Università degli Studi di Udine
Dipartimento di Matematica e Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Exploring the use of GPGPUs in Constraint Solving

Candidate:
Federico Campeotto

Supervisors:
Prof. Agostino Dovier
Prof. Enrico Pontelli

Academic Year December 9, 2014

Author's e-mail: campe8@nmsu.edu

Author's address:

Dipartimento di Matematica e Informatica
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

This Thesis is dedicated to whom it may concern.

Abstract

This dissertation presents an experimental study aimed at assessing the feasibility of parallelizing the constraint solving process using *Graphical Processing Units* (*GPU*s). GPUs support a form of data parallelism that appears to be suitable to the type of processing required to cycle through constraints and domain values during consistency checking and propagation. The dissertation also illustrates an implementation of a constraint solver capable of hybrid propagations (i.e., alternating CPU and GPU) and parallel search, and demonstrates the potential for competitiveness against sequential implementations. We consider the *Protein Structure Prediction* problem as a hard combinatorial real-world problem as case study to show the advantages of combining parallel search and parallel constraint propagation on a GPU architecture. We present the formalization and implementation of a novel class of constraints to support a variety of different structural analysis of proteins, such as loop modeling and structure prediction.. We demonstrate the suitability of a GPU approach to implement such MAS infrastructure, with significant performance improvements over the sequential implementation and other methods.

Acknowledgments

I thank my advisors Agostino Dovier and Enrico Pontelli for their guidance and encouragement in my research and for making this Thesis possible.

Contents

List of Figures

List of Tables

1 Introduction

Constraint programming (*CP*) is a declarative paradigm that aims to provide a general and high-level framework for solving combinatorial search problems. Based on strong theoretical foundations, it is attracting widespread commercial interest and it is now becoming the method choice for modelling many types of optimization problems [135]. Three main components characterize a constraint programming model: (1) variables, (2) variables' domains, and (3) relations among variables stated in the form of *constraints*. Differently from classical *imperative* approaches, there is no need to specify a sequence of steps to execute in order to find a solution for a given problem. Instead, the user specifies a set of properties that the solution must satisfy by posting constraints among variables. Then, a constraint solver explores the search space in order to find variable assignments that are consistent with all constraints. A *solution* is found when all variables have been assigned and all constraints are satisfied.

Different techniques from artificial intelligence, computer science, operations research, programming languages, and databases have been used to prune the search space and to find admissible assignments. These techniques try to overcome the inherent complexity of problems that usually are NP-hard, proving effective and allowing the constraint programming paradigm to be successfully applied to scheduling, bioinformatics, networks, configuration, and planning problems. Nevertheless, the cost of solving hard combinatorial real-world problems still motivates the exploration of new techniques to improve the exploration of the search space; parallelism has been recognized as a strong contender, especially with the wider availability of multi-core and cluster platforms.

The overall purpose of this dissertation is to show that the use of the *General-Purpose Computing on Graphics Processing Units* (*GPGPU*s) computational power in the form of *Single-Instruction Multiple-Threads* (*SIMT*) parallelism can improve the performance of constraint solvers providing a powerful parallel infrastructure at an affordable cost.

1.1 Research Objectives

In recent years, the computing industry has been involved in an architectural shift, increasing the number of computing cores and processors instead of the processor frequencies [77]. In the constraint solving domain, the wider availability of multi-core platforms raises the question of how to exploit this computational power and how to scale the current solving techniques. This dissertation aims to make a contribution to the domain of parallel constraint solving, by exploring ways of using Single-Instruction Multiple-Threads (SIMT) parallelism to reduce the cost of constrain propagation and search space exploration during the constraint solving process. Almost all modern desktops and laptops provide a powerful GPU, and there are several popular methods of utilizing GPUs, including CUDA [140] and OpenCL [159]. One of the aspects that has been brought to light is that GPUs are not a silver bullet, and direct ports of existing algorithms to a GPU architecture often perform poorly [85]. Nevertheless, this dissertation demonstrates the potential for using GPGPUs to speedup the constraint solving process. This is, to the best of our knowledge, the first comprehensive study investigating the use of GPGPUs in constraint solving (i.e., constraint propagation combined with parallel search); this study opens the doors to an alternative way to enhance performance of constraint solvers, through the unexploited computational power offered by GPUs.

1.2 Main Results

The typical constraint solving process is carried out by alternating two steps: (1) assigning values to variables (*labeling*), and (2) propagating constraints to prune the search space (*constraint propagation*). In this dissertation we present a feasibility study about the use of GPUs to speedup the constraint solving process.

GPU devices provide thousands of parallel cores and a well-structured multilevel data parallel decomposition represented by parallel blocks (coarse-grain parallelism) and parallel threads within each block (fine-grain parallelism). We consider this multilevel parallel decomposition to describe frameworks for parallel constraint solvers where the GPU computational power is exploited for both labeling and constraint propagation, leading to significant gains in terms of execution time. Our experimental results show that the use of GPUs in the constraint solving process can lead to remarkable speedups, outperforming corresponding sequential system and other implementations.

The combination of parallel search and parallel constraint propagation shows its major advantages when considering hard combinatorial real-world problems where the search space is usually large and the model is characterized by many hard *global constraints* (i.e., constraints defined on groups/sets of variables). As a real-world case study we considered the *Protein Structure Prediction* problem (*PSP* problem), i.e., the problem of predicting the three-dimensional structure of a protein given its amino acid sequence. PSP is one of the most important and complex problems in bioinformatics and theoretical chemistry. Suboptimal solutions computed by fast algorithms would be highly beneficial in biotechnology, and drug design. In this dissertation, we present a parallel constraint-based technology aimed at supporting structural studies of proteins. The solver relies on GPGPU computation to perform parallel local search and parallel constraint propagation. Our approach to the problem applies a *Multi-Agent System* (*MAS*) perspective, where concurrent agents explore the folding of different parts of a protein, leading to significant performance improvements (e.g., up to 100×) over the sequential implementation and other methods.

1.3 Thesis structure

The focus of this dissertation is on the study of a particular computing architecture (i.e., GPU architecture) to speed up the constraint solving process. Many hard, real-world combinatorial problems lend themselves to modeling as constraint satisfaction or optimization problems. Our efforts are then motivated by the need of efficient solvers for such diverse application domains as the areas of planning, job scheduling and bioinformatics [135]. We also consider a specific challenging constraint problem—the Protein Structure Prediction problem—as case study to test the implementation of a parallel constraint framework and assess the potential of GPU on a real-world problem. This dissertation is divided into three main parts organized as follows:

1.3.1 Constraint Programming: Definitions and Challenging Applications

In the first part of this dissertation (Chapter 2), we present some background information related to constraint problems and the constraint solving process in general. We introduce the notion of constraint problem, constraint engine and some techniques that are usually adopted to solve constraint satisfaction and optimization problems (e.g., arc-consistency notions, local search strategies, large neighborhood search). We also introduce the protein structure prediction problem, after a brief review of some basic biology notions (Chapter 3). In the last part of Chapter 3 (Section 3.2) we describe a sequential constraint resolution engine aimed at structural studies of proteins. The solver is suitable to address protein structure analysis studies, requiring the generation of a set of unbiased sampled diverse conformations which satisfy certain given constraints. The purpose of developing a sequential constraint solver for a challenging application is twofold. First, by describing a constraint solver we enter in detail in the aspects that characterize the solving process. The goal is to understand what kind of issues must be addressed in developing a solver and, in particular, which aspects of the solver can be parallelized in order to increase performance. Second, by considering an open research problem such as the PSP, we demonstrate the potential of constraint programming in a real-world problem.

1.3.2 Parallel Constraint Solving

In the second part (Chapter 4), we focus on parallel constraint solving. To this end, we introduce some notions about parallelism (e.g., metrics as *speedup*, *efficiency*, etc.), parallel constraint consistency (e.g., parallel *AC-3*, parallel *AC-4*, etc.), and parallel search in constraint programming. We will also touch some aspects regarding *DIStributed Constraint Satisfaction Problems* (*Dis-CSP*) and, in particular, *Distributed Constraint Optimization Problems* (*DCOP*) (Section 4.3). GPU computation and architecture, and the CUDA environment (i.e., the programming model used in this dissertation) are also presented as a part of background notions (Section 4.4).

The rest of the second part is organized as follows. In Chapter 5, we consider GPU-based propagation, presenting the structure of a constraint engine capable of hybrid propagation (i.e., alternating CPU and GPU) within a sequential exploration of the search space. Results are discussed with emphasis on the aspects that characterize the potential for effective exploitation of parallelism in the case of constraint propagation.

In Chapter 6, we analyze GPU-based local search in the domain of constraint programming. In particular, we focus on a parallel version of a popular local search method refereed to as *Large Neighborhood Search* (*LNS*). We describe a novel design and implementation of a constraint solver performing parallel search as well as a general framework that exploits SIMT parallelism to speedup local search strategies. We will present different local search strategies that can be used to explore in parallel multiple large neighborhoods. These strategies are implemented by making very localized changes in the definition of a neighborhood, namely, specifying the structure of a neighborhood. The main result of this section is a hybrid method for solving constraint optimization problems that uses local search strategies on large neighborhoods of variables. Usually, large neighborhood are explored using standard CP techniques. Instead, we present an approach based on local search to find the neighborhood that improves the objective function the most among a large set of different neighborhoods. Exploration of neighborhoods is performed in parallel on the GPU, leading to significant speedups in terms of time if compared with sequential implementation, standard CP, and standard LNS.

The second part of this dissertation ends by presenting some result on the use of GPUs in the domain of DCOPs (Chapter 7). DCOPs are defined as Distributed Constraint Optimization Problems, i.e., optimization problems where (sets of) variables represented by agents need to coordinate their value assignments to maximize the sum of resulting constrain utilities. We show that also in this context, the use of GPU computational power can lead to significant gains in terms of time and quality of the results when compared with standard methods.

1.3.3 Parallel Constraint Solving: Case Study

In the third part (Chapter 8), we re-consider the Protein Structure Prediction problem presented in Chapter 3 as case study for a parallel constraint solver that performs both parallel search and constraint propagation on GPU. We describe the design of the solver—based on a *Multi-Agent System* (*MAS*) infrastructure—and its implementation. The solver uses different agents for different parts of the protein performing parallel large neighborhood search, where spatial and geometric constraint are propagated and enforced on the protein structure using parallel threads. We show that our approach leads to remarkable speedups and is competitive with the state-of-the-art tools for protein structure prediction.

We conclude this dissertation (Chapter 9) summarizing the main results presented in the previous chapters and some research directions for future work.

I

Constraint Programming: Definitions and Challenging Applications

2 Background

2.1 Introduction

The set of conceptual tools provided by a programming language determines the way the programmer conceive and perceives the program. However, it is the way in which a programming problem is addressed that has a direct impact on the choice of the programming paradigm and the techniques used to solve it. Different programming paradigms, in fact, differ in the set of abstractions and tools available to the the programmer for representing the elements of the program (e.g., functions, objects, variables, constraints, etc.) and the procedures for processing the data.

Constraint Programming (*CP*) [135] is a declarative programming methodology that has gained a predominant role in addressing large scale combinatorial and optimization problems. As a paradigm, CP provides the tools necessary to guide the modeling and resolution of search problem– in particular, it offers declarative problem modeling (in terms of variables and constraints), the ability to rapidly propagate the effects of search decisions, and flexible and efficient procedures to explore the search space of possible solutions. In this chapter we consider a challenging constraint problem as case study to show several aspect related to the constraint programming paradigm.

We start by introducing the Constraint Programming paradigm 2.2, we present notions and definitions regarding constraint satisfaction and optimization problems, as well as the techniques and algorithms that are usually implemented in a constraint solver. Then, we focus our attention on the Protein Structure Prediction problem. We start by introducing the Constraint Programming paradigm (Section 2.2), we present notions and definitions regarding constraint satisfaction and optimization problems, as well as the techniques and algorithms that are usually implemented in a constraint solver. In Section 2.3 we focus our attention on strategies for solving constraint problems. We introduce the notions of *consistency techniques* and *constraint propagation* (Section 2.4). In Section 2.5 we present some *backtracking* techniques that are usually implemented in constraint solvers. We conclude the Chapter by presenting some background notions about *local search* strategies in Section 2.6.

2.2 Constraint Programming

Constraint satisfaction is the task of finding assignments to variables such that the assignments satisfy the constraints imposed on the variables. A *Constraint Satisfaction Problem* (*CSP*) $\mathcal{P}$ [135] is a triple (X, D, C) defined as follows:

- $X = \langle x_1, \dots, x_n \rangle$ is an n-tuple of variables;
- $D = \langle D^{x_1}, \dots, D^{x_n} \rangle$ is an n-tuple of *finite* domains, each associated to a distinct variable in X;
- C is a finite set of constraints on variables in X, where a constraint c on the m variables $x_{i_1}, \dots, x_{i_m}$, denoted as $c(x_{i_1}, \dots, x_{i_m})$, is a relation $c(x_{i_1}, \dots, x_{i_m}) \subseteq \times_{j=1}^{m} D^{x_{i_j}}$. The set of variables $\{x_{i_1}, \dots, x_{i_m}\}$ is referred to as the *scope* of c ($scp(c)$). If $m = 1$ or $m = 2$, the constraint is referred to as *unary* or *binary* constraint respectively. If $m > 2$ the constraint is referred to as *global* constraint.

A *solution* s of a CSP is a tuple $\langle s_1, \dots, s_n \rangle \in \times_{i=1}^n D^{x_i}$ s.t. for each $c(x_{i_1}, \dots, x_{i_m}) \in C$, we have $\langle s_{i_1}, \dots, s_{i_m} \rangle \in c(x_{i_1}, \dots, x_{i_m})$. $\mathcal{P}$ is (in)consistent if it has (no) solutions.

A *Constrain Optimization Problem* (*COP*) is a pair $\mathcal{Q} = (\mathcal{P}, g)$, where $\mathcal{P} = (X, D, C)$ is a CSP and

$$g : \times_{i=1}^n D^{x_i} \to \mathbb{R}$$

is a *cost function.* Given $\mathcal{Q}$, we seek a solution s of $\mathcal{P}$ such that $g(s)$ is maximal among all solutions of $\mathcal{P}$.

Example 2.2.1 *In the well-known* n-Queens *problem the task is to place n queens on a $n \times n$ chess board so that none of them can attack any other in one move. This problem can be modelled as a CSP as follows:*

- $X = \langle x_1, \dots, x_n \rangle$ *represents the rows of the chessboard;*
- $D = \langle D^{x_1}, \dots, D^{x_n} \rangle$*, where for* $1 \leq i \leq n, D^{x_i} = \{1, \dots, n\}$ *represents the column indexes of each queen placed on the chess board;*
- $C = \{|x_i - x_j| \neq |i - j| \wedge x_i \neq x_j \ : \ i = 1..n-1, j = 1..n\}$ *is the set of constraints used to impose that two queens cannot be placed on the same row, column or diagonal.*

Figure 2.1 shows a solution for the 8-Queens problem (solution $\langle 1, 7, 5, 8, 2, 4, 6, 3 \rangle$*).*

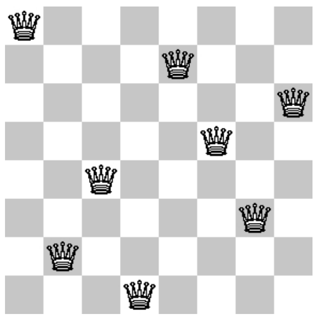


Figure 2.1: Example of a solution for the *8-Queens* problem.

Typical CSP solvers alternate two steps:

1. Selection of a variable and non-deterministic assignment of a value from its domain (*labeling*);

2. Propagation of the assignment through the constraints, to reduce the admissible values of the remaining variables and possibly detect inconsistencies (*constraint propagation*).

These two steps are alternated during the whole constraint solving process until either a variable's domain is empty or a solution is found. Let us observe that a variable assignment $x = d \in D^x$ implicitly defines a new decision node in a *search tree* by replacing the current CSP with another CSP where $D^x = \{d\}$. More formally, let us consider a CSP $\mathcal{P} = (X, D, C)$. We define the *search tree* for $\mathcal{P}$ as a tree defined as follows [6]:

- the nodes of the tree are CSPs;

- the root is $\mathcal{P}$;

- the nodes at an even level have exactly one child;

- if $\mathcal{P}_1, \dots, \mathcal{P}_m$, where $m \geq 1$, are direct descendants of $\mathcal{P}_0$, then the union of $\mathcal{P}_1, \dots, \mathcal{P}_m$ is equivalent w.r.t. X to $\mathcal{P}_0$.

It turns out that the resolution process is a sequence of transformations between CSPs that starts from the initial CSP and terminates when all the domains of a CSP are singletons or there is at least one that is empty [6]. It is possible to (informally) describe this process by means of *proof rules*. For example, the following rule

$$\frac{\phi}{\psi_1|\dots|\psi_n}$$

represents the transformation of the CSP ϕ to the CSPs $\psi_1,\dots,\psi_n$. Here, the "|" symbol represents an alternative choice between the possible transformations of ϕ, representing different nodes of the search tree.

Given a CSP (X,D,C), the labeling of a variable $x_i \in X$ where $D^{x_i} = \{d_1,\dots,d_m\}$, is represented by the following proof rule:

$$\frac{(X,D,C)}{(X,D,C\cup\{x_i=d_1\})|\dots|(X,D,C\cup\{x_i=d_m\})}.$$

Alternatively, the same labeling can be represented as follows:

$$\frac{(X,D,C)}{(X,\langle D^{x_1},\dots,D^{x_i}=\{d_1\},\dots,D^{x_n}\rangle,C)|\dots|(X,\langle D^{x_1},\dots,D^{x_i}=\{d_m\},\dots,D^{x_n}\rangle,C)}.$$

COP solvers use the same solving process scheme (i.e., alternation between labeling and constraint propagation) but they explore the space of possible solutions of the problem in order to find the optimal one (e.g., using branch and bound techniques). A complete COP solver stops whenever exploration is complete, while an incomplete COP solver might stop when a given limit is reached (e.g., time/number of improving solutions), returning the best solution found so far [135].

2.3 Search and Labeling

The most straightforward strategy that can be used to solve a CSP is the *Generate and Test* (*G&T*) search strategy. G&T is a very simple algorithm that guarantees to find a solution if one exists. It works as follows: it first generates a possible solution, i.e., a complete labeling of variables, checking whether constraints are satisfied. If a solution is found it returns the solution. Else, it generates another possible solution. The main drawback of this algorithm is its efficiency since it exhaustively explores the search space and eventually discovers inconsistencies on complete assignments of the variables. For example, given a CSP $\mathcal{P} = (X,D,C)$ such that $\mathcal{P}$ is inconsistent, $|X| = n$, and d is the maximum size of the domains, the G&T technique will explore $\mathcal{O}(d^n)$ possible complete labelling before discovering that no solution for $\mathcal{P}$ exists. More efficient techniques are presented in what follows.

2.3.1 Backtracking

Backtracking strategies [125] combine the generation of solutions with consistency checking. A solution is incrementally constructed by starting from partial assignments of values to variables and *backtracking* to previous assignments as soon as the current assignment does not satisfy one or more constraints.

Example 2.3.1 *Let us consider an* n-Queens *problem instance where $k < n$ queens have been already placed on the chessboard, i.e., the variables $x_1,\dots,x_k$ have been labeled and the current partial assignment satisfies all constraints. Instead of labeling all the remaining variables $x_{k+1},\dots,x_n$ and then checking for the consistency of the solution, a backtracking strategy first labels the variable x_{k+1} with a value $d_i \in D^{x_{k+1}}$, then it checks for consistency of constraints. If all constraints are satisfied by the current partial assignment, the search continues labeling the variables $x_{k+2},\dots,x_n$, otherwise a new assignment for x_{k+1} is tried. If none of the assignments for x_{k+1} is a valid assignment (i.e., it satisfies all constraints), the search* backtracks *to x_k and the process is repeated. Eventually, either the search backtracks to the first variable or a solution is found.*

Backtracking strategies are the "framework" in which labeling strategies usually operate: several choices can be made both for the next variable to label and for the value to assign to it. Simple enumeration of values usually performs poorly and different *heuristics* are provided by constraint solvers. Informally, a *heuristic* is a rule that is used to choose one of many alternatives. In the context of search in CSPs, a heuristic usually provides the rules to determine: (a) which variable to label, and (b) which value to select. For example, a heuristic can select the variable that has the smaller domain among all the other variables, or the variable that appears in the largest number of constraints. Another heuristic for selecting the value can choose always the minimum or the maximum element of a domain. There are a number of heuristics and, beside standard heuristics, they usually differ with the specific implementation of the constraint solver.

Although backtracking improves G&T algorithm by eliminating a large number of assignments that do not satisfy the constraints, there are two main issues that could dramatically decrease the performance of standard backtracking: (1) late detection of conflicts, and (2) repeated failures due to the same reason (e.g., a prefix of the current partial assignment may have the same property that leads to a failure). The former can be avoided using *consistency techniques* (see Section 2.4) to detect inconsistent partial assignment sooner, while the latter can be partially avoided by some kind of intelligent backtracking such as *Backjumping* or *Backmarking* (see Section 2.5).

2.4 Consistency Techniques and Constraint Propagation

The constraint programming solving process can be described by a sequence of transformations among equivalent CSPs (see Section 2.2).
Given two CSPs $\mathcal{P}_1 = (X_1, D_1, C_1)$ and $\mathcal{P}_2 = (X_2, D_2, C_2)$ and the set of variables $X^{\cap} = X_1 \cap X_2$, we say that $\mathcal{P}_1$ and $\mathcal{P}_2$ are *equivalent w.r.t.* $X^{\cap}$ if:

- For every solution s_1 of $\mathcal{P}_1$, $\exists s_2$ solution for $\mathcal{P}_2$ such that $s_{2|_{X^{\cap}}} = s_{1|_{X^{\cap}}}$;
- For every solution s_2 of $\mathcal{P}_2$, $\exists s_1$ solution for $\mathcal{P}_1$ such that $s_{1|_{X^{\cap}}} = s_{2|_{X^{\cap}}}$;

where $s_{|X}$ is the projection of the tuple s on the variables in X. The above definition can be further generalized for a sequence of CSPs $\mathcal{P}_1 = (X_1, D_1, C_1), \dots, \mathcal{P}_n = (X_n, D_n, C_n)$ w.r.t. the set of variables $X^{\cap} = \bigcap_{i=1}^{n} X_i$ and $\mathcal{P}_1$, if $\mathcal{P}_1$ and $\mathcal{P}_i$ are equivalent w.r.t. $X^{\cap}$, for $2 \leq i \leq n$.

During the constraint solving process, a constraint solver performs constraint propagation in order to create a sequence of CSPs which are equivalent to the original CSP but "simpler". The definition of "simpler" depends on the application but a CSP is usually considered simpler when its domains become smaller. This process is called *constraint propagation* and it is based on algorithms referred to as *constraint propagation algorithms* or *constraint propagators.* Constraint propagation is performed by repeatedly reducing domains while maintaining equivalence, i.e., they guarantee some form of "local consistency", ensuring that some subset of variables are consistent.

Example 2.4.1 *Let us consider a simple CSP*

$$\mathcal{P} = (\langle x_1, x_2 \rangle, \langle [l_{x_1}, u_{x_1}], [l_{x_2}, u_{x_2}] \rangle, \{x_1 < x_2\}),$$

where the domains of x_1 *and* x_2 *are represented by the intervals of integers* $[l..u]$*. Then, the constraint propagation algorithm for* $x_1 < x_2$ *is represented by the following rule:*

$$\frac{x_1 < x_2;\ D^{x_1} = [l_{x_1}, u_{x_1}], D^{x_2} = [l_{x_2}, u_{x_2}]}{x_1 < x_2;\ D^{x_1} = [l_{x_1}, \boldsymbol{min}(u_{x_1}, u_{x_2} - 1)], D^{x_2} = [\boldsymbol{max}(l_{x_2}, l_{x_1} + 1), u_{x_2}]}.$$

that transforms both domains to smaller ones, while preserving the equivalence of the two CSPs. For the sake of clarity, here we used a simplified but more intuitive version of the proof rule syntax defined in Section 2.2.

Consistency techniques are often based on the notion of constraint graphs. Given a CSP $\mathcal{P} = (X, D, C)$, a *constraint graph* for $\mathcal{P}$ is a graph (V, E), where $V = X$ and $E = C$, i.e., nodes correspond to variables and arcs (edges) represent constraints. Notice that the above definition assumes that all constraints are unary or binary. This is not a restriction since any global constraints can replaced by a semantically equivalent set of binary constraints introducing auxiliary variables [12].

2.4.1 Node Consistency.

The simplest notion of local consistency is called *node consistency.* We say that a CSP $\mathcal{P} = (X, D, C)$ is node consistent if for every variable $x \in X$, D^x is consistent with every unary constraint c on x:

- $\forall a \in D^x,\ a \in c(x)$;

A simple algorithm that achieves node consistency removes values from variables' domains that are inconsistent with unary constraints. It is easy to see that node consistency can be achieved in $\mathcal{O}(nd)$ time [109], where n is the number of variables and d the maximum size among all variables' domains.

