A constraint solver for discrete lattices, its parallelization, and application to protein structure prediction*

A. Dal Palù
Dip. Matematica
Università di Parma
alessandro.dalpalu@unipr.it

A. Dovier
Dip. Informatica
Università di Udine
dovier@dimi.uniud.it

E. Pontelli
Dept. Computer Science
New Mexico State University
epontell@cs.nmsu.edu

May 30, 2006

Abstract

This paper presents the design, implementation and application of a *Constraint Programming* framework on *3D crystal lattices*. The framework provides the flexibility to express and resolve constraints dealing with structural relationships of entities placed in a 3D lattice structure in space. The paper describes both sequential and parallel implementations of the framework, along with experiments that highlight its superior performance w.r.t. the use of more traditional frameworks (e.g., constraints on finite domains and integer programming) to model lattice constraints.

The framework is motivated and applied to address the problem of solving the *protein folding prediction* problem—i.e., predicting the 3D structure of a protein from its primary amino acid sequence.

Results and comparison with performance of other constraint-based solutions to this problem are presented.

Keywords. Constraint Programming, Protein Structure Prediction, Parallel Processing

1 Introduction

Constraint Programming (CP) is the *"study of computational systems based on constraints"* [12], where by *constraint* we intend a logical relation between some unknowns in a mathematical model of a domain of interest. Thus, constraints are employed to capture dependencies between components of a problem domain, and they are employed to restrict the acceptable values that can be assigned to the variables representing such components. The field of constraint programming has received increased attention over the years [28, 4, 49]; CP offers declarative languages for modeling computationally hard problems, allowing the programmer to keep a clean separation between the model and the resolution procedures employed for searching solutions to the modeled problem. Several real-world applications of CP technology have been developed [49], and constraint frameworks dealing with different types of domains have been proposed, such as finite domains [59], reals [47], intervals [13], and sets [31].

In this paper, we develop a constraint programming framework for *discrete three dimensional (3D) crystal lattices* (e.g., see Figure 1). In this type of framework, variables denote points that have to be placed in

*__Corresponding Author:__ Enrico Pontelli, Department of Computer Science, New Mexico State University, Box 30001/CS, Las Cruces, NM 88003, USA, epontell@cs.nmsu.edu.

the crystal lattice and constraints describe relationships between different points that have to hold when the placement is done. These lattice structures have been adopted in different fields of scientific computing [17, 39], to provide a manageable discretization of the 3D space and facilitate the investigation of physical and chemical organization of molecular, chemical, and crystal structures. In particular, in recent years, lattice structures have become of great interest for the study of the problem of computing approximations of the folding of protein structures in 3D space [57, 5, 23, 24, 39]. Given the molecular composition of a protein, i.e., a list of amino acids (known as the *primary structure*, the problem is that of determining the three dimensional (3D) shape (*tertiary structure*) that the protein assumes in normal conditions in biological environments. The problem can be modeled as a minimization problem for an energy function, which depends on the 3D shape of the protein. In the second part of this paper, we will show how our proposed constraint framework on crystal lattices contributes to the solution of this problem.

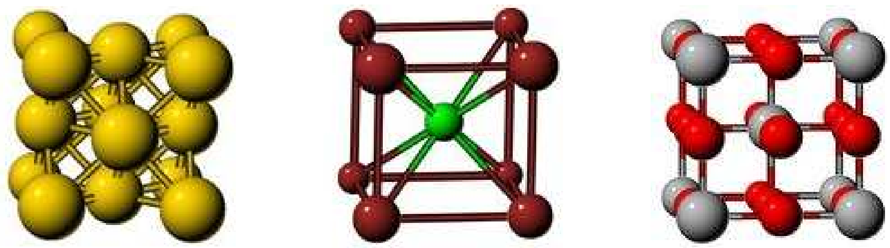


Figure 1: Sample Fragments of 3D Crystal Lattices

Traditional constraint solving domains (e.g., real numbers, finite domains) could be employed to model constraints in discrete lattices. For example, one could describe the possible placements of an object in the lattice structure through a collection of binary variables—one per lattice point, indicating whether the object is present or absent from that particular point. Alternatively, one could encode each *lattice point* (i.e., the atomic element of the lattice) as three independent values—representing the coordinates of the point in the lattice.

Various proposals have followed these approaches, e.g., [6, 23, 39]. These types of encoding lead to large constraint models, with many constraints and/or variables to be processed. Moreover, if we adopt the approach based on three variables to denote a lattice point, we will soon discover that, often, these variables are independent each other. This negatively affects the *propagation* stage—the individual coordinates are more "loosely" connected, making it harder to design constraints that allow changes to one coordinate of one object to be propagated to the coordinates of a different but related object [42, 52]. The outcome is a poor use of constraints and the need to explore large parts of the search space (i.e., the possible placements of points in the lattice) due to the inability of the constraints to prune it—leading to a *generate & test* solution, instead of the more desirable *constraint & generate* approach.

The approach we propose avoids these problems, by treating lattice points as *atomic values* of the constraint domain, and by designing constraint solving and constraint propagation techniques that directly target such atomic values. The resulting solver allows the native "finite domain" variables to represent 3D lattice points (*lattice variables*). Primitive constraints are introduced to capture basic spatial relationships within the lattice structure, such as relative positions, Euclidean, and lattice distances. We investigate constraint solving techniques in this framework, with a focus on propagation, search strategies, and automatic exploitation of parallelism. In particular, we develop an efficient built-in labeling strategy for lattice variables. We also investigate new search techniques for specific *rigid objects*—i.e., constraints representing collections of points that are part of a predetermined spatial structure, such as a cylinder with given entry and exit points and a known diameter. This last feature is essential to allow the representation and handling of secondary structure components in the previously mentioned protein structure prediction problem.

We describe the application of the constraint solver to the problem of *protein structure prediction*. The resulting encoding of the problem is highly declarative, allowing for ease of modification and for simple extendibility—e.g., simple introduction of new constraints representing additional knowledge about the protein we are modeling (e.g., knowledge about additional secondary structure elements). The experimental results show a dramatic improvement in performance over comparable encodings developed using traditional constraint logic programming over finite domains [24] (10^2–10^3 speedups w.r.t. SICStus 3.12.2 and ECLiPSe 5.8). The proposed solver also outperforms encodings of the same problem built using Integer Programming techniques—which are the most commonly used techniques to handle this type of problems in the literature.

The declarative nature of the problem encoding leaves complete freedom in the exploration of alternative search strategies, including the use of concurrent solutions to the problem. In particular, the exploration of the solutions search space is a highly non-deterministic process, with limited interaction between distinct branches of the search tree—each leading to a potentially distinct solution to the original problem. This provides a convenient framework for the concurrent exploration of different branches of the search tree, performed by distinct instances of the constraint solving engine. This paper explores the development of a parallel version of the proposed constraint solver, addressing the complex issues of dynamic scheduling and load balancing, and highlighting the significant improvements in performance that can be accomplished through the use of a parallel architecture—in our case, a Beowulf cluster.

The rest of the paper is organized as follows. Section 2 provides some basic definitions related to constraint satisfaction and constraint programming. Section 3 presents the formalization of our constraint solver over discrete 3D crystal lattices and discusses its sequential implementation. Section 4 describes a parallel version of the constraint solver, presenting the various design and implementation choices that have been made. In Section 5, we illustrate the application of the proposed constraint solver to the problem of protein structure prediction. This section includes experimental results using both the sequential and parallel version of COLA. This section includes also a comparison between our solution to the protein structure prediction problem and solutions developed using alternative frameworks (specifically, integer programming techniques and constraint logic programming over finite domains). Finally, Sections 6 and 7 provide a discussion of related works and the conclusions of the paper.

2 Preliminaries: Constraint Satisfaction Problems

In this section, we review some basic concepts and terminology associated to constraint satisfaction and constraint programming—the interested reader is referred to, e.g., [4, 28], for further details.

The modeling of a problem using constraint satisfaction makes use of a number of *unknowns*, described by *variables*, and related to each other by *constraints*. Let $\vec{X} = X_1, \ldots, X_k$ be a list of variables. Every variable X_i is associated to a set D_i, called its *domain*. If D_i is a finite subset of a totally ordered set, then we will denote with $\min(D_i)$ and $\max(D_i)$ the minimum and the maximum elements of D_i. For the sake of simplicity, if a domain D_i is the interval of integer numbers $\{a, a+1, a+2, \ldots, b\}$, then we will simply denote it with $a..b$.

Let `dom` be $D_1 \times \cdots \times D_k$. A ($n$-ary) *primitive constraint* C over `dom` is a relation $C \subseteq D_{i_1} \times \cdots \times D_{i_n}$, with $n \leq k$ and $\{i_1, \ldots, i_n\} \subseteq \{1, \ldots, k\}$. We say that $X_{i_1}, \ldots, X_{i_n}$ are the variables used by the constraint. A tuple $\langle d_{i_1}, \ldots, d_{i_n} \rangle \in D_{i_1} \times \cdots \times D_{i_n}$ *satisfies* the constraint C iff $\langle d_{i_1}, \ldots, d_{i_n} \rangle \in C$ (denoted by $\langle d_{i_1}, \ldots, d_{i_n} \rangle \models C$). We will often create more complex constraints, by composing primitive constraints via logical connectives. In particular, let C_1, C_2 be constraints over `dom`, where $C_1 \subseteq D_{i_1} \times \cdots \times D_{i_k}$ and $C_2 \subseteq D_{j_1} \times \cdots \times D_{j_h}$, then

- $C_1 \wedge C_2$ is a constraint, viewed as $(C_1 \wedge C_2) \subseteq D_{i_1} \times \cdots \times D_{i_k} \times D_{j_1} \times \cdots \times D_{j_h}$ and $\langle d_{i_1}, \ldots, d_{i_k}, d_{j_1}, \ldots, d_{j_h} \rangle \subseteq D_{i_1} \times \cdots \times D_{i_k} \times D_{j_1} \times \cdots \times D_{j_h}$ is a solution iff $\langle d_{i_1}, \ldots, d_{i_k} \rangle \in C_1$ and $\langle d_{j_1}, \ldots, d_{j_h} \rangle \in C_2$;
- $C_1 \vee C_2$ is a constraint, viewed as $(C_1 \vee C_2) \subseteq D_{i_1} \times \cdots \times D_{i_k} \times D_{j_1} \times \cdots \times D_{j_h}$ and $\langle d_{i_1}, \ldots, d_{i_k}, d_{j_1}, \ldots, d_{j_h} \rangle \subseteq D_{i_1} \times \cdots \times D_{i_k} \times D_{j_1} \times \cdots \times D_{j_h}$ is a solution iff $\langle d_{i_1}, \ldots, d_{i_k} \rangle \in C_1$ or $\langle d_{j_1}, \ldots, d_{j_h} \rangle \in C_2$;
- $\neg C_1$ is a constraint, viewed as $(\neg C_1) \subseteq D_{i_1} \times \cdots \times D_{i_k}$ and $\langle d_{i_1}, \ldots, d_{i_k} \rangle \in D_{i_1} \times \cdots \times D_{i_k}$ is a solution iff $\langle d_{i_1}, \ldots, d_{i_k} \rangle \notin C_1$.

Whenever it is clear from the context, we will denote a conjunction $C_1 \wedge \cdots \wedge C_k$ of constraints simply as the set $\{C_1, \ldots, C_k\}$. We will also denote with `fail` the constraint $\emptyset$—i.e., the constraint with no solutions.

A *Constraint Satisfaction Problem (CSP)* is described by three components:

- a list of variables $\vec{X} = X_1, \dots, X_k$,
- a corresponding collection of variable domains $D_1, \dots, D_k$, and
- a *finite* set of constraints $\mathcal{C}$ over $\mathtt{dom} = D_1 \times \cdots \times D_k$.

A CSP is denoted by $\mathcal{P} = \langle \mathcal{C}; D_\in \rangle$, where $D_\in$ is the formula $\bigwedge_{i=1}^{k} X_i \in D_i$—called the *domain expression* of $\mathcal{P}$. A tuple $\vec{d} = \langle d_1, \dots, d_k \rangle \in \mathtt{dom}$ is a *solution* of the CSP $\mathcal{P}$ if $\vec{d}$ satisfies every constraint $C \in \mathcal{C}$—or, more precisely, $\forall C \in \mathcal{C}$ we have that

$$\vec{d}\,|_{D_{i_1} \times \cdots \times D_{i_n}} \models C$$

where $\vec{d}\,|_{D_{i_1} \times \cdots \times D_{i_n}}$ is the projection of $\vec{d}$ on the domains $D_{i_1} \times \cdots \times D_{i_n}$ of C. The set of solutions of $\mathcal{P}$ is denoted by $\mathtt{sol}(\mathcal{P})$. If $\mathtt{sol}(\mathcal{P}) \neq \emptyset$, then $\mathcal{P}$ is *consistent*. Two CSPs $\mathcal{P}_1$ and $\mathcal{P}_2$, over the same list of variables $\vec{X}$, are *equivalent* if they admit the same set of solutions, i.e., $\mathtt{sol}(\mathcal{P}_1) = \mathtt{sol}(\mathcal{P}_2)$.

We often associate to a CSP $\mathcal{P} = \langle \mathcal{C}; D_\in \rangle$ a function $f : \mathtt{sol}(\mathcal{P}) \to E$, where E is a totally ordered set—e.g., $E = \mathbb{R}$ or $E = \mathbb{N}$. A *Constrained Optimization Problem (COP)* is a CSP $\mathcal{P}$ with an associated function f. A *solution* of a COP $\langle \mathcal{P}, f \rangle$ is a solution $\vec{d}$ of $\mathcal{P}$ that minimizes the function f on E, i.e., $\forall \vec{e} \in \mathtt{sol}(\mathcal{P})$ we have that $f(\vec{d}) \leq_E f(\vec{e})$—where $\leq_E$ denotes the total order relation on E.

A *Constraint Solver* (or, simply, a *solver*) is a procedure that transforms a CSP $\mathcal{P}$ into an equivalent CSP $\mathcal{P}'$. A solver is *complete* if $\mathcal{P}$ is transformed into an equivalent finite disjunction of CSPs that explicitly expresses all the solution(s) to the problem in the desired format (e.g., as a direct encoding of the tuples satisfying the CSP). In particular, if $\mathcal{P}$ is not consistent, we expect a complete constraint solver to return the constraint `fail`. Conversely, a solver is *incomplete* if it is not complete: $\mathcal{P}$ is transformed into a CSP (possibly "simpler" according to some comparison criteria), that however may not allow the immediate detection of (in)consistency and the extraction of the solutions.

Traditional constraint solvers are composed of two parts: a *constraint propagator* and a *solution search* component. Constraint propagation rewrites a constraint C into an equivalent one by applying rewriting rules, aimed at satisfying local consistency properties. The three most commonly used consistency properties are:

- *Node consistency:* A unary constraint C with variable X is *node consistent* whenever C is equal to the domain of X.
- *Arc consistency:* A binary constraint C, with variables X and Y, having domains D_X and D_Y, is *arc consistent* if (1) for each $v \in D_X$ there exists $v' \in D_Y$ such that $\langle v, v' \rangle \models C$ and (2) for each $v' \in D_Y$ there exists $v \in D_X$ such that $\langle v, v' \rangle \models C$.

- *Bounds consistency:* Let C be a binary constraint, with variables X, Y with domains D_X, D_Y. $\langle C; X \in D_X, Y \in D_Y\rangle$ is *bounds consistent* if $\langle C; X \in \min(D_X)..\max(D_X), Y \in \min(D_Y)..\max(D_Y)\rangle$ is *arc consistent.*

Let us observe that, if the variable domains are intervals, then bounds consistency and arc consistency coincide. If only one of the conditions for arc/bounds consistency is required, then the consistency is called *directional arc consistency.* A CSP $\langle \mathcal{C}; D_\in\rangle$ is *node* (*arc*, *bounds*) *consistent* if every unary (binary) constraint in $\mathcal{C}$ is node (arc, bounds) consistent. A node (arc, bounds) consistent propagator can be constructed to rewrite a non-consistent constraint into a consistent one, e.g., removing unacceptable values from the variables' domain. In general, node, arc, and bounds consistency propagators represent inconsistent solvers.

