A Constraint Logic Programming Approach to 3D Structure Determination of Large Protein Complexes

Alessandro Dal Palù

Dipartimento di Matematica,
Facoltà di Scienze MM FF e NN, Università di Parma,
Parco Area delle Scienze 53/A, 43100 Parma, Italy
Fax: +39 0521906950 E-mail: alessandro.dalpalu@unipr.it

Enrico Pontelli*, Jing He and Yonggang Lu

Computer Science Department,
New Mexico State University,
BOX 30001 MSC CS, Las Cruces, NM, 88003, USA
Fax: +1 5056461002 E-mail: epontell@cs.nmsu.edu
E-mail: jhe@cs.nmsu.edu E-mail: ylu@cs.nmsu.edu
*Corresponding author

Abstract: The paper describes a novel framework, constructed using constraint logic programming and parallelism, to determine the association between parts of the primary sequence of a protein and α-helices extracted from 3D low-resolution descriptions of large protein complexes. The association is determined by extracting constraints from the 3D information, regarding length, relative position, and connectivity of helices, and solving these constraints with the guidance of a secondary structure prediction algorithm. Parallelism is employed to enhance performance on large proteins. The framework provides a fast, inexpensive alternative to determine the exact tertiary structure of unknown proteins.

Keywords: Protein structure, constraint logic programming, parallelism

Reference to this paper should be made as follows: A. Dal Palù, E. Pontelli, J. He and Y. Lu. 'A Constraint Logic Programming Approach to 3D Structure Determination of Large Protein Complexes', *Int. J. Data Mining and Bioinformatics*, Vol. x, No. x, pp.xxx–xxx.

Biographical Notes:

Alessandro Dal Palù is a permanent researcher at University of Parma. He received his PhD Degree in the Department of Mathematics and Computer Science at Udine University, in 2006. His research interests include, constraint logic programming, bioinformatics, and algorithm optimization.

Enrico Pontelli is a professor of Computer Science at New Mexico State University. He received a Ph.D. from the Dept. of Computer Science at NMSU and a MS degree from the University of Houston. His research interests include bioinformatics, constraint logic programming, assistive technologies, and parallel computing.

Jing He is an assistant professor at New Mexico State University. She received her Ph.D. in Structural and Computational Biology from Baylor College

of Medicine in 2001. Her research interest are in the development of algorithms and software for biological systems.

Yongang Lu is a doctoral student in Computer Science at New Mexico State University. His research interests are in bioinformatics, with special focus on protein structure prediction.

1 Introduction

Proteins are responsible for nearly every function required for life. A functional protein can be thought of as a properly folded chain of amino acids (the *primary sequence*) in a 3-dimensional space (3D). The 3D structure of a protein crucially determines its interaction with other proteins and, thus, its biological functions. 3D protein structure determination is a challenging problem, particularly for large protein complexes and membrane proteins that are not easy to crystallize [6]. Protein structure determination methods can be generally classified in two categories. *Experimental methods* attempt to determine the 3D structure using experimental studies, frequently based on either X-ray crystallography or NMR spectroscopy. *Prediction methods*, instead, predict a protein structure based on its primary sequence information (e.g., using *homology* or *ab initio* methods). Experimental methods are limited by difficulties in protein expression, purification and crystallization—particularly severe in the case of large proteins. On the other hand, prediction methods are, on average, successful only 76% of the times, and their correctness is sensitive to a variety of factors [9, 12]

Electron Cryomicroscopy (EC) is a novel promising experimental technique, that determines 3D structures of large protein complexes at atomic resolution [6]. Current advances in EC techniques are able to determine a rough protein shape in the form of a *density distribution of electrons* (*density map*), with a resolution of 6 to 9Å [17]. At this resolution, amino acid side chains, used to distinguish different amino acids, are *not* generally visible. Nevertheless, this resolution allows the identification of specific features of the 3D structures, e.g., *occurrence* and *length* of α-helices [16, 8].

We propose a new methodology that improves protein structure determination by *combining*—using constraint logic programming (CLP($\mathcal{FD}$))—three sources of information: structural information (extracted from EC density maps), information from the primary sequence, and results from secondary structure prediction methods.

2 Overall Approach

2.1 Background Definitions

The *primary* structure of a protein is a sequence of linked units (*amino acids*). The amino acids are represented by symbols from a 20-element alphabet. 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 minimal value of free energy (*native* conformation). Native conformations are largely built from *Secondary Structure elements*—e.g., α-helices and β-sheets—often arranged in well-defined motifs. α-helices are composed of 5 to 40 contiguous amino acids, arranged in a regular right-handed helix. β-sheets are composed of extended strands of 5 to 10 amino acids.

Algorithms, e.g., based on neural networks, have been developed for the task of predicting the secondary structure (occurrences of helices, β-sheets, etc.) of a protein. For any protein sequences, prediction methods, such as PHD, generate a *measure of likelihood*—i.e., a value between 0 and 9—for each amino acid of being involved in an helix (or other secondary structure elements) [9]. On average, the accuracy of the best prediction methods is about 76% [9, 3]. These errors in the secondary structure prediction generate a lot of freedom in mapping the actual helices onto the primary sequence using the prediction as a guide.

2.2 Problem Description

Knowledge of the 3D position of each aminoacid in an unknown protein is vital to the understanding of the functions of the protein. Current techniques allow us to effectively determine the primary sequence of the protein, as well as the density map of the protein (using EC). Nevertheless, the ability to place each amino acid from the primary sequence on the density map is still missing—as the resolution of the density map is too low. Ab-initio protein-folding techniques (e.g., [2, 10]) could be used to address this problem, but they do not guarantee the correct 3D structure and they are, in general, extremely compute-intensive.

The general problem addressed in this work is to provide the first step towards such missing capabilities. Given the primary sequence of a protein, its 3D density map obtained using EC, and given the secondary structure prediction for that protein, we want to recover the correspondence between each helix identified in the 3D structure and the corresponding subsequence of amino acids in the primary structure.

