relevant contributions to the global energy come from the local angles formed by the amino acids.

The overall effect of this simplification is to exclude every contribution raising from the fact that three consecutive amino acids form a 90° angle. The intuition of our choice is that the "important" energy contributions that characterize a folding are those provided by pairs of amino acids that are "far apart" in the primary sequence. Indeed, we observed that the contributions deriving from local subsequences tend to mask the global energy evaluation and thus bias our search heuristic. This energy function better reflects the quality of the folding, even if the model is still too simple for a direct relation between our energy and the *RMSD* (*Root-Mean-Square Deviation*, the typical measure of structural diversity, Sect. 4.2.2).

4.1.4 Experimental Results

Let us discuss the impact of these new ideas on the performance of the computation. Table 1 compares the execution times of the old constraint structure of [15] and the modified one on different proteins. For a fair comparison, we have updated the contact constraints of the code [15] as described in Sect. 4.1.3. This explains the differences between results in column "Old version" and those reported in Table 3 for the code [15]. Both the codes include the same heuristics described in Section 3.2. The experiments have been performed on a 3GHz Pentium under Linux and they highlight a marked improvement in performance; only in one case we observed a slow-down, while in many cases we achieved from 4 to over 50 fold speedup. On the other hand, although the results are good for relatively small proteins, the new method are not sufficient to produce significant speedups on bigger proteins. In the next section, we introduce a new search heuristic that, combined with the ideas introduced here, is capable to efficiently explore the search tree generated by larger proteins.

ID	Size	New version	Old version	Speedup
1LE0	12	0.63 s	2.9s	4.7
1KVG	12	2.84 s	11.4s	4.0
1LE3	16	4.85 s	4.5s	0.9
1EDP	17	0.66 s	34.9s	52.9
1PG1	18	0.7 s	18.8s	26.8
1ZDD	34	3m 37s	29m21s	8.1

Table 1: Performance of the Modified Constraint Structure

4.2 Bounded Block Fails heuristic

In this section, we present a novel heuristic to guide the exploration of the search tree, called *Bounded Block Fails (BBF)*. This technique is general and can be applied to every search with a fixed ordering of variables—though it is particularly effective when applied to the protein folding problem at hand.

The heuristic involves the concept of *block*. Let us assume that V is a list $[V_1, \ldots, V_n]$ of variables and constants. A block B_i is a sublist of V of size k composed of unbound variables. The concatenation of all the blocks $B_1 B_2 \ldots B_\ell$ gives the ordered list of unbound variables present in V, where $\ell \leq \lceil \frac{n}{k} \rceil$. The blocks are selected dynamically, and they could exclude some of the original variables, that have already been instantiated due to constraint propagation. The number of blocks, thus, could be less than $\lceil \frac{n}{k} \rceil$ and it could be not constant during the whole search. In Figure 4 we depict a simple example for $k = 3$: we consider a list of 9 variables. The dark boxes represent ground assignments.

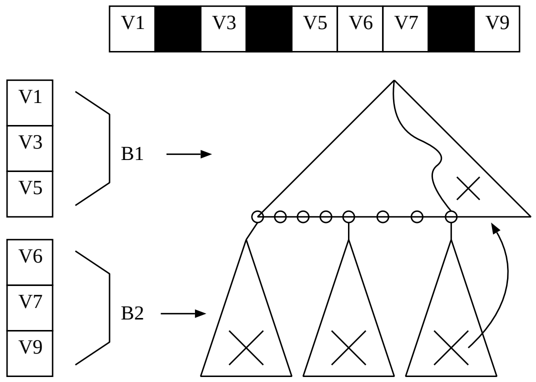

Figure 4: The BBF heuristics

The heuristics consists of splitting the search among the ℓ blocks. Internally, each block B_i is individually labeled according to the desired labeling strategy—in our case, the same heuristic employed in [15]. When a block B_i has been completely labeled, the search moves to the successive block B_{i+1}, if any. If the labeling of the block B_{i+1} fails, the search backtracks to the block B_i. Here there are two options: if the number of times that B_{i+1} completely failed is below a certain threshold t_i, then the process continues, by generating one more solution to B_i and re-entering B_{i+1}. Otherwise, if too many failures have occurred, then the Bounded Block Fail heuristic generates a failure for B_i as well and backtracks to a previous block. Observe that the count of the number of failures includes both the regular search failures as well as those caused by the Bounded Block Failure strategy. The list $t_1, \ldots, t_\ell$ of thresholds determines the behavior of the heuristic. In the Figure, we assume $t_1 = 3$; the figure shows that, after the third failure of B_2, the search on B_1 fails as well.

The BBF heuristic is effective whenever:

- suboptimal solutions are spread sparsely in the search tree;
- for each admissible solution, there are many others with small differences in variables assignments and energy.

In these cases, we can afford to skip solutions when generating block failure, because some others are going to be discovered following other choices in some earlier blocks. In particular, for our problem, it is reasonable to keep high threshold values for the first blocks, while exploring instead only a small fraction of the search space present in the last blocks. This is justified by the fact that many equivalent solutions can be typically found, just by changing the assignments in the last blocks, while the core of the problem lies in the first blocks.

In general, this technique can be effectively applied every time the variables are associated to some spatial properties and the corresponding physical object are related to each others, like in the case of a chain of amino acids. When assigning positions following the order of the amino acids on the chain, a failure in the current branch, means that the partial conformation does not allow to proceed without a collision. The BBF heuristic suggests to try to revise some earlier choices instead of exploring the whole space of possibilities depending on the block that collects failures. The high density and the great number of admissible solutions allow us to exclude some solutions, depending on the threshold values, and to still be able to find almost optimal solutions in shorter time.