For finding solutions to a CSP, constraint propagation procedures have to be combined with a *search component*, used to explore the space of alternative assignments of values to variables. Traditional search components make use of *splitting* rules (e.g., domain splitting, constraint splitting). The most used splitting rule is *domain labeling.* Given a domain expression $X \in \{a_1, \dots, a_k\}$, domain labeling performs a k-way non-deterministic choice: $X \in \{a_i\}$ for $i = 1, \dots, k$. In this way, the variable X is non-deterministically assigned a value drawn from its domain. The search of solutions is achieved by alternating a propagation stage and a splitting stage. The search tree generated by this process is commonly known as the *prop-labeling tree*[1] [4]. Observe that each node (and, thus, the subtree rooted at that node) represents a CSP. The search terminates when all the variables have been selected and assigned.

Example 1. *Consider for instance the CSP $\langle X < Y; X \in \{1, 2\}, Y \in \{1, 2, 3\}\rangle$. In Figure 2, the prop-labeling tree searching all its solutions is reported. Double-line edges represent propagation steps. In the first step, arc consistency allows the solver to remove* 1 *from the domain of Y. The nodes with multiple children denote domain labeling points; in the first one, the domain of Y is split. Note that nodes at even levels of the tree are generated by branching (except for the root), while nodes at odd levels are computed using propagation rules.*

Given a CSP (or a COP) $\mathcal{P}$, the prop-labeling tree is uniquely determined by the propagation algorithm, the rule for selecting the next variable X to be labeled, and the rule for selecting the sequence of values for X.

During the computation, variables, their domains, and all the constraints are stored in a suitable data structure, called the *constraint store.*

[1] We will often refer to it simply as *search tree.*

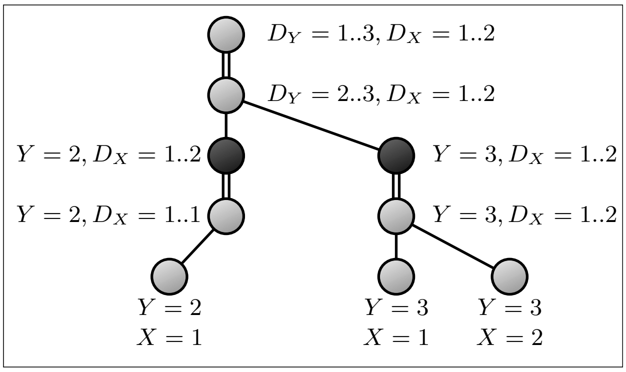


Figure 2: Prop-labeling-tree for $\langle X < Y; D_X = \{1, 2\}, D_Y = \{1, 2, 3\}\rangle$

3 COLA—COnstraint Solver on LAttices

In this section, we describe a framework developed to solve *Constraint Satisfaction Problems* (CSPs) modeled on 3D lattices. The solver allows us to define lattice variables, with their associated domains, constraints over these variables, and to search the space of admissible solutions.

3.1 Crystal Lattice models

Lattice models have long been used for protein structure prediction (see [57] for a survey) and in general for representing 3D structures.

Definition 1 (Lattice)**.** *A* crystal lattice *is a graph* (P, E)*, where* P *is a set of 3D points* $(x, y, z) \in \mathbb{Z}^3$*, connected by undirected edges* (E)*.*

Given an element $v \in P$, we denote with v_x, v_y, and v_z its three coordinates.

Lattices contain strong symmetries and present regular patterns repeated in the space. If all nodes have the same degree δ (i.e., the number of edges outgoing from a node), then the lattice is said to be δ*-connected.* Given $A, B \in P$, we define two notions of *distance* between elements of the lattice:

- the *squared Euclidean distance* of A, B is defined as:

$$eucl(A, B) = (B_x - A_x)^2 + (B_y - A_y)^2 + (B_z - A_z)^2$$

- the *norm infinity* of A, B is defined as:

$$norm_\infty(A, B) = \max\{|B_x - A_x|, |B_y - A_y|, |B_z - A_z|\}$$

Observe that $norm_\infty(A, B)^2 \leq eucl(A, B)$.

In this paper, we deal only with two types of lattices (CUBE and $\mathcal{FCC}$). Thus, in the rest of the discussion, whenever we talk about lattices, we will implicitly assume one of the two lattice structures described next. However, the framework can be adapted, with minor changes, to handle other types of discrete lattices.

Definition 2 (CUBE)**.** *A* cubic lattice (CUBE) (P, E) *is defined by the following properties:*

- $P = \{(x, y, z) \mid x, y, z \in \mathbb{Z}\}$*;*
- $E = \{(A, B) \mid A, B \in P, eucl(A, B) = 1\}$*.*

Let us observe that CUBE is 6-connected.

Definition 3 ($\mathcal{FCC}$)**.** *A $\mathcal{FCC}$ lattice (P, E) is defined by the following properties:*

- $P = \{(x, y, z) \mid x, y, z \in \mathbb{Z} \wedge x + y + z \text{ is even}\}$*;*
- $E = \{(A, B) \mid A, B \in P, eucl(A, B) = 2\}$*.*

The $\mathcal{FCC}$ model is based on cubes with sides of length 2, where the central point of each face is also an admissible point. The practical rule to compute the points belonging to the lattice is to check whether the sum of the points coordinates (x, y, z) is *even* (see Figure 3). Pairs of points at squared Euclidean distance 2 are linked and form the edges of the lattice; their distance is called *lattice unit*. Observe that, for lattice units, it holds that $|x_i - x_j| + |y_i - y_j| + |z_i - z_j| = 2$. The $\mathcal{FCC}$ lattice is 12-connected.

In [53] it is shown that the *Face-Centered Cubic Lattice* ($\mathcal{FCC}$) model is a well-suited, realistic model for 3D conformations of proteins (see also [1, 5, 39, 23]).

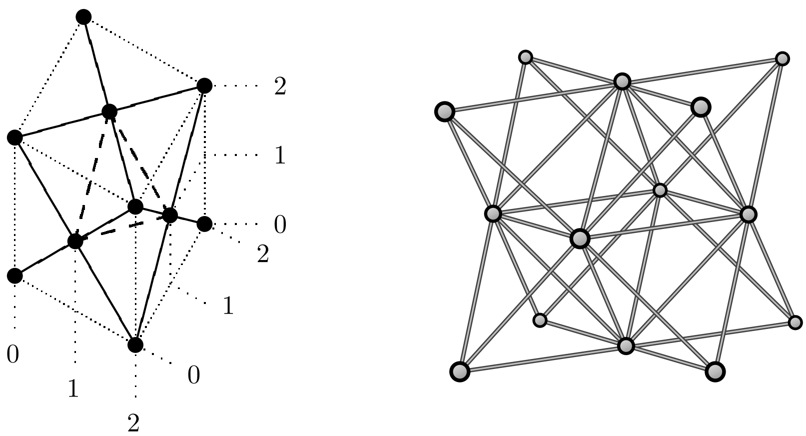


Figure 3: A cube of the $\mathcal{FCC}$ lattice. Thick lines are edges. Dashed lines represent edges inside the cube. On the right, the full set of edges in a unit cell.

3.2 Domains and Variables

Let (P, E) be the considered lattice. A *domain* D is described by a pair of points of $\langle \underline{D}, \overline{D} \rangle$, where $\underline{D} = (\underline{D}_x, \underline{D}_y, \underline{D}_z) \in \mathbb{Z}^3$ and $\overline{D} = (\overline{D}_x, \overline{D}_y, \overline{D}_z) \in \mathbb{Z}^3$. $\underline{D}$ and $\overline{D}$ are not necessarily lattice points. D implicitly defines a *box*:

$$Box(D) = \left\{(x, y, z) \in P \;:\; \underline{D}_x \leq x \leq \overline{D}_x \;\wedge\; \underline{D}_y \leq y \leq \overline{D}_y \wedge \underline{D}_z \leq z \leq \overline{D}_z \;\right\}$$

Intuitively, the box represents the intersection between the lattice structure and the volume, in the 3D space, delimited by the two points. We only handle the bounds of the effective domain, since a detailed representation of all the individual points in a volume of interest would be infeasible (due to the sheer number of points involved). The approach follows the same spirit as the manipulation of finite domains using bounds consistency (as mentioned in Section 2). We say that D is *admissible* if $Box(D) \neq \emptyset$. D is *ground* if it is admissible and $\underline{D} = \overline{D}$.

We introduce the following operations on domains:

- *Domain intersection*: Given two domains D and E, their *intersection* is defined as follows: $D \cap E = \langle \uparrow(D, E), \downarrow(D, E) \rangle$ where:
  - $\uparrow(D, E) = (\max\{\underline{D}_x, \underline{E}_x\}, \max\{\underline{D}_y, \underline{E}_y\}, \max\{\underline{D}_z, \underline{E}_z\})$
  - $\downarrow(D, E) = (\min\{\overline{D}_x, \overline{E}_x\}, \min\{\overline{D}_y, \overline{E}_y\}, \min\{\overline{D}_z, \overline{E}_z\})$
- *Domain dilation*: Given a domain D and a positive integer d, we define the *domain dilation* operation $D + d$, used to enlarges $Box(D)$ by $2d$ units, as:

$$D + d = \langle (\underline{D}_x - d,\ \underline{D}_y - d,\ \underline{D}_z - d),\ \ (\overline{D}_x + d,\ \overline{D}_y + d,\ \overline{D}_z + d) \rangle$$

- *Domain union:* Given two domains D and E, their *union* is defined as $D \cup E = (\min(D, E), \max(D, E))$, where
  - $\min(D, E) = (\min(\underline{D}_x, \underline{E}_x), \min(\underline{D}_y, \underline{E}_y), \min(\underline{D}_z, \underline{E}_z))$
  - $\max(D, E) = (\max(\overline{D}_x, \overline{E}_x), \max(\overline{D}_y, \overline{E}_y), \max(\overline{D}_z, \overline{E}_z))$

Each (lattice) variable V, that represents lattice points, is associated to a *domain* $D^V = \langle \underline{D}^V, \overline{D^V} \rangle$.

Figure 4 depicts, from left to right, the domains associated to variables V and W, the dilation of domain of variable V (i.e., $D^V + d$), the intersection between $D^V + d$ and D^W, and the union between D^V and D^W.

3.3 Constraints

Given two lattice variables V_1, V_2, $d \in \mathbb{N}$, $B_1 = Box(D^{V_1})$, $B_2 = Box(D^{V_2})$ and P_1, P_2 lattice points, we define the following primitive constraints:

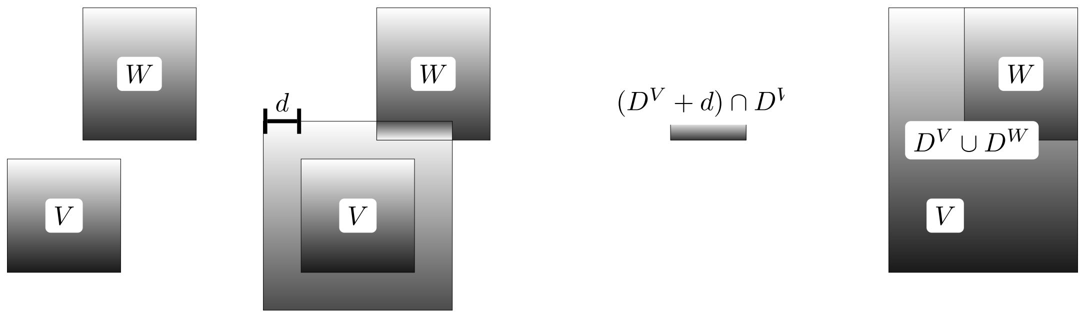


Figure 4: Example of dilation, intersection, and union

$$
\begin{array}{lcl}
\triangle(V_1, V_2) \leq d & \Leftrightarrow & \exists P_1 \in B_1, \exists P_2 \in B_2 \ s.t. \ norm_\infty(P_1, P_2) \leq d \\
\delta(V_1, V_2) \leq d & \Leftrightarrow & \exists P_1 \in B_1, \exists P_2 \in B_2 \ s.t. \ eucl(P_1, P_2) \leq d \\
\delta(V_1, V_2) \geq d & \Leftrightarrow & \exists P_1 \in B_1, \exists P_2 \in B_2 \ s.t. \ eucl(P_1, P_2) \geq d
\end{array}
$$

Intuitively, Δ encodes a constraint which restricts the $norm_\infty$ distance between two points, while the δ constraints encode lower and upper bound constraints on the Euclidean distance between points.

We will use also the constraint $\delta(V_1, V_2) = d$, which is a syntactic sugar for the conjunction of the constraints $\delta(V_1, V_2) \leq d$ and $\delta(V_1, V_2) \geq d$. The constraint $\triangle(V_1, V_2) \leq d$, based on the infinity norm, is introduced to provide an approximation of $\delta(V_1, V_2) \leq d$ which can be efficiently implemented.

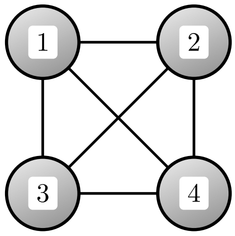
In this setting a CSP is a pair $\langle C; D_\in \rangle$, where C is a conjunction of constraints of the form above and $D_\in$ is a conjunction of domain expressions $V \in D^V$ (see Sect. 2).

Theorem 1. *The general problem of deciding whether a CSP in the lattice framework admits solutions is NP-complete.*

Proof sketch. The problem is clearly in NP, since a witness of a solution can be represented as a list of coordinates, containing a number of elements that has the same order as the input, and it can be verified in polynomial time. To show the NP-hardness, we reduce the *Graph 3-Colorability Problem* of an undirected graph $G(V, E)$ to our CSP. For the sake of simplicity, we refer here to the CUBE lattice.[2] For each node $n_i \in V$, we introduce a variable V_i with domain $D^{V_i} = \langle (0,0,0), (0,0,2) \rangle$. $Box(D^{V_i})$ contains three lattice points $(0, 0, j)$, corresponding to the color j. For every edge $e = (n_i, n_j)$, we add the constraint $\delta(V_i, V_j) \geq 1$, that constrains the points represented by the variables to be at a distance greater than 0 (i.e., have a different color). See Figure 5 for an example. It is easy to see that a solution of the constraint system can be used to determine a valid coloring of the original graph. □

When dealing with lattice variables, it can be convenient to describe some of their spatial properties by means of a global constraint. In particular, we introduce a *rigid block constraint*, that defines a layout of

[2] For other lattices, additional $\delta(\_, \_) \geq d$ constraints might be required, to identify 3 points in the box.



Variables:	V_1, V_2, V_3, V_4.
Domains:	$\forall i \in \{1,2,3,4\}$. $D^{V_i} = \langle (0,0,0), (0,0,2) \rangle$
Constraints:	$\forall i, j \in \{1,2,3,4\}$. $\delta(V_i, V_j) \geq 1$

Figure 5: An example of a non-satisfiable graph 3-colorability problem

points in the space that has to be respected by all admissible solutions. Let $\vec{V} = V_1, \ldots, V_k$ be a list of lattice variables, and $\vec{B} = B_1, \ldots, B_k$ a list of lattice points (that, intuitively, describe the desired layout of the rigid block). $\mathsf{block}(\vec{B}, \vec{V})$ is a k-ary constraint, whose solutions are assignments of lattice points to the variables $\vec{V}$, that can be obtained from $\vec{B}$ modulo translations and rotations. More precisely, we define a *rotation* of a lattice point $p = (p_x, p_y, p_z)$ as the formula[3] $rot(\phi, \theta, \psi)(p) = X \cdot Y \cdot Z \cdot p^T$, where

$$X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi \\ 0 & -\sin\phi & \cos\phi \end{bmatrix}, \ Y = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}, \ Z = \begin{bmatrix} \cos\psi & \sin\psi & 0 \\ -\sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Although the rotation angles ϕ, θ, ψ are real valued, only few combinations of them define automorphisms on the lattice in use. The total numbers of distinct automorphisms r depends on the lattice—e.g., in CUBE, we have that $r = 16$, and in the $\mathcal{FCC}$ we have that $r = 24$. We extend the definition of rotation to lists of values, $rot(\phi, \theta, \psi)(\vec{B})$, where $\vec{B}$ is a list of points and the result is a list in which every element of $\vec{B}$ is rotated according to the previous definition.