In the density map, it is possible to roughly recognize the helical regions, as dense cylinders, along with some of their key properties, such as their lengths and relative distances [8]. However, with only this piece of information, it is not straightforward to map each 3D cylinder to the corresponding primary structure area. The availability of a mapping between helices and the primary structure provides a *strong constraint* on the position of the amino acids in the 3D space, which can be exploited by molecular dynamics applications, e.g., CHARMM [4], to quickly determine an accurate high-resolution description of the protein.

2.3 Overall Approach

In this paper, we propose a general framework to integrate density maps and secondary structure information, and to predict the relationship between the primary structure and the 3D volumetric data. This first version of the framework deals only with helical information—future work will extend this approach to other secondary structure information (e.g., coils and β-sheets).

The framework is composed of two parts. The first one introduces new techniques for extracting 3D structural information from the density maps, and it is based on 3D volumetric data analysis. This phase leads to an abstract representation of certain features of the protein (e.g., helical regions and connectivity between regions). The second part exploits the high level information obtained from the volumetric analysis, and combines it with the secondary structure prediction. The idea is to join the constraints from the 3D structural information with the results of secondary structure prediction to map helices on the primary

protein sequence, in a way to maximize the likelihood for all helices. All this information is encoded as *constraints*, whose optimal solution determines the correspondence between 3D and 1D components—i.e., identify segments of the primary sequence that correspond to the helices recognized during volumetric data analysis. This second phase is implemented using *Constraint Logic Programming* techniques (i.e., CLP($\mathcal{FD}$)) [1] and parallelism. The first phase is not the focus of this paper, and it is only briefly discussed next, while the second phase is analyzed in the rest of the paper.

2.4 Constraints from the Density Maps

We developed a new technique to extract information from the density maps, using gradient-based analysis of volumetric data to determine helical regions [5]. In the second phase, we are interested in recovering some basic relationships between the predicted helices. The simplest relation to extract—Euclidean distance between extremities of different helices—is simple, but it provides a weak constraint. The proteins are often globular and each point is relatively close to the others—making the Euclidean distance small and not an effective constraint. Our idea, instead, is to extract the possible *expected paths* that might connect two helices, and to build a graph that describes the rough connectivity of areas in the density map. We can obtain this by properly analyzing the density levels and distributions in the map. This allows us to recover the actual topology the protein—along with various *spurious* connections, due to the low resolution of the density map. We accomplish this by first constructing a three dimensional graph, that represents the centroids of clusters of high density regions, and then by mapping on it the previously determined helices. The analysis of this graph allows us to extract stronger constraints: given two helices, it is possible to find the shortest path—*in terms of amino acids*—that separates them. This information is much stronger than the Euclidean distance, as it accounts for the possible links between helices observed in the density map. This information is correct: although two helices can be more distant (when the graph contains many spurious edges), it is not possible that two helices are separated by a smaller number of amino acids than predicted. Moreover, if there is only one path that links two helices, it is possible to state that the helices are *adjacent* in the primary sequence.

A detailed description of this analysis method and its evaluation can be found in [5].

3 A Constraint-based Solution

The information extracted from the analysis of the density maps has to be combined with the results of secondary structure prediction, to guide the creation of a mapping between helices and segments of the primary sequence. In general, the structural information extracted from the density map allows us to determine the number and lengths of the helices present in the protein (e.g., see the middle box in Fig. 1), as well as information about relative distances (in terms of amino acids) and positions. For each helix, we wish to explore its possible placements on the protein sequence—i.e., determine which segment of the protein sequence corresponds to the given helix (see bottom box in Fig. 1). The possible placements are expected to satisfy the distance, length, and position constraints, and they are *ranked* according to how well they match the secondary structure prediction produced by PHD [13] (top box in Fig. 1). The problem is reduced to a constraint opti-

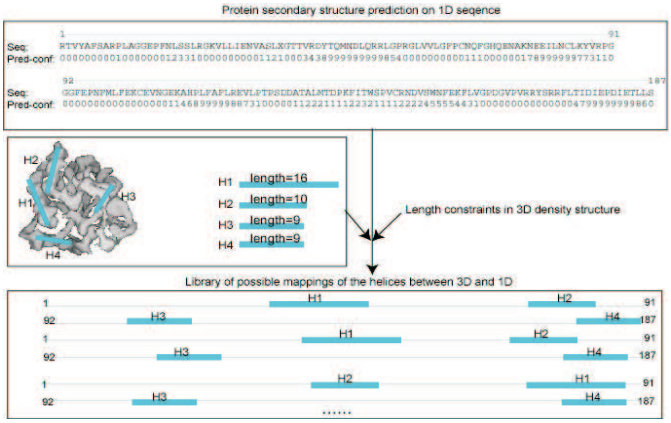

Figure 1 Combining structure prediction and helix information

mization problem. The optimal solution is a placement which satisfies all constraints and having the maximal rank.

Due to the exponential size of the search space, it is vital to employ the available constraints (e.g., length of the helices) to prune the search space and discard unlikely placements. Since we use the secondary structure prediction as a guide to score the placements, the helices are mostly placed near the high score populations (e.g., 899998) of the likelihood distribution produced by PHD.

3.1 A CLP Formalization of the Problem

The use of CLP($\mathcal{FD}$) for the encoding of our problem is suggested by two main needs: *(1)* the need for efficient solvers to handle the constraints extracted from the density maps, and *(2)* the need for modularity, as we wish to add new types of constraints as they become available from density maps analysis. These objectives can be easily achieved in CLP($\mathcal{FD}$) (while, e.g., modularity, is challenging in imperative languages).

Constraint Logic Programming: *Constraint Logic Programming (CLP)* [11] is a declarative programming paradigm which is particularly well-suited for encoding combinatorial optimization problems. It is the combination of two declarative paradigms: *Constraint* [1] and *Logic Programming* [15].

A *constraint* is a first-order formula that can be interpreted over various possible domains. For instance, $1 \leq X \wedge X < Y \wedge Y < 2$ is a constraint that is satisfiable over the domain $\mathbb{R}$ but not over the domain $\mathbb{N}$. A CLP language allows the programmer to encode problems as collections of logical rules of the form: $Head :- B_1, \ldots, B_k$, where the various B_i are constraints drawn from different classes of constraint domains. Intuitively, such a rule states that, whenever all constraints B_i are true, then also the consequence $Head$ will be true.