2.4.2 Arc Consistency.

The most common notion of local consistency is *Arc Consistency* (*AC*). Let us consider a binary constraint $c \in C$, where $\text{scp}(c) = \{x_i, x_j\}$ and $x_i, x_j \in X$. We say that c is arc consistent if:

- $\forall a \in D^{x_i},\ \exists b \in D^{x_j} (a, b) \in c(x_i, x_j)$;
- $\forall b \in D^{x_j},\ \exists a \in D^{x_i} (a, b) \in c(x_i, x_j)$.

Given a value $a \in D^{x_i}$ ($b \in D^{x_j}$), the corresponding value $b \in D^{x_j}$ ($a \in D^{x_i}$) s.t. $(a, b) \in c$ is referred to as the *support* of $a(b)$ in $D^{x_j}(D^{x_i})$ w.r.t. the constraint c. A CSP is *arc consistent* if all its binary constraints are arc consistent.

Example 2.4.2 *Consider the following CSP $\mathcal{P}$:*

$$(\langle x_1, x_2\rangle, \langle D^{x_1} = [5, 10], D^{x_2} = [3, 7]\rangle, \{x_1 < x_2\}).$$

Then, $\mathcal{P}$ is not arc consistent. For example, for $x_1 = 8, \nexists b \in D^{x_2}$ s.t. $8 < b$.

There are several algorithms to achieve arc consistency. These algorithms usually perform an iterative process that consider one edge of the constraint graph at a time, reducing the domains of the variables corresponding to the pair of adjacent nodes. The process iterates until the CSP is arc consistent (i.e., no further reduction is possible) or some domain is empty

The popular *AC-3* algorithm [108] ensures arc consistency following the same iterative process, as reported in Algorithm 1.

Algorithm 1 AC-3(X, C)

```
1: $Q \leftarrow \{(x_i, c) \mid c \in C, x_i \in$ scp$(c)\}$;
2: while $Q \neq \emptyset$ do
3:   select and remove $(x_i, c)$ from $Q$;
4:   if revise$(x_i, c)$ then
5:         if $D^{x_i} = \emptyset$ then return False;
6:         else $Q \leftarrow Q \cup \{(x_j, c') \mid c' \in C \wedge c' \neq c \wedge x_i, x_j \in$ scp$(c') \wedge j \neq i\}$;
7:   end if
8: end while
9: return True;
```

A queue of constraints Q, referred to as *constraint queue*, is initially filled with all the arcs of the CSP (line 1). Then the algorithm "revises" the arcs and adds to the queue those that are affected by previous revisions (line 6). Arcs are revised by the **revise** function, which tests for consistency along the arc, (i, j) for every adjacent node j (line 4). This function removes any value in D^{x_i} that has no *support* in D^{x_j} w.r.t. the constraint c. If at least one domain has been changed by the **revise** function, the constraint queue is updated accordingly. The procedure ends when either the constraint queue or some domain is empty. AC-3 ensures arc consistency in $\mathcal{O}(ed^3)$ time where e is the number of binary constraints in the CSP [109] and d is the maximum size of the domains.

Another popular algorithm for arc consistency is the *AC-4* [135] algorithm. The difference between AC-3 and AC-4 is on the way they revise the arcs. In particular, AC-4 keeps tracks of the supports of each value for each node w.r.t. corresponding arcs. As long as a label l at a node i has some support from one or more labels at all other nodes, l is considered a valid label for i. To check whether l has a support, the algorithm uses additional data structures: a *counter* that is used to count the number of supports for each label of each variable, and a *support* set of labels to update every time a label has been deleted. AC-4 improves AC-3 since it runs in $\mathcal{O}(ed^2)$ in the worst case.

2.4.3 k-Consistency.

The general notion of consistency is *Hyper-arc consistency* also referred to as *k-consistency* [135]. Hyper-arc consistency is defined as follows. A constraint c defined on k variables $x_1, \dots, x_k$ is k-consistent if:

- $\forall a \in D^{x_i} 1 \leq i \leq k,\ \exists d_1 \in D^{x_1}, \dots, d_{i-1} \in D^{x_{i-1}}, d_{i+1} \in D^{x_{i+1}}, \dots, d_k \in D^k$ s.t. $\langle d_1, \dots, d_{i-1}, a, d_{i+1}, \dots, d_k \rangle \in c(x_1, \dots, x_k)$.

This notion of consistency generalize arc consistency and, if applied systematically on a constraint graph, it can prune large portions of the search space. Nevertheless, achieving hyper-arc consistency requires $\mathcal{O}(d^n e)$ time, which is unacceptable for most problems. Therefore, it is usually preferred to apply some approximated forms of constraint consistency (i.e., there may still remain some inconsistent values) and use consistency techniques together with backtracking search strategies. For example, constraint solvers usually implement global constraints through functions that perform approximated propagation in *polynomial* time (e.g., for *NP*-hard constraints) still pruning considerable part of the search space (e.g., see [135], Chapter 6). Moreover, when the domains of the variables are integer sets of values, another form of consistency referred to as *bound consistency* can be defined. This form of consistency regards only the extreme values of the domains (i.e., minimum and maximum values) but it can be forced by efficient propagators. Bound consistency is usually adopted for arithmetic constraints and several global constraints on finite integer domains.

2.5 Intelligent Backtracking

Backtracking can be combined with constraint propagation, leading to more complex forms of search methods that are specific for constraint programming. These techniques are usually called *intelligent backtracking.*

2.5.1 Look-back Methods.

Look-back methods perform consistency checks on variables that have been already labelled, avoiding useless explorations of the search tree. The idea is to learn information while searching (i.e., learning from search failures), in order to avoid the repetition of the same mistakes. There are two general Look-back methods strategies [59]: (1) *backjumping* and (2) *backmarking.*

When using backjumping we allow the backtracking process to jump further back in the tree than just to the parent node. More precisely, whenever a failure is found, the current state of the search phase is analysed in order to identify the partial assignment that causes inconsistency, and the search process "jumps back" to the most recent conflicting variable rather than the immediately preceding variable.

Example 2.5.1 *Consider the following CSP $\mathcal{P}$:*

$$(\langle x_1, x_2, x_3, x_4, x_5\rangle, \langle[1,4],[1,4],[1,4],[1,4],[1,3]\rangle, \{x_2 = 1 \rightarrow x_5 > 3, \ldots\}).$$

Let us assume that at a certain point of the search process the variables x_1, x_2, x_3, x_4 are labeled with the following values: $x_1 = 2, x_2 = 1, x_3 = 1, x_4 = 1$ and the search performs simple backtrack. Since no value for x_5 can satisfy the constraint $x_2 = 1 \rightarrow x_5 > 3$ the search process backtracks to x_4 trying a different assignment, then to x_3, and eventually to x_2. On the other hand, if the search performs backjumping, the current partial assignment and the set of constraints are "analyzed" (i.e., the algorithm checks the prefixes of the current partial assignment w.r.t. the constraints involving the corresponding variables), proving that the assignment $x_2 = 1$ is not part of any solution. Therefore, the search process backtracks directly to the labeling of x_2.

Different algorithms use different techniques to identify the variable to jump to (e.g., analysis of the constraint graph, analysis of the violated constraints w.r.t. the domains of the variables in their scope, etc.) with different costs (see [45]).

Backmarking [135] improves backtracking by maintaining information about instantiated variables and changes that happened before each labeling. This information is used for every label to determine incompatible assignments for the other variables, avoiding some consistency checks. Let us observe that backmarking does not reduce the search space but it may only reduce the number of operations needed to find a solution.

2.5.2 Look-ahead Methods.

Look-back methods guide the search process by retrieving information from previous failures. Nevertheless, they do not prevent inconsistency from occurring.

Look-ahead methods are used to prevent future conflicts by suggesting the next variable to label and the sequence of values to try for that variable [119]. The simplest look-ahead technique is called *forward checking* (*FC*) and it is based on a weak form of arc consistency. It performs arc consistency only on the arcs that connect the assigned variables and the unassigned variables. If *full* arc consistency is used after each labeling step on all constraints, then this type of look-ahead is also referred to as *Maintaining Arc Consistency* (*MAC*). In other terms, AC enforces arc consistency on a constraint graph, while MAC refers to a backtracking scheme where after each labeling, arc consistency is maintained (or enforced) on the new CSP [103]. Let us observe that MAC prunes the search tree more than FC but at a higher computational cost.

Example 2.5.2 *Figure 2.2 shows how Forward Checking (left) and MAC (right) can prune the search space on the 4Queens problem. Filled dots represent the current partial assignment of the variables. The two algorithms propagate the constraints and filter the variables' domains w.r.t. the current partial assignment. Inconsistent values are represented by the symbol "x".*

2.6 Local Search

Given a sufficient amount of time, a *complete search strategy* finds (all) the solution(s) of a given problem (if there are any) by systematically exploring each path (or parts of it) of the search tree. However, in hard combinatorial problems the search space is usually large [22] and faster but incomplete methods are preferred to complete methods. Therefore, local search strategies have become popular.

Local Search (*LS*) techniques [1, 135] deal with constraint problems and optimization problems in general, and are based on the idea of iteratively improving a candidate solution s by minor "modifications" in order to reach another solution s' from s. The set of allowed modifications is called the *neighborhoods* of s and it is often defined by means of a neighborhood function η applied to s. LS methods rely on the existence of a candidate solution. Most problems typically have a naive (clearly not optimal) solution. If this is not the case, some constraints can be relaxed, i.e.,

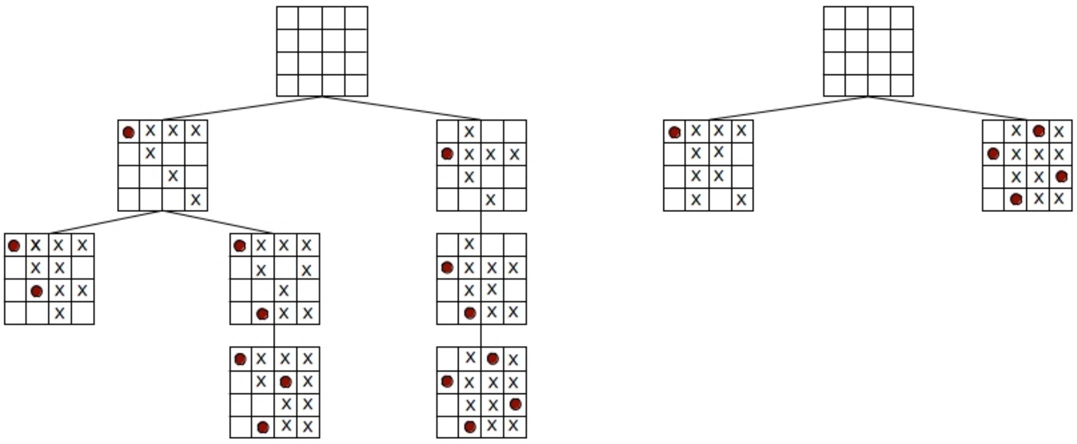


Figure 2.2: Forward checking (left) and MAC (right) on the 4-Queens problem.

an assignment is a solution whenever it satisfies all constraints but the relaxed ones. The cost function is therefore modified in order to take into account the number of unsatisfied constraint: when a solution of cost 0 is found, it will be used as a starting point for the original COP. Other techniques (e.g., a constraint solver) might be used to determine the initial candidate solution.

To ensure that the search process does not get stuck in a local optima or in an unsatisfactory solution, some kind of randomness is often used during the search. These algorithms are referred to as *Stochastic Local Search* methods.

Local search strategies are very effective for solving optimization problems, even though their effectiveness usually depends on the type and the landscape of the optimization problem [158]. On the flip side, their implementation usually requires few lines of code and their algorithmic structure is often relatively easy. In what follows we introduce some well-known local search algorithms.

2.6.1 Hill-climbing.

The simplest local search strategy is most probably the *Hill-climbing* technique [125]. Starting from an initial random assignment $s = \langle x_1 = d_1, \ldots, x_i = d_i, \ldots, x_n = d_n \rangle$, the neighborhood of s corresponds to all possible new assignments for any one variable x_i:

$$\eta(s) = \bigcup_{i=1}^{n} \{ \langle x_1 = d_1, \ldots, x_i = a, \ldots, x_n = d_n \rangle \mid a \in D^{x_i} \}.$$

The cost function $g(\cdot)$ is used to select an *improving* labeling w.r.t. s, i.e., a new complete assignment of variable s' randomly chose in $\eta(s)$ such that $g(s') < g(s)$. If a local minimum is reached, then it restarts from another initial random assignment. The algorithm stops as soon as the global minimum is found or it has reached some stopping criteria (e.g., time-out). Let us observe that the *Hill-climbing* re-labeling step is usually implemented by either a random selection of the variable or by a sequential scanning through all the variables.

The cost function $g(\cdot)$ is usually defined based on the specific problem/application. A common choice for CSP problems is to the *Min-conflict* heuristic by defining $g(\cdot)$ to prefer assignments that minimize the number of violated constraint. The heuristic iteratively evaluates the *conflict set* for the current assignment, i.e., the set of all variables that appear in an unsatisfied constraint. A variable from the conflict set is chosen randomly and it is re-labeled with a value that reduces the number of violated constraints or a random one if no value improves the cost function. To escape from local optima, the value of the cost function calculated by $g(\cdot)$ can be multiplied by a worsening factor with probability p:

$$g(s)' = \begin{cases} g(s), & \text{if } r \geq p \\ g(s) * k, & \text{if } r < p \end{cases}$$

where $g(\cdot)'$ is the modified version of the function $g(\cdot)$ that takes into account worsening labellings, $r \in [0..1]$ is a random value, and $k < 1$ is a constant that increases the value of $g(s)$.

2.6.2 Tabu Search.

Tabu search [65] strategies guide the search process avoiding being trapped in local optima. A Tabu search technique is based on the notion of *forbidden states*, i.e., a "short term memory" list of assignments to avoid in order to guide the search process always to new or non-recent nodes of the search tree. The term "non-recently" depends on the length k of the list of forbidden states. More precisely, the algorithm avoids partial assignments that have already been considered. This check is performed from the current state of the search process on, for the next k iterations. This strategy ensures that the search cannot being trapped in short term cycles and allows the search process to go beyond local optimality.

Tabu search can be combined with the Min-conflict heuristic. The *Min-conflict Heuristic with Tabu Search* generally performs better that both Min-conflict and Tabu search alone [160]. This algorithm works exactly as the Min-conflict technique, except that whenever a variable x is re-labeled with a new value d, the pair (x, d) is inserted in the Tabu list. This prevents the search process to re-consider previous labellings for the next k steps, unless such labellings lead to improvements.

2.6.3 Large Neighborhood Search.

LNS [144, 78] is an incomplete technique that hybridizes CP and LS to solve optimization problems. It is a particular case of local search where $\eta(s)$ generates a (random) neighborhood larger than those typically used in LS. The difference is that these sets of candidate solutions are explored using constraint based techniques, and the best improving solution is looked for. If after a timeout an improving solution is not found, a new random neighborhood is attempted. The process iterates until some stop criteria are met. Technically, all constraints among variables are considered, but the effect of $\eta(s)$ is to destroy the assignment for a set of variables. The stop criteria can include a global timeout or a maximum number of consecutive choices of $\eta(s)$ that have not lead to any improvements.

3 Protein Structure Prediction

Constraint Programming is a declarative programming methodology that has gained a predominant role in addressing large scale combinatorial and optimization problems. The declarative nature of CP enables fast and natural modeling of problems, facilitating not only development, but the rapid exploration of different models and resolution techniques (e.g., modeling choices, search heuristics).

In recent years, several research groups have started appreciating the potential of constraint programming within the realm of *Bioinformatics*. The field of Bioinformatics presents a number of open research problems that are grounded in critical exploration of combinatorial search space, highly suitable to be manipulated through constraint-based search. Constraint methodologies have been applied to analyze DNA sequences for instance, to locate Cis-regulatory elements [68], to DNA restriction maps construction [174], and to pair-wise and multiple sequence alignment [172, 173, 165]. Constraint methodologies have been applied to biological networks [31, 96, 133, 60, 61] and to other biological inference problems, such as Haplotype inference [67, 50], and phylogenetic inference [49].

A particular area of Bioinformatics that has witnessed an extensive use of CP techniques is the domain of *structural biology*—i.e., the branch of molecular biology and biochemistry that deals with the molecular structure of nucleic acids and proteins, and how the structure affects behavior and functions. Constraint Programming has progressively gained a pivotal role in providing effective ways to explore the space of conformations of macromolecules, to address problems like secondary and tertiary structure prediction, flexibility, motif discovery, docking [9, 93, 164, 39, 112, 149, 94, 41, 24, 178]. Two comprehensive surveys on the use of constraint-based methods in structural Bioinformatics have been recently proposed [37, 11].

Our focus in this chapter is on the use of constraint-based technology to support structural studies of *proteins*. We considered a *challenging constraint problem* as case study to introduce and analyze several aspects regarding the implementation of a constraint solver. We present the techniques adopted for propagating constraints and exploring the search space as well as their implementation.

Overview of the Chapter. Proteins are macromolecules of fundamental importance in the way they regulate vital functions in all biological processes. Their structural properties are critical in determining the biological functions of proteins [155, 10] and in investigating protein-protein interactions, which are central to virtually all cellular processes [3]. We refer to the *Protein Structure Prediction (PSP)* problem as the problem of determining the tertiary structure of a protein from knowledge of its primary structure and/or from knowledge of other structures (e.g., secondary structure components, templates from homologous proteins). The PSP problem is also often broken down to specialized classes of problems related to specific aspects of the tertiary structure of a protein, such as side-chain geometry prediction [48], loop modeling prediction [66, 170, 137, 156], and protein flexibility investigation [14].

All these classes of problems share common roots—the need to track the possible conformations of chains of amino acids. The variations of the problem relate to factors like the length of the chain being considered (from short peptides in the case of loop modeling to entire proteins in the general PSP case) and the diverse criteria employed in the selection of the solutions, as, for instance, the lowest basin of the effective energy surface, composed by the intra-molecular energy of the protein plus the solvation free energy [88, 98].

Modeling the variability of a protein chain involves many degrees of freedom which are needed to represent different protein conformations. Tracking this variability requires the exploration of a vast conformational space. Model simplifications can be adopted to reduce such computational cost, for instance backbone-only models represent only the backbone of proteins, the side-chain representation could be simplified to a single central point (centroid) describing its center of mass, or one can adopt approximated representation of the space though lattice models.

Nevertheless, even under strong simplifications, the search space remains intractable and prevents the use of brute-force search methods in the space of possible conformations [33].

Constraint programming methodologies have found natural use in addressing PSP and related problems—where structural and chemical properties have been modeled in terms of constraints over spatial positions of atoms, transforming the search of conformations into a constraint satisfaction/optimization problem. The proposed approaches range from pure ab initio methods [9, 39] to methods based on NMR data [95] to methods based on fragments assembly [42]. In spite of all these efforts, the design of effective approaches to filter the space of conformations and lead to a feasible search remains a challenging and open problem.

In this chapter we present a constraint solver targeted at modeling a general class of protein structure studies. In particular our solution is suitable to address protein structure analysis study, requiring the generation of a set of unbiased sampled diverse conformations which satisfy certain given restraints. One of the unique features of the solution presented in this work is its capability to generate a uniformly distributed sampling of target protein regions among a given portion of Cartesian space and with selected granularity—accounting both for spatial and rotational properties.

We abstract the problem as a general *multi-body* system, where each composing body is constrained by means of geometric properties and it is related to other bodies through joint relationships. Each body can represent an entity in the protein, such as an individual amino acid or a small peptide (e.g., a protein fragment). Bodies relate to the spatial positions and organization of individual atoms composing it.

The view of the exploration of protein structures as multi-body systems suggests a number of different constraints, that can be used to model different classes of structural studies and applied to filter infeasible (or unlikely) conformations. We propose an investigation of several classes of constraints, in terms of both their theoretical properties and practical use for filtering. Particular emphasis is given to the *Joined-Multibody* (JM) constraint, whose satisfaction we prove to be NP-complete. Realistic protein models require the assembly of hundreds of different body versions, making the problem intractable. We study an efficient approximated propagator, called *JM filtering* (JMf), that allows us to efficiently compute classes of solutions, partitioned by structural similarity and controlled tolerance for error. This perspective is novel and holds strong potential. The structural problems we are investigating are computationally intractable; the use of global constraints specifically designed to meet their needs enables a more effective exploration of the search space and a greater potential for effective approximations.

The multi-body model provides an interesting perspective in exploring the space of conformations –while the actual search operates on discrete sets of alternatives (e.g., sets of fragments), the filtering process avails of reasoning processes that operates with continuous domain; this allows the propagation and filtering to be effective.

The proposed multi-body constraints and filtering techniques constitute the core of the resolution engine of *FIASCO (Fragment-based Interactive Assembly for protein Structure prediction with COnstraints)*, an efficient C++-based constraint solver. We demonstrate the flexibility and efficiency of FIASCO by using its engine to model and solve a class of problems derived from loop modeling instances. Throughout the paper we show the ability of FIASCO of providing a uniform and efficient modeling platform for studying different structural properties (that have been, so far, addressed only using significantly distinct methods and tools). The declarative nature of constraint-based methods supports a level of elaboration tolerance that is not offered by other

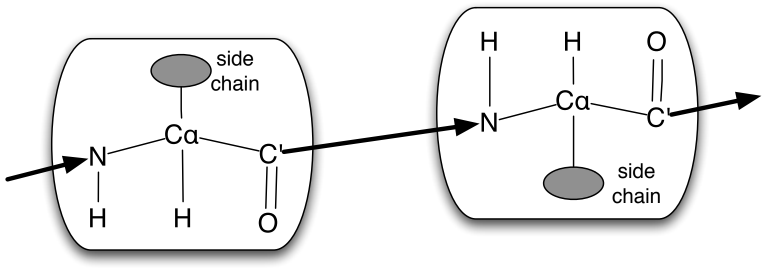


Figure 3.1: A schematic sequence of two amino acids showing the amino acid backbone and their side chains. The arrow from C' to N denotes the peptidic bond.

frameworks for protein structure prediction, facilitating the integration of additional knowledge in guiding the studies (e.g., availability of information about secondary structure elements).

The rest of the chapter is organized as follows. In Section 3.1, we provide a high-level background on the biological and chemical properties of proteins and review the most commonly used approaches to address structural studies. In Section 3.2, we develop the constraint framework for dealing with fragments and multi-body structures. Section 8.2.2 describes the implementation of the constraints and their propagation schemes in the FIASCO system. In Section 3.4 we report the experimental results from using FIASCO on a collection of benchmarks on loop modeling. Section 3.5 provides some concluding remarks.

3.1 Biological Background, General Context, and Related Work

In this section we will briefly review some basic Biology notions, introduce the problems we are tackling in this chapter and refer to a selection of the related literature.

3.1.1 Biological Background

A protein is a molecule made of smaller building blocks, called *amino acids.* One amino acid can be connected to another one by a *peptidic* bond. Several amino acids can be pairwise connected into a linear chain that forms the whole protein. The *backbone* of a protein, as illustrated in Figure 3.1, is formed by a sequence of N–C_α–C' atoms contained in each amino acid. The backbone is rather flexible and it allows a large degree of freedom to the protein.

Each amino acid is characterized by a variable group of atoms that influences the specific physical and chemical properties. This group, named *side chain*, ranges from 1 to 18 atoms and connects to the C_α atom of each amino acid. There are 20 kinds of amino acids found in common eukaryotic organisms.

Proteins can be made of 10 up to $1,000$ amino acids, while an average globular protein is about 300 amino acids long. Each amino acid contains 7–24 atoms, therefore the number of atoms and arrangements in the space can grow very easily beyond any computational power. Since the beginning of protein simulation studies, different algorithms for exploring the conformations have been devised, such as molecular dynamics, local search, Monte Carlo, genetic algorithms, constraint approaches, as well as different geometric representations [121].

In the literature, several geometric models for proteins have been proposed. One choice that influences the quality and the complexity of computational approaches is the *number of points* that describe a single amino acid.

The simplest representation is the one where each amino acid is represented by one point, typically the C_α atom, given its robust geometric property: *the distance between the C_α atoms of two consecutive amino acids is preserved with a low variance (roughly 3.81Å).* Usually, volumetric constraints are enforced over those points, in order to simulate the average occupancy of each amino acid. This representation can be visualized as a chain of beads that can be moved in the space.

More refined representation models store some (or all) the points of the backbone, plus a centroid of mass (CG) that represents the whole side chain that connects to the C_α atom. In these models, each amino acid is described by different C_α–CG distances and CG volumes. The centroid is an approximation of the side-chain flexibility and allows for more refined energetic models, while the number of points to be taken care of is still low. In this dissertation we use a particular case of these simplified models, the "5@" model, described precisely below. This is a particular instance of *coarse-grained* protein models [28, 146]. At the end of the spectrum, each atom in the amino acid is represented by one point. This representation is the most accurate, and at the same time allows for the most accurate energetic considerations. The drawback is that the computational demand for handling backbone and side-chain flexibility increases significantly. In Figure 3.3 we report three representations for the same protein.