Given a list of points $\vec{B}$, we define the concept of *templates* as the set:

$$\mathsf{Templ}(\vec{B}) = \{rot(\phi, \theta, \psi)(\vec{B}) \mid \exists \phi, \theta, \psi \text{ that generate an automorphism on the lattice}\}$$

which contains the distinct 3-dimensional rotations of the points $\vec{B}$ in the lattice. Note that, for a given list of points ($\vec{B}$), the cardinality of $\mathsf{Templ}(\vec{B})$ is at most r.

We say that $\vec{\ell} = (\ell_x, \ell_y, \ell_z)$ is a *lattice vector* if the translation by $\vec{\ell}$ of lattice points generates an automorphism on the lattice. Note that, for some asymmetric lattices, it is possible that lattice vectors do not exist.

Let $\vec{\ell}$ be a lattice vector; with $\mathsf{Shift}[\vec{\ell}]$ we denote a mapping that translates a rigid block according to the vector ℓ. Formally, for each $i = 1, \ldots, k$, $\mathsf{Shift}[\vec{\ell}](\vec{B})[i] = B_i + \vec{\ell}$. Shifts are used to place a template into the lattice space, preserving the orientation and the distances between points. A rigid block constraint

[3] The symbol $\cdot$ here denotes matrix multiplication.

$\mathsf{block}(\vec{B}, \vec{V})$ is satisfied by a variable assignment σ of $\vec{V}$ to lattice points if and only if there is a lattice vector $\vec{\ell}$ and a template $P \in \mathsf{Templ}(\vec{B})$ such that $\mathsf{Shift}[\vec{\ell}](P) = V_1\sigma, \ldots, V_k\sigma$.

3.4 Constraint Solving

The general approach to constraint solving adopted in this work relies on a standard combination of consistency techniques and systematic search. In particular, we make use of algorithms that attempt to extend a partial and consistent assignment (of variables to lattice points) to an assignment that is complete and satisfies all the constraints. The overall structure of a search algorithm that can be used is shown in Figure 6. Intuitively, the algorithm alternates consistency enforcing (e.g., bounds consistency)—performed by the `AC-3` procedure—with guessing the value to be assigned to a variable. The pick steps selects a variable X such that $|Box(D_X)| \geq 2$. Depending on the variable selection strategy, the variable selected is the one that satisfies the leftmost property (i.e., the non-labeled variable with the lowest index) and/or the first-fail property (i.e., the variable with the smallest $|Box(D_X)|$).

The `AC-3` procedure reduces domains of variables to ensure bounds consistency, and makes use of a queue to consider only the constraints for which changes have been applied. The procedure reaches a fixpoint when the propagation rules applied to $Vars$ are not able to restrict any domains. The structure of the algorithm is illustrated in Figure 7.

```
procedure chrono_search (Vars,C)
    AC-3(Vars,C)
    BT_search(Vars, ∅, C)

procedure BT_search(Vars, Done, C)
    if (Vars = ∅) then
        return Done
    pick X from Vars
    for each a in D_X do
        if (no constraint is violated by assigning a to X) then
            AC-3(C ∪ Done ∪ {X = a},Vars)
            R = BT_search(Vars \{X}, Done ∪{X = a}, C)
            if (R ≠ fail) then
                return R
    return fail
end
```

Figure 6: Backtracking search

If the problem is a COP, then the backtracking search should be enhanced to search for an optimal solution. If no further information is available (e.g., heuristics to guide the choice of variables and the order in which the different values for a variable are tried), then we can simply introduce a branch&bound component in the search.

```
procedure AC-3 (𝒞,Vars)
    Q = {C(X,Y) ∈ 𝒞 | C(X,Y) is a binary constraint and {X,Y} ∩ Vars ≠ ∅}
    while Q is not empty do
        select C(X,Y) from Q and remove it
        apply the propagation rule associated to C(X,Y)
        if ((D^X or D^Y have been modified) ∧ bounds_consistent(X,Y)) then
            Q = Q ∪ {C ∈ 𝒞 | C is a constraint on a modified variable}
end
```

Figure 7: AC-3 procedure for bounds-consistency

3.4.1 Consistency

We define the following rewriting rules, which are used to modify the domains of variables to ensure a form of bounds consistency for COLA constraints. Let us define the set D that contains the other domains as $D = D_{\in} \setminus \{A \in D^A, B \in D^B\}$.

The constraint $\triangle(A,B) \leq d$ states that the variables A and B are distant no more than d in $norm_\infty$. It can be employed to simplify domains through bounds consistency. The formal rule is:

$$[\triangle(A,B) \leq d] : \frac{\{A \in D^A, B \in D^B\} \cup D}{\{A \in ((D^B + d) \cap D^A), B \in ((D^A + d) \cap D^B)\} \cup D} \tag{1}$$

The constraint $\delta(A,B) \leq d$ states that A and B are at squared Euclidean distance less than or equal to d. The sphere of radius $\sqrt{d}$, that contains the admissible values defined by the constraint, can be approximated by the minimal surrounding box that enclose it (a cube with side $2\lceil\sqrt{d}\rceil$). The formal propagation rule is:

$$[\delta(A,B) \leq d] : \frac{\{A \in D^A, B \in D^B\} \cup D}{\{A \in ((D^B + \lceil\sqrt{d}\rceil) \cap D^A), B \in ((D^A + \lceil\sqrt{d}\rceil) \cap D^B)\} \cup D} \tag{2}$$

Performing propagation in the context of the $\delta(A,B) \geq d$ constraint is considerably harder, due to the coarse resolution of the box representation of the domain. A simple form of bounds consistency that can be applied in this case is described by the following rule:

$$[\delta(A,B) \geq d] : \frac{\{A \in D^A, B \in D^B\} \cup D, D^A \cup D^B = \langle u,v\rangle, \delta(u,v) < d}{\{A \in \emptyset, B \in \emptyset\} \cup D} \tag{3}$$

which is used to detect domains that do not contain points that are sufficiently far apart to satisfy the constraint.

Theorem 2. *The propagation rules* (1), (2), *and* (3) *are correct—i.e., if we have that* $\frac{CS_1}{CS_2}$ *using rule* (1), $(2), or (3)$*, then* CS_1 *and* CS_2 *are equivalent.*

Proof. The correctness proof is composed of two parts: we need to show that the rules do not introduce any new solutions and that the rules do not remove any solutions.

Introduction of new solutions: The introduction of new solutions is obviously impossible, since the operations performed in the three rules are based on a domain reduction, by means of the intersection operator (or by completely emptying the domain, as in rule (3)), and thus there is no possibility to add new points to the domains.

Removal of solutions: We now prove that no solution is removed. For the rule (1), the semantics of the constraint is $\triangle(V_1, V_2) \leq d \Leftrightarrow \exists P_1 \in D^{V_1}, \exists P_2 \in D^{V_2}$ *s.t.* $norm_\infty(P_1, P_2) \leq d$. It suffices to show that the points removed by R_1 from D^{V_1} and from D^{V_2} are such that $norm_\infty(P_1, P_2) > d$. This requires proving the following two (symmetric) cases: *(i)* each point removed from D^{V_1} has distance larger than d, in $norm_\infty$, from any point in D^{V_2}, and *(ii)* each point removed from D^{V_2} has distance larger than d, in $norm_\infty$, from any point in D^{V_1}. Let us prove the latter case (the first case can be proved in an analogous way). Let us define the set R of removed points as the difference between the original domain and the resulting domain, i.e., $R = D^B \setminus D^{B'}$. This leads to:

$$R = D^B \setminus D^{B'} \quad \text{iff} \quad R = D^B \setminus (D^A + d) \cap D^B \quad \text{iff} \quad R = D^B \setminus (D^A + d)$$

By definition of dilation, the points in $(D^A + d)$ are the ones in the Box:

$$\langle\ (\underline{D^A}_x - d, \underline{D^A}_y - d, \underline{D^A}_z - d),\ (\overline{D^A}_x + d, \overline{D^A}_y + d, \overline{D^A}_z + d)\ \rangle.$$

It follows that for each point $P \notin (D^A + d)$ and each point $Q \in D^A$ it holds that $norm_\infty(P, Q) > d$. By contradiction, if there is a $P = (P_x, P_y, P_z)$ such that $norm_\infty(P, Q) \leq d$, by definition of $norm_\infty$, $\max\{|Q_x - P_x|, |Q_y - P_y|, |Q_z - P_z|\} \leq d$. This is a contradiction, since $P \notin (D^A + d)$ implies that $|Q_x - P_x| > d$ or $|Q_y - P_y| > d$ or $|Q_z - P_z| > d$ for any point Q. Therefore, the points removed from D^B are certainly not admissible according to the semantics of the rule (1).

The proof for rule (2) follows from the previous one. We already noted that $norm_\infty(A, B)^2 \leq eucl(A, B)$. The propagation rule R_2 uses the $norm_\infty$ with distance $\lceil\sqrt{d}\rceil$ to approximate the $eucl$ distance of d. Since the points maintained by the rule (the box surrounding the sphere of radius $\sqrt{d}$) are more than the correct ones (the sphere of radius $\sqrt{d}$), it follows that (2) is correct as well.

The proof for rule (3) is straightforward. □

3.4.2 Solution Search

As shown in the general algorithm in Figure 6, the consistency phase is activated whenever the domain of a variable is modified.

Let us consider a situation where the variables $G = \{V_1, \ldots, V_{k-1}\}$ have been bound to specific values, V_k is the variable to be assigned next (as result of the `pick` step of the backtracking search algorithm), and let $NG = \{V_{k+1}, \ldots, V_n\}$ be all the remaining variables. The first step, after the labeling of V_k, is to check the consistency of the constraints of the form $C(V_k, V_i)$, where $V_i \in G$ (*node consistency* check). For efficiency reasons, the successive *propagation* phase is divided in two steps, that are equivalent to the `AC-3` procedure in Figure 7. First, all the constraints of the form $C(V_k, V_j)$ are processed, where $V_j \in NG$. This step propagates the new bounds of V_k to the variables not yet labeled. Thereafter, bounds consistency, using the same outline of `AC-3`, is applied to the constraints of the form $C(V_i, V_j)$, where $V_i, V_j \in NG$. We carefully implemented a constant-time insertion for handling the set of constraints to be revisited, using a combination of an array to store the constraints and an array of flags for each constraint. This leads to the following result:

Theorem 3. *Each propagation phase has a worst-case time complexity of $O(n+ed^3)$, where n is the number of variables involved, e is the number of constraints in the constraint store, and d the maximum domain size.*

Proof sketch. Let us assume that the variable V_i is labeled. Each propagation for a constraint costs $O(1)$, since only arithmetic operations are performed on the domain of the second variable. Let us assume that for each pair of variables and type of constraint, at most one constraint is deposited in the constraint store (this can be guaranteed with an initial simplification). In the worst case, there are $O(n)$ constraints of the form $C(V_i, V_j, d)$, where V_j is not ground. Thus, the algorithm propagates the new information in time $O(n)$, since each constraint costs constant time. The worst-case time complexity of `AC-3` procedure is $O(ed^3)$, where e is the number of constraints in the constraint store and d the maximum domain size. $\square$

Theorem 4. *The exclusive use of CSP rewriting based on consistency routines is, in general, incomplete.*

Proof sketch. Since it is possible to encode NP-complete problems (as shown in Theorem 1), and since the rewriting procedures described earlier are polynomial, the existence of a complete solver based only on such procedures would imply that P=NP. We present here a non-satisfiable instance of the graph 3-coloring problem that cannot be resolved using only application of the consistency procedures.

Let us consider the graph $G = (\{1,2,3,4\}, \{\{1,2\},\{1,3\},\{1,4\},\{2,3\},\{2,4\},\{3,4\}\})$—see also Figure 5, it can be seen immediately that the problem is not satisfiable. Let us show now how the problem is encoded into a CSP. We define variables V_1, V_2, V_3 and V_4. For each $i = 1, \ldots, 4$, we define the domains as follows: $D^{V_i} = \langle (0,0,0), (0,0,2) \rangle$. Finally, for every $i, j = 1, \ldots, 4$, we add the constraints $\delta(V_i, V_j) \geq 1$. Clearly, the application of constraint consistency to this CSP has no effect in reducing the domains, since the constraints $\delta(\_, \_) \geq d$, in this case, do not allow any simplification of domains. $\square$

To gain completeness, it is necessary to incorporate a backtracking search (as in Figure 6), which guarantees detection of solutions—possibly with an exponential worst-case time complexity.

3.4.3 An Example

We present here a CSP problem and its encoding in COLA. The problem we address is the one of connecting pins of a multi-layer hardware chip. The pins are located in specific positions in the 3D space, and the goal is to determine the possible connections by means of *wires* that can be physically arranged in the space without overlapping each other. We assume the problem to be discretized in the cubic lattice (even though it is possible to adapt it to other kinds of lattices).

The input of the problem consists of a set of lattice points (P), used to model the pins' positions, and a set of pairs of pins to be connected ($C \subseteq P \times P$). For each connection $c_i = (pb_i, pe_i) \in C$, there is a wire W_i, $i \in \{0, \ldots, |C| - 1\}$, that is modeled as a list of lattice points $W_i = [p^i_0, \ldots, p^i_{k-1}]$, where k is the number of points modeling each wire. The problem using a collection of constraints, stating that:

- The first and last lattice points of each wire W_i are equal to pb_i and pe_i, respectively.
- For each pair of consecutive points p^i_j and p^i_{j+1}, the distance between them is less or equal than 1.[4] Note that each wire is composed of k points. For solutions that require less than k points, we can maintain the same model and allow some of the k points to overlap (thus, certain consecutive points will have distance 0).
- For each $i \neq j$ and $l, m \in \{0, \ldots, k-1\}$, if $(l \neq 0 \vee l \neq k-1) \wedge (m \neq 0 \vee m \neq k-1)$, then the points p^i_l and p^j_m cannot overlap, i.e., they are at distance greater than or equal to 1. This states the non-overlapping constraint, except for those cases in which two extremes of two wires are selected—since the pins could be involved in more than one connection, the non-overlapping constraint is not applied.

Technically, we can encode this CSP in COLA as:

- $\forall i \in \{0, \ldots, |C| - 1\}.\ W_i = [p^i_0, \ldots, p^i_{k-1}]$ (Variables);
- $\forall i \in \{0, \ldots, |C| - 1\}.\ p^i_0 = pb_i, p^i_{k-1} = pe_i$;
- $\forall i \in \{0, \ldots, |C| - 1\}, 0 \leq j \leq k-1.\ \delta(p^i_j, p^i_{j+1}) \leq 1$;
- $\forall i, j \in \{0, \ldots, |C| - 1\}, i \neq j, \forall l, m \in \{0, \ldots, k-1\}.$

$$(l \neq 0 \vee l \neq k-1) \wedge (m \neq 0 \vee m \neq k-1) \Rightarrow \delta(p^i_l, p^j_m) \geq 1.$$

The complete encoding of this problem can be seen in Appendix 1.

[4] in the case of cubic lattice; for $\mathcal{FCC}$ it is $\sqrt{2}$

3.5 Sequential Implementation

In this section, we describe some of the design choices adopted in the development of a sequential implementation of the COLA framework.