4.2.1 *Experimental Results*

In Table 2, we report the performance obtained by applying the BBF strategy. We compute the execution times and the energy values for a set of proteins, using the implementation described in the previous section, with and without the application of the BBF heuristic. For the BBF heuristic, we use the values $k = 3$, $t_1 = 5, t_{\lceil n/k \rceil} = 3$, which have been experimentally shown to produce good performance. The other integer values t_i are determined as follows: $t_i = (1 - (\frac{i-1}{\lceil n/k \rceil})^2) t_1 + (\frac{i-1}{\lceil n/k \rceil})^2 t_{\lceil n/k \rceil}$. The entries marked (*) indicate timeouts (execution stopped after the specified amount of time).

		No BBF		BBF		
Protein	Len.	Time	Energy	Time	Energy	Speedup
1LE0	12	0.63s	-6799	0.80s	-6251	0.8
1KVG	12	2.84s	-14571	4.12s	-12661	0.7
1LE3	16	4.85s	-11008	1.53s	-11289	3.17
1EDP	17	0.66s	-16259	0.53s	-24492	1.2
1PG1	18	0.70s	-27016	0.83s	-27016	0.8
1E0N	27	1m 16s	-25493	3m	-22308	0.4
1ZDD	34	3m 37s	-9907	1m 51s	-18455	2.0
1VII	36	3h 35m	-24681	3h 40m	-25914	1.0
2GP8	40	1m 24s	-17425	1m 33s	-15187	0.9
1ED0	46	(*) 10h	-27410	1h 43m	-31565	≥ 5.8
1ENH	54	2h 24m	-26909	55m 6s	-28559	2.6
2ERA	61	(*) 10h	-37071	16m 21s	-45006	≥ 36.7

Table 2: Executions with and without the BBF heuristic.

The BBF strategy nicely integrates with the modifications described in Sect. 2. We would like to make two important observations. First of all, the current implementation of the BBF heuristic is performed in SICStus Prolog at a very high-level; e.g., when failing, many `fail` predicates are invoked, with a relatively high cost in the handling of the search tree. This makes the implementation rather inefficient, and this is evident especially for the smaller proteins. In spite of this, there are obvious advantages in terms of execution times and optima found for larger proteins. Clearly, a low-level implementation of the strategy (e.g., like the search strategies in ECLiPSe), will push the speedups to a much higher level. This leads us to the second observation—the BBF heuristics is designed to handle inputs with large sizes. For smaller proteins, the number of blocks tends to be small, and the heuristics accomplishes few block backtracks. In our experiments, we made use of blocks of size 3 (corresponding to the three coordinates of a single amino acid); attempts to reduce this value (i.e., increase the number of blocks) actually produced degradation of performance, due to the excessively small size of the blocks (that defeats the original idea of BBF). To handle larger proteins, it is possible to use larger blocks (e.g. $k = 9$), that include more nodes within the corresponding search subtree. Thus, a failure induced by the BBF strategy prunes a consistent portion of the tree and speeds up the search. To limit the consequent amount of speedup and loss of accuracy, it is advisable to tune properly the parameters t_i (e.g. high values for slow but more accurate results).

In Table 3, we report a comparison between the results obtained in the original constraint structure [15] and the corresponding ones obtained using the ideas reported in Section 4.1 and the BBF heuristic. In the column *CF* we indicate whether a specific compact factor was used (see Section 2)—when blank, we used a (high) default value (see [15] for more details). In the *AA* column, we indicate the number of amino acids in the protein, while in the *Core Zone* column we identify the subsequence of the protein without the tails—since the tails are less stable and less relevant for a quality test. The *RMSD* column reports the RMSD values (between brackets we indicate the RMSD for the Core Zone) of the computed solution w.r.t. the native structure deposited in the PDB (c.f. Section 4.2.2). At the bottom of the table (marked with (**)), we report two larger proteins, in order to demonstrate the power of the BBF heuristic, using larger size of the BBF blocks ($k = 9$). The thresholds are set to $t_1 = 4$ and $t_{\lceil n/k \rceil} = 2$. The foldings of these proteins were beyond the capabilities of the original constraint structure, while the computation time is extremely low using BBF; furthermore, the RMSD errors are also sufficiently low, thus making this folding a reasonable input to a molecular dynamics refinement step.

We also compare some of our results with those returned by the HMMSTR/Rosetta Prediction System [27]. This program does not use a lattice as underlying model: aminoacids can take any position in $\mathbb{R}^3$. We have used it as an *ab-initio* predictor (precisely, we have disabled the *homology* and *psi-blast* options). The comparison is obviously not fair because in our case secondary structure is known and not predicted. Times are obtained from the result files, but it is not clear to which machine they refer. Results are in Table 4. HMMSTR prediction runs presumably faster, but our predictions (which includes known secondary structure) improve the RMSD for bigger proteins.

Name	N	Our		Rosetta	
		Time	RMSD	Time	RMSD
1ZDD	34	1m.51s.	5.2	5m.35s.	3.5
1VII	36	3h.40m.	5.6 (4–32)	5m.35s.	4.2
1E0M	37	4h.30m.	6.3 (8–29)	6m.35s.	7.7
2GP8	40	1m.33s.	3.6 (6–38)	6m.35s.	6.4
1ED0	46	1h.43m.	6.2 (3–40)	7m.23s.	8.9

Table 4: Comparisons with Rosetta predictions

4.2.2 RMSD: Discussion

The *root-mean-square deviation (RMSD)* is the common measure of the *structural diversity* of two proteins and it measures the average distance between the atoms of two optimally aligned sets of amino acids. In our specific case, the reference model is taken directly from the sequences in the Protein Data Bank [6] (where a full atom model with real coordinates can be found).