In this work we select the intermediate representation for amino acids where the atoms N, C_α, C' of the backbone and the centroid of the side chain (CG) are accounted for. We also include an oxygen (O) atom attached to the C' atom, because this atom together with the C' and N identifies a triangle that is chemically stable along the backbone and it is used for the assembly of amino acids (see below for a complete formalization). The position of the two H atoms in the backbone can be deduced by the position the other atoms and we will not deal with them explicitly. In conclusion, we deal with *5 at*omic elements per amino acid: the 4 atoms $NC_\alpha C'O$ and the centroid CG. We briefly refer to this representation as to the "5@" model. Figure 3.4 illustrates how these atoms are involved in the concatenation of two consecutive amino acids. Inter-atomic distances between consecutive atoms are fixed—due to their chemical bonds; thus, the differences between these structures are identified by the differences between the angles involved. It is common to find substructures of a protein where consecutive amino acids are arranged according to repeated and characteristic patterns. This property is found in almost every protein; we refer to these typical patterns as *secondary structure elements.* The most common examples are α-helices and β-sheets (see Figure 3.3).

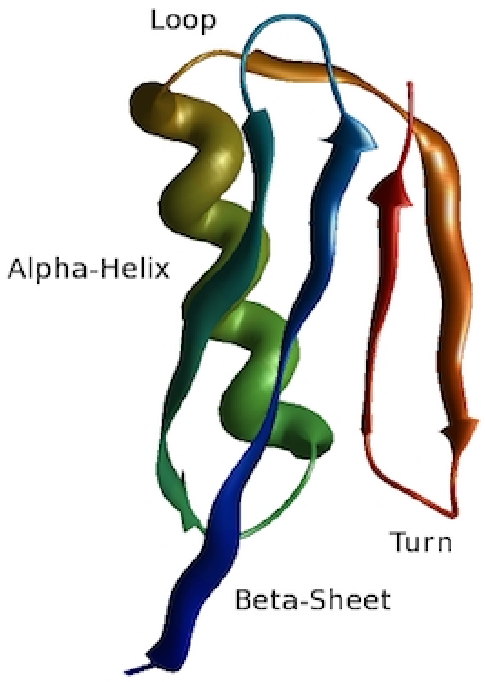


Figure 3.3: Secondary structure elements: α-helix, β-sheets, loops, and turns.

3.1.2 Context Of The Proposed Work

In this chapter we present a tool for assembling and reasoning about amino acids in the space. As in other similar approaches (e.g, [154]), the system relies on a set of admissible elementary shapes (or *fragments*) that represent the spatial dictionary of arrangements for every part of a protein.

Each element of the dictionary is general enough to describe the specific atomic structure of either a single amino acid or a longer sequence (even hundreds of amino acids long). For each amino acid sequence, several alternative arrangements are expected to populate the database, so that to offer various hypothesis about the local shape of the sequence. The protein is partitioned into contiguous fragments that can be arranged according to one of the possible shapes recorded

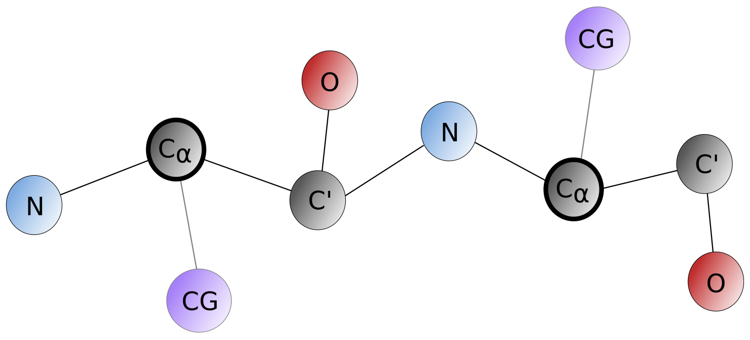


Figure 3.4: Amino acid concatenation in the 5@ model

in the database.

A sequence of amino acids is free to rotate its bonds in the space (typically two degrees of freedom along the backbone and several others along the side chain); however, due to chemical properties and physical occupancy that are specific to the types of amino acids involved and the surrounding environment, some arrangements are impossible and/or unlikely. The core assumption in assembling approaches is to rely on a statistical database of arrangements to describe local and feasible behavior, in order to direct the search to candidates that have high probability and are energetically favorable. The presence of multiple candidate fragments for every part of the protein requires a combinatorial search among the possible choices that, once assembled together, leads to alternative putative configurations for the protein. The search process is in charge of verifying the feasibility of each assembly, since the combination of local arrangements could generate a non-feasible global shape, e.g., one that leads to a spatial clash between atoms from different fragments. If one (or more) fragment is described by one single arrangement, that part of the protein is rigidly imposed. This particular degenerate case can be exploited to describe rigid parts of the protein. A specific combination of fragment length and number of instances for each fragment determines the type of protein problem being modeled. We can range from complete backbone flexibility (fragments made of hundreds of choices for each amino acids) to secondary structure - loop models (interleaving of longer fragments modeling helices/β-strands and shorter fragments).

The library of fragments is usually derived from the content of the Protein Data Bank (PDB, `www.pdb.org`) that contains more than 96,000 protein structures. The design adopted in our study is parametric on the choice of the library of fragments to use. For example, our experiments use a library of fragments derived from a subset of the PDB known as `top-500` [105], which contains non-redundant proteins and preserves statistical relevance. Alternative libraries of fragments can be obtained through the use of sophisticated protein database search algorithms, such as FREAD [26]. We retrieve information depending on the specific amino acid sequence, since local properties greatly influence the typical arrangements observed. Moreover, we build libraries for different sequences lengths h, even if for longer sequences the statistical coverage becomes weak. Nevertheless, [114] conjectured that a relatively small set of fragment shapes (a few dozens) of length 5, is able to describe virtually any protein. [73] demonstrates how the size and the structure of the search space is affected by the choice of the fragment length and how this can be used to optimize the search process. Similar considerations have been explored by others [76]. Recent work show how to efficiently build such dictionaries [54]. These models can be easily accommodated into our framework.

Each considered sequence is associated to several configurations of 5@ models, placed according to a standardized coordinate system. In this activity, we also consider the $C'O$ group of the preceding amino acid and the N atom of the following amino acid. This extra information is needed for fragments combination, assuming that the fragment will be connected by two peptidic bonds. Therefore, for a specific sequence, we store all the occurrences of

$$C'O\ \underbrace{NC_{\alpha}C'O}_{h \text{ times}}\ N$$

and relative positions. In order to reduce the impact of the specific properties of the database used, we cluster this set in such a way that if two fragments have a *Root Mean Square Deviation* (*RMSD*) value less than a given threshold, just one of them is stored. For example, for length $h = 1$ and a RMSD threshold of .2Å, we can derive a fragment database of roughly 90 fragments per amino acid. The RMSD captures the overall similarity in space of corresponding atoms, by performing an optimal roto-translation to best overlap the two structures.

The CG information is added later using statistical considerations about side-chain mobility, that are not accounted for during the clustering described above [55].

3.1.3 Protein Structure Prediction

In the protein structure prediction problem, the sequence of amino acids composing a protein (known as the *primary structure*) is given as input; the task is to predict the three dimensional (3D) shape (known as the *native conformation* or *tertiary structure*) of the protein under standard conditions.

The common assumption, based on Anfinsen's work [5], is that the 3D structure which minimizes some given energy function modeling the atomic force fields, is the candidate that best approximates the functional state of a protein. In such setting, the choice of the number of atoms used to represent each amino acid controls the quality and the computational complexity.

Moreover, the spatial domains where the protein's "points" (e.g., atoms, centroids) can be placed have an impact on the type of algorithms and search that can be performed. The domain can be either *continuous*, often represented by floating point coordinates, or *discrete*, often derived from a discretization of the space based on a crystal lattice structure.

Once the geometric model has been determined, it is necessary to introduce an energy function, mostly based on the atoms considered and their distances. In the structure prediction problem, the energy function is used to assign a score to each geometrically feasible candidate; the candidate with the optimal score represents the solution of the prediction problem.

Let us briefly review some popular approaches to this problem, with a particular emphasis on solutions that rely on constraint programming technology.

The natural approach of investigating protein conformations through simulations of physical movements of atoms and molecules is, unfortunately, beyond the current computational capabilities [84, 13, 90]. This has originated a variety of alternative approaches, many based on *comparative modeling*—i.e., small structures from related protein family members are used as templates to model the global structure of the protein of interest [86, 57, 154, 101, 87]. In these methods, often referred to as *fragments assembly*, a protein structure is assembled using small protein subunits as templates that present relevant sequence similarities (*homologous affinity*) w.r.t. the target sequence.

In the literature, *Constraint Programming (CP)* techniques have shown their potential: the structural variability of a protein can be modeled as constraints, and *constraint solving* is performed in order to infer the optimal structure [8, 11, 35, 42]. CP has been used to provide approximated solutions for *ab-initio* lattice-based modeling of protein structures, by using local search and large neighboring search [150, 47]; exact resolution of the problem on lattice spaces using CP, along with with clever symmetry breaking techniques, has also been investigated [8]. These approaches solve a constraint optimization problem based on a simple energy function (HP). A more precise energy function has been used by [35, 39], where information on secondary structures (i.e., α-helices, β-sheets) is also taken into consideration. Due to the approximation errors introduced by lattice discretization, these approaches do not scale to medium-size proteins. Off-lattice models, based on the idea of fragment assembly, and implemented using Constraint Logic Programming over Finite Domains, have been presented [42, 36], and applied not only to structure prediction but also to other structural analysis problems. For instance, [41] use this approach to generate sets of feasible conformations for studies of protein flexibility. The use of CP to analyze NMR data and the related problem of protein docking has also been investigated [11].

In the context of *ab-initio* prediction, a recent work [127] has shown that increasing the complexity of the conformational search space—by using a more refined fragment library—in combination with a sampling strategy, enhances the generation of near-native structure sets. The work of [145] and [117] illustrates various enhancement to the fragment-based assembly process leading to faster computations and an improved sampling of the conformation space—e.g., using tree-based methods inspired from motion planning to guarantee progress towards minimal energy conformations while maintaining geometrically separate conformations. In terms of energy landscape, the native state has generally lower free energy than non-native structures, but it is extremely difficult to locate. Hence, a targeted conformational sampling may aid protein structure prediction in that different near-native structure can be used to guide the search; several schemes based on Monte Carlo movements in sampling conformation space through fragments assembly have been proposed [151, 171, 44]. Methods based on non-uniform probabilistic mass functions (derived from previously generated decoys) have been proposed to aid in this problem [152]. Sampling, however, remains a great challenge for protein with complex topologies and/or large sizes [89, 151].

It is widely accepted that proteins, in their native state, should be considered as dynamic entities instead of steady rigid structures. Indeed, in recent years the research focus has shifted towards prediction schemes that take into account the non-static nature of proteins, supported by recent observations based on magnetic resonance techniques. Processes such as enzyme catalysis, protein transport and antigen recognition rely on the ability of proteins to change conformation according to the required conditions. This dynamic nature can be visualized as a set of different structures that coexist at the same time. The generation of such sets that capture non-redundant structures (in pure geometric terms) is a great challenge [89]. Robotics and inverse kinematics methods have been extensively explored both in sampling proteins' conformational space [179, 32] and for molecular simulations [2, 116, 126, 91].

A motivation for our work is to provide the ability of generating a protein set that contains optimal and sub-optimal candidates, in order to capture dynamic information about the behavior of a protein. A desirable property is that the conformations returned in the pool are sufficiently diverse and uniformly distributed in the 3D space.

3.1.4 Protein Loop Modeling

The protein loop modeling problem is a restricted version of the structure prediction problem. We will use this problem as a working example in the remaining part of the dissertation. In this context, the protein structure is already partially defined, e.g., a large number of atoms are already placed in the space. Usually, this common scenario derives from an X-ray crystallography analysis, where the spatial resolution of atoms degenerates in presence of some regions of the protein that are exposed on the surface and presents an increased instability. Since a crystal contains several copies of a protein in order to perform the measurement, such regions appear as more fuzzy, and therefore the placement of atoms in these regions may be ambiguous. Usually, these regions, referred to as *loops*, are not involved in secondary structures, which are instead more stable. When dealing with homology modeling, the same protein found in another organism, typically shows some variations in the sequence due to evolution, especially in the loop regions, since they are less essential for protein stability and functionality. Hence, starting from an homologous protein structure, usually loops need to be recomputed with a specialized loop modeling approach and the use of minimization techniques.

The length of a loop is typically in the range of 2 to 20 amino acids; nevertheless, compared to secondary structures, the flexibility of loops produces very large, physically consistent, conformation search spaces. Constraints on the mutual positions and orientations (dihedral angles) of the loop atoms can be deduced and used to simplify the search. Such restrictions are defined as the *loop closure* constraints. In Figure 3.3, we have a (simple) possible scenario where two macro-structures (two helices) are connected by a loop. In this setting, we can assume to know the position of the two helices, while the loop atoms are to be determined.

A procedure for protein loop modeling typically consists of 3 phases: *sampling*, *filtering*, and *ranking* [83]. Sampling is commonly based on a loop candidate generation, using dihedral angles sampled from structural databases [51], and subsequent candidate modification in order to satisfy the loop closure constraints. These conformations are checked w.r.t. the loop constraints and the geometries from the rest of the structure, and the loops that are detected as physically infeasible, e.g., causing steric clashes, are discarded by a filtering procedure.

Popular methods used for loop modeling include the *Cyclic Coordinate Descent (CCD)* method [21], the algorithms based on inverse kinematics [92, 147], the *Self-Organizing (SOS)* algorithm [104], which can simultaneously satisfy loop closure and steric clash restrictions by iteratively superimposing small fragments (amide and C_α) and adjusting distances between atoms, and the *Wriggling* method [18], that employs suitably designed Monte Carlo local moves to satisfy the loop closure constraints. Multi-method approaches have also been proposed—e.g., [102] proposes a loop sampling method which combines fragment assembly and analytical loop closure, based on a set of torsion angles satisfying the imposed constraints. *Ab initio* methods [132, 53, 81, 157, 43, 51, 170] and methods based on templates extracted from structural databases [26] have been explored.

Finally, a ranking step—e.g., based on statistical potential energy, like in DOPE [148], DFIRE [182], or the one proposed in [56]—is used to select the best loop candidates.

The sampling and filtering procedures should work together and direct the search towards structurally diverse and admissible loop conformations, in order to maximize the probability of including a candidate close to the native one and to reduce the time needed to analyze the candidates. Our work is motivated by the need of controlling the properties of the resulting set of candidates. In particular, we model structural diversity both in distance and orientation of the backbone and make the sampling phase guided by the loop constraints.

Fragment-based assembly methods have also been investigated in the context of loop modeling [102, 180]. [147] review in great detail loop modeling techniques.

3.2 Constraint Solving with 3D Fragments

In this Section, we introduce the formalization of an effective solution to tackle practical applications concerning with the placement of 3D fragments. Such applications are described as combinatorial problems, modeled as a set of *variables*, representing the entities the problem deals with, and a set of *constraints*, representing the relationships among the entities. In the context of a constraint programming system, variables and constraints are adopted to provide a *solution* for the CSP, that is, an assignment to the variables that satisfies all the constraints. We extend this concept by enabling the constraint solver to find a *representative solution* for the CSP that satisfies some additional properties expressed among the variables of the whole solution set.

3.2.1 Some Terminology

A *fragment* B is composed of an ordered list of at least three (distinct) 3D points, denoted by $\mathsf{points}(B)$. The number of points of a fragment is referred to as its *length.* The front- and end-anchors of a fragment B, denoted by $\mathsf{front}(B)$ and $\mathsf{end}(B)$, are the two lists containing the first three and the last three points of $\mathsf{points}(B)$ (we consider three 3D points in order to uniquely identify a plane in the space). With $B(i)$ we denote the i-th point of the fragment B. For two ordered lists of points $\vec{p}$ and $\vec{q}$, we write $\vec{p} \frown \vec{q}$ if they can be perfectly overlapped by a rigid coordinate translation and/or rotation (briefly, a *roto-translation*)—see Figure 3.5 (let us assume the z coordinate is 0 for all points and omitted for simplicity). In 3.5 (left) there are two fragments B_1 (light grey) and B_2 (dark grey) such that $\mathsf{points}(B_1) = ((0,0),(1,0),(1,1),(2,1))$ and $\mathsf{points}(B_2) = ((4,0),(3,0),(3,1),(4,1),(4,2))$. The arrows address their initial points. In 3.5 (right) an overlap of the two fragments is obtained by rotating B_2 of 90 degrees and then translating it by -3 units on the x-axis, the last three points of B_1 ($\mathsf{last}(B_1)$) and the first three points of B_2 ($\mathsf{first}(B_2)$). Thus, $\mathsf{end}(B_1) \frown \mathsf{front}(B_2)$.

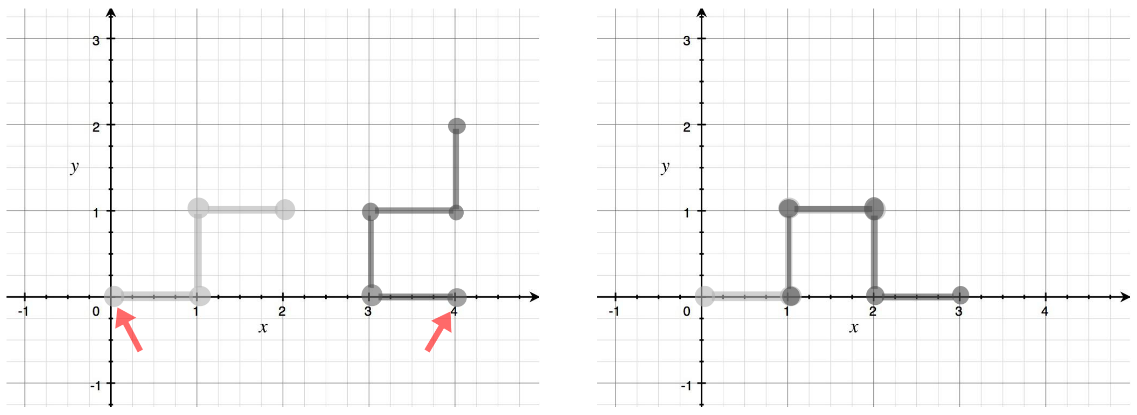


Figure 3.5: Overlap of two fragments in the plane.

A non-empty set of fragments with the same length is called a *body*. A body can be used to model a set of possible shapes for a sequence of points. We say that a body has *length* k if each fragment it contains has length k.

A *multi-body* is a sequence $S_1, \ldots, S_n$ of bodies.

Given a multibody $\vec{S} = S_1, \ldots, S_n$, a *rigid body* from $\vec{S}$ is a sequence of fragments $B_1, \ldots, B_n$, where $B_i \in S_i$ for $i = 1, \ldots, n$ and $\mathsf{end}(B_i) \frown \mathsf{front}(B_{i+1})$, for all $i = 1, \ldots, n-1$. A rigid body is uniquely identified by the sequence $B_1, \ldots, B_n$; however, when consecutive fragments are overlapped, the rigid body can be alternatively identified by a list of points that form a spatial shape. In Figure 3.6 we report examples of bodies, multi-bodies, and rigid bodies. As in the previous example, we assume that the z coordinate is 0 for all points. From left to right, top to bottom of Figure 3.6: the body S_1 composed by an unique fragment, and the bodies S_2 and S_3 composed by two fragments each. Arrows address the initial points of fragments. All the three bodies have length 4. $\vec{S} = S_1, S_2, S_3$ constitutes a multi-body. In the bottom-rightmost figure we report the spatial shapes associated to the four rigid bodies that can be obtained from the multi-body $\vec{S}$. One of them is identified by full lines, the other three by dashed lines. Observe that the rigid body identified by $((0,0),(1,0),(1,1),(2,1),(2,0),(3,0))$ can be obtained by a rotation of 180 degrees of the fragment $((2,0),(3,0),(3,1),(4,1))$ of S_2 on the x axis (flipping) and by a translation of -1 units on x and of $+1$ units on y. Observe moreover that the rigid body identified by $((0,0),(1,0),(1,1),(2,1),(2,0),(1,0))$ contains the same point $(1,0)$ twice.

Example 3.2.1 (Working Example) *These concepts are related to the loop-modeling problem. Points are atoms. A fragment is a spatial shape of some atoms. If the last three atoms of one fragment overlap with the first three atoms of another fragment, we can join them. A body is a set of admissible shapes for a given list of atoms. A multi-body $S_1, \ldots, S_n$ is a sequence of these elements, corresponding to a sequence of atoms (of amino acids). The idea is that the last three atoms of a body S_i are the same as the first three of the successive body S_{i+1}. A rigid body is a possible complete shape of those atoms, provided the last three atoms of the fragment selected in the set S_i overlap with the first three atoms of the fragment selected in S_{i+1}.*

The overlapping points $\mathsf{end}(B_i)$ and $\mathsf{front}(B_{i+1})$ constitute the i-th *joint* of the rigid body. The number of rigid bodies that can be obtained from a single multi-body $S_1, \ldots, S_n$ is bounded by $\Pi_{i=1}^{n} |S_i|$. Figure 3.7 provides a schematic general representation of a rigid body.

A rigid body is defined by the overlap of joints, and relies on a chain of relative roto-translations of its fragments. Each points in $\mathsf{points}(B_i)$ is therefore positioned according to the (homogeneous) coordinate system associated to a fragment B_{i-1}. Note that once the reference system for B_1 is defined, the whole rigid body is completely positioned. With the exception of the case where all points of a joint are collinear. Points $p_1, \ldots, p_n$, with $n \geq 3$ are *collinear* if the points $p_3, p_4, \ldots, p_n$

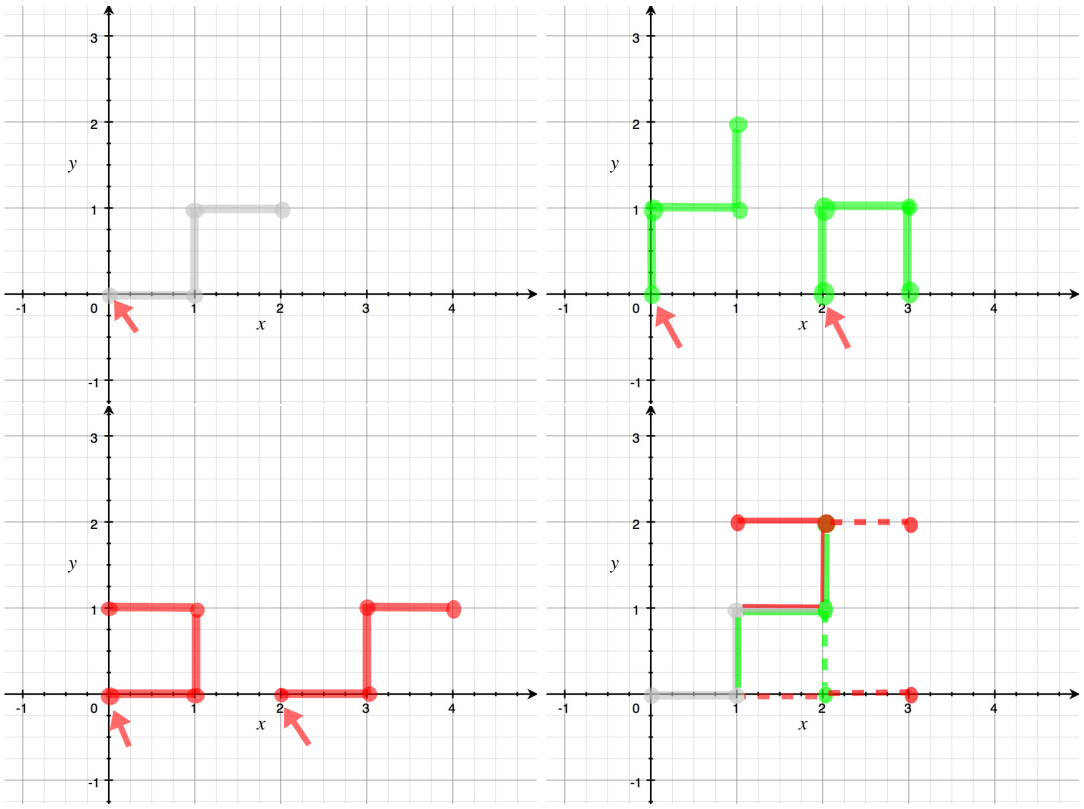


Figure 3.6: Examples of bodies, multi-bodies, and rigid bodies.

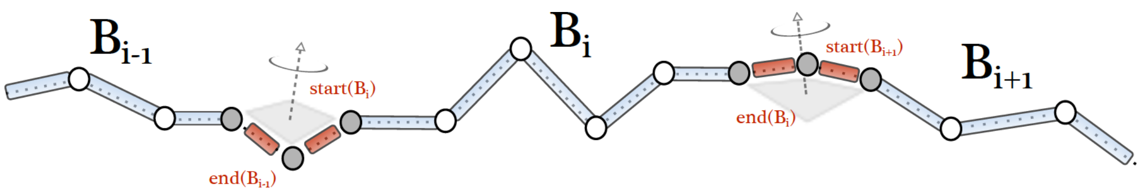


Figure 3.7: A schematic representation of a rigid body. The joints connecting two adjacent fragments are emphasized. The points in $\mathsf{points}(B)$ of each fragment are represented by circles.

belongs to the straight line containing the two points p_1 and p_2. The relative positions of two consecutive fragments B_{i-1} and B_i of a rigid body ($2 \leq i \leq n$) can be defined by a transformation matrix $T_i \in \mathbb{R}^{4\times4}$. Each matrix depends on the standard Denavit-Hartenberg parameters [75] obtained from the start and end of the fragments—the reader is referred to the work of [97] for details. We denote the product $T_1 \cdot T_2 \cdot \ldots \cdot T_i \cdot (x, y, z, 1)^T$ by $\mathcal{T}_i(x, y, z)$.