3.5.1 Variables and Constraint Representation

The set of variables adopted in the problem encoding (`Allvariables`) is represented by a static array, created during the problem definition phase. Each variable (see Figure 8) is a record that contains an identifier (`ID`, i.e., the record's position in the array), the flags telling whether the variable is `labeled`, `ground`, `failed` and `changed`, the `size` of $Box(D)$ (i.e., the integral volume of the box, used for variable selection strategies) and the points $\underline{D}$ and $\overline{D}$ that represent the domain D. The possible states of each variable are:

- `labeled` indicates that the variable has been selected (through the pick operation of the algorithm in Figure 6), labeled, and included in the search tree,
- `ground` indicates that the variable has a domain size equal to 1 (either because of an explicit assignment or because of application of consistency techniques),
- `failed` indicates that the domain of the variable has become empty, and
- `changed` indicates that the variable is in none of the other states and its domain has just been modified.

Each binary constraint over variables V_i and V_j is represented by a record that stores the identifiers of the variables (`var1, var2`, shown with dashed pointers in Figure 8), the `type` of binary constraint and the Euclidean distance associated to the constraint.

The collection of constraints present in the CSP is represented by a *constraint store*. The constraint store is realized as a dynamic array. Although all the constraints considered here are symmetric, for the sake of simplicity in the implementation, we treat them as *directional* constraints, using the information of the first (leftmost) domain to test and/or modify the second domain (bounds consistency—i.e., see Section 2). Consequently, every time a constraint over two variables has to be expressed, we require the addition of both directions of the constraint in the constraint store. E.g., the constraint $\delta(V_1, V_2) \leq d$ is actually implemented as $\delta(V_1, V_2) \leq d$ and $\delta(V_2, V_1) \leq d$.

Whenever a constraint is added to the store, a new *constraint object* is generated and introduced in the constraint store array. The constraint data type stores the information about the variables involved in the constraint (we assume here binary constraints), the type of constraint, and the distance parameter present in the constraint. In order to allow an efficient implementation of the consistency procedures, we introduce an additional data structure that provides direct access to constraints that involve a specific variable V_i. We define a set of dynamic arrays, one for each variable V_i, that contain pointers to each constraint $C(V_i, V_j)$

(see solid pointers in Figure 8). During the consistency phase, after the modification of a domain V_i, the set of constraints involved in possible further propagation operations are retrieved from the indexes contained in the array for V_i, without the need of repetitive scans of the complete constraint store.

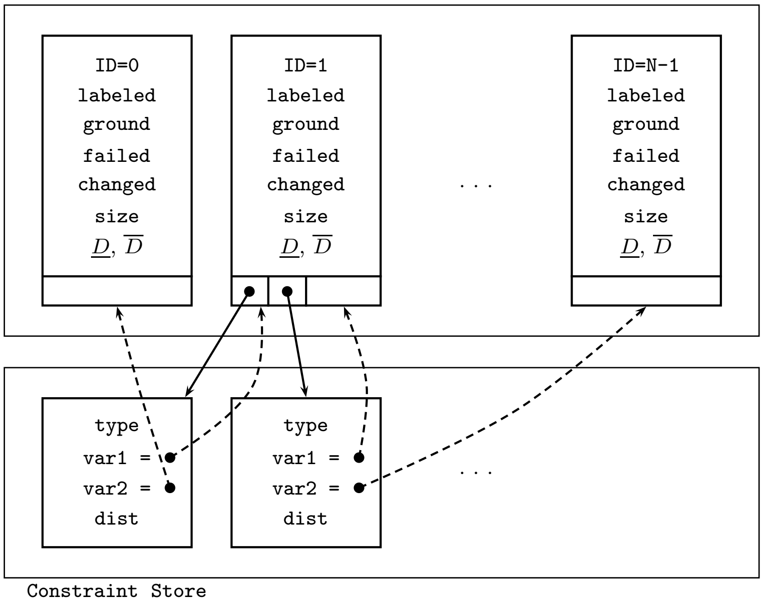


Figure 8: Constraint solver data structures

Rigid block constraints are handled with a different method. For each constraint $\mathsf{block}(\vec{B}, \vec{V})$, each allowed rotation in the lattice for the pattern $\vec{B}$ is precomputed and stored in a vector of list of points—namely the set $\mathsf{Templ}(\vec{B})$ defined in Section 3.3. During the search phase, the block constraint is analyzed whenever the first variable V_i in $\vec{V}$ is labeled—and, thus, all the remaining variables $\vec{V} \setminus \{V_i\}$ are not labeled. Let L be the lattice point associated to V_i. This choice uniquely determines the $\mathsf{Shift}[\vec{\ell}]$ for a template $P \in \mathsf{Templ}(\vec{B})$, in order to correctly place the block in the lattice. In particular, $\vec{\ell} = L - P_i$, i.e., the shift operator translates the rotated pattern in such a way that the pattern point associated to V_i is shifted to the lattice point L. For each rotation, and corresponding $\vec{\ell}$, the whole block is instantiated and consistency with other constraints is enforced (see also the successive Section).

3.5.2 Search Space

The evolution of the computation can be depicted as the construction of a search tree, where the internal nodes correspond to guessing the value of a variable (*labeling*) while the edges correspond to propagating the effect of the labeling to other variables (through consistency procedures).

We implement two *variable selection* strategies: a *leftmost* strategy—the collection of variables is viewed as a list, and the strategy selects the leftmost uninstantiated variable for the next labeling step—and a *first-fail* strategy—it selects the variable with `labeled`= 0 and the smallest domain size, i.e., the box with the smallest number of lattice points. The process of selecting the value for a variable V relies on D^V, on the structure of the underlying lattice, and on the constraints present.

For example, if the selected variable V is known to have distance 1 lattice unit from a known point in the lattice (a frequent occurrence in practical uses of lattice constraints), then the complete exploration of the box $Box(D^V)$ can be replaced by a direct exploration of the lattice neighbors of the given point (e.g., 12 points in the case of the $\mathcal{FCC}$ lattice and 6 points in the case of the CUBE lattice). This pruning can be pushed further following the same principle; if three variables X, Y, Z are known to be occupying three contiguous points in the lattice, and X, Y have already been placed in the lattice, then (in the $\mathcal{FCC}$ lattice) there are only 11 possible placements that can be explored for Z.

Moreover, we allow the possibility of collapsing levels of the search tree, by assigning a *set* of (related) variables in a single step. This operation is particularly useful when dealing with variables that belong to a rigid block constraint $\mathsf{block}(\vec{B}, \vec{V})$. When the first variable X of V is labeled, all the other variables in it are assigned, according to the precomputed templates. In particular, for each rotation template selected out of $\mathsf{Templ}(\vec{B})$, we open a branch of the search tree associated to the specific rotation.

At the implementation level, the current branch of the search tree is stored in an array; the i-th element of the array represents the i-th level of the current branch. Each level is associated to the corresponding variable chosen by the selection strategy. The variable is labeled with domain values and each choice creates a distinct branch in the search tree. A convenient enumeration of the domain elements is defined on every branch: each domain element is identified by a unique index. Each specific labeling choice and branch selection can thus be summarized by an extra counter. Storing this information on each level, allows an efficient handling of the array in case of backtracking and expansion of siblings (that are mapped on the same array element). The use of counters for each level allows an efficient detection of the completion of branching on a level and, at the same time, it avoids the maintenance of an explicit search tree (i.e., every sibling is recorded implicitly), with a significant benefits in terms of memory consumption.

As illustrated in Figure 6, backtracking is employed to explore different assignments of values to the variables (i.e., to move between branches of the search space). Since each step of propagation and of consistency leads to modifications of the data structures (e.g., modification of the domains), it is important to ensure that, during backtracking, the modifications to such data structures are properly undone, restoring the correct state of computation to restart with a different alternatives. To support this activity, we introduce a *value-trail stack*, used to keep track of the variables modified during propagation, and used to undo modifications during backtracking (in a fashion similar to the trail stack of a Warren Abstract Machine for

Prolog [2]).

3.5.3 Bounded Block Fails heuristic

In this Section, we recall the *Bounded Block Fails (BBF)* heuristics for searching solution that we presented in [24]. 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 $B_1B_2\ldots B_\ell$ 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 9 we depict a simple example for $k = 3$: we consider a list of 9 variables. The dark boxes represent ground assignments.

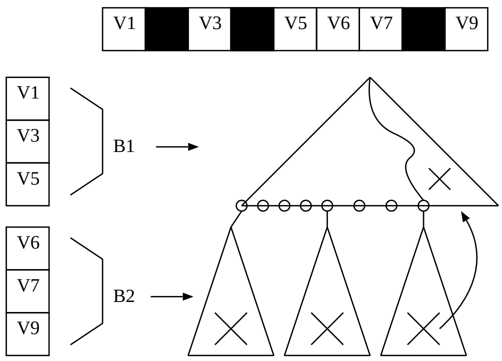


Figure 9: The BBF heuristics

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. 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

quality of the solution.

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.

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 Parallelizing COLA

Although the sequential implementation of COLA is fairly effective (see also Section 5.4.1), the performance of the implementation can be further improved, to enhance the applicability of the system to more time-consuming problems. In this Section, we explore the use of *parallelism* to enhance performance

4.1 Overall Organization

The parallel version of COLA is based on the exploitation of *search parallelism* (often referred to as *or-parallelism*) [51, 38] from the prop-labeling tree (cf. Section 2). Intuitively, the exploitation of parallelism is accomplished by allowing separate *agents* to concurrently explore different parts of the prop-labeling-tree in search of (optimal) solutions. Each agent can be implemented by a distinct process (or thread), possibly executed by a distinct processor, and each searching for a different solution to the problem.

We concentrate on the inherently non-deterministic stage of domain labeling (splitting rule) for a CSP (or COP). If two or more choices are generated in the prop-labeling-tree (see, e.g., the dark nodes in Figure 2), we try to solve each subtree rooted at the choice nodes independently and in parallel—e.g., see Figure 10. Note that the choice of subtrees is limited to roots on even levels: due to the definition of the prop-labeling-tree, odd levels contain nodes generated by propagation rules that do not involve branching.

In the context of resolution of a CSP, exploration of the different subtrees are independent, and as such they can be performed concurrently without the need for communication. In the context of the resolution of a COP, communication might be required, e.g., to propagate bound information during a branch&bound execution.

We propose a fully *decentralized* scheme for parallel scheduling of tasks and for load balancing. Each agent can alternate between active computation—i.e., exploration of parts of the prop-labeling-tree—and scheduling—i.e., trying to acquire new subtrees of the prop-labeling-tree for exploration. The general scheme is not dissimilar from traditional distributed approaches to the parallelization of discrete optimization [46, 33, 37].

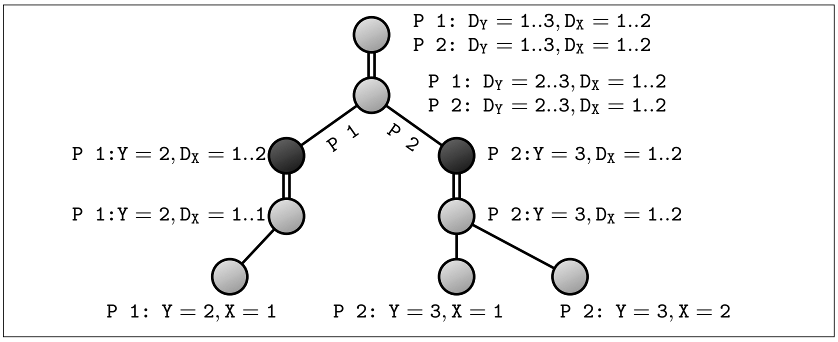


Figure 10: Intuitive view of search parallelism

4.2 Tasks

Given a CSP $\mathcal{P}$, the search of solutions is performed by traversing a prop-labeling tree. If variable selection rules and labeling rules are set, each node ν of this tree can be described in two, equivalent, ways:

- as a CSP $\mathcal{P}'$, or
- as a pair, consisting of the initial CSP $\mathcal{P}$ and the variable assignments performed in the path of the computation leading from the root of the tree ($\mathcal{P}$) to the the node ν.

A *task* is a subtree of a prop-labeling-tree for $\mathcal{P}$, whose root lies at an even distance from the root of the tree. The root node of a task is called the *task root*.

In a specific point of the computation, one node μ of the tree is being processed, and we call such node the *current node*. The path leading from the root of the prop-labeling-tree to the current node is called the *current path*.

Assuming a depth-first, left-to-right recursive search, the *processed nodes* are those nodes of the prop-labeling-tree which are located on the left of the current path, while the *subtasks* are the nodes on the right of the current path and at an even distance from the root. Some of the subtasks can be communicated and transferred to an idle agent, as a new task.

In the example of Figure 11, the task root (node 0) coincides with the tree root, and the current node is node 8. The dark nodes (2 and 3) represent the processed nodes; the shaded nodes $(0, 1, 4, 7$ and $8)$ represent the current path, while the white nodes $(5, 6$ and $9)$ are the subtasks.

4.3 Tasks Scheduling and Communication

Let us assume that n agents are available. The parallel exploration of the prop-labeling-tree relies on assigning distinct tasks to the agents. The key idea behind the use of a decentralized scheme is the notion of *dynamic rescheduling* of tasks among the agents. The rationale is that, every time an agent A terminates

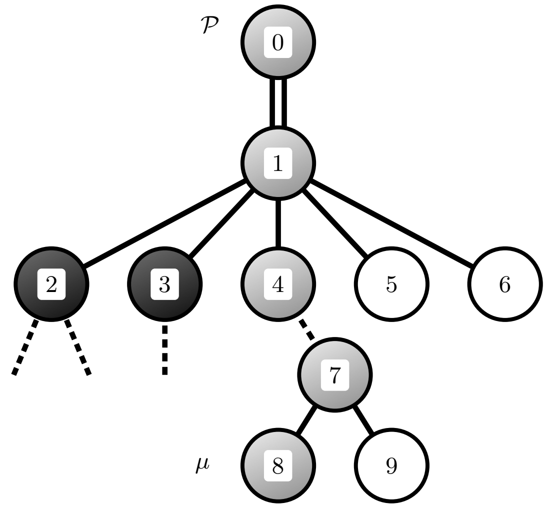


Figure 11: The search tree in the Parallel COLA system. Double lines mean propagation

its task, it queries the other agents to obtain a new task. If an agent B has an unexplored subtask, it can communicate such task back to A, allowing A to restart active computation on the newly received task. The need for dynamic rescheduling arises from the potentially unbalanced structure of the prop-labeling-tree, and the difficulty of estimating a-priori the size of a subtask.

A careful design has been adopted in order to reduce the communication traffic in the system during the search process. In this parallel version, we consider all-to-all communication and thus it is essential to devise a convenient protocol for efficient message handling.

We identify four types of messages that can be sent for handling task management:

- The message `REQ` is sent from agent i to agent j whenever agent i requests a new task from agent j.
- The message `TASK` is sent by agent j when a an unexplored subtask has been found and it can be given to the requesting agent i. The message actually contains a description of the new task, sent as the list of variable assignments encountered on the branch from the global root to the root of the subtask (e.g., corresponding to the nodes $0, 1, 4, 7, 9$ in Figure 11 if the subtask is rooted at 9).
- The message `WAIT` is sent by agent j to agent i if j itself is idle, it has requested a task and is awaiting for a reply. The `WAIT` message is used to indicate to agent i that no subtasks are currently available.
- Finally, the message `BUSY` is sent by agent j if the unexplored subtasks available in the agent j are considered too small for sharing.