When comparing this model to our prediction, we have to consider some important issues. First of all, our prediction deals with an approximation of each amino acid (i.e., with its α-carbon). When we compare the results, we consider the distance between the pairs of corresponding carbons in the models. Basically, we extract the backbone of the original protein, in order to obtain an equivalent chain of α-carbons.

The RMSD measure contains some intrinsic and unavoidable parts, that derive from the discretization of the backbone on the fcc lattice. Clearly, this reduces the possible placements in the space and, at the same time, forces the backbone to adapt to the closest position to the optimal folding. To quantify how this problem reflects on the RMSD measure, we consider a protein from the PDB, we select its α-carbons of the backbone, and we place them on the lattice. The placement fulfills our standard formalization of chain neighbors and allowed angles in the lattice. To maintain the folding properties in the lattice, we compute the Euclidean distance for each pair of amino acids and we add a constraint on the corresponding discrete pair. The quadratic number of distances is sufficient to recover the protein shape in the real space. Due to discretization, the distance on lattice cannot be kept exact and thus we relax the constraint including a *range* of allowed distances.

We tested the mapping on fcc with two short proteins: 1EDP (17 amino acids) and 1ZDD (34 amino acids). For 1EDP, the RMSD error is 6.58Å and for 1ZDD is 3.64Å. These values are relatively high, even if the fcc lattice is considered one of the best known discrete approximations. 1ZDD is composed mainly of α-helices, that can be well approximated on the lattice. Nevertheless, the error is still of the order of the distance between two consecutive amino acids. These considerations stress the fact that our results are affected by this kind of systematic errors, which are inherent in the use of the fcc lattice and independent from the quality of our solution strategies. As shown in [15], it is possible to eliminate these errors by running some steps of molecular dynamics—e.g., using the Generalized Born implicit solvent method [24] as implemented in the program CHARMM [9], applied to the initial fcc disposition of the backbone. After this relaxation, the molecule with a full atom description is closer to the local minimum of energy and, in general, eliminates the artifacts coming from the use of a discrete lattice structure.

4.3 SICStus vs ECLiPSe

The results presented in the previous sections were based on the CLP($\mathcal{FD}$) library provided by the SICStus Prolog system [10]. Another effective solver for finite domain constraints is offered by the ECLiPSe system [11]. In this section, we explore the impact of the different solvers on the performance of our protein folding problem solution. In particular, we tested the library CLP($\mathcal{FD}$) of SICStus Prolog 3.12.0 [10] and the libraries *ic* and *branch_and_bound* of ECLiPse 5.8 [11]. The tests have been performed using the optimized constraint system described in the previous sections. The aim is to compare the features of the two constraint solvers and, at the same time, understand which search strategy fits better to the problem.

As first test, we compared the efficiency of the built-in branch-and-bound solver. We coded the same program in SICStus and ECLiPSe—in the first case using the `minimize` option offered by the built-in `labeling` predicate, that starts a branch and bound

			Results from [15]			New results with BBF			Core	
Protein	CF	AA	Time	Energy	RMSD	Time	Energy	RMSD	Zone	Speedup
1LE0		12	1.3s	-9040	2.8 (2.6)	0.80s	-6251	3.9 (3.2)	2-11	1.6
1KVG		12	7.3s	-14409	2.7 (2.4)	4.12s	-12661	4.1 (4.1)	2-15	1.8
1LE3		16	2.3s	-13653	3.0 (2.7)	1.53s	-11289	4.6 (3.4)	3-11	1.5
1EDP		17	20.4s	-19389	4.3 (1.1)	0.53s	-24492	4.0 (1.1)	9-15	38.4
1PG1		18	14.6s	-10126	6.0 (5.2)	0.83s	-27016	4.2 (3.2)	4-17	17.6
1E0N		27	7m 54s	-12029	5.2 (5.1)	3m	-22308	6.4 (4.5)	3-24	2.6
1ZDD		34	17m 25s	-22350	4.0 (4.0)	1m 51s	-18455	5.3 (5.2)	5-34	9.4
1VII		36	7h 42m	-26408	10.2 (7.8)	3h 40m	-25914	6.0 (5.6)	4-32	2.1
1VII	0.3	36	3h58m	-28710	8.0 (7.4)	22m 41s	-19181	8.2 (8.1)	4-32	10.5
1E0M		37	(*) 24h	-30163	8.9 (4.4)	4h 35m	-26745	9.2 (6.3)	8-29	$\geq$ 5.2
2GP8		40	10h 39m	-29196	4.9 (3.5)	1m 33s	-15187	6.6 (3.6)	6-38	412.3
1ED0		46	9h 38m	-38218	8.0 (7.2)	1h 43m	-31565	6.9 (6.2)	3-40	5.6
1ENH		54	(*) 24h	-23373	9.9 (8.6)	55m 6s	-28559	11.1 (9.5)	8-52	$\geq$ 26.1
6PTI	0.25	58	(*) 48h	-42096	9.7 (9.4)	(*) 48h	-52258	8.0 (7.9)	3-55	1
2IGD	0.17	60	4h 59m	-40588	12.6 (11.5)	2h 35m	-45462	10.6 (10.5)	6-59	1.9
2ERA	0.19	61	(*) 1000s	-38138	11.6 (10.6)	(*) 1000s	-45006	11.9 (11.7)	3-55	1
1SN1	0.25	63	(*) 10h	-47121	8.6 (8.1)	(*) 10h	-47650	9.1 (9.2)	2-51	1
1YPA	0.17	63	(*) 10h	-60244	12.9 (9.8)	(*) 10h	-45617	11.5 (10.5)	12-52	1
1FVS	0.15	72	(**)			11m 49s	-58587	13.1 (13.5)	13-70	-
1B4R	0.16	80	(**)			25m 47s	-78140	13.1 (13.1)	2-79	-

Table 3: Computational results and comparisons.

search; for ECLiPSe we invoked a `complete` search of *ic* as parameter of the `bb_min` predicate in the *branch_and_bound* library. Table 5 compares the results. On average, the SICStus solver performs 12.3 times faster than the ECLiPSe solver. For these tests, we make use of a 2.4GHz Pentium 4 PC.