Let us analyze the first matrix T_1. The fragment B_1 can be forced to start in a given point and oriented in a given way; in this case the matrix T_1 defines the roto-translation of B_1 fulfilling these constraints. In the absence of such constraints, we assume that B_1 is *normalized* by T_1—i.e., its first point is $(0, 0, 0)$, the second point is aligned along the z axis and the third belongs to the plane formed by the x and z axes. This orientation is referred as the reference system Γ_0.

For $i = 1, \ldots, n$, the coordinate system conversion (x', y', z'), for a point $(x, y, z) \in \mathsf{points}(B_i)$ into the coordinate system of B_1, is obtained by:

$$(x', y', z', 1)^T = T_1 \cdot T_2 \cdot \ldots \cdot T_i \cdot (x, y, z, 1)^T = \mathcal{T}_i(x, y, z) \tag{3.1}$$

Homogeneous transformations are such that the last value of a tuple is always 1.

In the rest of the chapter, we focus on the 5@ model; however the proposed formalization and methods can be used also for other models, e.g., the C_α–C_α model. In the latter case, $\mathsf{points}(B_i)$ contains at least 3 amino acids, and the joints are guaranteed to be non-colinear, due to the chemical properties of the backbone. When combining C_α fragments, the specific rotational angles of the full-atom backbone are lost and a more imprecise multi-body assembly is produced.

A *fragment* is a body associated to a sequence of amino acids. A fragment for a sequence of $h \geq 1$ amino acids is described by a body of length $4h+3$, modeling the concatenation of the atoms represented by the regular expression: $C'O(NC_\alpha C'O)^h N$. In such representation the first and last sequence of $C'ON$ atoms coincide with the *front-* and *end-anchor*, respectively, and are employed during the process of assembling consecutive fragments (i.e., they are used in the roto-translation).

A discretized $\mathbb{R}^3$ space can be represented as a regular lattice, composed of cubic cells with side length equal to a given parameter k. Each cell is referred to as a *3D voxel* (or, simply, *voxel*); we assume that each voxel receives a unique identifier. We denote with $\texttt{voxel}(p, k)$ the identifier of the voxel that contains the 3D point p in the context of a discretization of the space using cubes with side length equal to k. This spatial quantization allows an efficient treatment of the approximated propagation required by some of the geometric constraints introduced in the following sections.

3.2.2 Variables And Domains

Let us now define the variables adopted to describe the entities of a problem with fragments. The *domain* of a variable V is the set of allowable values for V, and it will be denoted by D^V. To deal with fragments placements in the 3D space we adopt two distinct types of variables:

Finite Domain Variables (FDVs): The domain of a finite domain variable is a finite set of non negative integer numbers.

Point Variables (PVs): These variables will assume the coordinates of a 3D point in $\mathbb{R}^3$. Their domains are, initially, 3D boxes identified by two opposite vertices $\langle \min, \max \rangle$, as done in the discrete solver COLA [38, 39].

Example 3.2.2 (Working Example) *Following Example 3.2.1, FDVs are the identifiers of the various fragments in a body, while PVs are used to represent the 3D coordinates assigned to the various structural points (e.g., atoms, centroids) of interest for each molecule being considered. Clearly, the values of PVs will depend deterministically on the values of FDVs (and vice-versa).*

A variable is *assigned* if its domain contains a unique value; in the case of point variables, this happens if $\mathsf{D}^V = \langle \min, \max \rangle$ and $\min = \max$.

3.2.3 Constraints

In this section, we formalize the constraints that define the fragments placement, that can be used to describe Protein Structure problems in the context of fragment assembly.

Distance Constraints.

Distance constraints model spatial properties of point variables operating in the 3D space. Point variables P and Q can be related by a distance constraint of the form

$$\|P - Q\| \ \ op \ \ d \tag{3.2}$$

where $\|\cdot\|$ is the Euclidean norm, $d \in \mathbb{R}^+$ and op is $\leq$ or $\geq$.

The global constraint `alldistant` associates a minimal radius d_i to each point variable P_i $(i = 1, \ldots, n)$ and ensures that spheres surrounding each pair of point variables do not intersect:

$$\texttt{alldistant}(P_1, \ldots, P_n, d_1, \ldots, d_n), \tag{3.3}$$

This constraint is equivalent to the constraints $\|P_i - P_j\| \geq d_i + d_j$ for all $i, j \in \{1, \ldots, n\}, i < j$. It is used to avoid steric clashes among different atoms (and centroids), which have different volumes. Checking consistency of the `alldistant` constraint (given the domains of the variables P_i) is NP-complete [40]—the proof is based on an encoding of the bin-packing problem using the `alldistant` constraint, and holds true even in this particular setting, where the point variables have *intervals* of $\mathbb{R}^3$ as domains.

The `alldistant` constraint is introduced to avoid clashes when a rigid body is obtained from the multi-body $S_1, \ldots, S_n$. The distance constraints are useful when some extra information is known (e.g., one might have inferred by biological arguments that a pair of amino acid should stay within a certain distance).

Fragment Constraint.

Fragment constraints relate finite domain variables and point variables. Let us assume we have a database F of fragments, where $F[i]$ represents the i-th fragment in the database. Thus, given an FDV variable V, $F[V]$ denotes the fragment indexed by V when V is instantiated. The fragments are stored in F as an ordered list of 3D points.

Given a list of point variables $\vec{P}$, the constraint:

$$\texttt{fragment}(V, \vec{P}, F) \tag{3.4}$$

states that there exists a roto-translation `Rot` such that $\vec{P} = \texttt{Rot} \cdot F[V]$—namely, if $V = i$ then the list of points $\vec{P}$ should take the form of the fragment $F[i]$. For simplicity, we will omit the database F when clear from the context. Intuitively, these constraints ensure that any fragment choice will reproduce the correct shape for the associated 3D point, regardless of the space orientation of the fragment. The orientation is determined by the joined multi-body constraint presented in a following section.

Centroid Constraint.

The centroid constraint enforces a relation among four PVs. Intuitively, the first three of them are associated to the atoms N, C_α, C' of an amino acid and the fourth is related to the centroid CG. The constraint is parametric w.r.t. the type a of an amino acid and deterministically establishes the position of CG depending on the position of the other points:

$$\texttt{centroid}(P_N, P_{C_\alpha}, P_{C'}, P_{\mathsf{CG}}, a) \tag{3.5}$$

In Figure 3.8 the centroids are displayed along the backbone as purple circles and labeled "CG." This constraint can be used when the database of fragment contains only full backbone information.

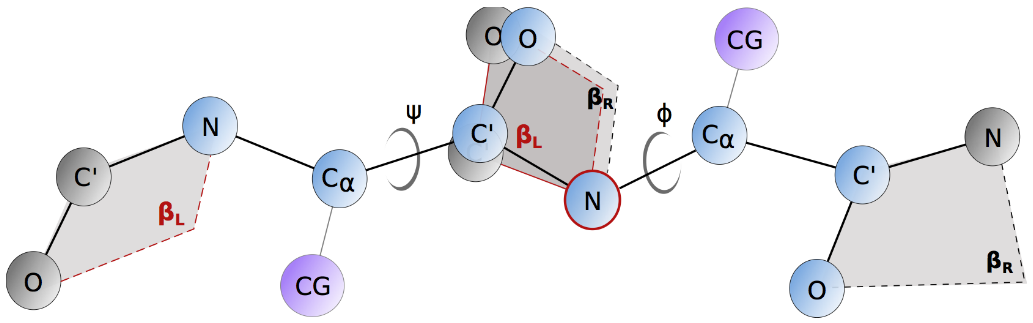


Figure 3.8: Fragments are assembled by overlapping the plane β_R, described by the rightmost C', O, N atoms of the first fragment (left), with the plane β_L, described by the leftmost C', O, N atoms of the second fragment (right), on the common nitrogen atom

The centroid information is used in place of the missing full-atom side chain. The side-chain centroid is computed by taking into account the average C_α-side-chain center of mass distance, the average bend angle formed by the side-chain center-of-mass-C_α-C', and the torsional angle formed by the N-C_α-C'-side-center of mass [55]. This abstraction allows us to reduce the number of fragments to consider, removing fragments that would geometrically conflict with the position of the CG. Consider that a single side chain may have up to 100 main configurations (rotamers).

Table Constraint.

This constraint is used to restrict the assignments of a set of FDVs (representing fragments) to specific tuples of choices. This is useful when modeling a specific local and collaborative behavior that involves more than one fragment; for example, this happens when modeling a secondary structure multiple arrangements of underlying amino acids.

Let F be a set of k-tuples of integer values and $\vec{V}$ a k-tuple of FDVs. A table (or combinatorial) constraint, of the form

$$\texttt{table}(\vec{V}, F) \tag{3.6}$$

requires that the list of variables $\vec{V}$ assumes values restricted to the tuples listed in F, i.e., there exists $t \in F$ such that $\vec{V}[i] = t[i]$, with i in $0, \dots, k-1$.

Example 3.2.3 (Working Example) *Going back to the loop-modeling problem, the role of the* `fragment` *constraint is evident: it relates the (IDs of the) selected fragments of a multi-body with the 3D positions of the various atoms involved. The* `centroid` *constraint is instead introduced to add the position of the centroid that represents the side chain in the 5@ representation.* `table` *constraint is a common constraint in constraint languages and it is useful when some info on consecutive fragments in a rigid body is known due to external knowledge.*

Joined Multibody Constraint.

The Joined Multibody (JM) constraint enforces a relation over a list of FDVs encoding a multibody. It limits the spatial domains of the various fragments composing the multibody in order to retain those fragments that assemble properly and that do not compenetrate. The *joined-multibody (JM) constraint* is described by a tuple: $J = \langle \vec{S}, \vec{V}, \vec{\mathcal{A}}, \vec{\mathcal{E}}, \delta \rangle$, where:

- $\vec{S} = S_1, \dots, S_n$ is a multi-body. Let $\mathcal{B} = \{B_1, \dots, B_k\}$ be the set of all fragments in $\vec{S}$, i.e., $\mathcal{B} = \bigcup_{i=1}^{n} S_i$.
- $\vec{V} = V_1, \dots, V_n$ is a list of FDVs, with domains $D^{V_i} = \{j : B_j \in S_i\}$.
- $\vec{\mathcal{A}} = \mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$, and $\vec{\mathcal{E}} = \mathcal{E}_1, \dots, \mathcal{E}_{3n}$ are lists of sets of 3D points such that:
  - $\mathcal{A}_1 \times \mathcal{A}_2 \times \mathcal{A}_3$ is the set of admissible points for $\mathsf{front}(B)$, with $B \in S_1$;

  - ∘ $\mathcal{E}_{3i-2} \times \mathcal{E}_{3i-1} \times \mathcal{E}_{3i}$ is the set of admissible points for $\mathsf{end}(B)$, with $B \in S_i$, $i = 1, \dots, n$;
- δ is a constant, used to express a minimal distance constraint between different point.

A *solution* for the JM constraint J is an assignment $\sigma : \vec{V} \mapsto \{1, \dots, |\mathcal{B}|\}$ s.t. there exist matrices $T_1, \dots, T_n$ (used in $\mathcal{T}$) with the following properties:

Domain: For all $i = 1, \dots, n$, $\sigma(V_i) \in \mathsf{D}^{V_i}$.

Joint: For all $i = 1, \dots, n-1$, let $(a^1, a^2, a^3) = \mathsf{end}(B_{\sigma(V_i)})$ and $(b^1, b^2, b^3) = \mathsf{front}(B_{\sigma(V_{i+1})})$, then it holds that (for $j = 1, 2, 3$):

$$\mathcal{T}_i(a^j_x, a^j_y, a^j_z) = \mathcal{T}_{i+1}(b^j_x, b^j_y, b^j_z)$$

Spatial Domain: Let $(a^1, a^2, a^3) = \mathsf{front}(B_{\sigma(V_1)})$, then $T_1 \cdot a^j \in \mathcal{A}_j \times \{1\}$ (the product $\times\{1\}$ is necessary as we use homogeneous coordinates). For all $i = 1, \dots, n$, let $(e^1, e^2, e^3) = \mathsf{end}(B_{\sigma(V_i)})$ then

$$\mathcal{T}_i(e^j_x, e^j_y, e^j_z) \in \mathcal{E}_{3(i-1)+j} \times \{1\}$$

where $1 \leq j \leq 3$ and $T_2, \dots, T_i$ (in $\mathcal{T}_i$) are the matrices that overlap $\mathsf{end}(B_{\sigma(V_{i-1})})$ and $\mathsf{front}(B_{\sigma(V_i)})$

Minimal Distance: For all $j, \ell = 1, \dots, n$, $j < \ell$, and for all points $a \in \mathsf{points}(B_{\sigma(V_j)})$ and $b \in \mathsf{points}(B_{\sigma(V_\ell)})$, it holds that:

$$\|\mathcal{T}_j(a_x, a_y, a_z) - \mathcal{T}_\ell(b_x, b_y, b_z)\| \geq \delta$$

Let us observe that this is a weak form of the `alldistant` constraint where different distances for each point are allowed. It is, in a sense, closer to the `alldifferent` constraint.

It has been proved that establishing *consistency*—i.e., existence of a solution—of JM constraints is NP-complete [20]. We have also proved that it remains NP-complete even assuming that all the fragments of the problem have the same three atoms with the same spatial position, and that the same holds for the last three atoms (of course fragments are allowed to contain more than three atoms otherwise the problem is trivial). The proof is reported in Appendix A.

Example 3.2.4 (Working Example) *The JM constraint contains exactly all the ingredients needed for modeling a loop problem. We have a multi-body $\vec{S}$, and the corresponding FDs $\vec{V}$, we have a set of possible 3D points where the loop starts $\vec{\mathcal{A}}$ and a set of possible 3D points where the loop ends $\vec{\mathcal{E}}$ and a weak version of the alldistant constraint between pair of atoms that avoid clashes, the solutions are the (non clashing) rigid bodies that starts in $\vec{\mathcal{A}}$ and ends in $\vec{\mathcal{E}}$.*

Let us observe that the JM constraint does not explicitly forbid spatial positions to PVs variables (save for the first three and the last three points of the loop). However, these additional constraints can be explicitly required during domain definition of the PVs variables used for the encoding.

Remark 3.2.5 *The choice of using three points of overlap resembles the method proposed by [92]. On the other hand, we should observe that it is only a technical exercise to modify the JM constraints and so that they allow a parametric overlap between contiguous fragments.*

3.3 The FIASCO Constraint Solver

We present the overall structure and implementation of a hybrid constraint solver capable of handling the classes of constraints described in the previous section.

3.3.1 Constraint Solving

A distinctive feature of FIASCO is the possibility to handle continuous domains at the cost of keeping a discrete library of choices (finite domain variables). The handling of fragments allows us to reason about spatial properties in a more efficient and descriptive way than the pure 3D domain modeling adopted in previous proposals. Moreover, FIASCO allows the solver to *uniformly* sample the search space by means of a spatial equivalence relation that is used to control the tradeoff between accuracy and efficiency. This is particularly effective when the finite domains are heavily populated, and is a critical component to model real-world problems.

The constraint solver builds on the classical prop-labeling tree exploration where constraint propagation phases are interleaved with non-deterministic branching phases used to explore different value assignments to variables [6]. The solver is able to handle both point variables and finite domain variables—this is the reason why we refer to it as an *hybrid* solver. In particular, the assignments to finite domain variables guide the search; their values imply assignments of the point variables, that in turn may propagate and reduce the domains of both point variables and finite domain variables. Moreover, the "propagation" technique implemented for the JM constraint is not a classical filtering technique—it is an approximated technique that we describe later.

The presence of point variables allows, in principle, an infinite number of domain values in $\mathbb{R}^3$. However, we noted that the information carried by assembling fragments (encoded by finite domain variables) is much more informative than any complex and demanding model for 3D continuous space (e.g., Oct-trees, CSG, no-goods). In particular, the direct kinematics encoded by a JM constraint is able to efficiently identify a set of admissible regions of a point variable in a fast, approximated, and controlled way. Therefore, the point variables can be seen as an internal aid to propagation. These variables are updated during the JM propagation phase and can interact with the JM propagator to prune the corresponding fragment variables. Distance constraints on point variables are included in a standard AC3 propagation loop for domains updates.

The other aspect that extends the classical solver structure is the capability of controlling the amount of the search tree to be explored. The search tree contains a large number of branches that are very similar, from the point of view of the geometric distance between corresponding point variables. The goal is to produce a subset of feasible solutions that exhibit significant 3D differences between themselves.This is accomplished by introducing the possibility to explore a subtree of a given depth, by enumerating a specific and limited number of branches, rather than following the standard recursion of propagation and expansion. To achieve this behavior, it is necessary to selectively interfere with the standard recursive call to the solver, and implement a non-deterministic assignment of partial tuples of finite domain variables. This resembles the implementation of a *table* constraint, which is dynamically created during the search. This strategy allows us to significantly reduce the number of branches explored in the subtree, and produces significant results when the selection of the branches is controlled by an adequate partitioning function. In this work, we propose an effective partitioning function based on a measure of 3D similarity for point variables; this is used to direct the search along specific branches of controlled depth that are adequately "separated" by the partitioning function. This is practically realized by introducing a form of look-ahead, controlled by the JM propagator, that returns a set of partial assignments as well as the filtered domains for the finite domain variables.

The general structure of the solver is highlighted in Algorithm 2. The solver is designed to process a list $\vec{V} = v_1, \ldots, v_n$ of finite domains variables, together with the domains $D_{v_1}, \ldots, D_{v_n}$ for them. Intuitively, each domain is a set of indices for the set of fragments. Moreover, the solver receives a list $\vec{P} = p_1, \ldots, p_{5n}$ of $5*n$ point variables, where the variables $p_{4*i}, \ldots, p_{4*i+4}$ are related to the fragment in the domain D_{v_i}. Each point variable p_j has, in turn, a spatial domain D_{p_j}. $\mathcal{C}$ represents the constraints between elements of $\vec{V}$ and $\vec{P}$. Finally, the solver receives also as input the "current" level ℓ in the exploration of the search tree (set to 1 the first time the procedure is called). For the sake of simplicity, the choice of variables to be assigned is based on their ordering in the input list (more sophisticated selection strategies can be easily introduced). When we enter the level ℓ, we assume that the variables $v_1, \ldots, v_{\ell-1}$ have already been assigned.

Let us briefly describe the algorithm. If all the variables in $\vec{V}$ have already been assigned (lines 1–4), then the search algorithm terminates and returns the computed solution, represented by the values assigned to the variables $\vec{P}$. Otherwise, we non-deterministically select a fragment index in the domain of the variable v_ℓ and assign it to the variable. Lines 6–7 indicate the execution of a standard constraint propagation step (using AC-3). If the propagation step fails, then we assume that another non-deterministic choice is made, if possible. Every reference to a non-deterministic choice in the algorithm corresponds to the creation of a choice-point that will be the

Algorithm 2 **search**$(\vec{V}, \vec{P}, \vec{D}, \mathcal{C}, \ell)$

```
Require: $\vec{V}, \vec{P}, \vec{D}, \mathcal{C}, \ell$
if $\ell > |\vec{V}|$ then
    output $(\vec{P})$
    return
end if
for each fragment index $f$ in $D_{v_\ell}$ do
    if AC-3$(\mathcal{C} \cup \{v_\ell = f\}, \vec{V}, \vec{P}, \vec{D})$ then
        $T^{n \times m} \leftarrow$ get_table_from_JM()
        if $n > 0$ then
            Non-deterministically select $i$ in $1..n$
            for $j = 1..m$ do
                $\mathcal{C} \leftarrow \mathcal{C} \cup \{v_{\ell+j} = T[i][j]\}$
            end for
            search$(\vec{V}, \vec{P}, \vec{D}, \mathcal{C}, \ell + m)$
        else
            search$(\vec{V}, \vec{P}, \vec{D}, \mathcal{C}, \ell + 1)$
        end if
    end if
end for
```

target of backtracking in case of failure (for simplicity, we assume chronological backtracking). If it succeeds, leading to a possible reduction of the domains $\vec{D}$, then the computation will proceed. A table constraint might be produced during the propagation of the JM constraint in the AC-3 procedure (see below for details). If this is the case (lines 8–9), some (m) variables are non-deterministically assigned with the values in the table (lines 9–12), and the search continues with m less variables to be assigned (line 13). If this is not the case, then the search will continue with only one less variable (v_ℓ) to be assigned (line 15).

A peculiar feature of our constraint solver (not reported in the abstract algorithm just defined) is that it can be used to avoid the search of solutions "too similar" to each others. Let us assume that the 3D space is partitioned in cubic voxels of size kÅ. Then, given a list of FDVs $\vec{V}$ and a list of PVs $\vec{P}$, the user can state:

$$\texttt{uniqueseq}(\vec{V}, \vec{P}, k) \tag{3.7}$$

This constraint forces the solver to prune the search tree in the following way. Given a partial assignment σ, let $v \in \vec{V}$ be the variable to be assigned at the next step and $p_1, \ldots, p_h \in \vec{P}$ the PVs to be consequently instantiated. The constraint ensures that for any two assignments σ_1, σ_2 extending σ to $v, p_1, \ldots, p_h$ it holds that there exists at least one $i \in \{1, \ldots, h\}$ such that $\sigma_1(p_i)$ and $\sigma_2(p_i)$ do not belong to the same voxel.

3.3.2 Constraint Propagation

In this section, we discuss the propagation rules associated to the various constraints introduced in Section 8.2.2; these are applied within the call to the AC-3 procedure (Section 2.4)(line 6 of Algorithm 2). The constraint propagation is used to reduce the domain size of the PVs and FDVs, ensuring constraint consistency.

Joined Multibody Constraint.

The JM constraint is a complex constraint that is triggered when the leftmost points involved in the constraint (anchors) are instantiated. The JM propagation (JMf) is based on the analysis of the distribution in the space of the points involved. The goal of the propagation is to reduce the domains of the FDVs through the identification of those fragments that cannot contribute to the generation of a rigid body that is compatible with the corresponding Point Variable domains. This can be viewed as a form of hyper-arc consistency over a set of fragments. Moreover, due to complexity and precision considerations, this propagator is approximated by the use of a spatial equivalence relation ($\sim$), that identifies classes of tuples of fragments; these classes have the property to be spatially "different" from one another.

This allows a compact handling of the combinatorics of the multi-body, while a controlled error threshold allows us to select the precision of the filtering. The equivalence relation captures those rigid bodies that are geometrically similar, allowing the search to "compact" small differences among them.

Algorithm 3 The JMf algorithm.

```
Require: $\vec{S}, \vec{V}, \vec{\mathcal{A}}, \vec{\mathcal{E}}, \delta, \mathcal{G}, \sim$
Ensure: $\mathsf{Tab}$
 1: $n \leftarrow |\vec{V}|; \mathsf{Tab} = \emptyset$
 2: $\mathcal{R}_1 \leftarrow \left\{ (B, T_1) \middle| B \in S_1, \exists T_1 \begin{pmatrix} T_1 \cdot \mathsf{start}(B) \in \mathcal{A}_1 \times \mathcal{A}_2 \times \mathcal{A}_3 \ \wedge \\ T_1 \cdot \mathsf{end}(B) \in \mathcal{E}_1 \times \mathcal{E}_2 \times \mathcal{E}_3 \ \wedge \\ \forall p \in \mathsf{points}(B). \forall q \in \mathcal{G}. \, \|(T_1 \cdot p) - q\| \geq \delta \wedge \\ \forall c \in \mathcal{C} \text{ involving } p.\mathsf{consistent}(c)) \end{pmatrix} \right\}$
 3: $\mathcal{P}_1 \leftarrow \{T_1 \cdot \mathsf{end}(B) \mid (B, T_1) \in \mathcal{R}_1\}$
 4: for each $i = 2, \ldots, n$ do
 5:     $\mathcal{P}_i = \emptyset$; $\mathcal{R}_i = \emptyset$;
 6:     for each $E \in \mathcal{P}_{i-1}/\sim$ do
 7:         $\mathcal{R}_i \leftarrow \mathcal{R}_i \cup \left\{ B \in S_i \middle| \begin{array}{l} T = \rho(E, \mathsf{start}(B)) \ \wedge \ T \neq \mathtt{fail} \ \wedge \\ T \cdot \mathsf{end}(B) \in \mathcal{E}_{3i-2} \times \mathcal{E}_{3i-1} \times \mathcal{E}_{3i} \ \wedge \\ \forall p \in \mathsf{points}(B). \forall q \in \mathcal{G}. \, \|(T \cdot p) - q\| \geq \delta \wedge \\ \forall c \in \mathcal{C} \text{ involving } p.\mathsf{consistent}(c)) \end{array} \right\}$
 8:         $\mathcal{P}_i \leftarrow \{\rho(E, \mathsf{start}(B)) \cdot \mathsf{end}(B) \mid B \in \mathcal{R}_i\}$
 9:     end for
10:     compute $\mathcal{P}_i/\sim$ and filter $\mathcal{R}_i$ accordingly
11: end for
12: for each representative $L$ of $\mathcal{P}_n/\sim$ do
13:     $\mathsf{Tab} = \mathsf{Tab} \cup \eta(L)$
14: end for
```

The JMf algorithm (Alg. 3) receives as input a JM-constraint $\langle \vec{S}, \vec{V}, \vec{\mathcal{A}}, \vec{\mathcal{E}}, \delta \rangle$, along with

- A set $\mathcal{G}$ of points that are not available for the placement of bodies, and
- The equivalence relation $\sim$.