Some preliminary tests (using a number of agents equal to the number of available processors), indicated that many concurrent communications between distinct pairs of agents lead to significant communication bottlenecks. Moreover, tasks fragmentation (i.e., production of small tasks) degrades the performances, due to the cost of frequent communication of tasks. To cope with these problems, we define a strict policy for handling messages, with a special care to load balancing. In the following list we summarize the main items considered in our policy:

- **Limited channels:** The agent $i \in \{0, \dots, n-1\}$ can request tasks only to a subset of agents $S_i \subseteq \{0, \dots, n-1\}$. In particular, we found that $|S_i| = n/2$ is sufficient to guarantee an acceptable communication delay and a good balancing of work;
- **Sequential task requests:** each agent i, after sending a task request to $j \in S$, must wait for an answer from j before sending a new task request. When j receives a task request, j has to answer either with a task or with a message with no task;
- **Requests addressing:** the set S_i is sorted and processed according to a simple round robin scheme. When a task has to be requested, an index that spans the set is updated and used to contact the corresponding agent;
- **Progress:** to avoid deadlocks, while waiting for a reply to a task request, an agent is expected to answer to every received task request;
- **Reduced traffic:** a minimum delay (`REQ_DELAY`) is imposed between two task requests sent to the same agent. This is necessary to avoid overloading an agent with task requests and to avoid delaying critical communication tasks. In the current experiments, the `REQ_DELAY` parameter is set to 0.01 seconds;
- **Enhanced load balancing:** subtasks are sent by an agent according to an ordering, constructed using the position of the subtask roots in the tree. In particular, nodes that are closer to the root of the task are considered first for sharing. Since the structure of the task changes dynamically (as new nodes are dynamically created and added to the task), a special procedure is in charge of retrieving the "highest" subtask available (a white node in Figure 11). The intuition is that nodes closer to the root of the tree have a greater likelihood of being the root of a large subtask. Observe that the actual size of a task (i.e., the number of nodes in it) cannot be precisely predicted, thus we use the depth of the root of the task as an estimate of the granularity of the task. This choice has been shown to reduce the amount of fragmentation, since the new subtasks are iteratively fetched from the received ones.
- **Optimized interaction:** if agent j receives a `REQ` message from agent i, and j has no subtasks available, then agent j will keep exploring its own task (expanding it), until there is a new subtask

that can be returned to i. This choice is better than returning a failure message to i, since the ratio of subtask generation is very high (every node expansion, usually generates more than one new subtask). In particular, this approach is cheaper than forcing agent i to start a new communication session for a new task request with some other agent;

- **Light message checks:** due to the above mentioned subtask generation ratio, it is convenient to insert a a check for pending `REQ` messages every x nodes expansions performed by the agent. Excessively frequent checks for messages (e.g., $x = 1$) would degrade the performance of the normal task computation, while a too infrequent test (e.g., $x = 256$) would cause a prolonged wait for the agents that sent the requests. In the current implementation, this parameter (called `ANSWER_EVERY_NODES`) has been fixed to $x = 16$.

Figure 12 and Figure 13 provide the pseudocode that describes the behavior of each agent. In particular, in Figure 12 we show the code that defines the outer loop (search_handler) of the agent, in charge of obtaining a task and submitting it to the search procedure—described in Figure 13.

Let us discuss the code in Figure 12. The procedure `prepare_initial_task` (Line 5) produces a first task for each processor, whose structure is dependent on the shape of the search tree—this construction is discussed in more detail in Section 4.4. Lines 11–21 are part of the loop executed by the agent while waiting for a new task to arrive. At each iteration, termination of the loop is detected in Line 11. We introduce a boolean variable, `ask_task`: if its value is *true*, then the agent can issue a task request (`REQ`) to agent i (Lines 12–14). After a task request, `ask_task` is set to *false*, and can be reset to *true* only if a reply from agent i is received. Before ending the loop (Lines 20–21), the agent needs to update the address of the next agent to contact for a task request, according to the round-robin scheme. This update is performed whenever the previous request has already been answered or it could not be issued due to time reasons (Line 12).

Finally, Line 23 is reached only if a new task to be explored has been received or a termination signal has arrived. In the case of a new task (Line 24), the procedure search is called, the task is processed, and a new iteration of loop 2–24 is executed.

In Figure 13, we describe the overall structure of the search routine—a modification of the sequential search routine developed for COLA. The main novelty of this routine is in Lines 5–9, where we introduce the check for task requests. If the request arrives when there are some subtasks available, the highest one is sent back (Line 8). If no subtasks are available, then the request is kept pending until new adequate subtasks are generated (and Line 8 will be activated). In this way, the answer to agent i is delayed to avoid further communications. Note that, in Line 6, the check for communication is executed every `ANSWER_EVERY_NODES` task expansion steps, where the counter is global to the current agent. Lines 11–13 represent the normal

```
search_handler(D)
1  initial_task = true
2  while !terminated
3  do
4     if initial_task
5       then task = prepare_initial_task()
6            initial_task = false
7       else
8            task = NIL
9            ask_task = true
10           repeat
11                  terminated = handle_termination()
12                  if ask_task ∧ last request to i older than REQ_DELAY
13                    then send REQ to i
14                         ask_task = false
15                  if i returned a message ∧ message = TASK
16                    then task = get_task(i)
17                  if received REQ from j
18                    then send WAIT to j
19                  if i returned a message ∨ ask_task
20                    then ask_task = true
21                         i = next process ∈ S
22             until task ≠ NIL ∨ terminated
23    if !terminated
24      then search(task)
25 endwhile
```

Figure 12: Pseudocode of Parallel Process Manager

exploration of the search tree.

4.4 The initial tasks

Specialized actions are performed, at the beginning of the computation, to assign an initial task to the different agents. A simple implementation, which follows the more traditional model of assigning the root task to an initial agent and letting the other agents obtain subtasks from there [38], revealed to be highly inefficient, due to the immediate saturation of the communication channels generated by task requests of $n-1$ idle agents.

A more effective choice for this phase (see also Line 5 in Figure 12) is to devise a method to assign a task to each process to begin with. Given n agents, the idea is to produce an initial partition of the whole search tree into n subtrees, that are balanced as much as possible; we also aim at achieving this goal without requiring any communications between agents.

Since a task is represented by the branch leading from the root of the search tree to the root of the task, the goal is to create a parallel, coordinated and communication-free exploration of the tree that leads each agent along a different branch. At the end of this parallel step, each agent is mapped to a distinct node,

```
search(Lev)
if leaf
  then return
expand a level
for each node to process on current level
do
   if explored ANSWER_EVERY_NODES nodes  ∧ subtask available
     then
          if received REQ from i
            then reply i with a new task
                 update available subtasks
   pick an expanded node (if not given as subtask)
   propagation
   search next level
```

Figure 13: Pseudocode of Parallel Search

guaranteeing at the same time that the assigned nodes are as high as possible in the search tree (to enhance the likelihood of a large grain task).

This initial exploration of the search tree is concurrently performed by the different agents, following a modified depth-first strategy. Whenever one node u is expanded with its children, the group of agents working on u is partitioned between the children to continue the exploration. The distribution scheme is realized in a simple manner, by defining a mapping from agents to nodes of the search tree. A node u can be explored by $[i..j]$ agents, meaning that the agents having id in the range from i to j will expand the node u. The expansion of u generates nodes $u_1, u_2, \ldots, u_k$. Each agent that is assigned to u selects one of these k nodes, according to a common partitioning strategy, and continue the exploration of the search tree. The partition is made such that the same number of agents is assigned to each node. If there are fewer agents than nodes, then each agent is assigned to a node, and the nodes without any association are gathered and assigned to an arbitrary agent as additional subtasks. If there are more agents than nodes, the interval $[i..j]$ is partitioned uniformly in k sets, one for each expanded node, and each agent follows the depth-first descent on the node that has an interval containing its agent id. Figure 14 depicts this phase. Nodes are represented by circles, and the agent associated to them are represented by boxes. In this representation, for the sake of simplicity, the nodes produced by constraint propagation are collapsed into the parent, i.e., we omit the double lines as in Figure 2.

As soon as agent i moves to a node which is assigned the agent ids interval $[i..i]$, the agent is ready to begin the task exploration, since it is guaranteed to be the only one to handle that task. If, for propagation reasons, a node cannot generate any children, the agents assigned to that node will start in idle state and begin to request tasks in the usual manner. This event depends on the structure of the CSP problem, but, from our experience, is relatively infrequent.

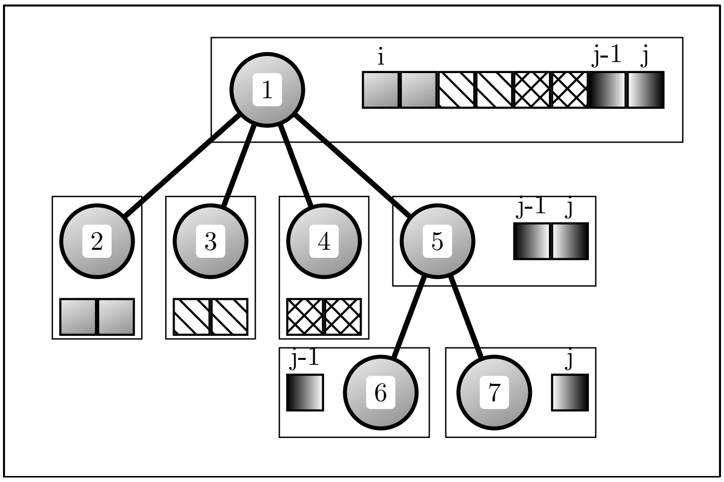


Figure 14: Initial Task Parallel Assignment

4.5 Some implementation details

The parallel system described has been developed on a Beowulf cluster—using C++ and mpicxx. Each agent is implemented as an MPI process, and communication is explicitly realized using message passing. Since we make use of MPI-1, we rely on a static process structure—i.e., agents are created at the moment of launching the program, and no agent can be added or removed from the system during the execution.

Agents are assigned linear ids, directly obtained from the corresponding MPI process ids. Access to the CSP/COP problem description is concurrently performed by all the agents, to avoid additional communication.

Since we rely on a distributed scheduling structure, termination detection becomes an issue. We implement the standard token ring termination detection by Dijkstra [30]. In particular, we pass a black/white token every time the agent is in the search_handler loop (Line 10 of Figure 12) and the agent has received a token from the preceding agent (viewing the agents as part of a ring, where agent $n-1$ out of n is followed by agent 0). Note that, while processing a task, the token passing activity is suspended, thus making the token traffic very light.

5 An Example: the Protein Structure Prediction Problem

We report some results deriving from the implementation and testing of the framework described above on a challenging application coming from the Bioinformatics area: the Protein Structure Prediction Problem. We briefly recall here its mathematical formulation (see also [19, 23]).

5.1 Protein Structure Prediction Problem

The *Primary* structure of a protein is a sequence of linked units called *amino acids*. The amino acids can be identified by an alphabet $\mathcal{A}$ of 20 different symbols, associated to specific chemical-physical properties. The protein tends to reach a 3D conformation with the minimal value of free energy (*native* conformation), also called its *tertiary structure*. Native conformations are largely built from *secondary structure* elements, namely some local rigid structures (e.g., α-helices and β-sheets) that involve some short sequences of amino acids arranged in a predetermined fashion. Some of these local structures can be predicted accurately using neural networks or homology; we will use this information in our predictions. Moreover, it is possible to predict that two atoms will be close in the native state, e.g., thanks to disulfide bonds (*SS-bond*).

We can encode the problem as a COP on the $\mathcal{FCC}$ domain, which has been considered suitable to model proteins on discrete spaces by various researchers [40, 39, 5, 23]. Given a primary sequence $S = s_1 \cdots s_n$, with $s_i \in \mathcal{A}$, let $\omega(i)$ be the *position* of the amino acid s_i in the $\mathcal{FCC}$. It is assumed that two consecutive amino acids are always separated by a fixed distance (typically, 3.8Å). Given two lattice points ω_1, ω_2, we indicate with $\texttt{next}(\omega_1, \omega_2)$ the fact that they are contiguous in the lattice. In the $\mathcal{FCC}$ lattice,

$$\texttt{next}(\omega_1, \omega_2) \Leftrightarrow \delta(\omega_1, \omega_2) = 2$$

It is assumed that the energy of the protein is the sum of the energies generated from all the pairs of amino acids. These local energies depend on their distances and their types. We assume that the energy contribution is 0 if they are at a distance grater than a certain threshold. In particular, we employ the binary (boolean) function `contact`, that states that two amino acids s_i and s_j are sufficiently close to be able to interact, and thus they contribute to the energy function. In the $\mathcal{FCC}$ we state that

$$\texttt{contact}(A, B) = 1 \Leftrightarrow \delta(A, B) = 4$$

Given an $\mathcal{FCC}$ lattice (P, E) and a primary sequence $S = s_1 \cdots s_n$, with $s_i \in \mathcal{A}$, a *folding* of S in (P, E) is a function $\omega : \{1, \ldots, n\} \to P$ such that:

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

Every time a contact between a pair of amino acids is detected, a specific energy contribution, dependent on the specific pair of amino acids (and drawn from a 20×20 table) is applied [15]. 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 (PSP)* can be modeled as the problem of finding the folding ω

of S such that the following energy cost function is minimized:

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

In the $\mathcal{FCC}$, each point is adjacent to 12 neighboring points, and the angle between three adjacent residues may assume values 60°, 90°, 120°, and 180°. Volumetric constraints and energetic restraints in proteins make the values 60° and 180° infeasible. Therefore, in our model, we retain only the 90° and 120° angles [58, 35].

As already mentioned, a *contact* between two non-adjacent residues in $\mathcal{FCC}$ occurs when their separation is two lattice units. 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 unit. This is achieved by adding, for the pair i and j, the constraint: $\delta(s_i, s_j) \ge 4$. Moreover, as explained in [23], it is reasonable to bound the maximum distance between two amino acids using a *compact factor*.

5.2 Modeling PSP on COLA

We summarize the formalization in use with COLA constraints. Given $S = s_1 \cdots s_n$, with $s_i \in \mathcal{A}$, we represent with lattice variable V_i the *lattice position* of amino acid s_i. The modeling leads to the following constraints:

- For each $i \in \{1, \dots n-1\}$, we have that $\delta(V_i, V_{i+1}) = 2$: adjacent amino acids in the primary sequence are mapped to lattice points connected by one $\mathcal{FCC}$ lattice unit (*next property*);
- For each $i \in \{2, \dots n-1\}$, we have that $\delta(V_{i-1}, V_{i+1}) \le 7$: three adjacent amino acids may not form an angle of 180° in the lattice (*bend property (i)*);
- For each $i, j \in \{1, \dots n\}$ and $|i - j| \ge 2$, we have that $\delta(V_i, V_j) \ge 4$: two non-consecutive amino acids must be separated by more than one lattice unit (non overlapping property), and angles of 60° are disallowed for three consecutive amino acids (*bend property (ii)*).
- For each known ssbond present between amino acids s_i and s_j, we have that $\triangle(V_i, V_j) \le 4$ (*ssbond property*).
- For each $i, j \in \{1, \dots n\}$ we have that $\triangle(V_i, V_j) \le \mathsf{cf} \cdot n$, where cf is the *compact factor*, expressed as a coefficient in $[0..1] \subset \mathbb{R}$ (see [23]).

An evaluation of the performance of using COLA on this modeling of the PSP problem is presented in Section 5.4.1.

5.3 An ad-hoc Branch and Bound strategy

We present a branch and bound (BB) strategy, adapted to the specific needs of the PSP problem. In the case of the protein fold problem, a generic branch and bound scheme, based on the estimation of the energy of the conformation, proved to be rather ineffective with large input sizes. Our intuition is that the cost function can collect many contributions at the very end of a branch and drastically change its value. This behavior is particularly evident when processing large proteins. As a result, the prediction of the bounds for the energy function, computationally expensive, reveals to be potentially inaccurate.