ID	AA	SICStus	ECLiPSe	Optimum	Ratio
1LE0	12	25.7 s	314 s	-9072	12.22
1KVG	12	23.2 s	319 s	-14571	13.75
1LE3	16	85.0 s	990 s	-11840	11.65
1PG1	18	6.3 s	70.5 s	-27853	11.14

Table 5: ECLiPSe vs. SICStus branch-and-bound performance

On the other hand, ECLiPSe offers a number of additional built-in search heuristics, that can be selected when solving a minimization problem. We tested the following built-in strategies: the *Limited Discrepancy Search (LDS)*, the *Bounded Backtracking Search (BBS)*, and the *Depth Bounded Search (DBS)*—together with LDS.

The LDS strategy looks for neighbors of an admissible solution that differ only in a limited number of different choices (discrepancies) in labeling [32]. The BBS strategy limits to the number of backtracking steps performed in the search tree. The DBS strategy expands completely a specified number of levels from the root and then explores the remaining tree with a specified technique (LDS in our experiments).

In Tables 5 and 6 we report the results obtained from various benchmarks using the different ECLiPSe heuristics. In these experiments we made use of the optimized constraint structure but *without* the BBF strategy. Table 5 compares SICStus to the LDS and the BBS strategies. Table 6 focuses on the DBS strategy. Even if the SICStus solver is faster than ECLiPSe, it does not offer a comparable selection of built-in search strategies. Our pruning heuristic is implemented at a very high level, and thus not as efficient as possible. Nevertheless, it performs better in terms of time and best solution found. None of the tested ECLiPSe strategies is able to produce faster *or* better results than our heuristic in SICStus.

One of the problems we observed is that the ECLiPSe heuristics do not properly fit the problem. Conceptually, we would like to explore the space considering that:

- small changes in the protein folding are not relevant to the global energy—which is the opposite of what the LDS strategy does;

	1PG1 (54 Vars)		1E0N (81 Vars)	
Heuristic	Time	Energy	Time	Energy
SICStus	0.73	-27016	78.0	-25493
Lds(0)	0.22	-20767	No	No
Lds(1)	1.79	-21412	11.17	-13715
Lds(2)	8.8	-27082	93.0	-14958
Bbs(10)	0.3	-20676	1.3	-12017
Bbs(100)	2.17	-22253	5.6	-12017
Bbs(1000)	40.8	-27853	75.0	-17221

Figure 5: SICStus and ECLiPSe times (in seconds)

- once a solution is found, it is likely that another interesting solution lies in a complete different part of the search tree—which is in contrast to the BBS strategy, that performs a fixed number of backtracks, and thus it keeps the search in the proximity of the solution found;
- the complete exploration of the first levels of the tree (as done in DBS) is time consuming, and a more selective heuristic should be employed.

1E0N	Y=0		Y=1		Y=2	
	Time	Energy	Time	Energy	Time	Energy
X=1	No	No	21.5	-13715	166	-15382
X=2	No	No	30.3	-12688	219	-15382
X=3	No	No	45.9	-14052	326	-18046

Figure 6: Strategy Dbs(X,lds(Y)) test. Times in seconds

Nevertheless, the experiments performed with ECLiPSe highlight that some of the search strategies (e.g., the BBS strategy) can produce solutions that are suboptimal in terms of global energy, but in a significantly shorter period of time. This could be useful in situations where an optimal solution requires too much time and an approximation is satisfactory to work with. These considerations suggest also that a more efficient search could be performed on a specific solver in which we handle the search tree at low level and implement some new heuristics, more suitable to our problem.

4.4 Further Experiments

In this section, we summarize some additional promising ideas that we explored in this project; in spite of looking theoretically promising, once implemented, these strategies failed to provide a benefit. They are not part of our system, but they deserve to be discussed.

In our approach we explore the search space on the lattice structure, instantiating variables with ground values drawn from their finite domains. Another approach is to bisect the search space at each labeling step, non-deterministically adding some extra constraints that specify, e.g., whether a variable is even or odd. The idea is that the interaction of these constraints should efficiently prune the search space, and allow propagation to force more variables to ground values. The idea seems promising, since the fcc lattice relies on properties based on the even relationships of its points. Unfortunately, the implementation of these idea provided relatively poor performance. The reasons can be found in the lack of propagation due to the distance between the points that get assigned values.

We also tried to investigate the order in which variables should be explored. We tried to first label the X coordinates of every point, and later the other two coordinates, in the hope that some propagation could already prune parts of the search space. The intuition is that the lattice structure creates tight dependences between coordinates; thus, fixing one coordinate is expected to quickly affect the others. Also in this case, the resulting performance was very poor—mostly due to lack of propagation (fixing one coordinate is not sufficiently strong to propagate on the neighboring points).