For the sake of readability, we assume that the domain information for variables are available. The algorithm builds a table constraint `table` $(\vec{V}, \mathsf{Tab})$. In this process, the algorithm makes use of a function ρ (lines 7 and 8); this function takes as input two lists $\vec{a}$ and $\vec{b}$ of 3D points, and computes the homogeneous transformation to overlap $\vec{b}$ on $\vec{a}$. A call to this function will fail if $\vec{a} \not\frown \vec{b}$. For simplicity, the fourth component (always 1) of the homogeneous transformation is not explicitly reported in the algorithm.

For $i = 1, \ldots, n = |\vec{V}|$, the algorithm computes the sets $\mathcal{R}_i$ and $\mathcal{P}_i$, that will respectively contain the fragments from S_i that can still lead to a solution, and the corresponding allowed 3D positions of their end-points. For each fragment $B \in \mathcal{R}_{i+1}$ we denote with $\mathsf{parent}(B)$ the set of fragments $B' \in \mathcal{R}_i$ such that $\mathsf{end}(B') \frown \mathsf{front}(B)$ via ρ. For each fragment B, we denote with $\mathsf{label}(B)$ the corresponding FD value associated.

In computing/updating $\mathcal{R}_i$ and $\mathcal{P}_i$, only fragments that have end-anchors contained in the bounds $\mathcal{E}_{3i-2}, \mathcal{E}_{3i-1}, \mathcal{E}_{3i}$ are kept. Fragments that would cause points to collapse—i.e., due to a distance smaller than δ from previously placed points—are filtered out (lines 2 and 7). Moreover, the spatial positions of the points of the first fragment are validated against $\mathcal{A}$ (line 2); finally, we enforce the consistency check of each constraint $c \in \mathcal{C}$ involving points in $\mathsf{points(B)} \in S_i$ to retain only those points that can potentially reach the admissible positions (lines 2 and 7).

The algorithm performs $|\vec{V}| - 1$ iterations (lines 4–11). First $\mathcal{R}_i$ and $\mathcal{P}_i$ are computed on the basis of the sets of end-anchors of the previous level $\mathcal{P}_{i-1}$ and the starting point of a selected fragment B, filtering out those that are not overlapping and those that lead to wrong portions of space (lines 7–8). The filtering based on $\sim$ is applied (line 10). During this step, the set of triples of 3D points $\mathcal{P}_i$ is clustered using $\sim$. A representative of each equivalence class is chosen (within $\mathcal{P}_i$) and the corresponding fragment in $\mathcal{R}_i$ is identified; all the other (non-identified) fragments are filtered out from $\mathcal{R}_i$. Let us also note that the filtering based on clustering is not performed for the initial step $\mathcal{P}_1$, as typically this is already captured by the restrictions imposed by $\mathcal{A}$.

Once the fragments reachable at last iteration are determined and their representatives selected, we populate the Tab with the set of tuples associated to each representative L. The function $\eta(L)$ returns the assignments to $\vec{V}$ that allows us to overlap the last point to L.

The JMf algorithm is parametric w.r.t. the clustering relation and the function selecting the representative; they both express the degree of approximation of the rigid bodies to be built. The proposed clustering relation for loop modeling takes into account two factors: *(a)* The positions of the end-anchors in the 3D space and *(b)* The orientation of the plane formed by the fragment's anchor β_L w.r.t. a fixed reference system Γ_0 adopted by FIASCO (c.f. Figure 3.8). This combination of clusterings allows to capture local geometrical similarities, since both spatial and rotational features are taken into account.

The spatial clustering *(a)* used is the following. Given a set of fragments, three end points $C'ON$ (end anchors) of each cluster are considered, and the centroid of the triangle $C'ON$ is computed. We use three parameters: $k_{min}, k_{max} \in \mathbb{N}$, $k_{min} \leq k_{max}$, and $r \in \mathbb{R}$, $r \geq 0$. We start by selecting a set of k_{min} fragments, pairwise distant at least $2r$. These fragments are selected as representatives of an equivalence class for other fragments that fall within a sphere of radius r centered in the centroid of the representative. This clustering ensures a rather even initial distribution of clusters, however some fragments may not fall within the k_{min} clusters. We allow to create up to $k_{max} - k_{min}$ new clusters, each of them covering a sphere of radius r. Remaining fragments are then assigned to the closest cluster. The employed technique is a variant of the k-means clustering algorithm called *leader clustering algorithm* [17]; it allows a fast implementation and acceptable results.

The orientation clustering *(b)* partitions the fragments according to their relative orientation of planes β_R w.r.t. Γ_0. A plane spatial orientation is described by the Euler angles ϕ, θ, ψ of its frame w.r.t. Γ_0. This algorithm produces a variable number of partitions depending on β. In particular, given a threshold $\beta > 0$ there are $3 \cdot (360/\beta)$ possible partitions describing equal regions on a sphere though the interval $(\phi \pm / \frac{\beta}{2}, \theta \pm / \frac{\beta}{2}, \psi \pm / \frac{\beta}{2})$. Each fragment is allotted to the partition determined by β.

The final cluster is the intersection of the two partitioning algorithms. This defines an equivalence relation $\sim$ depending on k_{min}, k_{max}, r, and β. The representative selection function selects the fragment for each partition according to some preferences (e.g., most frequent fragment, closest to the center, etc.).

Note that for $r = 0$, $\beta = 360$, and k_{max} unbounded, no clustering is performed and this would cause the combinatorial explosion of every possible end-anchor on the whole problem. The spatial error introduced depends on r and β. With $\beta = 360$, the error introduced at each step can be bounded by $2r$ for each dimension. At each iteration the errors are linearly increased, since a new fragment is placed with an initial error gathered from previous iterations, thus resulting in a $2nr$ bound for the last end-anchor. Clearly this bound is very coarse, and on average the experimental results show better performances. Similar considerations can be argued for rotational errors, however the intersection of the two clusterings, provide, in general, a much tighter bound.

Figure 3.9 is a graphical representation of the propagation of a JM constraint over the variables $V_i, \ldots, V_{i+3}$. **(a)** A simultaneous placement of all the elements in the domain of the variable V_{i+1} is simulated, by overlapping each corresponding fragment with the end-anchor of the fragment associated to the element in the domain of V_i. The set of points $\mathcal{P}_{i+1}$ is computed and clustered using the relation $\sim$ (points within the dotted ellipses). For each cluster one fragment representative is hence chosen (highlighted fragments with filled rightmost circle). The collection of representatives

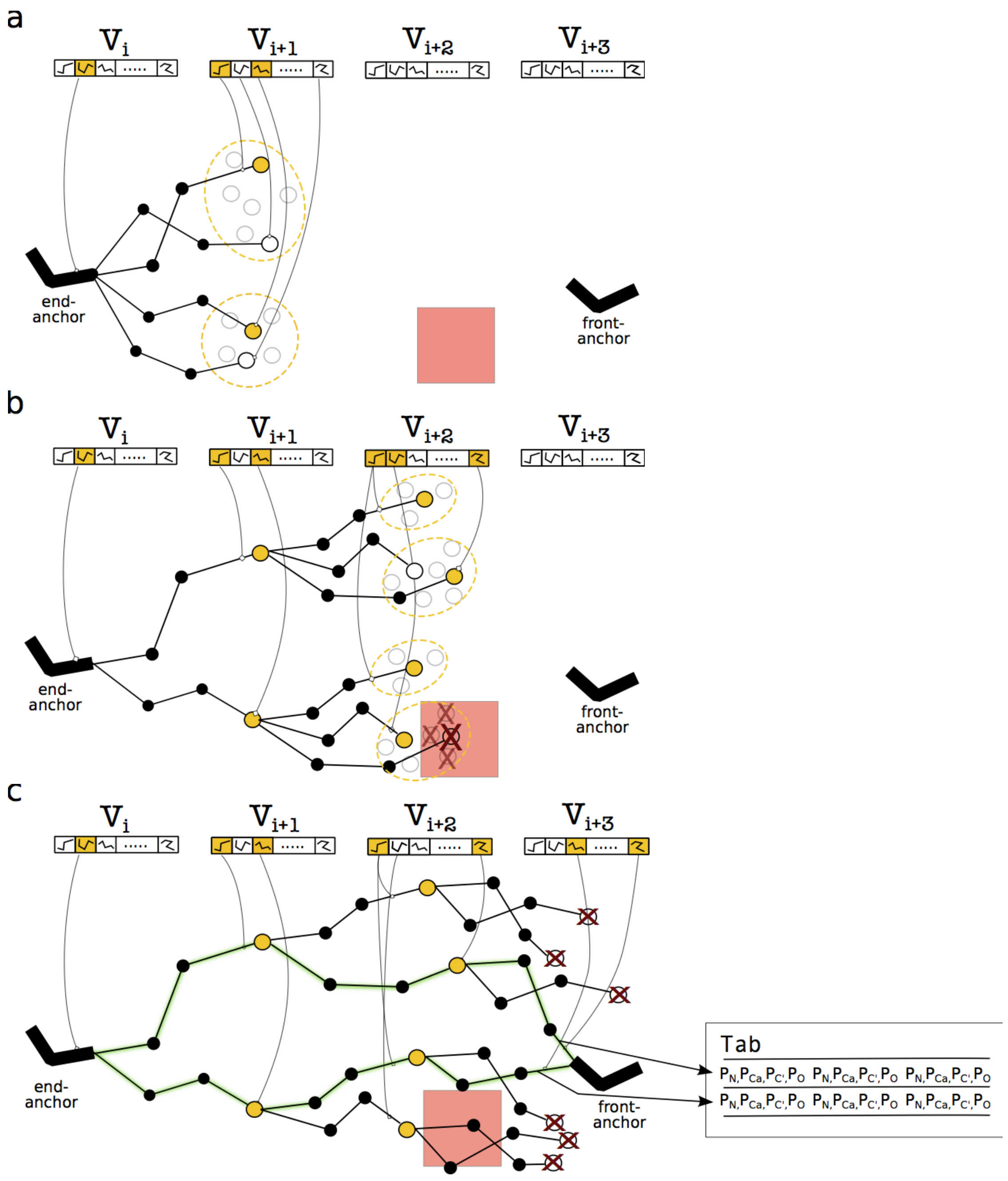

a
b
c
V_i
V_{i+1}
V_{i+2}
V_{i+3}
end-anchor
front-anchor
Tab
$P_N,P_{Ca},P_{C'},P_O$ $P_N,P_{Ca},P_{C'},P_O$ $P_N,P_{Ca},P_{C'},P_O$
$P_N,P_{Ca},P_{C'},P_O$ $P_N,P_{Ca},P_{C'},P_O$ $P_N,P_{Ca},P_{C'},P_O$

Figure 3.9: Graphical representation of the propagation of a JM constraint.

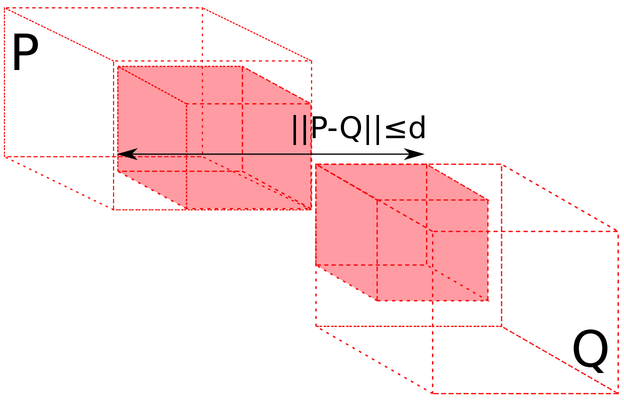


Figure 3.10: The effect of a distance constraint $||P - Q|| \leq d$ propagation. Empty boxes represent the original PVs domains and the full boxes represent the reduced PVs domains after the effect of constraint propagation.

constitutes the set $\mathcal{R}_{i+1}$ **(b)** The previous step is performed again on the basis of the end-anchors related to the fragments representatives chosen in the previous level. The filled box, represents the set of points $\mathcal{G}$ that are not available for the placement of bodies (for instance due to a distance constraint). and the fragment falling in such area are discarded. **(c)** In the last iteration of the JMf algorithm the set of points $\mathcal{P}_{i+3}$ is not clustered, but only those that reach the desired position are retained, for instance the front-anchor associated to the fragment of the next variable, and the sequence of fragments able to lead to such condition (marked by thick lines) are selected to populate the table Tab.

Distance Constraints.

The propagation of the distance constraints is an approximated technique that reduces the size of the box domains. We introduce the following operations over PVs box domains of two variables P and Q that will be used to describe the propagation rule in this and in the following subsections:

$$\begin{array}{rrcl}
\textbf{Domain intersection:} & \mathsf{D}^P \cap \mathsf{D}^Q & = & \langle \max(P_{\min}, Q_{\min}),\ \min(P_{\max}, Q_{\max}) \rangle \\
\textbf{Domain union:} & \mathsf{D}^P \cup \mathsf{D}^Q & = & \langle \min(P_{\min}, Q_{\min}),\ \max(P_{\max}, Q_{\max}) \rangle \\
\textbf{Domain dilatation:} & \mathsf{D}^P + d & = & \langle P_{\min} - d, P_{\min} + d \rangle
\end{array}$$

where $\max(P, Q) = (\max(P_x, Q_x), \max(P_y, Q_y), \max(P_z, Q_z))$, (and similarly for min), and $P+d = (P_x + d, P_y + d, P_z + d)$.

Given two point variables P and Q, with domains D^P and D^Q, respectively, the simplification rule for the constraint $||P - Q|| \leq d$ updates the domains as follows:

$$D^P = ((\mathsf{D}^Q + d) \cap \mathsf{D}^P) \qquad D^Q = ((\mathsf{D}^P + d) \cap \mathsf{D}^Q) \tag{3.8}$$

which ensures that the points in D^P and D^Q are positioned within an approximation of a sphere of radius d. The sphere is approximated by considering the box inscribing it (a cube of side $2d$), as illustrated in Figure 3.10.

The propagation of the constraint $||P - Q|| \geq d$ is harder as the coarse representation of the box domains adopted in this work to model PVs does not allow the description of more complex polyhedron. We hence apply a simple form of bound consistency described by the following rule:

$$||P - Q|| \geq d : \frac{\{(\mathsf{D}^P \cup \mathsf{D}^Q) = \langle l, u \rangle,\ ||u - l|| < d\}}{\{\mathsf{D}^P = \emptyset, \mathsf{D}^Q = \emptyset\}} \tag{3.9}$$

that establishes unsatisfiability of the constraint.

Fragment Constraint.

The propagation of a fragment constraint $\texttt{fragment}(V, \vec{P}, T)$ is exploited during the solution search to enforce the assembly process of the fragment $T[V]$ along the point variables $P_1, \ldots, P_n$ of $\vec{P}$. Recall that D^V is the domain of V containing the references $\{j_1, \ldots, j_k\}$ to the database of fragments T.

$$\texttt{fragment}(V, \vec{P}, T) : \frac{\{\mathsf{D}^{P_1} = \{p_1\},\ \mathsf{D}^{P_2} = \{p_2\},\ \mathsf{D}^{P_3} = \{p_3\}, \mathsf{D}^V = \{j_1, \ldots, j_k\}\ \}}{\left\{ \bigwedge_{i=1}^{n} \mathsf{D}^{P_i} = \mathsf{D}^{P_i} \cap \bigcup_{f=j_1}^{j_k} \{\rho((p_1, p_2, p_3), T[f]) \cdot T[f](i)\} \right\}} \tag{3.10}$$

where $\rho((p_1, p_2, p_3), T[f])$ is the roto-translation to be applied to overlap the first three points of the fragment $T[f]$ with the start-anchor (p_1, p_2, p_3).

The conjunction in the bottom part of the rule re-evaluates the domains for P_1, P_2, P_3, and it may reduce the singleton domains to empty whenever there is no compatible ρ for the selected fragment.

Centroid Constraint.

When the positions of the atoms N, C_α and C' for an amino acids a are determined, the propagation algorithm enforces the value for the PV $P_{\mathtt{CG}}$ involved in the centroid constraint.

$$\texttt{centroid}(P_N, P_{C_\alpha}, P_{C'}, P_{\mathtt{CG}}, a) : \frac{\{\mathsf{D}^{P_N} = \{p_N\},\ \mathsf{D}^{C_\alpha} = \{p_{C_\alpha}\},\ \mathsf{D}^{P_{C'}} = \{p_{C'}\}\}}{\{\mathsf{D}^{P_{\mathtt{CG}}} = (\mathsf{D}^{P_{\mathtt{CG}}} \cap \{\mathtt{cg}(p_N, p_{C_\alpha}, p_{C'}, a)\})\}} \tag{3.11}$$

where $\mathtt{cg}(p_N, p_{C_\alpha}, p_{C'}, a)$ is a support function which returns the center of the mass for the side chain of the amino acid a by considering the points $p_N, p_{C_\alpha}, p_{C'}$, as described in Sect. 3.2.3.

Some Implementation Details

The proposed solver relies on an efficient C++ implementation, and it is carefully designed to allow additional tailored solving capabilities without the need of reshaping the core structures.

The internal representation of the domains of the finite domain variables can be abstracted by two arrays of the same length of the size of the initial domain. One array points to the values and the other is a Boolean bit-mask that states whether a value is still in the domain. If all flags are set to 0, the current partial assignment cannot be a part of a solution of the overall constraint; if exactly one is set to 1, then the variable is assigned to a value. This representation implies a linear scan of the domains during the propagation but it is justified by the reasonably small size of the domains of the target application (typically less than 100 values). The internal representation of the domains for point variables is simply a pair $\langle \min, \max \rangle$ that uniquely characterizes a 3D box in $\mathbb{R}^3$. Since these variables are used mostly in distance constraints, this representation is expressive enough (*Oct-trees* have been considered but with no significant advantage).

Point Variables propagation has been described above; these variables are instantiated after fragment selection.

For the management of the `uniqueseq` property (3.7) we implemented a dedicated data structure based on hash tables. Every time a PV is assigned, its value is mapped into a 3D voxel of fixed size. The 3D grid is implemented via a hash table with voxel indexes as keys and points contained in such voxels as values. All the operations can be performed in $\tilde{O}(1)$ (amortized complexity).

3.3.3 Multiple JM Constraints

We briefly describe how we have modeled two problems with FIASCO. The JM constraint is able to model geometrically assembly of fragments and therefore it is used for every protein model. A single JM that covers a protein ensures its flexibility, however for long proteins some computational and precision issues arise. It can be beneficial to model a protein by multiple JM constraints, e.g., $JM(i, j)$ and $JM(j, k)$ so that the amino acids from i to j are covered and the JM constraints overlap on a common amino acid. This practical choice improves the approximate search and allows us to increase the number of different solutions produced. In practice, each protein section handled by a JM constraint is potentially combined to the different arrangements for the other sections. Therefore, it is expected that the number of solutions found grows exponentially in the number of JM constraints. The other JM constraint parameters can be used to control clustering precision and number of conformations found.

3.4 Experimental Results

We report on the experimental results obtained with the FIASCO system (available at `http://www.cs.nmsu.edu/fiasco`). Experiments are performed on a Linux Intel Core i7 860, 2.5 GHz, memory 8 GB, machine. The solver has been implemented in C++.

The fragment database adopted is the FREAD database which has been shown to be effective in loop structure prediction [26]. For the parameters analysis 3.4.1 we use a database of fragments of length 1. These fragments are classified by their amino acid class and their frequency of occurrence over the whole `top-500`.

We set the system to model the two applications described below. In particular, in Section 3.4.1 we analyze the loop modeling scenario and we focus on the performances of `JM` filtering by examining the filtering power and computational costs. Next, we compare the quality of the loop conformations generated, by measuring the RMSD of the proposed loop with respect to the native conformation. We then present some relationships among the JM parameters to control quality and efficiency.

In Section 3.4.2 we show some examples of ab-initio protein structure prediction and we conclude with a comparison of FIASCO against other constraint solvers, for protein models that can be described by a common subset of constraints.

3.4.1 Loop Modeling

The loop modeling problem is formalized by the presence of two known (large) fragments that are both fixed in the space. A sequence of amino acids of length n is given for connecting these two parts of the protein. A JM constraint is defined over the sequence, with particular attention to the starting and ending points that are fixed. The start of the first fragment and the end of the last fragment, namely a sequence $C'ON$ (initial points) of coordinates $\vec{a} = (a^1, a^2, a^3)$, and a sequence $C'ON$ (final points) of coordinates $\vec{e} = (e^1, e^2, e^3)$ are known. There is one caveat about the end points: due to the discrete nature of fragment assembly, it is unlikely to exactly reach the final points. We accommodate for some errors, and require that the JM constraint produces results that fall within some threshold from the corresponding final points.

In Figure 3.11 we show an example of loop computed by FIASCO (the parts of the protein to be connected are shown on the left and the connecting loop on the right).

Additional spatial constraints about points (e.g., no-good regions determined by presence of other atoms) are given. The constant δ (now $\delta = 1.5$Å) asserts a minimum distance between pairs of atoms.

Filtered Search Space And Performances.

We selected 30 protein targets from a set of non-redundant X-ray crystallography structures as done by [21]. We partitioned the proteins into 3 classes according to their loop region lengths ($n = 4, 8$, and 12). We model a CSP that uses fragment assembly to model the loop, in particular using the JM constraint over the loop region.

To assess the filtering capabilities of FIASCO, we perform an exhaustive search generating all the solution for each of the protein targets. Using a clusterization of 0.2Å, a number of different

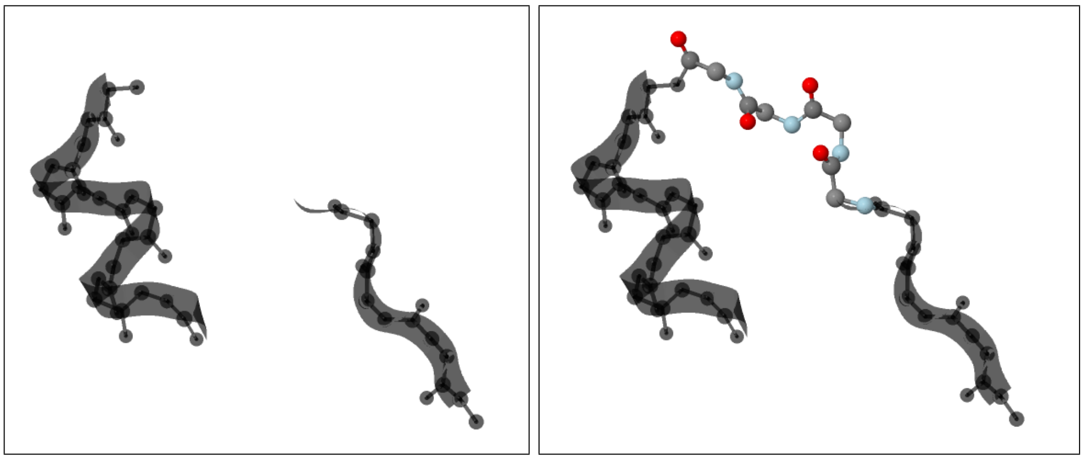


Figure 3.11: An example of loop computed by FIASCO.

fragments of length 1 is found for each amino acid (see Fig. 3.12). The size of the domains for the corresponding FDVs is bound by 100—this is an adequate sample size to describe a reasonable amino acid flexibility. In those cases where the number of fragments exceeds 100, the 100 most frequent ones are kept.

A loop of length n generates an exponential search space of size bounded by 100^n. The selected variable is the leftmost one. Fragments are selected in decreasing frequency order. The clustering parameters are set as follows: the k_{min} value is equal to the size of the domains, while we have used different values for k_{max} based on loop lengths. The values for r and β are set to 120 and 0.5 in each setting. A summary of the parameters is listed in Table 3.1.

In Table 3.1 we report the average times needed to exhaustively explore the loop search space, and the average number of solutions generated. Let us observe that times and number of solutions increase with the lengths of the loops. Therefore, the k_{max} value is decreased to keep "acceptable" running times for longer loops.

n	**JM Parameters**					**Full JM**	
	# JM	k_{min}	k_{max}	β	r	# Solutions	Time (s)
4	1	100	1000	120	0.5	597	3.13
8	2	100	500	120	0.5	98507	10.12
12	3	100	100	120	0.5	328309	28.87

Table 3.1: Loop Modeling settings and average running times (in seconds) and number of solutions generated.

JM Approximated Propagator Quality

Even if the approximated JM produces a small set of solutions, we show here that this is a good representation of the overall variability of the protein structure. For this test, we compare the solutions by means of RMSD from the original structures. The experiments were carried out with the same 30 protein targets and settings described in Table 3.1, with the only exception of k_{max} for the loop set of size 12, which was set to 500.