For combinatorial problems it is common to use *finite domain constraints*, namely relational constraints (e.g., equalities and disequalities) between arithmetic expressions, where each variable ranges over a predefined finite domain. The instance of CLP which allows the

use of finite domain constraints is typically called CLP($\mathcal{FD}$). The library `clpfd` of SICStus [14] provides the capability to develop CLP programs using this type of constraints.

The CLP approach to encode combinatorial optimization problems relies on the so-called *Constrain & Generate* technique. In the constrain & generate approach, first a deterministic phase introduces a number of constraints, and then a non-deterministic phase generates the solution space—under the given constraints. The constraints introduced allow a substantial pruning of the search space (via constraint propagation), significantly extending the class of problems that can be feasibly tackled. Moreover, in this phase one can take advantage of built-in strategies (such as constraint propagation and branch and bound) and it is possible to further guide the solution search using problem-dependent heuristics.

For example, if we want to maximize the objective function $6x + 10y - 2z$ with the conditions $x \geq 0, y \geq 0, z \geq 0$ and $10x + 17y + 4z \leq 40$, then we can write the code (using the concrete syntax of SICStus):

$$\begin{array}{l} \textit{constrain} \left\{ \begin{array}{l} X \texttt{\#>=} 0, Y \texttt{\#>=} 0, Z \texttt{\#>=} 0, \\ 10 * X + 17 * Y + 4 * Z \texttt{\#=<} 40, \\ Result \texttt{\#=} 6 * X + 10 * Y - 2 * Z, \end{array} \right. \\ \textit{generate} \ \{ \ \textit{labeling}([\textit{maximize}(Result)], [X, Y, Z]) \end{array}$$

where the *labeling* is a built-in procedure to generate the possible assignments to X, Y, Z. The procedure creates a *search tree*, which contains the choices made during the enumeration of assignments to variables. Each node of the tree represents a variable and each of its children is associated to an assignment of an admissible value of that variable.

Problem Formalization: Let n be the number of helices extracted from the density map, and, for each helix i, let L_i denote its length. Let m be the length of the primary sequence. A *placement* π is a tuple $\langle \pi_1, \ldots, \pi_n \rangle$, where $1 \leq \pi_j \leq m$ denotes the initial position of helix i. The collection of all possible placements is denoted by Π. A *partial* assignment $(\pi, i), 1 \leq i \leq n$ is the tuple $\langle \pi_1, \ldots, \pi_i \rangle$. Note that $(\pi, n) = \pi$.

Let $pred_i$ denote the confidence level assigned by PHD for the presence of an helix in position i in the protein sequence—i.e., the likelihood that the amino acid in position i belongs to an helix.[a] Given the position P_j of the beginning of helix j in the sequence, the score associated to P_j is

$$S_j(P_j) = \sum_{i=P_j}^{P_j+L_j-1} pred_i$$

where L_j is the helix length. The average score of assigning n helices is the total score divided by the total length of the n helices.

To each placement is associated a *score* defined as $S(\pi) = \sum_{i=1}^{n} S_i(\pi_i)$. We define the *average score* of a partial assignment:

$$A(\pi, i) = \frac{\sum_{j=1}^{i} S_j(\pi_j)}{\sum_{j=1}^{i} L_j}$$

The *objective function* can be described as: $\max_{\pi \in \Pi} S(\pi)$.
A set of constraints should be imposed over Π to restrict the admissible placements:

[a] Remember that $0 \leq pred_i \leq 9$.

- *Non-overlap constraint:* for $i < j$: $(\pi_i \geq \pi_j + L_j) \vee (\pi_i + L_i \leq \pi_j)$. The constraint forbids placements where one amino acid belongs to two helices.
- *Trim scores:* $\forall 1 \leq i \leq n : A(\pi, i) > T_m \frac{i-1}{n-1} + T_M \frac{n-i}{n-1}$, where $T_m < T_M$ are respectively minimal and maximal thresholds on the score. Basically, for each level of the tree, we set a minimum threshold on the partial score, used to prune those branches that contain an average partial contribution which is too low. The threshold varies linearly with the depth of the tree, and ranges from T_m to T_M. The idea is to require a low threshold for the root (T_m), and increase it while placing different helices.
- *Minimal distance constraint:* if the density map indicates that helices $i \neq j$ are separated by at least $\delta_{i,j}$ amino acids, then $(\pi_i \geq \pi_j + L_j + \delta_{i,j})$ or $(\pi_i + L_i + \delta_{i,j} \leq \pi_j)$.
- *Adjacency constraint:* if the density map clearly identifies that helices i and j are adjacent *and* separated by $\delta_{i,j}$ amino-acids, then $(\pi_i \geq \pi_j + L_j + \delta_{i,j} \wedge \pi_i \leq \pi_j + L_j + \delta_{i,j} + V)$ or $(\pi_i + L_i + \delta_{i,j} \leq \pi_j \wedge \pi_i + L_i + \delta_{i,j} + V \geq \pi_j)$. This is a special case of Minimal Distance Constraint, where the number of amino acids between the two helices is bounded to be at least $\delta_{i,j}$ and at most $\delta_{i,j} + V$, where $V \in \mathbb{N}$ is a small number to accommodate errors when identifying the exact bounds of helices (the exact case would use $V = 0$). Even if the constraint still allows the presence of other helices in between the two adjacent ones, this is actually avoided because of the distances predicted from the density maps, i.e., the disequality $\delta_{j,k} \geq \delta_{i,j} + L_i + \delta_{i,k}$ holds. In this way, helix k cannot be inserted between i and j without violating a minimal distance constraint.