We explored other variations in the order of labeling of the variables. We considered *key* amino acids, i.e., a subset of the sequence composed of samples of the original primary sequence. The subsequence has been created by taking one amino acid each k positions (amino acids s_{ik}, for every allowed i). The idea is that, by labeling these amino acids first, we should be able to considerably restrict the number of choices left for the amino acids lying in between two key amino acids. The labeling order is, thus, all the key amino acids first, followed by all the others. This idea showed that, whenever we increase the value k, the tree explored increases in size as well. This can be explained by the fact that, once s_{ik} is ground, there is an exponential number (in k) of choices for the placements of $s_{(i+1)k}$. Thus, the possible reduction in the number of conformations for the amino acids between key elements does not sufficiently balance the number of alternatives for the key elements. We concluded that labeling amino acids that are next to ground ones (as done in the system described earlier) is really the choice that limits the most the exponential growth of the search tree.

Finally, we observed that performing the search by operating directly on the complete list containing first the amino acids in the central part of the protein and then the amino acids in the tails (i.e., the list [Body,Tails]) is 10% faster than searching first [Body], and later invoking another labeling search on [Tails]. We believe this derives from the greater effectiveness of the interaction between heuristics on the full search tree.

5. A PARALLEL SOLUTION

In this section we provide an overview of a parallel scheme we designed to solve the protein folding problem. The overall parallelization scheme has been designed as a customization of general or-parallelism techniques [19] to the structure of the problem at hand. The parallel scheme builds on the constraint structure described in the previous sections. The search for solutions is divided among a number of processors, and the search tree is fragmented into subtrees (called *tasks*) and distributed for parallel exploration.

There are two main issues related to the task assignment. The first is that the tasks should be *uniformly* distributed among processors during the execution; this is easier if the number of tasks is large. The second issue relates to constraint propagation and pruning of the search tree through problem-dependent heuristics, that are more effective when applied to large search trees—i.e., fewer tasks. Hence, the task scheduling strategy should strike a proper balance between these two conflicting requirements.

5.1 Overview of the System

The system is composed of three components: a *loader*, a *scheduler*, and a set of *clients*. Figure 7 shows the components and the main interactions.

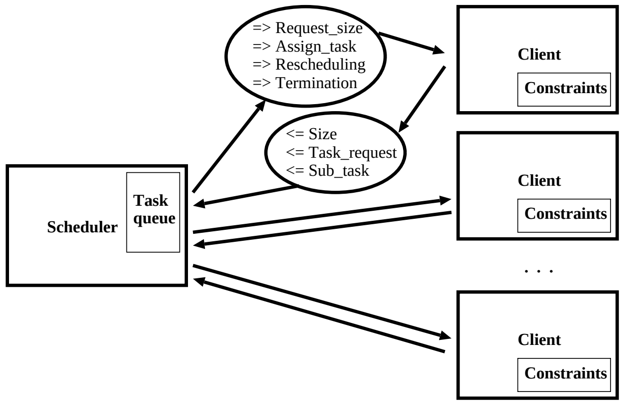

Figure 7: The parallel system

The loader is a C program, in charge of creating the communication channels—realized using shared memory segments—between the scheduler and the clients. In addition, the loader is in charge of launching both the scheduler and the clients, as parallel processes.

The system we developed makes use of a centralized scheduling mechanism. The scheduler is also a C program, that handles the dynamic distribution of tasks to the clients, and implements strategies for load balancing. Each client is a CLP program, that explores the subtree (i.e., task) assigned to the client by the scheduler. When the task is exhausted, the client will notify the scheduler that it is ready to receive an additional task.

5.2 Scheduling and Communication

The centralized scheduler implements a *direct scheduling* strategy. It relies on a static partitioning of the search tree, performed according to user defined parameters. During direct scheduling, tasks are assigned to clients upon request.

The scheduler determines the initial pool of tasks to be assigned (*task queue* in Fig. 7)—according to a static expansion of a user-specified number of levels (`Levels`) of the search tree. Since the scheduler is a C program, it does not have access to the collection of constraints; thus, the initial pool of tasks is generated by the clients, and retrieved by the scheduler during the initialization phase. Here, the first client is in charge of precomputing the expansion of `Levels` levels of the search tree and of returning the result of the expansion to the scheduler. The task queue is initialized with a set of subtrees of the search tree, all with roots at the same depth in the tree (`Level`). Each task is described by the list of nodes in the branch that connects the root of the search tree to the root of the task subtree. The scheduler assigns a task to a client whenever the client sends a `task_request` message. The task is assigned to the client by communicating the list of nodes describing it. The message that brings the task to the client is called `Assign_task`.

Due to the irregular structure of the search subtrees—because of the pruning performed by the constraint propagation process and by the heuristics—it is necessary to provide load balancing mechanisms. These mechanisms are employed when the scheduler has an empty task queue, and there is a mix of active and idle clients in the system. The purpose of load balancing is to dynamically generate new, smaller tasks, by further partitioning some of the tasks that are still active. Such smaller tasks can then be reassigned to the idle clients. The load balancing is implemented by a *rescheduling* procedure, activated by the scheduler every time there is at least one idle client and the task queue is empty. In this case, the scheduler selects, with a `Rescheduling` message, the client that has the estimated highest load of work. Due to lack of space, we do not provide technical details on the rescheduling procedure.

The scheduler takes also care of detecting global termination. Termination occurs when the task queue is empty and task requests have been submitted from all clients. In such a situation, the scheduler returns a `Termination` message to each client.

5.3 Structure of the Client

Each client is a CLP program that implements the process of solving the constraints on a given subset of the search space—i.e., a subset of the domains of the variables in the problem. When launched, a client imports protein data, defines variable domains and applies constraints. After the initial loading, the first client communicates back to the scheduler the list of partial assignments (`Tasks`) obtained from the expansion of the search tree for `Levels` variables.