In Figure 3.13 we show the bar chart for the RMSD of the predictions for each protein loop within the group of targets analyzed. Precisely, in the x-axis there are the 30 (10 for each loop

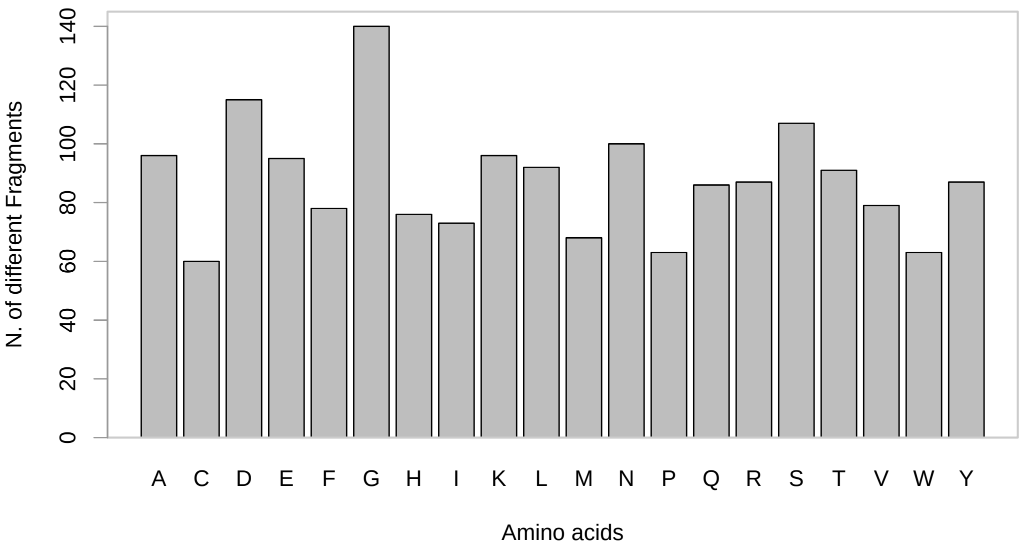


Figure 3.12: Number of different fragments (after clustering) per amino acid in the dataset

length) protein targets. Each bar reports the best RMSD (dark), the average RMSD (grey), and the worst RMSD (light grey) found. Numbers over the bars represent the number of loops found (multiplied by the factors indicated underneath). The results are biased by the fragment database in use: we excluded from it the fragments that belong to the deposited protein targets. Therefore, it is not possible to reconstruct the original target loop and none of the searches are expected to reach a RMSD equal to 0.

For loops of length 8 and 12, the exploration of the whole conformational search space using a simple search procedure would result in an excessively long computation time. For example, consider the search space for loops of length 12 and domains of size 100, that is 100^{12}. This enforces the need for a propagator such as JM, as its filtering algorithm successfully removes redundant conformations and it allows us to cover the whole search space in a short period of time.

In Fig. 3.13 loop predictions are calculated using fragments of length 1. To study how this choice affects both time and accuracy of the sampling we also model the loops of length 12 using fragment of length 3, 6, and 9. Best RMSDs are reported in Figure 3.14. For these experiments we kept the settings used above ($kmax = 500$). Moreover, each JM constraint is imposed on the fragments in order to cover the whole fragment (e.g, for fragments of length 3 we set a JM constraint every three consecutive amino acids) and we set a time-out of 3600 Seconds.

Notice that increasing the length of the fragments the accuracy decreases due to the reduced size of the domains. Nevertheless, the time is also reduced since the sampling is performed on a smaller search space and the JM constraints cover longer sequences of amino acids. The average times are: 1580.14, 0.98, and 0.74 seconds using fragments of length 3, 6, and 9 respectively.

Comparison With State-of-the-art Loop Samplers.

In this section, we compare our method to three state-of-the-art loop samplers: the *Cyclic Coordinate Descent (CCD)* algorithm [21], the *Self-Organizing algorithm (SOS)* [104], and the *FALCm* method [102].

Table 3.2 shows the average of the best RMSD for the benchmarks of length 4, 8 and 12 as

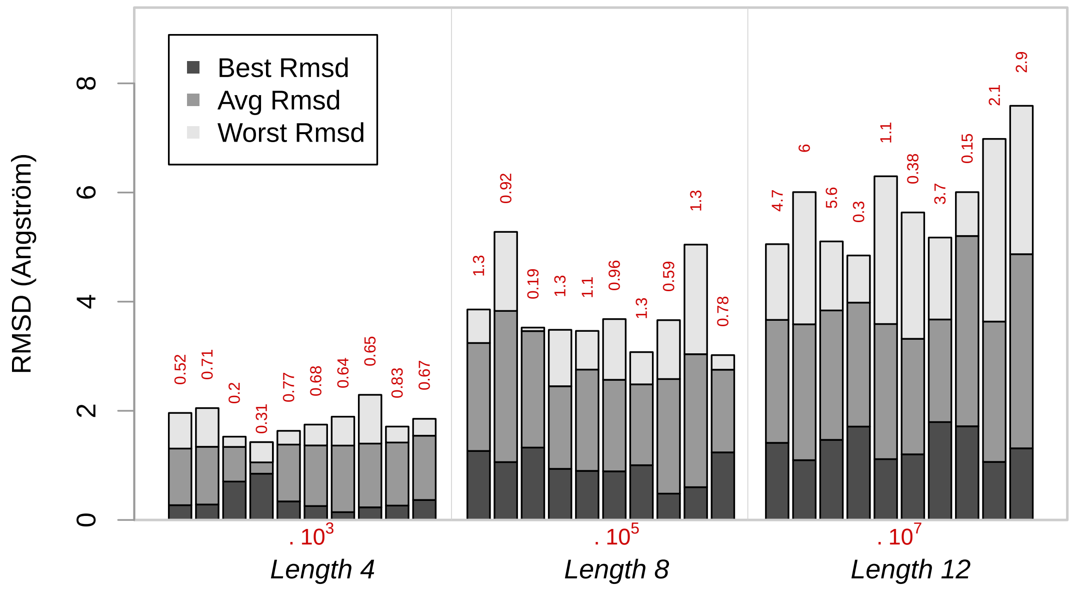


Figure 3.13: RMSD comparison for each Loop Set (x-axis: the 30 protein targets)

computed by the four programs. We report the results as given in Table 2 of [21] for the CCD, Table 1 of [104] for SOS, Table II of [102] for the FALCm method, and the RMSD's obtained adopting the settings for JMf that provided the best results in the previous section (see also Subsection 3.4.1). It can be noted that our results are competitive with those produced by the other systems.

Loop	**Average (best) RMSD**			
Length	CCD	SOS	FALCm	JMf
4	0.56	**0.20**	0.22	0.27
8	1.59	1.19	**0.72**	0.93
12	3.05	2.25	1.81	**1.58**

Table 3.2: Comparison of loop sampling methods

The execution time we reported appear to be very competitive (e.g., see [156]).

JM Parameters Analysis.

In this section, we analyze the impact of the `JM` parameters on the quality of the best solutions found and on the execution times. In particular, the aim of these experiments is to shed light on the relationship between the `JM` constraint settings and the results.

In Figure 3.15, 3.16, 3.17, we analyze the impact of the k_{max} on the execution times (left) and on the precision (right) of the filtering of the JM constraint. From top to bottom, we use $\beta = 60, 120, 360$. The tests are performed over the protein loops of length 4 (see section above),

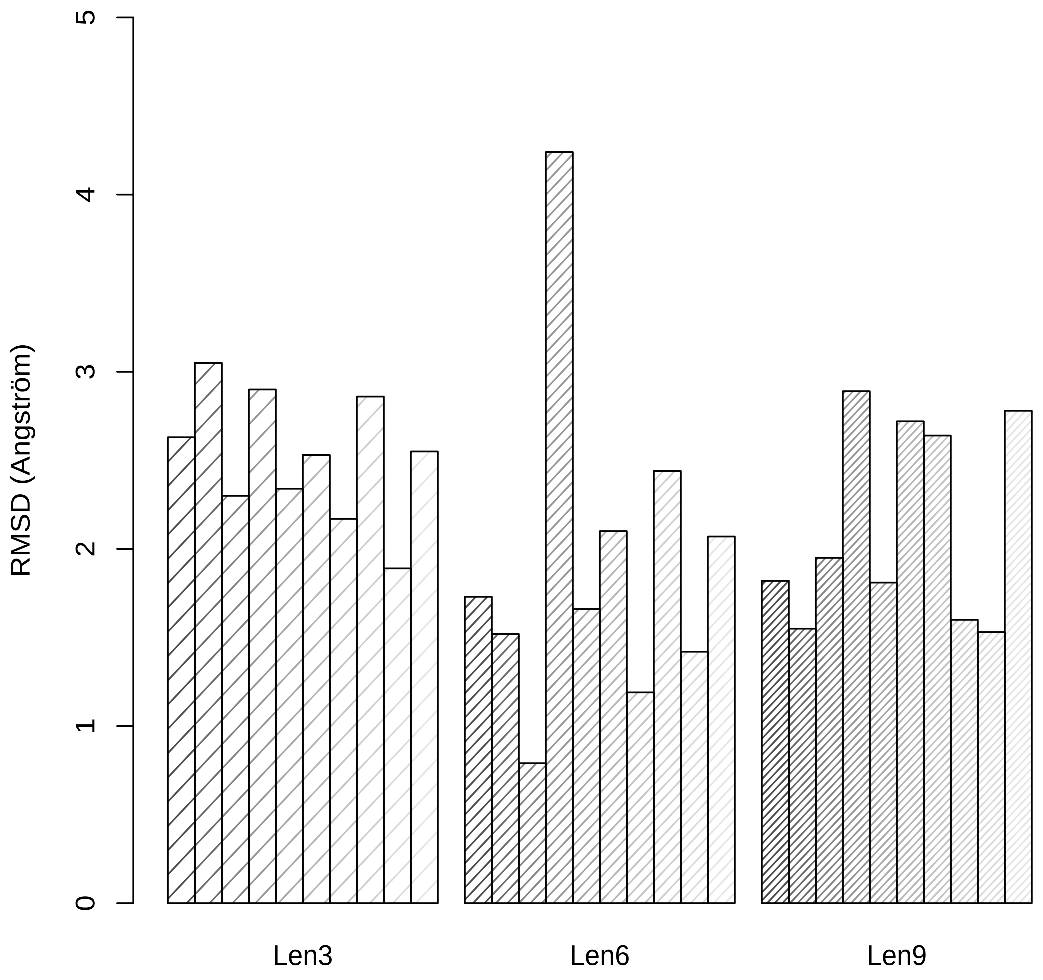


Figure 3.14: RMSD comparison for loop sampling on loops of length 12 using fragments of length 3, 6, and 9.

adopting as cluster parameters, r in $\{0.5, 1.0, 3.0, 5.0\}$, and $k_{min} = 100$. Each dot in the plots represents the average of the best RMSD found by each predictions (left) and the average execution time (right). The RMSD values tend to decrease for smaller clustering parameters r and β and as the number of clusters increases, while the filtering time increases as k_{max} increases.

In Figure 3.18 we study the relation between the RMSD and both the number of JMs that cover a given target loop or protein and the voxel-side parameter. For these experiments we used the values $\{100, 250, 500, 800, 1000\}$ for the k_{max}, we set $r = 1$, $\beta = 120$, and we averaged the RMSDs values on the resulting sample set of structures. The relation between the RMSD and number of JM as well as the average and worst computational times are shown in Fig. 3.18 left. Here we use a medium-length loop taken from the protein 1XPC (res. 216-230) and we vary the number of JMs that cover the loop (the side of the voxel has been set to $3\mathring{A}$). From the figure we observe that increasing the number of JMs (i.e. covering less amino acids with a single JM) the RMSD decreases but the computational cost is higher. Notice that the best RMSD is given when the loop is covered by 4 JM constraint (i.e., a JM constraint each four consecutive amino acids). As a rule of thumb we suggest to use a JM constraint to cover from 3 to 4 consecutive amino acids since this setting produces the best results within an acceptable time. In Fig. 3.18 right we report the best RMSD (solid line) and the average RMSD (dotted line) of the structures found using multiple `JM` constraints that cover sequences of 4 consecutive amino acids through the whole target proteins. Namely, if the protein target has length n, we set the `JM` constraints from i to $i + 3$, where $i = 3 \cdot j, 0 \leq j < n/3$. For these experiments, we considered three proteins of relatively short length, in order to obtain a complete exploration of the search space in reasonable computational time: 1LE0 (length 12), 1MXN (length 16), and 1FDF (length 24). Moreover we used the values $\{3, 5, 10, 20, 30, 50, 100\}$ for the side of the voxels used for the clustering.

From the Figure 3.18 we observe that the voxel size (enabled by the `uniqueseq`) has an impact on the clustering for values lower than $30\mathring{A}$ (recall that these proteins have a diameter less than $30\mathring{A}$). For voxel sides lower than $3\mathring{A}$ we observe no substantial improvement in terms of quality, while the time required by the solver to compute the solutions increases exponentially.

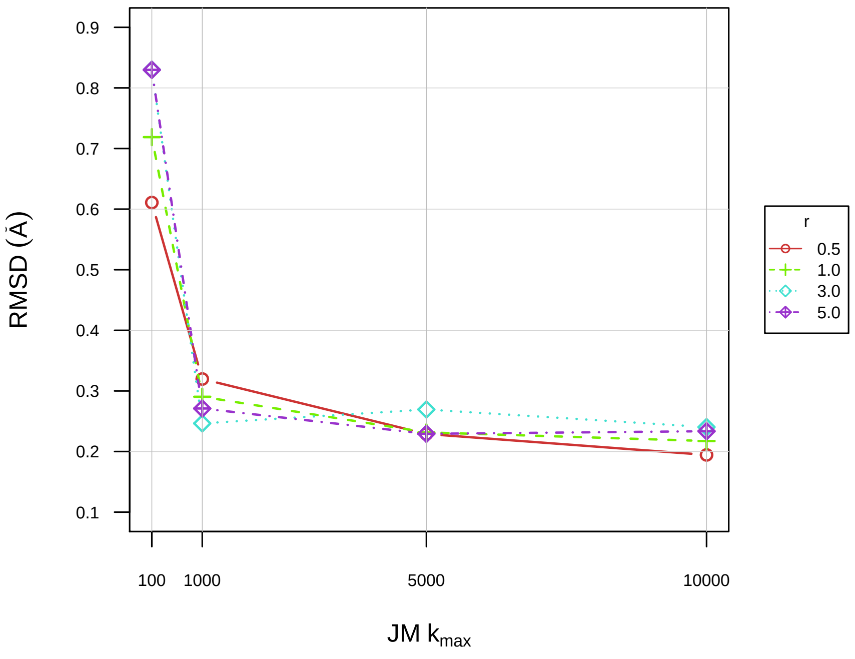
Results Summary and Default Parameters.

We now provide some guidelines that may be helpful to tune the JM parameters for a given protein modeling problem. We suggest several levels of parametrization that might be used according to the user needs with respect to running time or prediction accuracy. We stress that these are merely guidelines, outlined from our empirical evaluations, and that several tests should be done to establish the desired tuning.

We suggest to set a JM to model a sequence of at least 3 amino acids and in general not longer than 8, to payoff the computational load of the JM clustering. The default choice for k_{min} is set to be the average size of the variable domains involved in a JM constraint, while we suggest to set k_{max} to be at least as k_{min} and not greater than $10,000$. The latter, together with the number of consecutive JM constraints, will have the greatest impact on the computational cost and prediction accuracy. Computational costs will grow as the number of consecutive JM increases, and at the same time it will also produce in general higher accuracy. The same trend is exhibited by the growing k_{max} parameter. Table 3.3 illustrates five basic settings that could be used incrementally to establish a trade off between running times and prediction accuracy. The first level (Lev. 1) is associated to faster computational times and lower accuracy while the last one (Lev. 5) is the slowest but also the most accurate. The second column of the table indicates the length of the amino acid sequence modeled by a single JM.

3.4.2 An Application in Protein Structure Prediction

In the protein structure prediction problem, we model a generic backbone through multiple JM constraints. In principle, an unique JM constraint can model the whole problem. As in the previous cases, we split it into smaller parts, moreover, the presence of secondary structure is a valid help in the placement of JM constraints that can handle loops between each consecutive pair. A simple



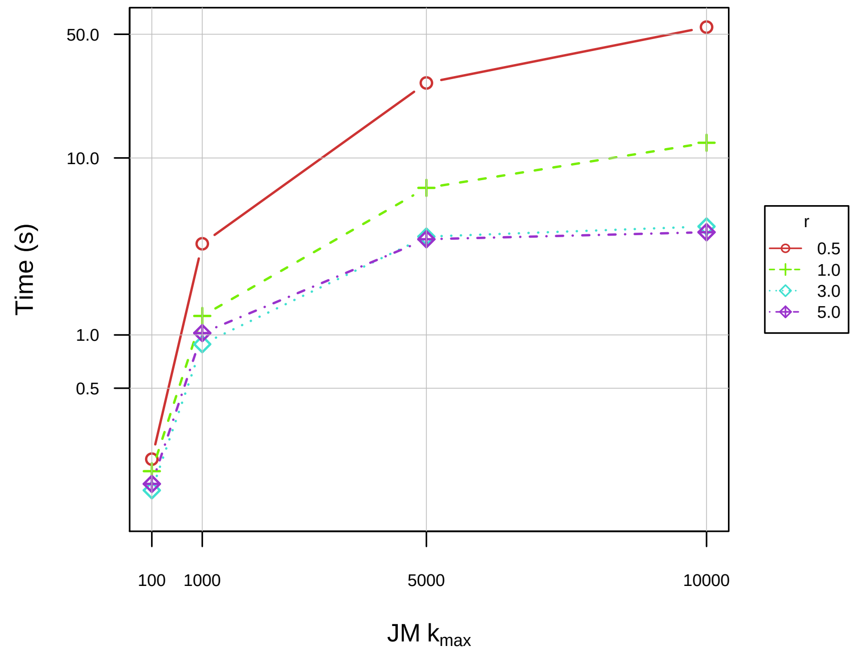
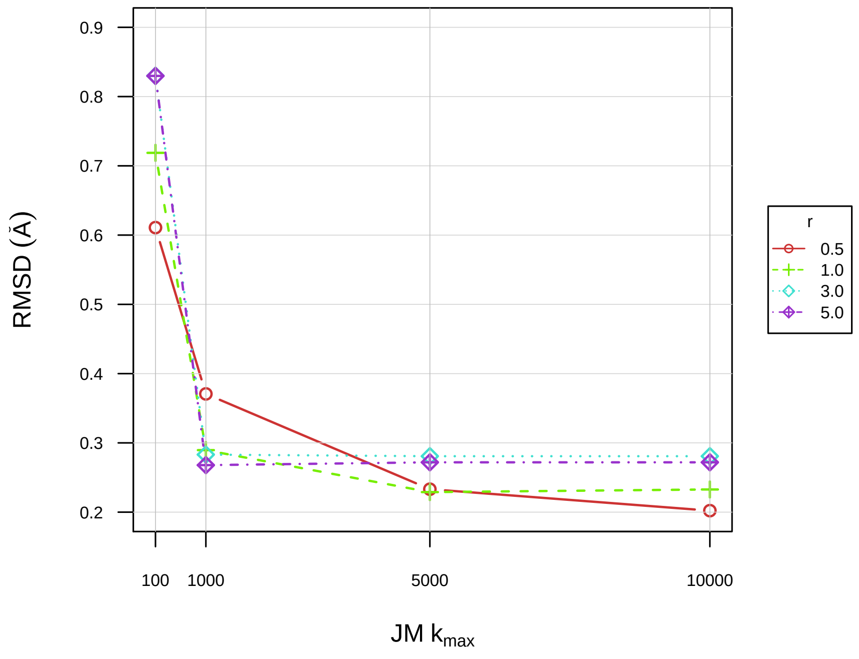


Figure 3.15: Comparison of the best RMSD values and execution times at varying of the k_{max} clustering parameter for $\beta = 60$.



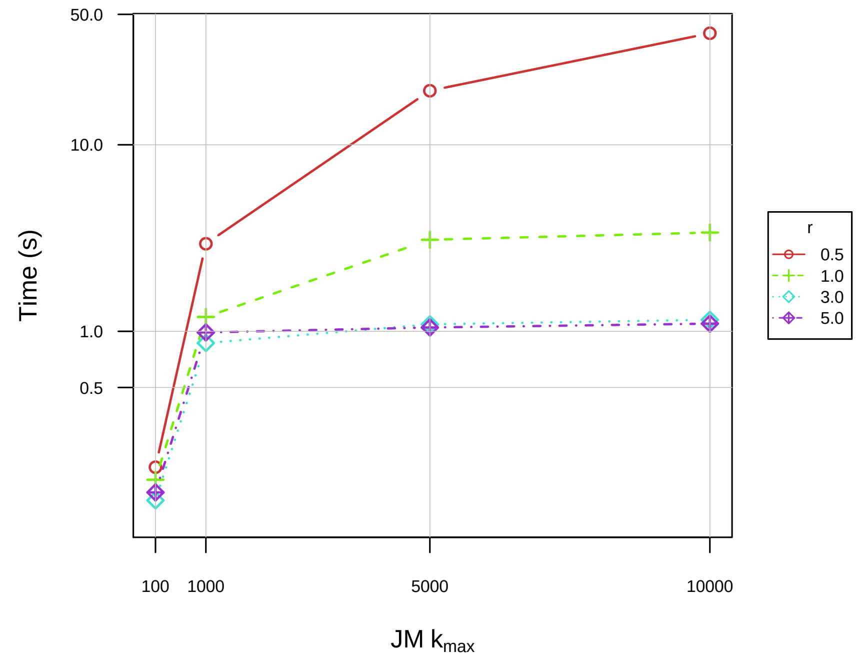
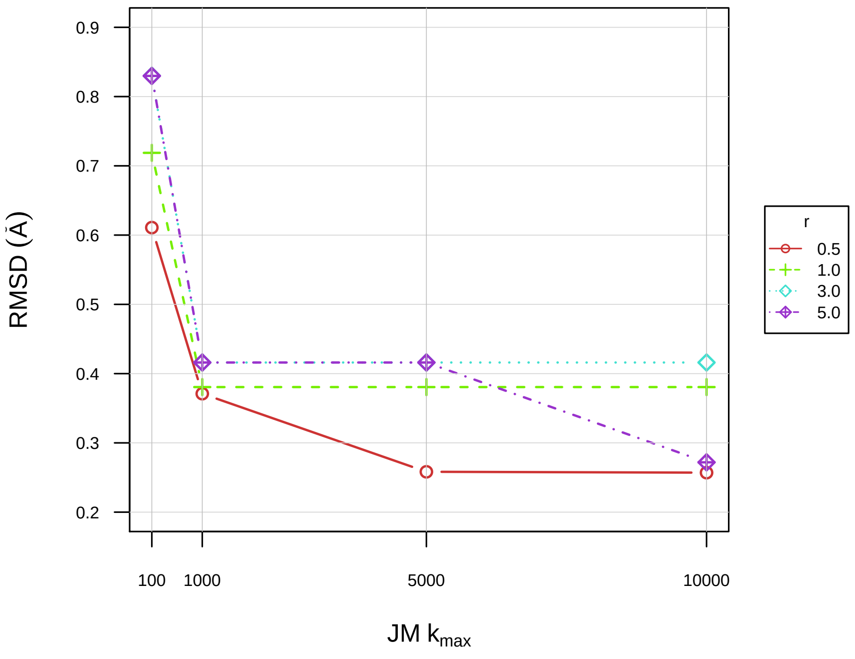


Figure 3.16: Comparison of the best RMSD values and execution times at varying of the k_{max} clustering parameter for $\beta = 120$.



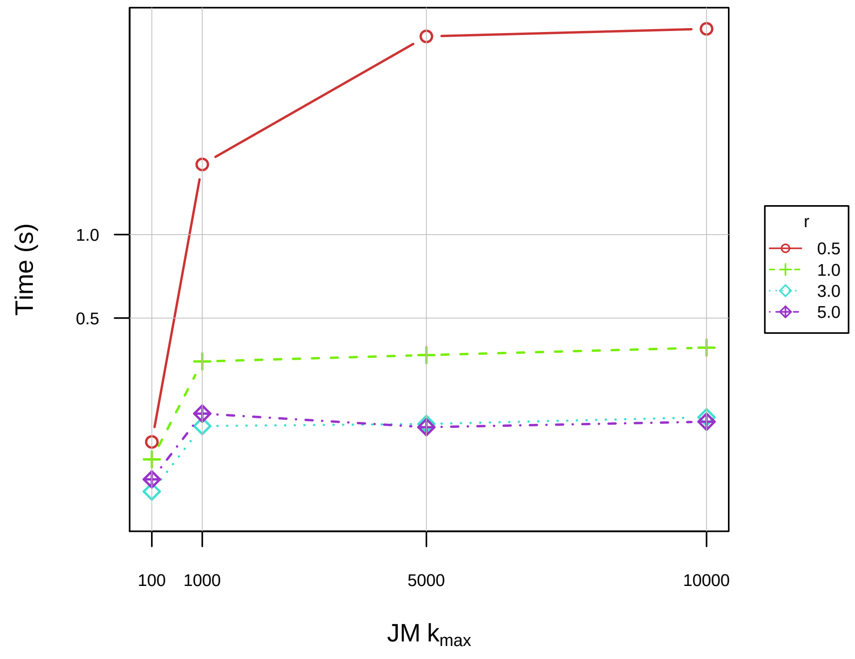
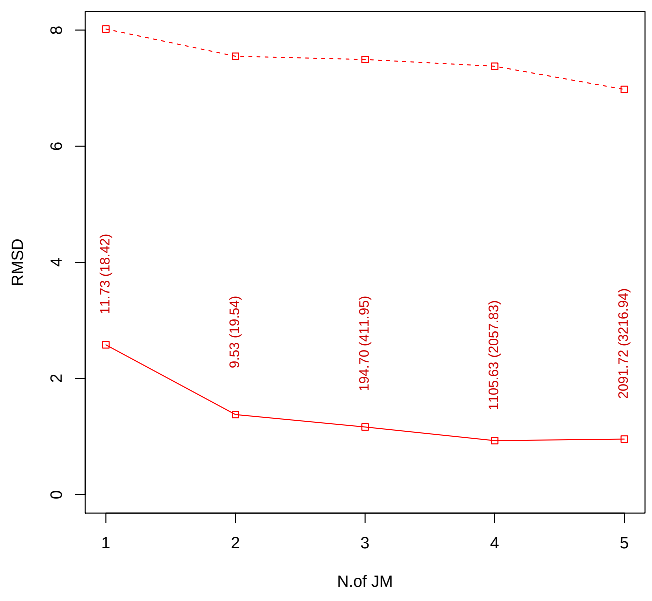


Figure 3.17: Comparison of the best RMSD values and execution times at varying of the k_{max} clustering parameter for $\beta = 360$.