A CLP($\mathcal{FD}$) Encoding: We are interested in modeling the properties of helices in order to find a suitable placement over the primary structure. Each helix ℓ is modeled by two variables, P_ℓ and S_ℓ, that represent respectively the *position* of the helix in the primary sequence and the *score* associated to such placement. The variable L_ℓ denotes the length of the helix. The domains associated to P_ℓ and S_ℓ represent the possible placements of the helix and the corresponding quality score. The program selects, for each helix ℓ, a set of positions $p_i \in \mathsf{P}_\ell$, that are good candidates for the search. For each p_i, the corresponding score $s_i \in \mathsf{S}_\ell$ is included in the score domain. The candidates are selected performing a convolution of a window of length L_ℓ on the PHD prediction array. The score associated to each position is equal to the sum of predicted reliability for each amino acid in the current window. In order to reduce the search space from the beginning, a heuristic is employed in the definition of the domains for the `P` of each helix. The heuristic selects, from each domain, a set of representative positions, removing positions that lead to a very low score and merging domain positions that are very close to each others. For an helix ℓ, we refer to `DomMinDist` as the minimal distance required between two positions in the position domain P_ℓ, i.e., $\forall x, y \in \mathsf{P}_\ell$. `abs(x-y) #>= DomMinDist`.

The program has the classical *constrain & generate* structure. The program defines the variables' domains and the constraints among them, and then launches a backtrack search, that explores the space of admissible solutions. The input `Data` for the program is a list of records of the following form:

```
[Hn, Length, [P_list], [S_list]]
```

where, for each helix `Hn`, `Length` is its length, `P_list` is the list of allowed positions in the primary structure (referred to the beginning of the helix), and `S_list` is the corresponding list of scores associated to each position. Note that, for convenience, `S_list` is

sorted in decreasing order (the most valuable positions are listed first). The input data contains also the adjacency list (`AdjList`) obtained from density map analysis, composed of elements of the form: `[Hi, Hj, GDistij, Strict]` where `Hi` and `Hj` are helices and `GDistij` is the minimal distance in terms of amino acids that separates them in the primary sequence. The flag `Strict` defines whether `GDistij` is a lower bound or an exact estimate. The input data is encoded as constraints over FD variables. The code:

```
set_domains(Data,P,S,DomMinDist),
get_length(Data,L), get_index(Data,Index),
constrain_no_ovrl(P,L),
constrain_scores(Data,P,S),
trim_scores(S,L,Tm,TM),
(UseGraph==1 -> cons_pos(P,L,AdjList,Index)
             ; true).
```

implements the *Constrain* phase. Here, the domains for the variables `P` and `S` are established (`set_domains`), so that every domain contains the values specified in the input data. The domains of `P` are simplified as described above, using the `DomMinDist` user defined parameter. The predicate `get_length` (`get_index`) recovers from `Data` the list of lengths (helix indices `Hn`) associated to the helices. The predicate `constrain_no_ovrl` adds a constraint for each pair i, j of helices. The constraint expresses the fact that two distinct helices may not overlap when placed on the primary sequence; this is encoded in CLP($\mathcal{FD}$) by restricting the range of relative positions the two helices can assume:

```
(Pi #>= Pj+Lj) #\/ (Pi+Li #=< Pj)
```

,

where `Pi`, `Pj`, `Li`, and `Lj` are the position variables and lengths associated to the helices i and j.

The second set of constraints is added by `constrain_scores`. For each helix i, we call `bind_score(DomP,DomS,Pi,Si)`, that binds each position in the list `DomP` to the corresponding score `DomS`. Note that the lists contain the same elements defined for the domains of variables `Pi` and `Si`. The predicate `bind_score` is defined as follows:

```
bind_score([DP | DomP], [DS | DomS], P, S) :-
      (P #= DP) #=> (S #= DS),
      bind_score(DomP, DomS, P, S).
bind_score([], [], _P, _S).
```

The implication `#=>` ensures that, whenever a value is assigned to an helix position, the corresponding score variable is properly assigned as well. The predicate `trim_scores` adds the *Trim score* constraints, aimed at pruning the search tree as the placement of the helices is generated. A varying minimum threshold on the score of the partial solution is enforced.

Finally, if the flag `UseGraph` is set, then the predicate `cons_pos` is called, that introduces the distance/adjacency constraints (in terms of amino acids) obtained from the density map. For each pair of helices i, j we restrict the distance between the corresponding two blocks on the primary sequence to be at least the value computed using the density map information, i.e.,

```
(Pi#>= Pj+Lj+GDistij) #\/ (Pi+Li+GDistij#=<Pj)
```

where `GDistij` is the minimal amino acids distance predicted from the density map.

A special case is considered when two helices are predicted to be consecutive in the primary sequence (`Strict` flag set to 1). This information is encoded as an additional

constraint, stating that the minimal distance predicted is actually very close to the correct one. This adds an upper limit for the distance between the two helices. In this case, to accommodate for errors in predictions, the lower and upper limit for the relative distance of two consecutive helices differ by few amino acids (`Var`). In CLP($\mathcal{FD}$):

```
(Pi #>= Pj+Lj+GDistij #/\ Pi #=< Pj+Lj+GDistij+Var)  #\/
  (Pi+Li+GDistij #=< Pj #/\ Pi+Li+GDistij+Var #>= Pj)
```

We make use of the built-in search capabilities of CLP($\mathcal{FD}$) to support the *generate* phase of the constraint resolution process. We implement two types of search. In the first (*Maximize* search), we search the best solution using branch&bound, via the built-in predicate `labeling([maximize(Score)],Positions)` where `Score` contains the average score of a placement, and `Positions` is the list of position variables for the n helices. In the other search (*Enumeration* search), we list the suboptimal set of solutions whose score is within a small percentage from the selected maximum. The constraint: `T is integer(Enum*Perc*100), Score*100 #>= T*SumLen` is introduced, where `SumLen` is the sum of each helix length, `Enum` is the minimal average score required to enumerate a solution and `Perc` is a coefficient (between 0 and 1) that lowers that threshold. It is useful, when enumerating suboptimal solutions, to equal `Enum` to the maximum found with the branch and bound search and to select (using `Perc`) the amount of solutions required. To explore the admissible solutions that fulfill the constraints, we combine the build-in `labeling([ff],Positions)` and `findall`, to collect the results produced by `labeling`. As before, we search assignments of the list `Positions` of all helices positions that satisfy all the constraints.