After that, each client starts a loop composed of three operations:

1. delivery of a `Task_request` to the scheduler,
2. wait for assignment of a task (`Assign_task`), and
3. execution of the task.

The processing of a task is based on the CLP scheme described earlier. During each task execution, the client checks for eventual requests for `Rescheduling`—realized by breaking down the labeling process into labeling of smaller lists. If the request is received, the client stops the execution and communicates all the subtasks left to the scheduler. The client stops its execution when it receives the special termination signal.

5.4 Implementation Details

The parallel system has been implemented using a combination of C programming and Constraint Logic Programming (specifically SICStus Prolog 3.12.0 [30]). The activation of the processes implementing the scheduler and the clients is accomplished by standard `fork` calls of C issued by the loader. Separate processes are created for the scheduler and for the different clients.

The communication is realized using shared memory — in detail, using the IPC `shm` library. Since the clients are written in Prolog, it is necessary to encapsulate some low level C routines to access the shared memory. This has been realized using the foreign interface of SICStus Prolog. The use of a low level implementation of shared memory communication mechanisms has been dictated by the need for fast interaction between processes. SICStus Prolog provides existing libraries for interprocess communication, e.g., those based on the Linda paradigm, but tests revealed that access to shared data using Linda blackboard takes about 10ms, while our implementation using shared memory requires only 1μs (10^5 times faster). The reason is that we require a less stringent synchronization behavior.

The message exchange between clients and the scheduler is realized using shared memory queues. For boolean messages, e.g., `Request_size` and `Task_request`, a single byte is written, while the reading process checks the same memory location. For passing more structured data, e.g., lists specifying the tasks, the n list elements are written in an array from location 1 to n, and the length n of the list is stored in the array position 0. The first byte of the array is used also as a synchronizing flag, to ensure that the reader accesses the data only once they have been completely transferred in the communication buffer.

During the execution of Prolog predicates, we also relied on low-level C routines to maintain an efficient representation of the current state of execution of the task. This allows us to maintain a simplified representation of the state of the execution across the branches of the subtree (which are explored via backtracking by the Prolog system) and it simplifies communication with the scheduler during load balancing.

5.5 Experimental results

In this section, we report the experimental results obtained from the execution of our parallel system. The experiments have been performed using a HP RP8400 NUMA architecture, with 14 PA-RISC processors, 8GB RAM, and running HP-UX 11.1.0.

Table 6 shows the times, in seconds, of the parallel execution of the program, using the BBF strategy, no rescheduling and up to 8 parallel clients. We have parallelized the code on numbers of processors that are powers of 2. For each protein, the first 4 `Levels` of the tree are fully expanded. For the protein 1E0N we also included the results obtained using a value of `Levels` equal to 3—as this is sufficient to generate a sufficient number of tasks. The corresponding speedup curves are depicted in Fig. 8. The last column reports the execution time, on one CPU of the parallel machine, of the code from [15].

	Processors				[15]
Protein	1	2	4	8	1
1PG1	12.51	6.21	3.5	2.11	48.7
1KVG	13.03	7.56	3.99	2.73	24.9
1LE3	16.36	10.63	5.66	2.51	8.2
1EDP	11.91	6.4	3.42	1.19	67.8
1E0N	239.9	160	136.2	118.5	598.4
1E0N 3 `Levels`	2249	1367	141.72	82	598.4

Table 6: Parallel Execution Time (in seconds)

These performance results show that the scalability factors are strongly dependent on the specific problem. We discuss here some of the reasons that generate this behavior. First of all, the subdivision of the search tree into tasks does not imply an equal partition of the workload. In fact, since constraints and heuristics are applied to the trees during the search, their effect on a local portion of the search tree can result in a different pruning compared to the same subtree during a sequential search. This effect can have appreciated side-effects: for instance, the superlinear speed-up obtained in the cases of 1EDP and 1E0N (3 levels). Let us observe that, although the sequential speed up for some proteins is of the order of 40 (e.g., for 1EDP—see Table 3) and the parallel speed up is for some proteins of the order of 10 with eight processors (e.g., for 1EDP), the combined speed up is only of the order of 60 (instead of 400). The reason is the overheads associated to the management of parallel tasks—e.g., task exchange, communication costs, and the cost of resetting the constraint store when a new task is acquired. The latter cost, that seems to be the most significant one, could be removed by introducing a more ad-hoc handling of constraints—i.e., bypassing some of the constraint-handling functionalities of SICStus.

An aspect that is important to emphasize is that clients share their intermediate results by placing them in shared memory, where it becomes accessible to every other client. In particular, the shared memory allows the agents to share their current best solution, effectively parallelizing the branch-and-bound process. Let us assume that the tasks $T_1, \ldots T_n$ are explored in that order in the equivalent sequential algorithm. In the parallel case, every task can take advantage of an earlier-found new bound for the search. For example, T_i and T_j, with $i < j$, are explored in parallel, and a new best solution is found in T_j. This allows a client to prune T_i as well, resulting in a speedup in the search. On the other hand, though, the symmetric case can arise: T_j can be explored extensively because of a late discover of a bound in T_i. In the sequential case T_j would have been pruned from the very beginning, because T_i would have been completed before even starting the exploration of T_j. Thus, the way the partitioning of the search tree in tasks is performed will dramatically affect the search. As reported in the experimental data, for the protein 1E0N, we launched two different sets of initial tasks, resulting from an expansion of respectively 3 and 4 levels of the search tree. The effect of this choice is dramatic, and it shows how the early discover of some local optima in the search can drastically reduce the entire search. We plan to investigate in the future the design of problem-dependent partitionings, in order to take advantage of this parallel sharing of information.