We adopted a more coarse and constant time cost estimation. The strategy we propose implements branch and bound using the *number of contacts* generated by the given conformation as the information to perform pruning. In general, the global energy and the number of contacts are strongly related. Nevertheless, since the energy function is composed of weighted contributions of amino acids in contact, the two values may occasionally diverge. The computation of estimates of the number of contacts is facilitated by the peculiar properties of the $\mathcal{FCC}$ lattice; e.g., each amino acid can form at most 3 contacts with other ones. When a new best conformation is found, we compute the number c of contacts realized. Assuming that, in the worst case, the last amino acids to be labeled generate 3 contacts each, at $c/3$ levels before the leaves, each subtree can be safely pruned whenever the number of contacts is less than c. This heuristic can be computed in constant time since, given a partial assignment, an upper bound for possible contacts is immediately known. Since the energy is not precisely expressed by the number of contacts, we can not guarantee the completeness of the heuristics. Nevertheless, empirical tests showed that this is not a significant problem; our experiments indicated also that the pruning of the last levels of the tree provides significant speedup during search.

In Table 1, we show some experimental tests of enumeration of the complete search tree, with and without the pruning heuristic presented above (under Windows on an Intel Centrino 2.0GHz, 1GB RAM). Each known protein has an official ID assigned in the *Protein Data Bank* [14] that we have used in the first column in the table. The second column (N) denotes the number of amino acids in the corresponding protein. We run a complete enumeration and then perform the same search with the contacts heuristic activated. In all cases, the heuristic improves time and reduces the number of nodes explored, without significantly changing the optimal solutions discovered.

5.4 Experimental Results and Comparisons

In this Section,we provide the results obtained from running a collection of tests using the sequential and parallel implementations of COLA. In particular, we present experiments that are related to the use of COLA to solve instances of the PSP problem. Our focus is not only on proving the practical effectiveness

Table 1: Effectiveness of contact pruning heuristic

		Enumeration			BB Heuristic		
ID	N	Energy	Nodes	Time	Energy	Nodes	Time
1kvg	12	-6,881	318,690	0.250s	-6,881	124,722	0.187s
1le0	12	-4,351	1,541,107	1.125s	-4,351	487,105	0.703s
1le3	16	-5,299	1,544,830	1.515s	-5,299	439,969	0.969s
1pg1	18	-10,352	56,934	0.047s	-10,352	7,908	0.016s
1zdd	34	-12,315	234,314	1.609s	-12,097	34,748	1.390s

of COLA as a constraint solver on ($\mathcal{FCC}$) lattice structures, but also on demonstrating the effectiveness of using COLA to address the PSP problem on $\mathcal{FCC}$, compared to other solutions to this problem presented in the literature.

In Section 5.4.1, we analyze the sequential performance of the COLA implementation, including comparisons with the encoding of the same problems using traditional finite domain constraints (specifically, using SICStus Prolog [55] and ECLiPSe Prolog [60]). We also analyze the impact of the use of the BBF heuristic on the quality of the results, and discuss the scalability to large instances of the problem. Section 5.4.2 analyzes the results in terms of exploitation of parallelism. In Section 5.4.3, we compare the performance of COLA in solving PSP problems in the $\mathcal{FCC}$ lattice with the performance of various *Integer Programming (IP)* solvers—as IP has been identified as a valid methodology for this class of problems [39].

5.4.1 Sequential Performance

For experimental results regarding the sequential version of the solver, we run the tests on a Intel Centrino 2GHz with 1GB RAM. The operating system is Windows XP and we compile the code using MinGW32 v. 3.1.1 with the expensive optimization flags enabled. We noted that for different versions the efficiency of the code produced is different. The COLA source, other Prolog programs and some results cited in this Section are available at `www.dimi.uniud.it/dovier/PF`.

Efficiency: The first test we discuss is designed to benchmark the speed of our solver. Our goal is to compare the solution to the protein folding problem using our lattice solver with the solution obtained by mapping the problem to finite domain constraints—using SICStus 3.12.2 (`clpfd`) and ECLiPSe 5.8 (`ic`). We run complete enumerations of the search tree using the first-fail strategy and we retain the best solution computed, i.e. cost function handling is included in these tests. To perform a fair comparison, we did not make use of branch and bound strategies in any of the implementations.

We implement the PSP problem in SICStus and ECLiPSe using the formalization described in [24]. This formalization is equivalent to COLA formalization of Section 5.1. In Table 2, we compare the running times

Table 2: Complete Search

ID	COLA	SICStus		ECLiPSe	
1edp	0.031s	5.28s	(170x)	34.1s	(1,100x)
1pg1	0.079s	10.37s	(131x)	58.1s	(735x)
1kvg	0.281s	26.69s	(95x)	138.9s	(494x)
1le0	1.469s	271.2s	(184x)	1044s	(711x)
1le3	2.219s	392.3s	(177x)	1898s	(855x)
1zdd	2.062s	8520s*	(4131x)	> 6h.	(>10,000x)

,

required to explore the whole search space. In the first column, we report the protein selected, in the second the time (in seconds) required by the COLA solver to explore the search tree, while the last two columns report the corresponding running times using SICStus and ECLiPSe (in brackets the speedups of COLA w.r.t. the finite domain CLP solvers). For these examples, we use proteins whose search tree can be exhaustively explored in a reasonable time. The (*) reports that an instantiation error occurred. Table 2 shows that the choices made in the design and implementation of the new solver allow us to gain speedups in the order of 10^2–10^3 times w.r.t. standard general-purpose FD constraint solvers. Moreover, our implementation is robust and scales to large search trees with a limited use of memory. E.g. in SICStus, the set of admissible elements that form the domain is maintained as a disjunction of intervals, which experimentally causes an average of 10 times more memory consumption than COLA.

Table 3: BBF experimental results (Windows, Intel Centrino 2GHz, 1GB RAM).

ID	n	**CF**	**BBF**	**Time**	**Energy**	**PDB on $\mathcal{FCC}$**	**PDB**
1kvg	12	0.94	50	0.06s	-18,375	-17,964	-28,593
1edp	17	0.76	50	0.03s	-46,912	-38,889	-48,665
1e0n	27	0.56	50	1.75s	-52,558	-51,656	-60,728
1zdd	34	0.49	50	0.094s	-63,079	-62,955	-69,571
1vii	36	0.48	50	4.93s	-76,746	-71,037	-82,268
1e0m	37	0.47	30	16m12s	-72,434	-66,511	-81,810
2gp8	40	0.45	50	0.25s	-55,561	-55,941	-67,298
1ed0	46	0.41	50	7.62s	-124,740	-118,570	-157,616
1enh	54	0.37	50	49.5s	-122,879	-83,642	-140,126
2igd	60	0.35	20	3h51m	-167,126	-149,521	-201,159
1sn1	63	0.18	10	22m2s	-200,404	-242,589	-367,285
1ail	69	0.32	50	2m53s	-220,090	-143,798	-269,032
1l6t	78	0.30	50	1.12s	-360,351	-285,360	-446,647
1hs7	97	0.20	50	36m19s	-240,148	-246,275	-367,687
1tqg	104	0.15	20	10m24s	-462,918	-362,355	-1,242,015

Heuristics tests: To show the power of our constraint solver in handling ad-hoc search heuristics, we test a set of selected proteins, with lengths ranging from 12 to 104. The heuristics in use are the Bounded Block

Fails, presented in Section 3.5.3, and the branch&bound strategy, described in Section 5.3.

Table 3 reports the results of the executions. The Table indicates the PDB protein identifier (**ID**), the protein length (n) in terms of amino acids, the compact factor parameter (CF, see Section 5.2), the BBF threshold values assigned to $t_1 = \cdots = t_\ell$ (**BBF**, i.e., the number of allowed failures for each block), the time to complete the search (**Time**), the measure of the quality of the best solution computed (**Energy**), the estimation of the quality of a solution that can be obtained by discretizing the protein on the $\mathcal{FCC}$ lattice (**PDB on** $\mathcal{FCC}$), and the value of energy for the protein conformation in 3D space without the spatial constraints imposed by the use of a lattice (**PDB**).

For BBF, we decided to define the block size equal to 3 for $n \leq 30$ and equal to 5 for larger proteins. The number of allowed failures within each block is selected as expressed in the Table in BBF column. We empirically noticed that larger block sizes provide less accurate results, due to the greater pruning, when failing on bigger blocks.

Proteins with more than 100 amino acids can be handled by our solver. This result is significantly better than what reported in the literature for the solution of this problem (using CLP(FD) or Integer Programming)—e.g., using CLP(FD) it is possible to handle proteins of up to 80 amino acids. This improvement is non-trivial, because of the NP-completeness of the problem at hand. The new heuristics provide more effective pruning of the search tree, and they allow to collect better quality solutions. The tradeoff between quality and speed is controlled by the BBF threshold: higher values provide a more refined search and higher quality solutions.

For large proteins, it is an open problem in the literature how to precisely estimate the errors arising from discretizing the protein structure in a lattice space. In our estimations, we enumerated some of the possible arrangements of a PDB protein on the $\mathcal{FCC}$ lattice, (10,000 structures) and we retained the best arrangement found. The application of our energy function on that candidate provides an idea of the possible energy that can be reached by the $\mathcal{FCC}$ optimization. The quality comparison between our folding and the mapping of PDB on $\mathcal{FCC}$ and the PDB itself, reveals that our solution, even for larger proteins, are comparable to foldings of PDB on $\mathcal{FCC}$. Note also that, for larger proteins, the size of the pool of the selected solutions for PDB on $\mathcal{FCC}$ mappings, becomes insufficient. This causes an underestimation of the possible best energy that can be obtained. For our purposes, this confirms that the BBF search heuristic tends to retain meaningful conformations associated to suboptimal energy values.

Scalability on bigger instances: We would like to study the scalability of our solver. To accomplish this, we use the protein examples to test larger input size instances. We report some tests on artificial proteins having a structure of the type XYZ, i.e., composed of two known subsequences (X and Z), while Y is a short connecting sequence. This reflects a common practice when dealing with large proteins—where the structure of various subsequences is known (e.g., determined by homology) and the problem is to place this large rigid blocks (e.g., linked together by short coils). We can show that our framework can easily handle proteins of size up to $1,000$ amino acids. We run some *complete* enumerations varying the length of Y and the proteins used as pattern for X and Z.

Table 4: Processing proteins XYZ

X	Z	$\lvert X\rvert$	$\lvert Y\rvert$	$\lvert Z\rvert$	Time
1e0n	1e0n	27	4	27	3.53s
1e0n	1e0n	27	5	27	19.4s
1e0n	1e0n	27	6	27	106s
1ail	1ail	69	4	69	10.1s
1ail	1ail	69	5	69	54.5s
1ail	1ail	69	6	69	290s
1hs7	1hs7	97	4	97	17.5s
1hs7	1hs7	97	5	97	94.6s
1hs7	1hs7	97	6	97	507s
1e0n	1e0n	27	3	27	0.64s
1e0n-2	1e0n-2	57	3	57	2.48s
1e0n-4	1e0n-4	117	3	117	2.79s
1e0n-8	1e0n-8	237	3	237	1.17s
1e0n-16	1e0n-16	477	3	477	4.87s

Table 5: Ratios sphere/box approach

ID	Nodes	Time
1pg1	1.00	1.34
1kvg	1.95	2.39
1le0	1.00	1.06
1le3	1.02	1.16
1edp	2.96	2.00
1zdd	1.30	2.18

In our tests, we load the proteins X and Z as predicted in Table 3. We link them with a coil of amino acids with length $|Y|$ (leaving X and Z free of moving in the lattice as rigid objects). The search is a simple enumeration using Leftmost variable selection. Table 4 shows that the execution time is low, and dominated by the size of Y, instead of the size of XYZ. Note that, for each increase of the length of Y from 4 to 6 amino acids, the computation time roughly increases by 6 times, i.e., the number of admissible lattice points for a variable, when the previous ones have been assigned.

In the second part of the Table, we consider proteins constructed as follows: we start with X and Z equal to the 1E0N protein (whose folding can be optimally computed), and every successive test makes use of $X' = Z' = XYZ$—i.e., at each experiment we make use of the results from the previous experiment. This approach allows us to push the search to sequences of size up to $1,000$ amino acids. In these experiments, our concern is not only the execution time, but the ability of the solver to make use of known structures to prune the search tree. As the results show, COLA is able to efficiently handle short connecting sequences (the Y part of the protein).

Spherical representation of domains: We tested a different formalization of the variable domains, where domains are represented as spheres instead of using *Box*. We reimplemented, in our solver, the domain description of a variable in terms of a center and a radius (with discrete coordinates) and the definition of an intersection of spheres as the smallest sphere that includes them. The idea is that a sphere should be more suitable to express the propagation of Euclidean distance constraints.

Unfortunately, the results reported in Table 5 show that this idea is not successful. The Table reports in the first column the test protein used, in the second the ratio of visited nodes in the search tree between sphere over box implementations. The last column provides the ratio of computation time between the two implementations. In particular, observe that many more internal nodes are expanded in the sphere implementation. There are two reasons for this:

- Computing spheres intersection is more expensive than intersecting boxes;
- Often, two intersecting spheres are almost tangent. In this case the correct intersection is approximated by another sphere that includes a great amount of discarded volume.

5.4.2 Parallel results

We implemented the ideas described in Section 4. In particular, we constructed a distributed implementation, built using the MPI library. This version is, thus, highly portable. We tested it on a Linux-based Beowulf cluster (Intel Xeon 1.7GHz processors, Myrinet-2000 interconnection network).

In the implementation, we make use of non blocking receive calls (`MPI_Test` and `MPI_Wait`), in order to allow the computation to proceed while waiting for messages from other agents. For efficiency reasons, we also make use of user-defined buffers for supporting communication (using `MPI_Attach` and `MPI_BSend` routines).

We run some experiments to test the quality and scalability of the system. In particular, we concentrate on exhaustive searches. We run a complete enumeration of some proteins defined with our formalization. We choose two real proteins (1LE0 and 1ZDD) with a relatively small set of admissible conformations (in the order of millions). These tasks are small and it is interesting to analyze the performance of the parallel system when handling high fragmentation and load balancing in a light work condition. Moreover, we tested the solver with a bigger protein (1E0N). Since the whole protein would require days to be completely computed, it has been reduced by the last 2 and 3 amino acids (that are not essential in the determination of the shape). We name these proteins 1E0N2 and 1E0N3, respectively, since it is actually a subsequence of the original protein. Additionally, we test COLA with a sequence made of 13 consecutive Alanines (a type of amino acid), without any secondary structure imposed (the solution space is roughly $O(6^n)$). These last three examples are useful to show the behavior of the system in heavy load conditions, which is the typical situation for protein application.

In Figure 15, we show the speedups achieved using different numbers of processors. In the plot, the solid line represents the theoretical linear speedup; as we can observe, the speedups are high and very close to the linear one. These results are possible thanks to the scheduling policy adopted. It is important to stress that, in this framework, there is no prior knowledge of the size of each task. In this sense, the need of a decentralized scheduling with load balancing is essential, since the interaction of constraint propagation and search is unpredictable. As expected, smaller tasks (1LE0 and 1ZDD) scale slightly worse, due to the frequent balancing of small tasks, while for the bigger protein, this effect is more contained.