In general, it is not expected that the actual solution lies exactly on the maximum, due to approximations and errors in predictions. It is important to study the quality of a set of quasi-optimal solutions. In the optimization search, the selection order of the position variables is determined by decreasing order of helices length, since the prediction quality is higher for longer helices. Thus, in the labeling predicate we present a list ordered accordingly. In the enumeration search, we use a first-fail (`ff` option) variable selection rule, which is appropriate to explore the search tree without a branch and bound algorithm. The actual code that implements the formalization described above is concise and it is easily modifiable, i.e., when new constraints need to be added.

3.2 Experimental Results

The system has been implemented using the `clpfd` library of SICStus. In our tests, we use the proteins with the largest number of helices from the benchmark set of [7]. We perform two types of comparisons. In the first, we compare the running times of the sequential version as in [7] and the *equivalent* implementation in CLP($\mathcal{FD}$). In the second analysis, we run a set of tests to analyze the speedup obtained by including the constraints gathered by the preliminary density map elaboration. We also test different thresholds for the `DomMinDist` heuristic and we show the results of the two different search strategies employed.

Table 1 reports the running times of the same problem instances (without the distance constraints from density maps—these are too difficult to introduce in C++), implemented in C++ and CLP($\mathcal{FD}$). For the CLP($\mathcal{FD}$) section, we report the branch and bound time (maximize) and the enumeration time of the solutions over a certain average threshold (dependent on the specific protein). The tests are run on an Intel 3GHz Linux machine. We use `ThrMin` = 5, `DomMinDist` = 10, and we enumerate all solutions with score greater

Protein	Thr Max	C++	CLP($\mathcal{FD}$) Maximize	CLP($\mathcal{FD}$) Enum
1CC5	7	0.10 s	$\leq$ 0.01 s	0.01 s
1ECA	7	0.66 s	0.16 s	1.93 s
2PHH	8	2.66 s	0.65 s	1.48 s
2TSC	7	15.1 s	0.44 s	3.54 s
3TIM	8	92.8 s	0.83 s	34.4 s
2CYP	6	40.9 s	31.9 s	131 s

Table 1 C++ vs. CLP($\mathcal{FD}$)

than `ThrMax` (i.e., `Enum`=`ThrMax`, `Perc`=1.0), in order to explore the same solutions as in the maximize search.

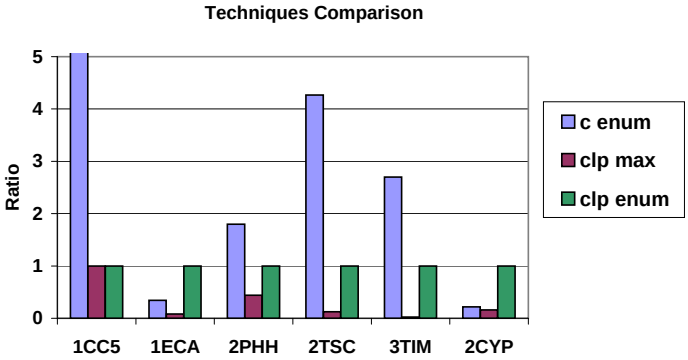

Figure 2 C++, branch & bound, CLP($\mathcal{FD}$) enumeration

The times show that the CLP($\mathcal{FD}$) approach is comparable, and frequently superior, to the optimized C++ implementation. Note that the times in CLP($\mathcal{FD}$) are not proportional to C++. The main reason is that pruning in CLP($\mathcal{FD}$) is handled in conjunction with constraint propagation, which can be very effective, depending on the specific domains at hand. Observe that the search for the optimal solution in CLP($\mathcal{FD}$), using built-in branch and bound techniques, is very efficient, even in this not very constrained scenario, yet maintaining a simple and modular structure. Figure 2 compares the performance of the 3 techniques. For each protein, we normalized the three running times against the CLP($\mathcal{FD}$) enumeration time.

The next set of experiments illustrates the introduction of distance/adjacency constraints to reduce the search space. Moreover, we also show the differences in the search space and time, when varying the search technique (i.e., maximization and enumeration). and the `DomMinDist` heuristic parameter. If not specified differently, we used the same parameters as in the previous table. We set `ThrMax`=6 and for the enumeration case we selected `Enum` equal to the maximal value found during the maximization with the same parameters.

In Table 4 (Appendix **A**), we report the detailed summary of these experimental results. The column *Distance Max Real* reports whether the enumerated set of solutions contains the real solution, and, in that case, reports the distance in terms of score from the maximal

solution found. Note that, for these tests, we consider as real solution a solution that presents the same helices' order and positions as in the original protein in the PDB.

Fig. 3 depicts, for each protein, the speedup in terms of time and nodes enumerated, obtained by introducing distance/adjacency constraints. The values reported are taken, for each protein, from the rows with the lowest `DomMinDist` parameter using enumeration search.