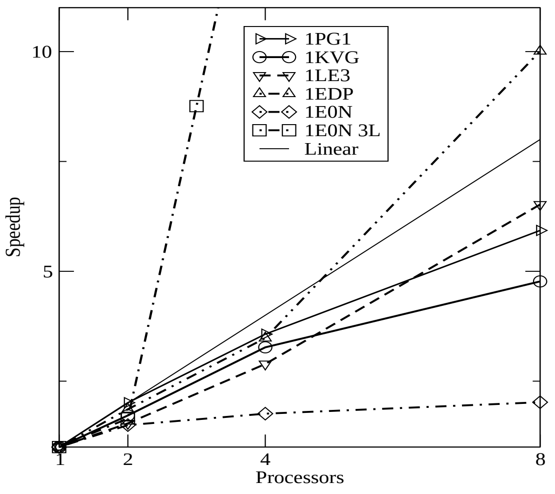

Figure 8: The parallel speedup without rescheduling

Let us conclude by observing that, during our experiments, the rescheduling procedure has been rarely effective in improving performance. This is due to the poor interaction between the fairly "sequential" way of rescheduling tasks and the more sophisticated exploration imposed by our heuristic strategies. Work is in progress to adapt rescheduling to better match our search strategies.

6. SCALABILITY ON MACRO BLOCKS

In this section we provide some ideas about the possible applications of our program to tackle larger proteins. When dealing with large proteins, it is common to encounter situations where the conformations of various subsequences are already known (e.g., by homology). Thus, the problem of predicting the structure of the whole protein is conceptually equivalent to predicting the placement of few rigid macro blocks, that are linked together by some coils. These ideas could be exploited by our prediction tool.

We defined the following example, to demonstrate that the current model can be feasibly used to attack larger proteins. Let us assume that we know in advance two substructures of a protein. In particular, we use the sequence XYX, where X is a known protein sequence and Y is a fixed-length linking coil of non-interacting amino acids. The idea is to predict the structure of the sequence XYX, using as input some strong constraints derived from the already known conformation of X. In our example, the two subunits X represent two generic structures that are known. In detail, we use the Euclidean distances between every pair of amino acids in the conformation of X as an additional constraint. This would be sufficient to recover the exact shape of the two Xs, if the original model was predicted on lattice (see the discussion in Sect. 4.2.2).

In our tests, we used as shape model for X the proteins 1EDP, 2GP8, and 1ZDD, as predicted by our tool on fcc. In Table 7, we report the length (Len) of $|XYX|$ and the time (in seconds) required to predict the structure of XYX, for different lengths of Y. For the search, we used SICStus Prolog 3.12.0 and our pruning heuristic (without BBF strategy) on a 2.4GHz Pentium PC.

	X=1EDP		X=1ZDD		X=2GP8	
$\|Y\|$	Len	Time	Len	Time	Len	Time
0	34	2.9s	68	10.7s	80	47.9s
1	35	16.5s	69	22.2s	81	1m 45s
2	36	27.1s	70	25.5s	82	1m 21s
3	37	28.1s	71	40.4s	83	2m 35s
4	38	1m 3s	72	50.9s	84	7m 5s

Table 7: Resolution of the XYX problem

The execution times grow according to the length of the coil Y. As expected, for proteins of this size but with partially known structure, the times are significantly lower than a prediction that uses only the secondary structure information. This allows our system to handle larger protein complexes.

In Table 8 we recomputed the execution times, disabling our pruning heuristic and replacing it by the SICStus built-in branch and bound search. It shows the exponential growth depending on the length of the inner coil Y. Note how our heuristic (see Table 7) considerably reduces the execution times.

$\|Y\|$	Len	X=1EDP	Len	X=1ZDD	Len	X=2GP8
0	34	7.0s	68	15.8s	80	3m15s
1	35	36.6s	69	1m6s	81	16m40s
2	36	3m1s	70	6m14s	82	1h25m
3	37	14m15s	71	33m6s	83	7h36m

Table 8: XYX problem using built-in branch-and-bound

7. CONCLUSION AND FUTURE WORK

In this paper, we provided a formalization of the protein folding problem on face-centered cubic lattice structures. The formalization has been transformed in a constraint system, and solved using constraint solving over finite domains. We analyzed different ways of organizing the constraint structure, and different heuristics and search strategies to solve them. We presented and tested a new search heuristic (Bounded Block Fails), well suited for this problem. We also provided a way to parallelize the process of exploring the search space, allowing concurrent constraint solvers to cooperate in the search of an optimal folding.

The results collected from the different approaches to the problem (sequential search strategies, parallel implementations, different implementations of solvers) converge on the need of a new dedicated and efficient solver. The analysis in Section 4.3 suggests that the only way to significantly improve our framework is to access the search tree at a lower level, and to implement new heuristics more suitable to the problem. Since this is not allowed by the current implementations of SICStus and ECLiPSe, we plan to develop our own ad-hoc constraint solver. The new solver will be dedicated to problem on lattices (and thus more efficient) and will implement ad-hoc search strategies. The new solver, applied to the protein folding problem of fcc, is expected to allow us to tackle larger problems—the final goal is to manage proteins composed of 500 amino acids. As shown in Table 3, currently we can solve (without using scalability—Section 6) proteins of length in the order of 80 amino acids.

We also plan to continue the development of the parallel solution; in particular, we intend to develop rescheduling strategies that better match the heuristics employed by the sequential system, allowing for a more effective load balancing and scalability.