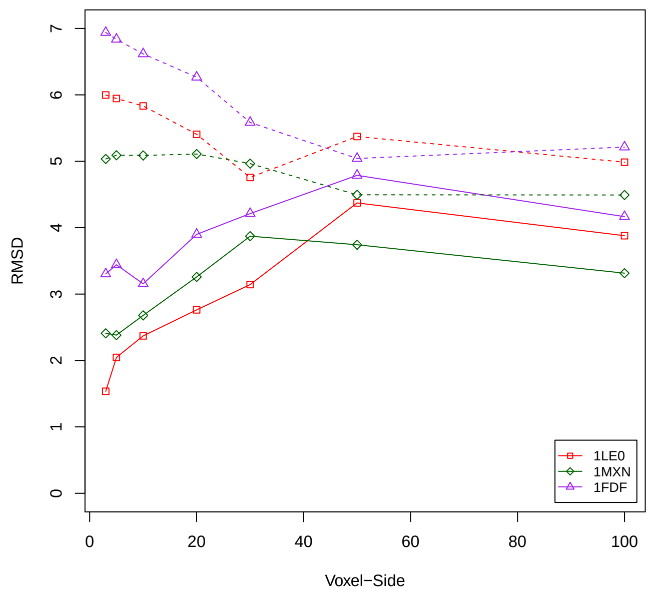


Figure 3.18: Top: RMSD (best and average) and Time (average and worst) values increasing the number of JM constraints that completely cover a target loop of length 15. Bottom: Average (dotted line) and best (solid line) RMSD for the targets 1LE0 of length 12 (top), 1MXN of length 16 (medium), and 1FDF of length 25 (low).

Lev.	$n.JM$	k_{min}	k_{max}	β	r	Speed	Accuracy
1	8	$\|D\|$	500	120	5	$* * * *$	$*$
2	8	$\|D\|$	1000	120	3	$* * *$	$* *$
3	6	$\|D\|$	100	120	3	$* * *$	$* *$
4	4	$\|D\|$	500	120	3	$* *$	$* * *$
5	4	$\|D\|$	1000	120	1	$*$	$* * * *$

Table 3.3: JM default parameters

search can generate a pool of conformations, then energy scoring can select the best candidate. We have used a statistical energy function developed for the 5@ model, but any other energy function can be used instead.

In this section, we study the applicability of FIASCO to the protein structure prediction problem. In particular, we consider prediction problems where the secondary structure elements of the protein are given. Furthermore, in order to assess the potential structure, we introduce an energy function—the same that we have adopted in previous studies, and more precisely described in `http://www.cs.nmsu.edu/fiasco`.

For the modeling, we have used the information about the location and the type of the secondary structure elements on the primary sequence provided directly by the Protein Data Bank. We have imposed a sequence of JM constraints between every consecutive pair of secondary structure elements. The number of consecutive JM constraints varied according to the length of the unstructured sequence being modeled, covering at most 5 amino acids with a single JM constraint. In addition one JM constraint was imposed from the first amino acid to the beginning of the first secondary structure element and another from the end of the last secondary structure element and the last amino acid (the tails of the protein). The domains for the initial and end points of the JM constraints are the set of all admissible points (a box large enough to contain the protein). In the search phase, the "first" secondary structure is deterministically set in the space. Then the labeling proceeds with the JM constraint attached to it leading to the next secondary structure and so on. Tails are instantiated at the end.

The propagation of the constraints generates a set of admissible structures, that represents the possible folds of the target protein. From this set, we select the structure with minimum energy; we extract also the structure with minimum RMSD, in order to evaluate the quality of the energy function. For these tests we adopt the FREAD database. Table 3.4 reports the best energy values found by FIASCO. In the RMSD columns is reported the corresponding RMSD associated to the conformation with best energy found by the solver. The $\#JM$ column reports the total number of JM used to model each protein, together with the maximum number of consecutive JM adopted to model a contiguous sequence of amino acids (within parentheses).

Protein ID	Len.	# JM	Energy	RMSD	Time (Min.)
1ZDD	35	4(2)	−100513	2.05	11.42
2GP8	40	4(2)	−138110	6.28	8.55
2K9D	44	5(2)	−204693	2.52	2.69
1ENH	54	4(1)	−309896	8.21	31.67
2IGD	60	7(2)	−295882	10.50	26.47
1SN1	63	7(3)	−358874	5.55	14.82
1AIL	69	4(1)	−411077	4.59	4.46
1B4R	79	11(2)	−313590	6.11	8.41
1JHG	100	7(1)	−572950	4.51	4.50

Table 3.4: Ab initio prediction with FIASCO.

The results show that the quality of the predictions computed by FIASCO (6.3 as average RMSD) is competitive (and, as shown in the following section, at par or better than what produced by other methods). The results are particularly encouraging for proteins of longer length, where the sampling of the search space aids in development of admissible structures. The time required by FIASCO to completely explore the search space depends strongly on the type and the mutual arrangement of secondary structure elements of the target. For example, the protein *2K9D* and the protein *1ENH* have the same length, but FIASCO is significantly faster on the first protein than on the second one. The same observation can be made for the proteins *2IGD* and *1SN1*. The results reported in Table 3.4 are promising and they suggest that this is a feasible approach to solve the ab initio prediction problem. As a future work, we will explore the integration of local search techniques (e.g., large-neighboring search), in order to sample the search space and to further decrease the time needed to explore it.

3.4.3 Comparison of FIASCO with State-of-the-Art Constraint Solvers

In this section, we motivate our choice of designing an ad-hoc solver instead of using a general-purpose constraint solver. In particular we provide a comparison between FIASCO and state-of-the-art constraint solving. The results justify the choice of implementing a new solver from scratch instead of using an available constraint programming library or a constraint programming language. The solver chosen for this comparison is Gecode [62], a very efficient solver and the winner of the most recent MiniZinc challenges [120].

Gecode has introduced (in version 4.0) the handling of floating point variables. Nevertheless, since Gecode is the fastest solver for FD variables, we have first encoded the PSP by discretizing fragments and positions. In particular, we multiplied each real number by a scaling factor (100) to obtain integer values. Each spatial position is encoded by a triple of variables, representing the coordinates of the point. Each operation (e.g., multiplications) applied to such variables requires re-scaling of the result; this unfortunately leads to ineffective propagation. This is particularly evident when dealing with distance constraints, that require the implementation of Euclidean distance between pairs of triples of variables.

In order to understand the solvers capabilities to propagate constraints on the placement of overlapping fragment we implemented three versions of the code, that considered a different number of constraints, precisely:

1. An implementation that uses only JM constraint (JM only);
2. An implementation that adds the `alldistant` constraint;
3. An implementation that adds the `alldistant` and `centroid` constraints.

In all cases we use a complete search (in particular, the clustering and tabling constraints of lines 10 and 12–14 of Algorithm 3 are disabled).

In Table 3.5, we report the execution times required by FIASCO and by Gecode (with the same considered constraints) to determine an increasing number of solutions, from $1,000$ to $1,000,000$. These solutions are computed for the target protein 1ZDD which has length 35. Table 3.5 shows that the execution time of both solvers increases proportionally with the number of solutions found. However, FIASCO is one order of magnitude faster than Gecode in the unconstrained case, and two orders of magnitude faster in the other cases. The main reason is that FIASCO is specifically developed to handle the finite domains and 3D point variables, while these are approximated by FD variables in Gecode. Constraints on these approximations propagate poorly and slowly. Moreover, the approximation of fragments using finite domain variables introduces approximation errors, that grow during the search phase (and consequently, less solutions are returned in the constrained cases). These errors may result in final structures that are relatively imprecise when the coordinates of the atoms are converted back into real values.

In Table 3.6, we consider a small sequence of four amino acids (SER TRP THR TRP—the first four amino acids of the protein 1LE0), and we generate all solutions. We report the values of the

Number of	FIASCO			Gecode		
solutions	JM only	`alldistant`	`alldistant + centroid`	JM only	`alldistant`	`alldistant + centroid`
1000	**0.030**	**0.051**	**0.059**	0.358	2.531	3.807
10000	**0.312**	**0.476**	**0.612**	2.571	21.056	35.370
100000	**3.006**	**4.794**	**6.040**	25.407	209.569	347.831
1000000	**29.859**	**47.669**	**61.385**	252.815	2186.83	3632.39

Table 3.5: Comparison of the execution times of FIASCO and Gecode, for increasing number of solutions and with different sets of considered constraints.

best and the average RMSD among the structures of the sets of solutions computed using FIASCO and the Gecode implementation after a complete enumeration of the domains. We can observe that FIASCO is significantly faster in exploring the search space, moreover, the approximation introduces errors that leads to the loss of feasible solutions.

	FIASCO				Gecode			
	N. sol.	Time (sec.)	RMSD	Avg. RMSD	N. sol	Time (sec.)	RMSD	Avg. RMSD
JM only	**810000**	**20.493**	**0.167**	**1.570**	**810000**	181.102	0.190	1.596
`alldistant`	**805322**	**33.493**	**0.167**	**1.564**	774463	252.974	0.190	1.591
`alldistant + centroid`	**805322**	**38.953**	**0.167**	**1.564**	169441	140.644	0.580	1.880

Table 3.6: Number of solutions, time, best RMSD, and average RMSD on the set of structures found by FIASCO and Gecode after a complete enumeration of the solution space using different constraints

We have encoded the same constraint satisfaction problem using the new version of Gecode that allows to employ float variables. We labeled the finite domain variables that allow to select fragments, while values for the point variables are obtained by constraint propagation. Since constraint propagation of float variables is based on interval arithmetics, it turns out that after few amino acids these intervals are too large for being able of reconstructing the protein and or evaluating the energy value. For instance, after a complete assignment of the variables related to fragments of protein 1ZDD, while the domains of the float variables associated with the position of the first two amino acids are singletons, those related to the tenth amino acids are intervals with size from 1 to 2 Å; even worse, the domains of the atoms of the eleventh amino acids are unbounded. A further stage of labeling of the float variables required computational time of orders of magnitude higher than those reported in Table 3.6 for the finite domain Gecode implementation.

Constraint solvers like ECLiPSe [23] and Choco [25] also support the mixed use of integer and real variables. ECLiPSe is a Prolog-based language which handles integer and real variables together. However, the great number of matrix operations required in our application does not fit well with a Prolog implementation. Furthermore, the current trend of ECLiPSe is to replace a direct constraint solving with a translation to FlatZinc. In the case of Choco, the current support of floating point variables is still under development (c.f. `http://choco.sourceforge.net/userguide.pdf`—page 3). Things may change with the next releases.

We also experimented with another constraint solver, by implementing the multi-body constraints using the JaCoP library [82], in a similar way as done for Gecode. Eventually, we tested the same protein used for the results reported in Table 3.5, and we did not observe any substantial difference in terms of execution time, from the Gecode implementation.

In terms of protein structure prediction, the design of FIASCO has been influenced by our own previous work on the TUPLES system [36]. TUPLES is also a constraint solver for protein structure prediction, based on fragments assembly. Figure 3.19 compares the performance of TUPLES and FIASCO on the same set of proteins discussed in Section 3.4.2. To make the comparison fair, we make use of the same energy function in both systems and assume that the secondary structure elements are known. Note that there are some important differences between the two systems. TUPLES is implemented using constraint logic programming techniques, specifically, SICStus Prolog [162]; TUPLES does not make use of floating point variables; on the other hand, TUPLES introduces a heuristic search mechanism based on large neighboring search.

The results show that the quality of the predictions computed by FIASCO (6.3 as average RMSD) is better than the quality of the predictions computed by TUPLES (9.4 as average RMSD). The complete sampling of the search space allows us to obtain better results for the proteins of longer length in the benchmark ($\geq$ 63). Instead, for shorter proteins, we obtain comparable results. The similarity of the quality depends on the use of the same energy function for both the systems. Notice that the energy function used is designed for the simpler model adopted in TUPLES (C_α–C_α). Moreover, TUPLES is based on a Prolog implementation and hence each value must be rounded and approximated. These aspects explain both the quality differences between the RMSD and the Best RMSD found by FIASCO and the behavior for which for some proteins (e.g., *1ZDD*, *2GP8*) the (energy) RMSD values are better in FIASCO even if their corresponding energy (RMSD) values are higher than in TUPLES. The execution times of FIASCO are significantly faster than TUPLES, in spite of FIASCO's lack of a sophisticated search heuristic.

We also performed a comparison with the state-of-the-art online Robetta predictor [131] for the first four proteins of Table 3.6. We built the dictionary for 3 and 9 amino acid long peptides through the Robetta interface, and we disabled any homology inference, in order to maintain a fair comparison. The results are reported in table 3.7. It can be noted that our results are comparable with Robetta predictor.

		Robetta		FIASCO	
Protein ID	Len.	RMSD	Time (Sec.)	RMSD	Time (Sec.)
1ZDD	35	5.92	21.00	2.05	11.42
2GP8	40	5.44	16.00	6.28	8.55
2K9D	44	4.65	22.00	2.52	2.69
1ENH	54	2.74	39.00	8.21	31.67

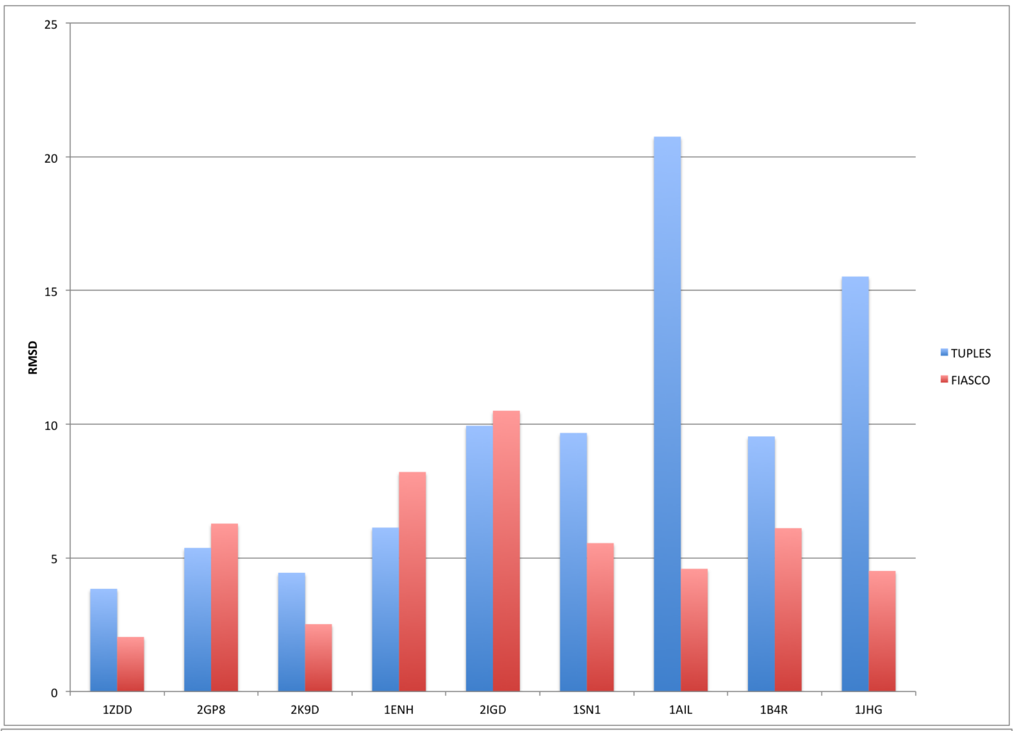
Table 3.7: Ab initio prediction with FIASCO.

Let us conclude this section mentioning that the results reported in the this section (where we compared FIASCO with TUPLES) provide also an implicit comparison with another off-the-shelf solver, the SICStus Prolog constraint logic programming solver [162].

3.5 Summary

In the first part of this chapter, we introduced some notions related to the constraint programming paradigm. As a paradigm, CP provides the tools necessary to guide the modeling and resolution of search problemsin particular, it offers declarative problem modeling (in terms of variables and constraints), the ability to rapidly propagate the effects of search decisions, and flexible and efficient procedures to explore the search space of possible solutions. We presented several aspects related to the constraint solving process such as complete and incomplete search strategies, constraint propagation, and consistency notions.

In the second part of the chapter, we presented a real-world problem tackled with constraint programming techniques, namely the problem of predicting the three-dimensional structure of a protein given its sequence of amino acids. We described a novel constraint (joined-multibody) to model rigid bodies connected by joints, with constrained degrees of freedom in the 3D space. We presented a polynomial time approximated filtering algorithm of the joined-multibody constraint, that exploits the geometrical features of the rigid bodies. In particular, the filtering algorithm is combined with search heuristics that can produce a pool of admissible solutions that are uniformly sampled. This allows for a direct control of the quality and number of solutions. The filtering



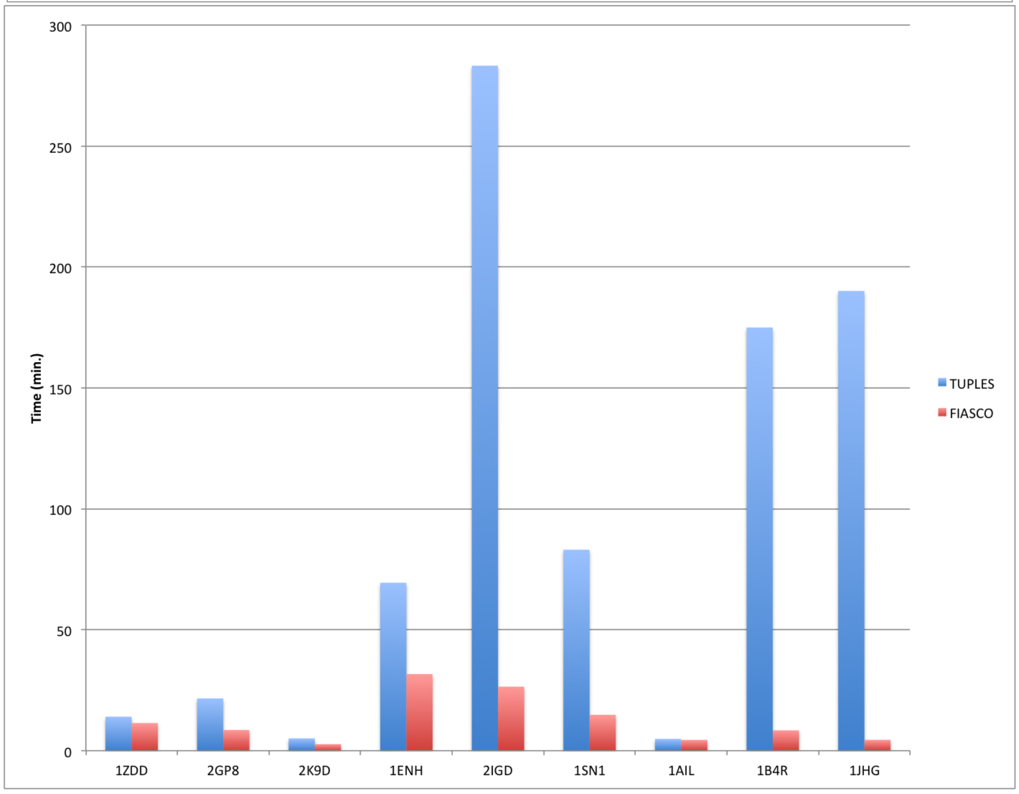


Figure 3.19: Comparison of RMSD and Execution Time between TUPLES and FIASCO

algorithm is based on a 3D clustering procedure that is able to cope with a high variability of rigid bodies, while preserving the computational cost. The practical advantages of the joined-multibody constraint are shown by an extensive set of real protein simulations for two main categories: protein loop reconstruction and structure prediction (ab-initio). The tests showed how the parameters of the constraint are able to control effectively the quality and computational cost of the search. In conclusion, the constraint solver FIASCO is able to model effectively various common protein case-studies analyses.

In the last part of this dissertation, we shall describe the porting of FIASCO within a GPU-based framework. The CP model adopted for FIASCO will remain the same but propagation algorithms and search space exploration will take advantage of GPU computation, leading to a significant speed-up in terms of computational time.

II

Parallel Constraint Solving

4 Background

Constraint programming has gained prominence as an effective and declarative paradigm for modeling and solving complex combinatorial problems. In spite of the natural presence of concurrency, there has been relatively limited effort to use novel massively parallel architectures to speedup constraint programming algorithms. Recent technological trends have made massive parallel platforms and corresponding programming models available to the broad users community—transforming high performance computing from a specialized domain for complex scientific computing into a general purpose model for everyday computing. One of the most successful efforts is represented by the use of modern *Graphical Processing Units* (*GPU*s) for general purpose parallel computing: General Purpose GPUs (GPGPUs). Several libraries and programming environments (e.g., the *Compute Unified Device Architecture* (*CUDA*) created by *NVIDIA*) have been made available to allow programmers to access GPUs and exploit their computational power. Nevertheless, obtaining good speed-ups from parallel algorithms that run on GPU architectures is not an easy task. While it is relatively simple to develop correct CUDA programs (e.g., by incrementally modifying an existing sequential program), it is challenging to design an efficient solution. Direct porting of sequential algorithms often perform poorly and new strategies must be adopted in order to reduce the drawbacks inherently associated with the use of a GPU architecture (e.g., small amount of shared memory between threads, slow communication channel between CPU and GPU, etc.), and maximize the benefits of having thousands of parallel threads running in parallel. Thus, optimization of CUDA programs require a thorough understanding of the hardware characteristics of the GPU being used, as well as an accurate design of the algorithms to run in parallel.

In the second part of this dissertation we focus our attention on parallel constraint programming. To this end, we start by reviewing some aspects regarding parallelism and parallel complexity notions (Section 4.1). We present parallel and distributed algorithms for arc consistency and constraint propagation, and we evaluate their efficiency w.r.t. their sequential counterparts. In Section 4.3 we describe parallel search strategies. In particular, we focus on parallel local search algorithms which have been proven to be efficient on real-world problems. We also report on multi-agent search, where a problem is solved by a collection of agents that exchange messages in order to solve distributed versions of constraint satisfaction and optimization problems. We conclude this background review with a brief introduction on GPU computation and the CUDA programming environment.

4.1 Parallel Computing

Parallel computing is a form of computation based on the principle that complex computational tasks can be broken down in many simpler ones, which can be solved concurrently. There are different forms of parallelism, according to the criteria used to classify it. For example, it is possible distinguish between parallelism at the level of instructions, data or task; multi-core parallelism (i.e., using shared memory), distributed computation, or combinations of both. In a multiprogramming context, it is also possible to classify parallelism as *Single Instruction, Multiple Data* (*SIMD*) parallelism (i.e., the same set of instructions performed in parallel on different data) or *Multiple*

Instruction, Multiple Data (*MIMD*) parallelism (i.e., different instructions performed in parallel on different data) [167].

GPUs offer (a variant of) SIMD parallelism on shared memory architectures, at the level of data. Therefore, they entail the restriction of providing data level parallelism but not concurrency (i.e., multiple instructions at a given moment). On the other hand, thousands of parallel thread are available as computational units at the same time, with no context switch in a well-defined multi-dimensional hierarchical organization.

This form or parallelism shows its major advantages on embarrassingly parallel problems. A problem is referred to as *embarrassingly parallel* or *perfectly parallel*, when there is minimal or no communication between runs and little to no effort for load balancing. It follows that, usually, little effort is required to decompose it in many parallel task. Problems in which there is no communication or dependency between parallel tasks are usually embarrassingly parallel. For example, given n inputs values $x_1, \dots, x_n$, output variables $y_1, \dots, y_n$, and $f_1, \dots, f_n$ *pure* functions (i.e., no side effects) taking the corresponding input values as parameters, the following n assignments:

$$y_i = f_i(x_i),\ 1 \le i \le n$$

can be performed in parallel by n different threads and represent an embarrassingly parallel task (e.g., if $f_1 = \cdots = f_n$ the above assignment corresponds to the C++ *STL* function `std::map`). Other examples of embarrassingly parallel problems suitable for GPU computation—and actually used—are: *brute force* searches in cryptography, rendering i computer graphics, sum/min/max reductions, parallel implementations of genetic algorithms, and parallel samplings (e.g., Monte Carlo sampling) [46].