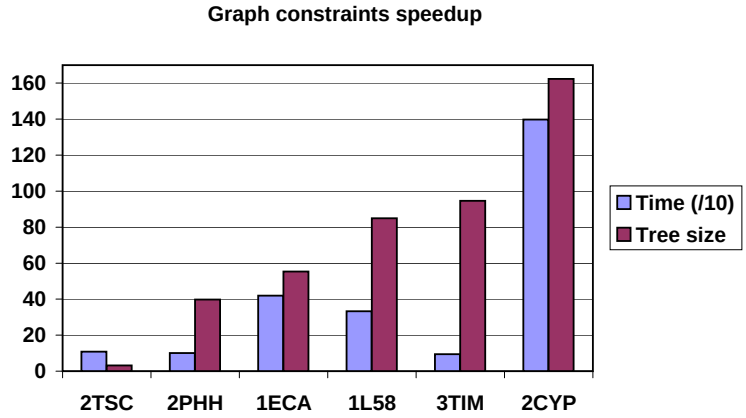

Figure 3 Speedup using distance/adjacency constraints

We can conclude that the additional constraints provide significant speedups. In particular, note that the minimal amino acid distance between two helices extracted from the density map is a conservative estimate. This fact has various consequences: first of all, the use of an improved feature extraction procedure from the density maps could produce higher correct minimal distances between helices and speedup the search by several orders of magnitude. Moreover, for the protein 1ECA, note that, using the graph constraints in combination with a strict `DomMinDist` heuristic, the real solution is not found for `DomMinDist`=10. This shows clearly that the constraints correctly eliminate the possible real solution candidates that are not physically allowed, but nevertheless considered in a generic blind enumeration (as the one in [7]). This also shows that the `DomMinDist` heuristic is very useful to reduce the search, but also it cannot be used with too high values, since it could exclude some domain values essential to the identification of the real solution. The same points above can justify the fact that, in certain proteins, the introduction of distance/adjacency constraints increases the distance between the best solution score and the real one: the enumerated space is pruned by the additional constraints, and it presents a correct real solution associated to a lower average score. Observe that, even without strong constraints, we can find the solution in the optimal position for the protein 2PHH.

4 A Parallel Constraint Solution

The exploration of a search tree induced by the variables, domains, and the constraints, is highly non-deterministic, suggesting the possibility of partitioning the search tree into subtrees (*tasks*) and exploring them *in parallel*.

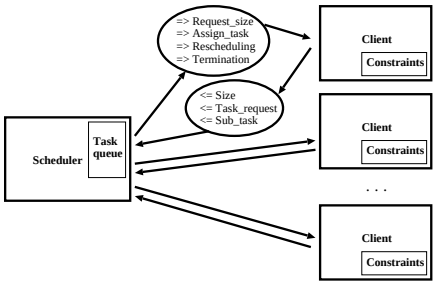

Figure 4 The parallel system

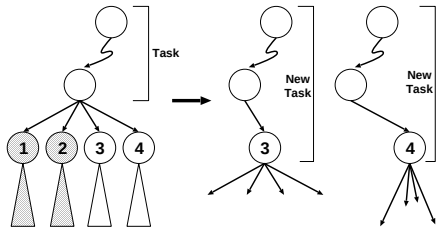

Figure 5 Example of rescheduling

4.1 Overview of the Parallel System

The parallel system (see Fig. 4) is composed of a *scheduler* and a set of *clients*. The system makes use of a *centralized* scheduling mechanism. The scheduler is a C program that handles the distribution of tasks to the clients in a fully dynamic fashion, and implements load balancing strategies. Each client explores the subtree (i.e., *task*) assigned to the client by the scheduler, in search of solutions. In addition, the client's execution might be preempted if load balancing activities are requested by the scheduler. In such a case, the client will select parts of its task and return it to the scheduler for redistribution.

4.2 Clients

Each client is a CLP($\mathcal{FD}$) 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 defines constraints. After the problem is loaded, it communicates back to the scheduler the cardinality (`Size`) of each variable domain representing a level in the search tree. After that, each client starts a loop composed of three operations:

- send a `Task_request` to the scheduler,
- wait for assignment of a task, and
- execute the task.

The processing of a task is based on the CLP($\mathcal{FD}$) scheme described earlier. During task elaboration, the client checks for eventual `Rescheduling` requests; this test is performed every time one of the subtrees of the initial task's root is completed. If the request is received, the client stops and communicates all the remaining `sub_tasks` to the scheduler.

4.3 Scheduling and Communication

The distribution of tasks to the clients needs to strike a balance between uniform work distribution (easier for many tasks) and effectiveness of constraint propagation (more efficient for fewer, larger tasks).

The centralized scheduler statically determines the initial pool of tasks to be assigned (*task queue*) according to a user-defined parameter (minimal number of tasks—MinTask)

and the size of each level of the tree. The size of each level of the tree depends on the size of the domain of the variables in the constraint problem. The task queue is initialized with a set of subtrees of the search tree (all having roots at the same depth). The scheduler assigns a task—using an `Assign_task` message—to a client in response to a `Task_request` message. Each task is described by the path in the tree from the root of the tree to the root of the task subtree.[b] E.g., task 1 in Fig. 5 is described by the path from the root to node 1.

The pruning performed by the constraint propagation process leads to irregularly structured search trees, requiring the use of 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 (*rescheduling*) is to dynamically generate smaller tasks from tasks that are still active, and move them from active clients to idle ones. The rescheduling is initiated by the scheduler, which selects, with a `Rescheduling` message, the client that has the estimated highest work load. Fig. 5 illustrates the rescheduling operation; the dashed areas have already been explored. The selected client will return to the scheduler the unexplored parts of its current task. These parts consist of a set of root nodes that partition the non-consumed task in a set of subtrees (e.g., nodes 3 and 4 in the Figure). The scheduler creates a new task for each subtree identified by the nodes received from the client (on the right in the Figure). This operation fragments the leftover task into smaller new tasks, added to the task queue and redistributed to the idle clients. To avoid excessive communication in determining the exact work-load in each client, we approximate the work-load by assuming that each task is a complete tree; the work-load is computed as the product of the number of unexplored children of the task's root and the average size of the subtrees. This estimate has worked well in practice. To maintain an acceptable task grain size, we impose a minimal threshold on the size of tasks (i.e., number of nodes and tree height) that can be redistributed.

This rescheduling strategy has been refined to better deal with cases where there are few, heavy tasks. In our initial implementation, clients accept rescheduling requests only when they have completed one of the children of the root of their assigned task (e.g., when the client has completed the subtree rooted at node 2 in Fig. 5). Each of the subtrees (nodes 1–4 in Fig. 5) is individually generated using the built-in labeling predicates of CLP($\mathcal{FD}$). This approach introduces a delay in the rescheduling phase—ideally, we would like the client to immediately return the available subtasks (e.g., nodes 3 and 4 in Fig. 5) without keeping the scheduler (and the idle clients) on hold. We limited this problem by increasing the frequency of the test for presence of rescheduling requests—the test is performed each time the computation backtracks into the top k levels of the task (k being a user-defined parameter). The choice of k is critical—to make the test possible, the first k levels cannot be implemented using the built-in labeling predicate, but using a (slower) user-defined labeling procedure. Thus, there is a trade-off between the increased computational time for exploring a tree and the delay before returning the preempted tasks. The results presented next make use of a factor $k = 2$ (in Fig. 5 this corresponds to the nodes 1–4 and their immediate children). The scheduler takes also care of detecting global termination.