Acknowledgments

We thank Federico Fogolari, for his guidance and comments.

The research has been partially supported by NSF grants HRD-0420407 and EIA-0220590, by the MIUR Project *Sybilla*, by GNCS 2005 *Vincoli e codici* and by FIRB 2003 project RBNE03B8KK.

8. REFERENCES

[1] R. Agarwala et al. Local rules for protein folding on a triangular lattice and generalized hydrophobicity in the HP model. *J. of Computational Biology*, pages 275–296, 1997.

[2] K. R. Apt. *Principles of constraint programming*. Cambridge University press, 2003.

[3] R. Backofen. The protein structure prediction problem: A constraint optimization approach using a new lower bound. *Constraints*, 6(2–3):223–255, 2001.

[4] R. Backofen and S. Will. Fast, constraint-based threading of HP sequences to hydrophobic cores. *Int. Conf. on Principle and Practice of Constraint Programming*, 494–508, 2001. Springer Verlag.

[5] R. Backofen and S. Will. A Constraint-Based Approach to Structure Prediction for Simplified Protein Models that Outperforms Other Existing Methods. *ICLP*, 2003, Springer Verlag.

[6] H. M. Berman et al. The protein data bank. *Nucleic Acids Research*, 28:235–242, 2000. http://www.rcsb.org/pdb/.

[7] M. Berrera, H. Molinari, and F. Fogolari. Amino acid empirical contact energy definitions for fold recognition in the space of contact maps. *BMC Bioinformatics*, 4(8), 2003.

[8] R. Bonneau and D. Baker. Ab initio protein structure prediction: progress and prospects. *Annu. Rev. Biophys. Biomol. Struct.*, 30:173–89, 2001.

[9] B. R. Brooks et al. Charmm: A program for macromolecular energy minimization and dynamics calculations. *J. Comput. Chem.*, 4:187–217, 1983.

[10] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. *PLILP*, Springer Verlag, 1997.

[11] A. M. Cheadle et al. ECLiPSe: An Introduction. Technical Report IC-Parc 03–1, IC-Parc, 2003.

[12] D. Clark, J. Shirazi, and C. Rawlings. Protein topology prediction through constraint-based search and the evaluation of topological folding rules. *Protein Engineering*, 4:752–760, 1991.

[13] P. Clote and R. Backofen. *Computational Molecular Biology: An Introduction*. John Wiley & Sons, 2001.

[14] P. Crescenzi et al. On the complexity of protein folding. In *Proc. of STOC*, pages 597–603, 1998.

[15] A. Dal Palù, A. Dovier, and F. Fogolari. Constraint logic programming approach to protein structure prediction. *BMC Bioinformatics*, 5(186), 2004.

[16] A. Dal Palù, A. Dovier, and F. Fogolari. Protein folding in $CLP(\mathcal{FD})$ with empirical contact energies. In *Recent Advances in Constraints*, Springer Verlag, 2004.

[17] F. Fogolari et al. Modeling of polypeptide chains as C-α chains, C-α chains with C-β, and C-α chains with ellipsoidal lateral chains. *Biophysical Journal*, 70:1183–1197, 1996.

[18] S. Forman. *Torsion Angle Selection and Emergent Non-local Secondary Structure in Protein Structure Prediction*. PhD thesis, U. of Iowa, 2001.

[19] G. Gupta, E. Pontelli, M. Carlsson, M. Hermegildo, K. Ali. Parallel Execution of Prolog: a Survey. *ACM TOPLAS*, 23(4):472–602, 2001.

[20] W. Hart and S. Israil. Fast protein folding in the hydrophobic-hytrophilic model within three-eighths of optimal. *J. of Computational Biology*, pages 53–96, 1996.

[21] S. Miyazawa and R. L. Jernigan. Residue-residue potentials with a favorable contact pair term and an unfavorable high packing density term, for simulation and threading. *J. of Molecular Biology*, 256(3):623–644, 1996.

[22] A. Newman. A New Algorithm for Protein Folding in the HP Model. In *Symposium on Discrete Algorithms* . Springer Verlag, 2002.

[23] Committee on Mathematical Challenges from Computational Chemistry. National Research Council, 1995.

[24] D. Qiu, P. Shenkin, F. Hollinger, and W. Still. The gb/sa continuum model for solvation. A fast analytical method for the calculation of approximate born radii. *J. Phys. Chem.*, 101:3005–3014, 1997.

[25] G. Raghunathan and R. L. Jernigan. Ideal architecture of residue packing and its observation in protein structures. *Protein Science*, 6:2072–2083, 1997.

[26] B. Rost and C. Sander. Prediction of protein secondary structure at better than 70% accuracy. *J. Mol. Biol.*, 232:584–599, 1993.

[27] K. Simons et al. Ab initio protein structure prediction of CASP III targets using ROSETTA. *Proteins* 1999, 3:171–176.

[28] J. Skolnick and A. Kolinski. Computational studies of protein folding. *Computing in Science and Engineering*, 3(5):40–50, 2001.

[29] J. Skolnick and A. Kolinski. Reduced models of proteins and their applications. *Polymer*, 45:511–524, 2004.

[30] Swedish Institute for Computer Science. Sicstus Prolog. http://www.sics.se/sicstus/.

[31] L. Toma and S. Toma. Folding simulation of protein models on the structure-based cubo-octahedral lattice with contact interactions algorithm. *Protein Science*, 8:196–202, 1999.

[32] M.L.G. William and D. Harvey. Limited discrepancy search. *IJCAI*, pages 607–615, Morgan Kaufmann, 1995.