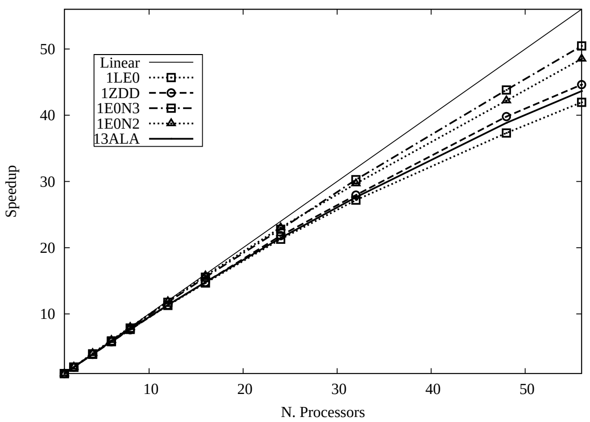


Figure 15: Parallel Speedup

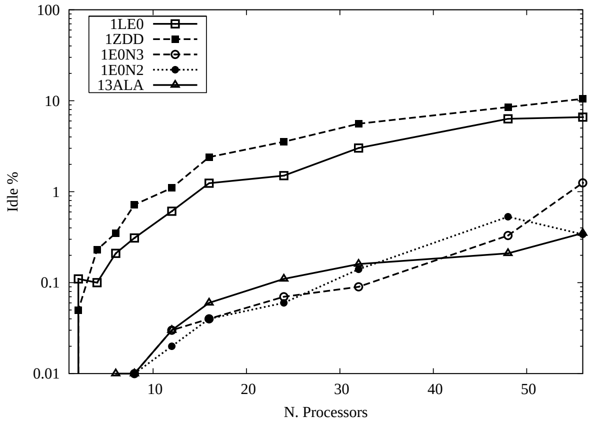


Figure 16: Parallel Idle Time

In Figure 16, we plot—in a logarithmic scale—the fraction of time spent to receive new tasks (idle time). Here, it is even more evident how the communication becomes a bottleneck for small tasks, since most of the time is spent in waiting for a new task to process. For bigger problems, the degradation is relevantly smaller.

In conclusion, using up to 56 processors provides an excellent speedup in computation. For completeness, in Table 6, we report the detailed results from these experiments. In the first column we report the number N of processors employed (i.e., the '-np' parameter used with the `mpirun` command) and for each protein we report the parallel time in seconds (**Time**), the percentage of idle time (**% Idle**) and the Parallel Speedup achieved (**P.S.**).

5.4.3 Comparing COLA to IP

The goal of the investigation presented in this section is to compare COLA with state-of-the-art Integer Programming (IP) and Constraint Logic Programming solvers.

Simplified versions of the PSP problem have been presented and studied in the literature [39, 7]. The simplified version relies on the use of the *HP* model, where amino acids are classified in two classes (H,

Table 6: Parallel Experimental Results

N proc	1LE0 Time	1LE0 % Idle	1LE0 P.S.	1ZDD Time	1ZDD % Idle	1ZDD P.S.	13ALA Time	13ALA % Idle	13ALA P.S.
1	13.511	0.00	1.00	18.028	0.00	1.00	1001.781	0.00	1.00
2	6.836	0.11	1.98	9.195	0.05	1.96	504.258	0.00	1.99
4	3.461	0.10	3.90	4.630	0.23	3.89	255.981	0.00	3.91
6	2.320	0.21	5.82	3.115	0.35	5.79	170.784	0.01	5.87
8	1.764	0.31	7.66	2.368	0.72	7.61	129.977	0.01	7.71
12	1.197	0.61	11.29	1.591	1.10	11.33	88.238	0.03	11.35
16	0.921	1.24	14.67	1.220	2.40	14.78	67.671	0.06	14.80
24	0.635	1.50	21.28	0.826	3.54	21.83	46.643	0.11	21.48
32	0.497	3.02	27.19	0.645	5.59	27.95	36.266	0.16	27.62
48	0.362	6.33	37.32	0.453	8.51	39.80	25.789	0.21	38.85
56	0.322	6.61	41.96	0.404	10.49	44.62	22.949	0.35	43.65

N proc	1E0N3 Time	1E0N3 % Idle	1E0N3 P.S.	1E0N2 Time	1E0N2 % Idle	1E0N2 P.S.
1	1831.609	0.00	1.00	7980.564	0.000	1.00
2	932.983	0.00	1.96	3980.610	0.013	2.00
4	460.594	0.00	3.98	1965.987	0.047	4.06
6	308.834	0.00	5.93	1323.383	0.064	6.03
8	232.621	0.01	7.87	997.069	0.107	8.00
12	155.756	0.03	11.76	671.106	0.164	11.89
16	117.846	0.04	15.54	505.641	0.190	15.78
24	80.381	0.07	22.79	346.162	0.193	23.05
32	60.513	0.09	30.27	268.844	0.379	29.68
48	41.794	0.33	43.82	189.069	0.999	42.21
56	36.292	1.25	50.47	164.519	0.565	48.51

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 [21, 50], 2D triangular lattices [1], 3D square models [39], and face-centered cubic lattices ($\mathcal{FCC}$) [53, 7]. For the sake of simplicity, we consider in this section a lattice composed of a subset of $\mathbb{N}^2$. These simplifications exclude from the tests some technicalities generated by using a refined energy model and the $\mathcal{FCC}$ lattice. However, this model is sufficient to offer the sources of complexity that are typical of the more general versions of the PSP problem.

In these tests, we are interested in the performance of the process of finding an optimal conformation, given the constraints (encoded as linear disequalities in IP) and the associated cost function. In particular, we focus on measuring the execution time, being the only factor that can be related between the different frameworks—other parameters, such as the number of branch points and the number of nodes explored, are not as easily comparable across frameworks.

The guideline for comparing the frameworks is to use *equivalent* constraints, and to provide the most natural implementation. We deliberately avoid optimizations and heuristics in all the frameworks, in order to test the power of the native solvers. Clearly, specific heuristics and specialized encodings can improve performance within each framework, but they would not be portable across frameworks.

An Integer Programming model: In Integer Programming, it is convenient to deal with many variables, each with a small domain, instead of having fewer variables with large domain. Thus, we converted the domains used to represent the coordinates, used in the previously described model, into a set of *boolean variables*. This model introduces a set of boolean variables X_ℓ (*layer*) for each amino acid ℓ. Each layer X_ℓ contains boolean variables $X_\ell[i, j]$, where (i, j) represent a specific coordinate of a position of amino acid ℓ. If $X_\ell[i, j] = 1$, then the amino acid ℓ will occupy the position (i, j) in the map. Linear constraints are introduced to model the same relationships described in Section 5. In particular,

- For each ℓ we have $\sum_{i,j} X_\ell[i, j] = 1$, stating that each amino acid has a unique placement (in its layer).
- For each i, j we have that $\sum_\ell X_\ell[i, j] \leq 1$, stating that, for every position i, j, at most one amino acid is present in such location—i.e., there are no loops.
- For each $0 \leq \ell \leq n-1$ we have that $X_\ell[i, j] \leq X_{\ell+1}[i+1, j]+X_{\ell+1}[i-1, j]+X_{\ell+1}[i, j+1]+X_{\ell+1}[i, j-1]$, stating that, if the amino acid ℓ is assigned to the position (i, j), then the amino acid $\ell+1$ has to occupy a position in space which has distance 1 from (i, j). Note that similar constraints are introduced to constrain amino acid $\ell - 1$ as well.

Finally, to compute the energy contributions, we make use of the following AND implementation: if A, B, C are boolean variables, we can express the relation $C = A \wedge B$ using the linear constraints: $C \leq A$, $C \leq B$ and $A + B - 1 \leq C$.

We consider an energy contribution every time there are two amino acids at distance 1. We separate two cases, where the amino acids are at distance 1 along the horizontal axis (h) and along the vertical axis (v). For the first case, the constraints are expressed by

$$A_{i,j,h} = \sum_\ell X_\ell[i, j], B_{i,j,h} = \sum_\ell X_\ell[i + 1, j]$$

for every i, j. The other case is similarly handled ($A_{i,j,v}$ and $B_{i,j,v}$ are defined). By implementing $C_{i,j,h} = A_{i,j,h} \wedge B_{i,j,h}$ as described above, we are able to to define $C_{i,j,h}$ as an energy contribution; the coexistence of two amino acids in contact ($A_{i,j,h}$ and $B_{i,j,h}$ equal to 1) forces the corresponding $C_{i,j,h}$ boolean value to assume the value 1. The cost function can be computed as the sum of the $C_{i,j,h}$ and $C_{i,j,v}$ values, for every pair i, j, with $i + 1 < j$ (to exclude consecutive amino acids).

A Model for COLA: Defining these same constraints for COLA is simple, and it is equivalent to the ones used for the CLP(FD) model of this problem (as described, for example, in [22]). In COLA, variables represent three dimensional points as native object. The encoding, using only 2D coordinates for each point, is even simpler than a CLP(FD) program:

```
/// next constraint for consecutive amino acids
for (int i=0;i<n-1;i++)
    cstore_add(EUCL_EQ,i,i+1,1);

/// non occupancy constraints
for (int i=0;i<n;i++)
   for (int j=i+2;j<n;j++)
      cstore_add(EUCL_GEQ,i,j,1);
```

An interesting component of the problem is to observe how the branch and bound strategy has been realized. Bound estimation strategies are effective at the last levels of the prop-labeling-tree. In this case, the estimation is activated for the last 2 levels of the search tree and provides a pruning of 50% of nodes. The estimate evaluates the contributions between labeled pairs and considers the maximal contacts that a non-completely specified pair can provide. The estimation process involves domain analysis, to establish whenever two domains could generate a contribution. Other considerations help in tightening the bound. For n amino acids, there are $n-1$ contributions that are independent from the specific conformation (consecutive pairs). Moreover, each amino acid i, $1 < i < n$, can form at most 2 extra contacts ($max\_contribs$) since, out of the 4 neighbors, two of them are already occupied by $i-1$ and $i+1$. For $i = 1$ and $i = n$, the number of extra contacts is 3, since there is only one linked neighbor (i.e., 2 and $n-1$, respectively). We provide in Figure 17 the pseudocode for the estimation.

Experimental Results: We implemented the model described in this section using different solvers. In particular, we tested

- CLP systems, using built in branch and bound (SICStus `clpfd` [55] and GNU Prolog CLP(FD) [29]);
- Integer Programming (IP) systems (GLPK [36], CPLEX [41], and PICO [32]);
- our COLA solver.

We also tested an additional version with COLA, in which we compute and provide the exact lower bound. This is to show that most of the time spent during the search with constraint systems is used to guarantee the optimality of the solution.

Table 7 reports the experimental results. For licenses and practical reasons, we run the CPLEX code on a Sun (**Sun** columns), 64-bit dual-processor Ultrasparc machine, with 300 MHz clock and 512MB RAM and PICO code (**Linux** columns) on a Linux-based machine, Athlon 1.5GHz, 1GB of RAm. The remaining results are computed with a Windows 2Ghz Centrino, 1GB Ram laptop (**Win** columns). In order to relate

```
D is the domain of variables (partially assigned)
s is the number of contacts provided by ground pairs
p is the number of possible contacts (at least one variable is non ground)

estimate(D)
s ← 0
p ← 0
for i = 0 to n − 2
do max_contribs = 2;
   if i = 0
      then max_contribs = 3
   possible = 0
   known_contribs = 1
   avail_spots = 0
   for j = i + 3 to n − 1
   do if i%2 ≠ j%2
         then avail_spots + +
              if var(i), var(j) are ground
                 then if contrib_hp(i, j) ≠ 0
                         then known_contribs + +
                              max_contribs − −
                 else  if i and j domains could generate a contribution
                         then possible + +
   s+ = known_contribs
   p+ = min(avail_spots, max_contribs, possible)
return s + p
```

Figure 17: The branch and bound estimation for our constraint solver

the results among different machines, we run for every architecture the same version of COLA.

The columns of the table are: the length of the input sequence (n); the execution times under Windows using COLA, COLA with lower bound information (COLA LB), GNU Prolog (GNU), SICStusProlog (SICStus) and GLPK; the execution times under Sun using COLA and CPLEX; the execution times under Linux using COLA and PICO. All execution times reported are in seconds.

In the Table the summarized performances for each solver are intended to give an overview of the different systems. The rigorous comparison between each pair of systems is not possible since we use different architectures, however these results can be collected to compare the performances of other systems w.r.t. our COLA solver.

In Figure 18 we merge the Table 7 results into a single overview. To obtain the Figure, we scale the performances with different architectures using COLA performances as the common denominator.

We would like to comment these results, referring to the corresponding Figure 18. In particular, we limit our analysis to *general* aspects of the results (also in consideration of the different platforms used in the experiments). In the Figure, the y axis is logarithmic, in order to better show an increase of the protein size translates in an exponential growth of the execution time (roughly linear plot). The only exception is

Table 7: Summary of execution times

n	Win					Sun		Linux	
	COLA	COLA LB	GNU	SICStus	GLPK	COLA	CPLEX	COLA	PICO
5	0,00	0,00	0,01	0,00	0,10	0	0,23	0,000	0,8
6	0,00	0,00	0,01	0,00	0,10	0	0,61	0,001	6,2
7	0,00	0,00	0,02	0,02	1,80	0,01	1,23	0,001	19,1
8	0,00	0,00	0,02	0,03	2,60	0,03	2,36	0,001	31,6
9	0,00	0,00	0,05	0,09	5,30	0,04	11,77	0,002	60,9
10	0,00	0,00	0,09	0,24	83,35	0,12	91,47	0,003	260,8
11	0,00	0,00	0,19	0,52	134,19	0,25	94,61	0,006	285,1
12	0,02	0,00	0,34	1,11	206,37	0,55	16,65	0,012	482,7
13	0,02	0,00	1,90	2,38	1013,35	1,26	163,7	0,024	2354,4
14	0,03	0,00	4,18	5,06		2,42	254,4	0,049	4841,2
15	0,06	0,00	9,47	10,20		4,79	262,9	0,090	3898,1
16	0,38	0,01	20,40	62,50		28,61	79,8		
17	0,88	0,00	45,56	145,10		65,15	3042		
18	1,95	0,01	99,61	319,20		154,8	2070		
19	3,84	0,02	399,0	726,50		322,5	3592		
20	7,96	0,04	460,0	1545,00		625	4837		
21	18,81	0,01	2589			1485	3019		
22	41,18	0,01				3229	4703		
23	84,79	0,02				6634	4021		
24	179,70	0,05				14076	8051		
25	1117,00	6,23				>86400	7516		
26	2719,00	0,18				>86400	>86400		

represented by CPLEX. In this case, the growth increases significantly when the length n overpass a full square core (i.e., k^2). This behavior suggests that protein lengths between k^2 and $(k+1)^2$, $k \in \mathbb{N}$, offer some properties, e.g., a common core of amino acids of type h composed of a square of side k, that provide some bounds to the search.

It is very interesting to note that the performance of SICStus and CPLEX are very different. SICStus shows an exponential behavior, while CPLEX has a step like function. We expect that for $n = 26$ (i.e., the next expected jump in the step function for CPLEX), the two execution times should be again comparable. Note that, for $n = 26$, CPLEX did not produce the optimal solution within 24 hours. For COLA, we expect an exponential growth similar to the one computed using the Windows machine and depicted in the graph.

Comparing the two CLP(FD) frameworks reveals that GNU Prolog performs roughly 3 times better than SICStus.

It is important to observe that the implementation of COLA is between 2 and 3 orders faster than SICStus Prolog. This also translates into a competitive alternative to CPLEX. The introduction of the exact lower bound (which is harder to compute in constraint systems than in IP systems), reduces dramatically the search. This could suggest that importing some ideas of IP into constraint frameworks could provide a great benefit.

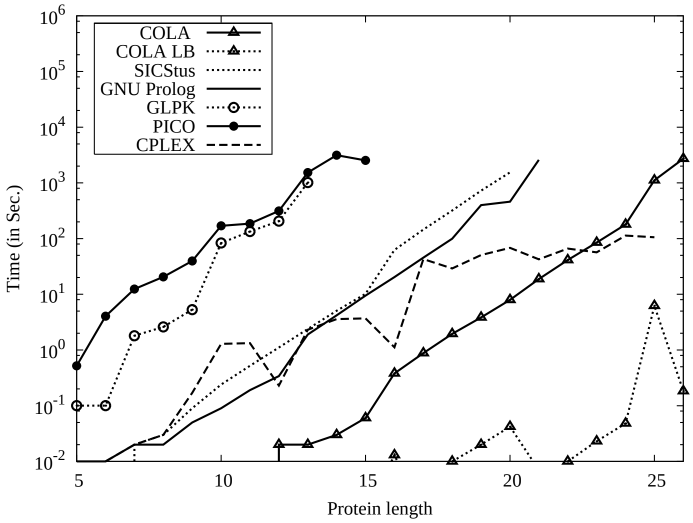


Figure 18: Comparison between CLP(FD), IP solvers, and COLA

We also wish to report that a direct translation of the constraint model into IP is about 2 orders of magnitude slower than the IP version we presented (tested on GLPK). This result is somehow expected, since IP performs better with simple domain variables (boolean) rather than variables with large domains. It is also interesting to note that the commercial version (CPLEX) is several orders of magnitude faster than the free version (GLPK). Moreover the IP formalization, once encoded using constraints, performs several orders slower when tested with SICStus. These facts suggest that the encodings used for both programming styles are the most reasonable ones.