4.4 Experimental results

The experiments have been conducted using an HP RP8400 NUMA architecture. The core constraint handling part is coded in SICStus. We consider four different modalities. In

[b] We use a compact bitmap representation of the path.

the first set of tests, we run the program in its simplest formulation, i.e., with no rescheduling and no distance/adjacency constraints provided by the analysis described in Section 2. In the second set of tests, we enable the basic rescheduling of tasks. In the third, we test the second rescheduling strategy ($k = 2$). In the last set, we use rescheduling and distance constraints.

In Table 2 (*Standard*), we report the running times for a set of six proteins from the PDB. The columns contain the protein ID, the number of tasks (# T) in which the search space is partitioned, and the parallel time in seconds required when using 1,2,4 and 8 clients. Next, we report the same tests using a larger initial set of tasks. The number of tasks in the left side is 10 and in the right side is 100. The speedups from the data in the Standard part of Table 2 appear, when using fewer tasks, to be poor. This happens when the problem is subdivided into few and heavy tasks, without the possibility to break them into smaller units. The performance improves using 100 tasks, since the load is fragmented in smaller tasks and it is more likely that every client ends at the same time (theoretical optimal scaling). Note also that the execution time on a single client increases when partitioning the problem into more tasks, since there is an overhead for constraint handling every time a subproblem is initialized. Moreover, during search in constraint programming, it is preferable to have a single search tree, since information for backtracking and pruning is lost when starting an exploration on a subtree from scratch. This suggests that the number of tasks should be kept small, and at the same time the smallest number of tasks should be rescheduled to guarantee that every processor is idle for a minimal amount of time. These considerations indicate that the rescheduling strategy, described in Sect. 4, should be applied whenever a client is idle. Table 2 (First rescheduling) reports the results for the same proteins using the rescheduling strategy. Comparing the Standard and First rescheduling tables, it is clear that the introduction of the strategy is effective, as it helps in balancing the load. The speedups and overall time are improved, even if an overhead for the rescheduling phase is introduced.

We now compare the benefits given by the introduction of the second rescheduling strategy (which provides a higher frequency of checking for rescheduling requests). Table 2 reports the times for the 10 tasks case, using the second rescheduling strategy. We can see that the speedups, using the second rescheduling strategy, are closer to the linear one. This strategy, though, costs more in terms of exploration time. There is a trade off between the search time and the wait before rescheduling a task. The table shows that the longer exploration time is balanced by the quicker rescheduling, when big tasks are handled by many clients. In these cases, it is preferable to split tasks as soon as possible, reducing delay and improving overlapping of actual computations. To scale the problem to more clients, it is reasonable to employ the second strategy and a small number of initial tasks.

The last results deal with the inclusion of the distance/adjacency constraints, coming from our helix search described in Section 2. In Table 3 we compare the three most computationally expensive proteins, with 100 initial tasks. The most striking difference is that the addition of simple constraints, which are implemented in few Prolog lines of code, is able to reduce the times by a factor up to 12 (see Table 3). Moreover the speedups of the parallel computations are only slightly decreased. We can also notice that the speedups are less uniform—due to the more irregular structure of the tasks, caused by the pruning from the new constraints.

	#	Standard "10 tasks"			
ID	T	1	2	4	8
1ECA	9	1.19	0.63	0.33	0.21
1L58	9	3.93	2.24	1.59	1.26
2TSC	11	4.70	2.60	1.50	0.89
2CYP	11	34.80	19.80	12.50	7.79
2PHH	10	55.80	29.70	17.70	17.40
3TIM	11	105.4	57.90	56.80	56.80
	#	Standard "100 tasks"			
ID	T	1	2	4	8
1ECA	81	1.42	0.72	0.37	0.19
1L58	90	4.10	2.11	1.12	0.64
2TSC	77	4.90	2.50	1.30	0.69
2CYP	154	35.10	17.80	9.10	4.80
2PHH	160	56.30	28.50	14.50	8.00
3TIM	143	105.6	55.20	33.20	21.40
	First rescheduling "10 tasks"				
1ECA	9	1.21	0.65	0.35	0.2
1L58	9	4.16	2.16	1.32	0.67
2TSC	11	4.73	2.41	1.28	0.82
2CYP	11	33.50	17.05	8.85	4.69
2PHH	10	56.11	29.26	14.79	8.92
3TIM	11	110.00	55.32	29.48	22.11
	First rescheduling "100 tasks"				
1ECA	81	1.52	0.76	0.40	0.20
1L58	90	4.26	2.16	1.09	0.57
2TSC	77	4.93	2.49	1.26	0.67
2CYP	154	34.98	17.57	8.78	4.62
2PHH	160	57.11	28.59	14.70	7.41
3TIM	143	108.23	54.33	27.60	14.76
	Second rescheduling "10 tasks"				
1ECA	9	1.34	0.72	0.37	0.22
1L58	9	4.14	2.11	1.28	0.62
2TSC	11	4.82	2.45	1.32	0.69
2CYP	11	34.54	18.03	9.16	4.73
2PHH	10	56.60	28.37	14.48	7.70
3TIM	11	107.38	56.14	27.47	14.06

Table 2 Execution Times

	No constraints (100 tasks)				Constraints (100 tasks)			
ID	1	2	4	8	1	2	4	8
2CYP	35.0	17.6	8.7	4.6	31.4	15.7	8.0	4.1
2PHH	57.1	28.6	14.7	7.4	3.9	2.0	1.0	0.6
3TIM	108.2	54.3	27.6	14.7	19.9	10.3	5.5	2.8

Table 3 Application of Additional Constraints

5 Conclusion and Future Work

In this paper, we presented a new constraint framework to determine the correspondence between the primary and tertiary structure of complex proteins. We addressed the problem using CLP($\mathcal{FD}$), and implemented a working prototype on a parallel machine.