Finally, the results of PICO system are significantly worse than COLA. Moreover a coarse comparison among the other solvers shows that the performances of PICO are similar to the ones of GLPK and several orders worse than CPLEX.

It is very hard to develop general conclusions, since the execution times rely heavily on the structure of the constraints and on the search strategies employed. A highly specific program for a framework can save great amount of time, thanks to the effort in coding the problem. This last parameter is usually hard to evaluate, especially since human skills and tricks used are often problem and paradigm dependent. We find, here, that the best way to test these paradigms on this problem is to code it in the most natural and simple way in each framework (i.e., CLP(FD) and IP). In particular, in CLP(FD) we use Finite Domains, that are expected to produce good performances when propagating interval arithmetic information. On the other side, in IP we made use of a great amount of boolean variables, that usually are better managed by the Linear Programming solving techniques.

These results are based on implementations without optimizations and heuristics. In this sense, both paradigms allow us to introduce many search heuristics and directives for the exploration of the search space. In IP, usually, the most effective strategy is on the splitting of variables' domain after a cycle of LP. Usually bisection is the most common choice, while many alternatives are possible in the selection of the variable. In CLP(FD), a popular choice is the *first fail* variable selection, where the variable with the smallest domain is selected for labeling.

It seems that, in the context of this example, domain propagation, which is not present in IP, is the key to achieve a good performance. Moreover, CLP(FD) allows the presence of non linear constraints, that are more difficult to handle in other frameworks. Nonetheless, non-linear constraints can propagate some information, and thus be effective in pruning the search tree.

6 Related Work

To the best of our knowledge, very limited work has been conducted in the field of *generic* constraint solving in lattice structures.

The constraint programming model adopted is similar in spirit to the model used in [42]—as they also make use of variables representing 3D coordinates and box domains. The problem addressed in [42] is significantly different, as they make use of a continuous space model, they do not rely on a energy model, and they assume the availability of rich distance constraints obtained from NMR data, thus leading to a more constrained problem—while in our problem we are dealing with a search space of $O(6^n)$ conformations in the $\mathcal{FCC}$ lattice for proteins with n amino acids. Every modification of a variable domain, in our version of the problem, propagates only to a few other variables, and every attempt to propagate refined information (i.e., the good/no good sub-volumes of [42]) when exploring a branch in the search tree, is defeated by the frequent backtracking. Thus, in our approach we preferred a very efficient and coarse bounds consistency. The ideas of [42], i.e., restricting the space domains for rigid objects is simply too expensive in our framework. We opted for a direct grounding of rigid objects, since in lattices there are few possible orientations. In our case, the position of objects can be basically anywhere, due to the lack of strong constraints. The techniques of [42] would be more costly and produce a poor propagation.

Our work is also related to the various proposals on spatial constraints, e.g., the C^3 system (implemented using CPLEX) [16]), and the various algorithms for consistency checking of (2D) Euclidean constraints [48, 3].

The bibliography on the protein folding problem is extensive [56]; the problem has been recognized as a fundamental challenge [20], 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 [21, 50], 2D triangular lattices [1], 3D square models [39], and face-centered cubic lattices ($\mathcal{FCC}$) [53]. Backofen and Will have extensively studied this last problem [5, 7, 6]. 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 $\mathcal{FCC}$ lattices are one of the best and simplest approximation of the 3D space (Sect. 3.1). 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 [15]. 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 [6]. 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 [18]. 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 [34].

7 Conclusion

In this paper, we presented a formalization of a constraint programming framework on crystal lattice structures—a regular, discretized version of the 3D space. The framework has been realized into a concrete solver (COLA), with various search strategies and heuristics. The solver has been applied to the problem of computing the minimal energy folding of proteins in the $\mathcal{FCC}$ lattice, providing high speedups and scalability w.r.t. previous solutions. The speedups derive from a more direct and compact representation of the lattice constraints, and the use of search strategies that better match the structure of the problem. We proposed branch and bound and problem-specific heuristics, showing how they can be integrated in our constraint framework to effectively prune the search space.

The parallel version of COLA, presented in this paper, proved to be robust and highly scalable. In particular, this version can be run on a generic cluster supporting MPI, and thus it is suitable for distributed computing and large-scale parallelism. Compared to the parallel implementation on a shared memory machine, and using CLP(FD), as presented in the literature [24], there are two main benefits to highlight. The first is the use of the COLA solver is the key to improve parallel scalability. The capability to distribute tasks during the exploration of a search tree, thanks to the free access to data structure and code of our solver, allows us to write efficient code. In CLP(FD), on the other hand, each task sent to the CLP(FD) solver, is solved as a whole, and this denies any possibility to interact at a finer granularity. The results show the better scalability. The second benefit is that a parallel version written in MPI allows us to port the code to a greater number of different architectures, and, at the same time, opens the possibility to employ large clusters, that play a fundamental role in solving the problem in a faster manner.

As future work, we will further investigate the possibility of using alternative forms of consistency, that will allow for greater propagation and more effective reduction of the search space. We will also explore the possibility to design new types of constraints, that will more directly address the presence of rigid structures.

Acknowledgments

We would like to thank Federico Fogolari for sharing his knowledge on the protein folding problem and Giuseppe Lancia for fruitful discussions on Section 5.4.3. The work is partially supported by NSF grants CNS 0544373, CNS 0454066, HRD 0420407, and CNS0220590, and by MIUR projects FIRB RBNE03B8KK and PRIN 2005015491.

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*, 275–296, 1997.

[2] H. Ait Kaci. *Warren's Abstract Machine: a Tutorial Reconstruction.* MIT Press, 1991.

[3] J. Allen. Maintaining Knowledge about Temporal Intervals. In *Communications of the ACM*, 26(11):832–843, 1983.

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

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

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

[7] R. Backofen and S. Will. Fast, constraint-based threading of HP sequences to hydrophobic cores. *Proc. of CP 2001*, LNCS 2239:494–508, 2001.

[8] R. Backofen. Constraint techniques for solving the protein structure prediction problem. *Proc. of CP 1998*, LNCS 1520:72–86, 1998.

[9] R. Backofen, S. Will and E. Bornberg-Bauer. Application of constraint programming techniques for structure prediction of lattice proteins with extended alphabets. *Bioinformatics*, 15(3):234–242, 1999.

[10] R. Backofen. A polynomial time upper bound for the number of contacts in the hp-model on the face-centered-cubic lattice (fcc). *J. of Discrete Algorithms*, 2(2):161–206, 2004.

[11] R. Backofen and S. Will. Optimally compact finite sphere packings – hydrophobic cores in the FCC. *Proc. Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, LNCS 2089:257–272, 2001.

[12] R. Barták. Constraint Programming: In Pursuit of the Holy Grail In *Proceedings of the Week of Doctoral Students (WDS99)*, Part IV, MatFyzPress, Prague, June 1999, pp. 555-564.

[13] F. Benhamou, L. Granvilliers, F. Goualard. Interval constraints: results and perspectives. In *New Trends in Constraints*, Springer Verlag, pp. 1–16, 1999.

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

[15] 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.

[16] A. Brodsky, V.E. Segal, P.A. Exarkhopoulo. Constraint Object-oriented Database Systems. Technical Report, George Mason University, 1996.

[17] Center for Computational Materials Science, Naval Research Labs, Crystal Lattice Structures, `cst-www.nrl.navy.mil/lattice/`.

[18] 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.

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

[20] Committee on Mathematical Challenges from Computational Chemistry. Mathematical Challenges from Theoretical-Computational Chemistry. National Research Council, 1995.

[21] P. Crescenzi, D. Goldman, C. Papadimitrou, A. Piccolboni and M. Yannakakis. On the complexity of protein folding. *Proc. of STOC*, 597–603, 1998.

[22] A. Dal Palù, A. Dovier and F. Fogolari. Protein Folding in CLP(FD) with Empirical Contact Energies. *Recent Advances in Constraints*, LNAI Volume 3010, 2004.

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

[24] A. Dal Palù, A. Dovier and E. Pontelli. Heuristics, Optimizations, and Parallelism for Protein Structure Prediction in CLP(FD). *Principles and Practice of Declarative Programming*, ACM Press, 2005.

[25] A. Dal Palù, A. Dovier and E. Pontelli. A Constraint Logic Programming Approach to 3D Structure Determination of Large Protein Complexes. *Proc. of LPAR'05*, Springer Verlag, 2005.

[26] A. Dal Palù, A. Dovier, E. Pontelli and G.Rossi. Integrating Finite Domain Constraints and CLP with Sets. D. Miller, ed., *Proc. of Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 219–229, 2003.

[27] A. Dal Palù, E. Pontelli, J. He, and Y. Lu. A Constraint Logic Programming Approach to 3D Structure Determination of Large Protein Complexes. *Proc. of Symposium on Applied Computing (SAC 2006)*, ACM Press, 2006.

[28] R. Dechter. *Constraint Processing*. Morgan Kauffman, 2003.

[29] D. Diaz. GNU Prolog: a Native Prolog Compiler with Constraint Solving over Finite Domains. `gnu-prolog.inria.fr/manual/index.html`, 2005.

[30] E. W. Dijkstra and C. S. Scholten. Termination Detection for Diffusing Computations. In *Information Processing Letters*, 11:1–4, 1980.

[31] A. Dovier, C. Piazza, E. Pontelli and G. Rossi. Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931,2000.

[32] J. Eckstein, C.A. Phillips, W.E. Hart. PICO: an Object-oriented Framework for Parallel Branch-and-Bound. In *Inherently Parallel Algorithms in Feasibility and Optimization and their Applications*, pp. 219–265, 2001.

[33] R. Finkel and U. Manber. DIB – a Distributed Implementation of Backtracking. In *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, 1987.

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

[35] 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 J.*, 70:1183–1197, 1996.

[36] GLPK: GNU Linear Programming Kit. `www.gnu.org/software/glpk/glpk.html`, 2003.

[37] A. Grama, A. Gupta, G. Karypis, V. Kumar. *Introduction to Parallel Computing, 2nd Edition.* Addison Wesley, 2003.

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

[39] W. E. Hart and A. Newman. The Computational Complexity of Protein Structure Prediction in Simple Lattice Models. *Handbook on Algorithms in Bioinformatics*, CRC Press, 2003.

[40] D. A. Hinds and M. Levitt. A lattice model for protein structure prediction at low resolution. *Proc. Natl. Acad. Sci. USA*, 89:2536–2540, 1992.

[41] ILOG. ILOG CPLEX: High Performance Software for Mathematical Programming. `www.ilog.com/products/cplex`, 2006.

[42] L. Krippahl and P. Barahona. Applying Constraint Propagation to Protein Structure Determination. *Proc. of CP 2003*, LNCS 1713:289–302, 1999.

[43] L. Krippahl and P. Barahona. PSICO: Solving Protein Structures with Constraint Programming and Optimisation. *Constraints* 7:317–331, 2002.

[44] L. Krippahl and P. Barahona. Propagating N-Ary Rigid-Body Constraints. *Proc. of CP 2003*, LNCS 2833:452–465, 2003.

[45] L. Krippahl and P. Barahona. Applying Constraint Programming to Rigid Body Protein Docking. *Proc. of CP 2005*, LNCS 3709:373–387, 2005.

[46] V. Kumar, A. Grama, V. Rao. Scalable Load Balancing Techniques for Parallel Computers. In *Journal of Parallel and Distributed Computing*, 22(1):60–79, 1994.

[47] J.L. Lassez and J. Jaffar. Constraint Logic Programming. In *Principles of Programming Languages (POPL)*, ACM Press, 1987.

[48] X. Liu, S. Shekhar, S. Chawla. Maintaining Spatial Constraints Using a Dimension Graph Approach. In *International Journal of Artificial Intelligence Tools*, 10(4):639–662, 2001.

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

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

[51] L. Perron. Search Procedures and Parallelism in Constraint Programming. In *Principles and Practice of Constraint Programming (CP)*, Springer Verlag, 1999.

[52] C-G. Quimper and T. Walsh. Beyond Finite Domains: the All Different and Global Cardinality Constraints. *Principles and Practice of Constraint Programming*, Springer Verlag, pp. 812–816, 2005.

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

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

[55] SICStus Prolog User's Manual, Release 3.12. Swedish Institute of Computer Science, 2005.

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

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

[58] 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.

[59] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[60] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe: a platform for constraint logic programming. IC-Parc, Imperial College, 1997.

Appendix 1.

We report an example of CSP coding in COLA. We encode here the wire problem described in the paper.

```
int wires=6;
int bounds=4;
int plugs[wires][4];  // inx iny outx outy
plugs[0][0]=1;plugs[0][1]=1;plugs[0][2]=1;plugs[0][3]=3;
plugs[1][0]=1;plugs[1][1]=1;plugs[1][2]=3;plugs[1][3]=1;
plugs[2][0]=3;plugs[2][1]=3;plugs[2][2]=1;plugs[2][3]=3;
plugs[3][0]=3;plugs[3][1]=3;plugs[3][2]=3;plugs[3][3]=1;
plugs[4][0]=1;plugs[4][1]=3;plugs[4][2]=2;plugs[4][3]=5;
plugs[5][0]=3;plugs[5][1]=3;plugs[5][2]=2;plugs[5][3]=5;
plugs[6][0]=1;plugs[6][1]=1;plugs[6][2]=2;plugs[6][3]=5;

////// NECESSARY (1): define n = number of variables (global variable)
n=wires*bounds;

////// NECESSARY (2): variables initialization
AllVariables* vars=(AllVariables*)malloc(sizeof(AllVariables));
allvar_init(vars);

///// set domains for variables (default is 0..MAXVAL)
int point[3],point2[3];
point[0]=0;point[1]=0;point[2]=0;
point2[0]=255;point2[1]=255;point2[2]=255;
for (int i=0;i<n;i++)
  var_init_bounds(vars->variables+i,point,point2);

////// NECESSARY (3): init constraint store
cstore_init(STORE_AC3);

///////////// constraints posting //////////////////////////////////////
//// NOTE: each constraint has to be inserted: C(v1,v2) and C(v2,v1)
/// wire
for (int i=0;i<wires;i++)
for (int j=0;j<bounds-1;j++){
    cstore_add(CONSTR_EUCL_LEQ,i*bounds+j,i*bounds+j+1,1);
    cstore_add(CONSTR_EUCL_LEQ,i*bounds+j+1,i*bounds+j,1);}

/// link io/outs
for (int i=0;i<wires;i++){
 point[0]=plugs[i][0];point[1]=plugs[i][1];point[2]=1;
 var_init_point(vars->variables+i*bounds,point);
 point[0]=plugs[i][2];point[1]=plugs[i][3];point[2]=1;
 var_init_point(vars->variables+i*bounds+bounds-1,point);}

///non overlap
/// skip plug-plug intersection
for (int i1=0;i1<wires;i1++)
for (int j1=0;j1<bounds;j1++)
for (int i2=0;i2<wires;i2++)
for (int j2=0;j2<bounds;j2++)
if (i1!=i2){
  if ((j1!=0 && j1!=bounds-1) || (j2!=0 && j2!=bounds-1))
    cstore_add(CONSTR_EUCL_G,i1*bounds+j1,i2*bounds+j2,1);}
///////////// end of constraints posting//////////////////////////////////////

////// NECESSARY (4): start search
search(vars,SEARCH_SIMPLE, SEARCH_FF);
```