We used the density maps obtained by means of EC, and defined a methodology to extract different types of constraints to help the mapping process. Constraints are employed to automatically determine the best placement of helices on the primary sequence, under the guidance of scoring functions from protein structure prediction techniques. The experimental results showed that the problem can be effectively parallelized, and the use of constraints drastically reduces the running times, allowing the effective analysis of large proteins with good accuracy.

In the future, we plan to improve the framework in many directions. First, we want improve the constraint extraction from the density maps. We expect that stronger constraints will further prune the search space; this will also raise some delicate issues in terms of ensuring proper load balancing during parallel execution (as already highlighted in this work in the rescheduling process). We will also refine the rescheduling strategies, to achieve better speedups, e.g., focusing on the tasks fragmentation and redistribution.

References

[1] K. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

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

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

[4] B.R. Brooks et al. CHARMM: A Program for Macromolecular Energy Minimization and Dynamics Calculations. *J. Comput. Chem.* 1983, **4**:187–217.

[5] A. Dal Palù et al. Identification of α-helices from low resolution protein density maps. *Computational Systems Bioinformatics Conference*, 2006.

[6] W. Chiu et al. Deriving Folds of Macromolecular Complexes through Electron Cryomicroscopy and Bioinformatics Approaches. In *Curr Opin Struct Biol.*, 12:263–269, 2002.

[7] J. He, Y. Lu, E. Pontelli. A Parallel Algorithm for Helices Mapping between 3D and 1D Protein Structure Using the Length Constraints. In *ISPA*, Springer Verlag, 2004.

[8] W. Jiang et al. Computational Tools for Intermediate Resolution Structure Interpretation. *J Mol Biol.*, 308, 2001.

[9] D.T. Jones. Protein Secondary Structure Prediction Based on Position-Specific Scoring Matrices. *J Mol Biol.*, 292, 1999.

[10] S. Kirkpatrick et al. Optimization by simulated annealing. *Science*, (220):671–680, 1983.

[11] K. Marriott and P. J. Stuckey. *Programming with Constraints*. The MIT Press, 1998.

[12] G. Pollastri, D. Przybylski, B. Rost P. Baldi. Improving the prediction of protein secondary structure in three and eight classes using recurrent neural networks and profiles. *Proteins*, 47(2):228-235, 2002.

[13] B. Rost. Protein secondary structure predication continues to rise. *J. Struct. Biol.*, 134:204–218, 2001.

[14] Swedish Institute of Computer Science. *The SICStus Prolog Home Page*. `www.sics.se`.

[15] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 2nd edition, 1997.

[16] Z.H. Zhou et al. Seeing the herpesvirus capsid at $8.5\mathring{A}$. *Science*, 288(5467):877–880, 2000.

[17] Z.H. Zhou et al. Electron Cryomicroscopy and Bioinformatics Suggest Protein Fold Models for Rice Dwarf Virus. In *Nat Struct Biol.*, 8, 2001.

A Experimental Results

Protein ID	Hel.	Dist/Adj Constr.	DomMin Dist	Search Type	Enum Perc	N. Sol found	Distance Max Real	Time
1ECA	6	Y	10	Max	-	1	-	0.15 s
		N	10	Max	-	1	-	0.31 s
		Y	5	Max	-	1	-	1.11 s
		N	5	Max	-	1	-	8.66 s
		Y	10	Enum	0.95	1	N.F.	0.95 s
		N	10	Enum	0.95	355	2.68%	1.60 s
		Y	5	Enum	0.95	88	2.86%	3.89 s
		N	5	Enum	0.95	4,875	1.44%	16.34 s
2TSC	7	Y	10	Max	-	1	-	1.62 s
		N	10	Max	-	1	-	1.72 s
		Y	5	Max	-	1	-	6.69 s
		N	5	Max	-	1	-	5.60 s
		Y	10	Enum	0.90	571	5.40%	5.41 s
		N	10	Enum	0.90	1,597	5.40%	6.63 s
		Y	5	Enum	0.95	573	3.54%	22.61 s
		N	5	Enum	0.95	1,849	3.54%	24.65 s
1L58	7	Y	10	Max	-	1	-	1.31 s
		N	10	Max	-	1	-	1.49 s
		Y	5	Max	-	1	-	5.75 s
		N	5	Max	-	1	-	11.46 s
		Y	10	Enum	0.95	13	0.16%	2.94 s
		N	10	Enum	0.95	307	4.18%	5.69 s
		Y	5	Enum	0.95	13	0.16%	13.32 s
		N	5	Enum	0.95	1,105	4.18%	44.41 s
2PHH	8	Y	10	Max	-	1	-	6.53 s
		N	10	Max	-	1	-	5.17 s
		Y	7	Max	-	1	-	4.25 s
		N	7	Max	-	1	-	8.66 s
		Y	10	Enum	0.95	19	N.F.	32.69 s
		N	10	Enum	0.95	9,289	N.F.	30.53 s
		Y	7	Enum	0.97	69	0.00%	21.58 s
		N	7	Enum	0.97	2,749	2.13%	21.70 s
3TIM	9	Y	10	Max	-	1	-	13.82 s
		N	10	Max	-	1	-	3.37 s
		Y	5	Max	-	1	-	28.80 s
		N	5	Max	-	1	-	3.62 s
		Y	10	Enum	0.95	257	0.82%	32.74 s
		N	10	Enum	0.95	17,321	0.93%	42.80 s
		Y	5	Enum	0.95	883	0.81%	2m 24 s
		N	5	Enum	0.95	83,521	0.66%	2m 14 s
2CYP	9	Y	10	Max	-	1	-	3m 58 s
		N	10	Max	-	1	-	6m 40 s
		Y	8	Max	-	1	-	6m 40 s
		N	8	Max	-	1	-	13m 40 s
		Y	10	Enum	0.97	641	N.F	7m 28 s
		N	10	Enum	0.97	11,961	N.F	24m 00 s
		Y	8	Enum	0.94	1,636	6.06%	12m 31 s
		N	8	Enum	0.94	265,545	3.88%	2h 55m

Table 4 CLP($\mathcal{FD}$) program: tests among different parameters.