4.1.1 Metrics

Speedup. The first question we are interested in when we develop a parallel algorithm is how much faster the algorithm is compared to its sequential implementation, i.e., how much *speedup* we gain from parallel computation. The most common definition of speedup is the ratio of the computational time (T_1) taken by an algorithm executed on a single processor machine to the time taken by a parallel version of the same algorithm on a p-processors machine (T_p) [167]:

$$S(p) = \frac{T_1}{T_p}.$$

Let us observe that usually T_1 corresponds to the time taken by the fastest known sequential algorithm for the given problem. The definition above is a good indicator about the performance of a parallel implementation. Nevertheless it does not give any hint regarding how the execution times T_1 and T_p should be calculated. For example, the value T_1 depends on the actual implementation and the structure of the sequential algorithm, which may differ significantly from the structure of parallel algorithm used to calculate T_p. Let us observe that in the worst case scenario, the sequential algorithm has a completely different structure w.r.t. its parallel implementation.

In a theoretical analysis, a more accurate indicator of the speed-up gained by using a parallel machine is given by the ratio of the number of computational steps using one processor (C_1) to the number of parallel computational steps with p processors (C_p):

$$S(p) = \frac{C_1}{C_p}.$$

This measure takes into account the differences of the structure between the two implementations (i.e., sequential and parallel), but it does not consider the communication costs between parallel processes that are, usually, more expensive than computational steps.

Linear speed up & Efficiency. The upper bound on the maximum achievable speedup is usually a *linear speedup*, i.e., a speedup of p using p processors:

$$S(p) \le \frac{T_1}{T_1/p} = p.$$

Linear speed up can be obtained on embarrassingly parallel problems or, in general, when the parallel task running on p processors can be divided in p processes of equal-duration and with no additional overhead.

Sometimes, a *super linear* speedup (i.e., $S(p) > p$) can be also obtained. This usually happens when the ratio is calculated on a sub-optimal implementation of the sequential algorithm [167].

The ratio between the speedup obtained and the total number of processors used by the algorithm is called *efficiency* and it indicates how effectively the processors are used:

$$\eta = \frac{S(p)}{p} \times 100\%$$

Higher efficiency means better utilization and, vice-versa, better utilization of the parallel architecture increases the speed up.

Inherent parallelism indicates how much speed up can be expected from a parallel algorithm. The inherent parallelism is based on the notion of *unbounded parallel complexity* of an algorithm A, i.e., the time taken by the algorithm given an infinite number of processors, and it is defined as follows:

$$IP(A) = \frac{\textbf{Sequential complexity of } A}{\textbf{Unbounded parallel complexity of } A}.$$

$IP(\cdot)$ represents the amount of parallelism intrinsic to the problem, and larger values usually mean large amount of computation that can be performed in parallel.

Amdahl's law. *Amdahl's law* [4] is a simple observation regarding the maximum expected speed up achievable on a parallel machine. The Amdahl's law relates the percentage b of a program that must be executed sequentially to the remaining $1 - b$ part that admits an arbitrary level of parallelism. Therefore, the expected speed up is:

$$S(p) = \frac{T_1}{T_p} = \frac{T_1}{T_1 * b + \frac{T_1 * (1-b)}{p}} = \frac{1}{b + \frac{1-b}{p}}.$$

Let us observe that, according to Amdahl's law, as p tends to infinity the maximum speed up tends to the constant $1/b$.

Example 4.1.1 *Let us suppose that a given (sequential) algorithm can be parallelized by a factor of* 95% *w.r.t. its original sequential code. It follows that* $b = 5/100 = 1/20$ *is the percentage of inherently sequential computational steps and therefore, applying the Amdahl's law, the maximum achievable speed up is* 20*, regardless of the number of available processors.*

This argument was one of the reasons that encouraged hardware development away from multi-processor and towards faster processors. Let us observe that Amdahl's law is overly pessimistic [70, 79] and it does not reflect the actual performance of the system. Amdahl's law shows the general trend but it does not take into account real systems and some non-trivial processes. Amdahl's law assumes that given infinite number of cores the parallel part is executed instantly. This assumption is accurate but not true due to both software scheduling policies and hardware characteristic of the parallel architectures. Moreover, Amdahl's law does not take into account the effect of critical section, atomic operation, barrier synchronization, and other inter process communication methods. Therefore, in order to calculate the expected speed-up, it is usually simpler to plot the Amdahl's curve by measuring the performance of the parallel application and comparing it with the theoretical results.

4.2 Parallel Consistency in Constraint Programming

Exploitation of multi-processors architectures for parallel arc consistency has led to different results, varying from to 2× to quasi-linear speedups. In order to parallelize constraint propagation different strategies have been adopted. In what follows we present two main parallel strategies for arc consistency and constraint propagation: (1) parallelism at the level of constraints in the constraint queue using shared memory, and (2) distributed computation and message passing protocols for arc consistency algorithms.

Parallel AC algorithms. Constraint propagation can be easily parallelized by implementing the parallel versions of the sequential algorithms for node and arc consistency. In [139] Samal and Henderson present a parallel implementation for AC-3, and AC-4 called PAC-3, and PAC-4 respectively. The strategy adopted in these algorithms consist on the concurrent call of the *revise* function for all the arcs in the constraint queue (see Section 2.4). Algorithm 4 shows the pseudo-code for PAC-3. In algorithm 4, the statement

$$\textbf{forall } i \leftarrow 1 \textbf{ to } m \textbf{ do } \mathbf{f}(i)$$

is used to represent the execution of m parallel tasks; the $i^{\mathbf{th}}$ task executing $\mathbf{f}(i)$.

Algorithm 4 PAC-3(X, C)

```
1: $Q \leftarrow \{(x_i, c) \mid c \in C, x_i \in \textbf{scp}(c)\}$;
2: repeat
3:   forall $(x_i, c) \in Q$ do {Parallel loop.}
4:       if revise$(x_i, c)$ then
5:           add_arcs$(x_i, c)$;
6:       end if;
7:   end for;
8: until $Q = \emptyset$;
```

Both the procedures **revise** and **add_arcs** can be done in parallel; the former checks supports for all the labels in parallel (up to the square of the maximum size of the domains), while the latter updates the queue of constraints and checks whether a domain has become empty. If d is the maximum size of domains, up to d^2 processors are needed. The function **add_arcs** can be executed in constant time by representing the queue of constraints through a Boolean array of size e shared between the parallel processes, and a global flag to indicate an empty domain.

The algorithm requires up to ed^2 processors since there could be the case where all the e arcs are revised in parallel, and the **revise** functions has to check every possible combination of values for each arc. Let us observe that there are at most nd labels and at least one label is deleted in each *loop* iteration. Therefore, PAC-3 runs in $\mathcal{O}(nd)$ time, and its inherent parallelism is equal to $\mathcal{O}(ed^3)/\mathcal{O}(nd) = \mathcal{O}(ed^2/n)$.

Using a similar implementation strategy it is possible to show that PAC-4 reduces the complexity of AC-4 from $\mathcal{O}(ed^2)$ to $\mathcal{O}(ed)$, using $\mathcal{O}(d^2n^2)$ processors [139]. The inherent parallelism is $IP = \mathcal{O}(ed/n)$.

Other implementations of parallel constraint propagation schemes split the queue of constraint in small queues local to each parallel process [27]. Synchronization is used to update domains after each local propagation, possibly discovering inconsistencies. This version leads to higher speedups on average (up to 6×) but it could be more vulnerable to synchronization costs.

Distributed AC algorithms. A different strategy for parallelizing arc consistency is through distributed computation, i.e., performing concurrent propagation of constraints on a message-passing infrastructure (e.g., a network of workstations).

A distributed version of AC-4 for binary CSPs, referred to as *DisAC-4*, is presented in [124]. DisAC-4 is based on message passing protocols and variables that are distributed among concurrent agents. Each agent performs AC-4 on its local subset of variables, communicating with other agents as soon as a domain becomes empty. The system reaches a global fix-point when all the agents terminate their local AC-4. Experimental results show linear speedup w.r.t. the number of processors.

An improvement of DisAC-4 referred to as DisAC-9 is presented in [71]. The strategy adopted in DisAC-9 is similar to the one used for DisAC-4 but the algorithm is optimal w.r.t. the number of messages exchanged between agents, therefore, increasing the speedup (the authors show that DisAC-9 actually outperforms its sequential implementation).

4.3 Parallel Search in Constraint Programming

Parallel search in constraint programming has been investigated more extensively than parallel constraint propagation since, in general, search algorithms can be "easily" parallelized, outperforming their sequential implementation [142].

A common strategy for parallelizing the search process splits the search tree into many (disjoint) subtrees. Each subtree is then assigned to a different processor and the search process on the entire search tree is performed in parallel. This intuitive strategy requires special attention to ensure that the whole search space is partitioned in a balanced manner between parallel processes, in particular when the search process is based on backtracking techniques.

Some issues related parallel search strategies involving backtracking can be easily addressed considering the properties of the AND-OR structure of the search trees common in the context of logic programming [29]. Let us observe that the ideas are also applicable to the OR-trees common in constraint solving. An *AND-OR* tree is a tree representing the structure of a (logic) program where a node is either labeled as *AND*-node or *OR*-node: the former is solved once all its descendants are solved (in terms of logic unification), while the latter represents an alternative choice for a possible solution. This node classification is used to split the search process between independent paths, assigned to different processors and distinguishing respectively between *And-parallelism* or *OR-parallelism*. This schema avoids the overhead due to shared memory or communication contentions (e.g., copies of computation states between processors) but it also leads to duplicate work (e.g., when two paths share the same structure for at most of their length and they differ only at the bottom). A comprehensive survey of the issues arising in parallel execution of logic programming languages along with the most relevant approaches explored to date in the field can be found in [58]. The authors focus mostly on the challenges emerging from the parallel execution of *Prolog* programs, describing the major techniques used for shared memory implementation of And-parallelism, Or-parallelism, and combinations of the two.

Shared memory approaches define communication layers through shared memory between parallel processes, in order to create a "global view" of the current state of the search process. In [129] the author applies this technique on the top of the commercial solver ILOG [80]. The nodes of the search tree are split among the following three different sets: (1) the set of *open nodes* (or *search frontier*), (2) the set of *closed nodes*, and (3) the set of *unexplored nodes*. Shared memory is used to move nodes from one set to another. This framework is general enough to allow the user to implement many search algorithms and to assign different search nodes to different processors with relatively ease. Each processor performs its own sequential search on a portion of the tree and communicates with other processors through shared memory, ensuring load balancing and termination detection.

Constraint problems can be solved in a distributed environment by systems that combine constraint reasoning, concurrency, and message-passing communication protocols. The *Mozart* [166] implementation of *Oz* [34] is a concurrent language that supports distribution and programming of search engines from computation spaces [143]. Mozart provides a simple, high-level, and reusable design for parallel search on a network of computers [142].

Better performance are obtained by parallelizing local search strategies for solving constraint problems. The constraint solver COMET [113] explores the search space by a constraint-based local search strategy (see Section 2.6). Different initial solutions are distributed among different machine on a distributed system. Each parallel process therefore tries to improve its correspondent solution using local search techniques and the best solution among all processes is collected and imposed as global solution. A further level of parallelization is given by a transparent multi-core parallel search shared between machines. Each processor is associated to a *worker* agent. When a worker expands a node, it generates a set of new unexpanded nodes. These nodes are added to a central *pool*; workers that run out of work take unexpanded nodes from the pool, and in case of optimization problems they also communicate new improvements for the evaluation function. To keep the exploration of the search space consistent, communications from and to the central pool are synchronized among the workers. In [113] the authors report speedups about 3.41 and 3.08 with four machines.

4.3.1 Combining Parallel Search with Parallel Constraint Propagation

Parallelizing the entire solving process in a constraint programming environment is not an easy task since parallel search and parallel consistency typically suit different types of problems. If a problem is highly constrained, parallel search can slow down the solving process due to communication costs between parallel branches and resources issues. On the other hand, if a problem is highly constrained, there will be many inconsistent branches for which parallel search cannot be fully exploited [134].

In [134], the authors present a framework that combines parallel search with parallel constraint propagation. The strategy adopted by the authors is to distinguish between search threads and consistency threads and to associate every search thread several consistency threads. Search threads deal with variable assignments, while consistency threads are in charge of propagating constraints considering the current labeling performed by their corresponded search thread.

Example 4.3.1 *Figure 4.1 shows the combination of parallel search and parallel consistency within the same framework. The domain* $D^{x_1} = [0, 9]$ *is split between 2 parallel processors* p_1 *and* p_2*. Constraints* c_1, c_2 *involving* x_1 *are propagated in parallel after the assignment by* p_1 *and* p_2*, and two more available processors* p_3 *and* p_4*.*

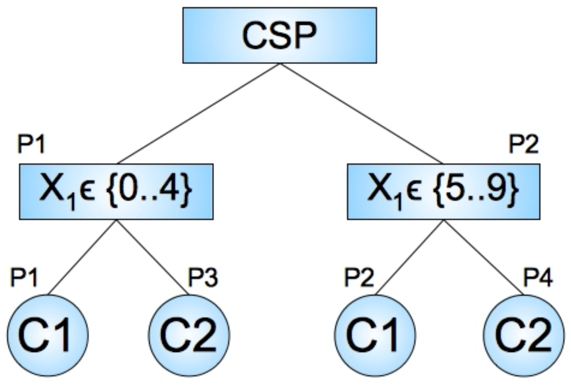


Figure 4.1: Example of parallel search combined with parallel constraint propagation.

The domain of each variable is divided among threads using a depth first search strategy. It follows that there is no data dependency between the different parts of the search space. Moreover, the set of constrains to propagate is split among consistency threads, rather then parallelizing the propagation of a single constraint. Note that search threads can be used as consistency threads as soon as they make an assignment, since consistency is performed after each labeling.

The experiments presented in [134] consider up to 8 parallel search and consistency threads on the *Sudoku* and *n-queens* problems. The speedups obtained are of 2 on average. The authors identify several drawback of their system design, in particular related to inefficiency in parallel consistency, synchronization of pruning, memory contention, and processor cache.

4.3.2 Distributed Constraint Programming

Here we present a slightly different view of parallel constraint solving, by introducing the notion of Distributed Constraint Satisfaction Problems [135]. The main difference from standard constraint satisfaction problems lies on the fact that every variable is controlled by a corresponding *agent*, and the solving process involves communication between agents, usually located on different machines. Agents do not have a global view of the problem or knowledge about the solving process as a whole. Instead, each agent holds private information and other agents only see information they are required to see for the solving process.

A distributed algorithm generally requires a high amount of messages exchange between agents, which slows down the overall solving process w.r.t. a centralized version. Nevertheless, there are a number of reasons why a distributed framework may be necessary, such as the dynamic nature of the problem (e.g., some agents may disappear while other may appear), privacy-security policies, fault-tolerant computations, etc.

A *Distributed Constraint Satisfaction Problem* (*DisCSP*) is a quadruple (X, D, C, A) defined as follows:

- $X = \langle x_1, \ldots, x_n \rangle$ is an n-tuple of variables;
- $D = \langle D^{x_1}, \ldots, D^{x_n} \rangle$ is an n-tuple of *finite* domains, each associated to a distinct variable in X;
- C is a finite set of constraints on variables in X, where a constraint c on the m variables $x_{i_1}, \ldots, x_{i_m}$, denoted as $c(x_{i_1}, \ldots, x_{i_m})$, is a relation $c(x_{i_1}, \ldots, x_{i_m}) \subseteq \times_{j=i_1}^{i_m} D^{x_j}$;
- $A = \{a_1, \ldots, a_m\}$ is a set of m *agents*.

Let us observe that the original definition of DisCSP includes a function that maps each variable to an agent. Here we assume an implicit mapping where each variable is mapped to exactly on agent and vice-versa. It is assumed that each agent a_i knows D^{x_i} and all constraints involving x_i. The definition of *Distributed Constraint Optimization Problem* (*DCOP*) is similar to the definition of DisCSP but with the set of constraint C representing a set of cost functions $c_1, \ldots, c_m$ s.t. for $0 \leq i \leq m$, $c_i : \times_{i=1}^{n} D^{x_i} \rightarrow \mathbb{R} \cup \{-\infty, +\infty\}$. The cost of a solution s for a DCOP is therefore calculated aggregating all the values given by the cost functions evaluated on s. An *optimal solution* is a solution with minimum cost.

Solving DisCSPs

A simple algorithm for solving DisCSPs is the *synchronous backtracking* search algorithm [177]. Synchronous backtracking is the distributed extension of the backtracking algorithm used for solving CSPs (see Sec. 2.2). More precisely, given an ordering among the agents (known to all agents), partial assignments are passed from agent to agent. Whenever an agent receives a partial solution from the previous agent, it instantiates its variable and it performs consistency. If no instantiation can satisfy its constraints, then it sends a *backtracking* message to the previous agent in the ordering. Otherwise it sends the extended partial solution to the next agent. While this algorithm preserves the simple schema of its sequential counterpart, it cannot take advantage of parallelism.

Asynchronous backtracking (*ABT*) [177] allows agents to run concurrently and asynchronously. It requires a *total* order among agents, inducing a direction in the constraints from *value sending* agent to *constraint-evaluating* agent, according with the order (see Figure 4.2).

Example 4.3.2 *Figure 4.2 shows a DisCSP with three agents x_1, x_2, and x_3, which is also the variable ordering. Domains are $D^{x_1} = \{1, 2\}, D^{x_2} = \{2\}, D^{x_3} = \{1, 2\}$, while the constraints are $x_1 \neq x_2, x_2 \neq x_3$.*

To ensure asynchronicity and completeness, each agent keeps two different data structures: (1) a set of values that the agent believes are assigned to higher priority agents (i.e., assignments of the variables held by value sending agents), referred to as *self* or *agent view*, and (2) a set of conjunctions of assignments variable-value that does not satisfy one or more constraint, refereed to as the set of *nogoods*. Two agents $x_i < x_j$ exchange four types of messages during search:

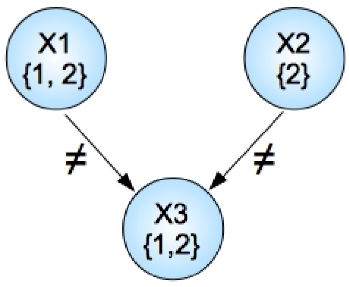


Figure 4.2: Example of Distributed CSP. x_1 and x_2 are *value-sending* agents, and x_3 is a *constraint-evaluating* agent.

- $OK?(i, j, val)$: agent x_i informs agent x_j that x_i has taken value val;
- $NGD(j, i, ng)$: agent x_j informs agent x_i that a nogood ng that involves x_i has been detected;
- $ADDL(i, j)$: agent x_i asks agent x_j to set up a link from x_j to x_i;
- $STOP(i, j)$: agent x_i informs agent x_j that there is no solution since an empty nogood has been generated.

When ABT starts, each agents chooses the assignment for its variables and communicates the choice to the low priority agents using an $OK?$ message. On the other hand, whenever an agent receives an $OK?$ message, first it updates its self view, it removes nogoods that are inconsistent with the new assignment and it checks for consistency with its new updated agent view. If an agent receives a NGD message, it checks if the nogoods are consistent with its current agent view. If it is the case it updates its nogood set and it searches for a new variable assignment. When the agent cannot find any value consistent with its self and nogood sets, new nogoods are generated and sent to the lowest higher priority agents in its agent view. $ADDL(i, j)$ messages are used whenever an agent x_i finds that is not connected with an agent x_j and x_j appears in a nogood for x_i.

Solving DCOPs

In DCOPs, agents communicate and coordinate while looking for an optimal solution through messages.

One of the reference algorithm for solving DCOPs optimally is the *Asynchronous Distributed OPTimization with quality guarantees* (*ADOPT*) algorithm [115]. ADOPT is a polynomial-space algorithms where, agents execute asynchronously and in parallel, finding the globally optimal solution. The key idea is to use a weak form of backtracking, namely, the backtrack condition is related to the value of the *lower bound* of the current assignment. Moreover, it uses an efficient reconstruction of abandoned solutions based on the notion of *backtrack threshold*. This requires polynomial space in the worst case, instead of exponential space as needed for memorizing partial solutions. Computational times of ADOPT can be improved by *BnB-ADOPT* [175] algorithm, where the optimal solution is obtained by propagating upper bounds on the quality of the solutions. In *BnB-ADOPT*, agents are organized in a DFS pseudo-tree, i.e., an arrangement of the constraint graph such that it does not contains cycles and variables involved in the same cost function appear in the same branch of the tree. Once the DFS pseudo-tree is defined, BnB-ADOPT performs an asynchronous depth-first-branch-and-bound search until an optimal solution is found. Three different messages are exchanged between two agents x_i, x_j:

- $VALUE(i, j, val, th)$: agent x_i informs agent x_j that it has assigned the value val to its variable x_i with a threshold th, representing the value used for pruning (see [175] for a formal definition of threshold).

- $COST(i, j, ctx, lb, ub)$: agent x_i informs agent x_j about its lower and upper bounds lb, ub for the current set of assignments ctx involving x_i's ancestors;
- $TERMINATE(i, j)$: agent x_i informs agent x_j about termination.

Each agent executes a loop: it first assigns a value to its variable, and then it sends a $VALUE$ message to each child and a $COST$ message to its parent. Based on these information items the other agents modify their assignment in order to reduce the difference between lower and upper bound. $TERMINATE$ message is sent from parents to children to inform that the optimal solution has been found, i.e., each agent has assigned the optimum value for its variable.

4.4 CUDA Computing

Modern graphic cards (*Graphics Processing Units*) are multiprocessor devices, offering hundreds of computing cores and a rich memory hierarchy for graphical processing (e.g., DirectX and OpenGL). Efforts like NVIDIA's *CUDA—Compute Unified Device Architecture* [140] aim at enabling the use of the multicores of a GPU to accelerate general applications—by providing programming models and APIs that enable the full programmability of the GPU. In this dissertation, we consider the CUDA programming model. The underlying conceptual model of parallelism supported by CUDA is *Single-Instruction Multiple-Thread (SIMT)*, a variant of the SIMD (Single-Instruction Multiple Data) model. In SIMT, the same instruction is executed by different threads that run on identical cores, while data and operands may differ from thread to thread. CUDA's architectural model is summarized in Figure 4.3.

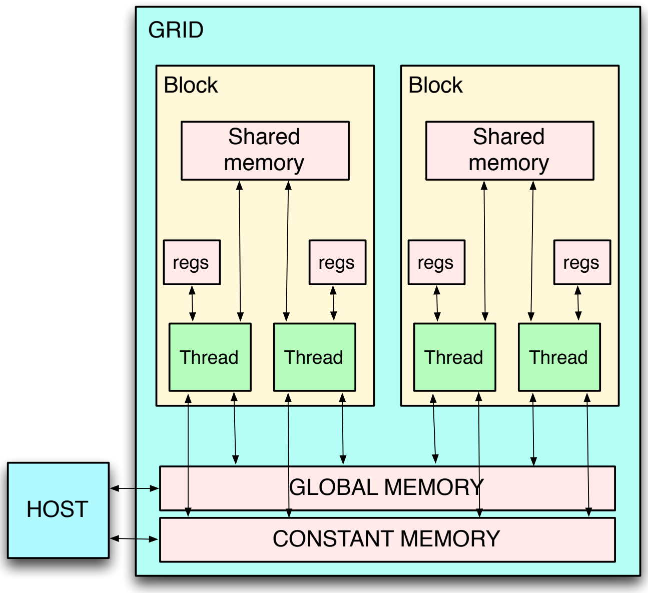


Figure 4.3: CUDA Logical Architecture.

Different NVIDIA GPUs provide varying numbers of cores, their organization, and amounts of memory. The GPU is constituted by a series of *Streaming Multi-Processors (SMs)*; the number of SMs depends on the specific class of GPUs—e.g., the Fermi architecture provides 16 SMs. In turn, each SM contains a number of computing cores (each with a fully pipelined ALU and floating-point unit); the number of cores per SM may range from 8 (in the older G80 platforms) to 32 (e.g., in the Fermi platforms). Each GPU provides access to on-chip memory (for thread registers and shared memory) and off-chip memory (L2 cache, global memory and constant memory).

CUDA introduces a logical view of computations, allowing programmers to define abstract parallel work and to schedule it among different hardware configurations (see Fig. 4.3). A typical CUDA program is a C/C++ program that includes parts meant for execution on the CPU (referred to as the *host*) and parts meant for parallel execution on the GPU (referred as the *device*). A parallel computation is described by a collection of *kernels*. Each kernel is a function to be executed by several threads. Threads spawned on the device to execute a kernel are hierarchically organized to facilitate the mapping of the threads to the (possibly multi-dimensional) data structures being processed: threads are organized in a 3-dimensional structure (called *block*), and blocks themselves are organized in 2-dimensional tables (called *grids*). CUDA maps blocks (coarse-grain parallelism) to the SMs for execution; each SM schedules the threads in a block (fine-grain parallelism) on its computing cores in chunks of 32 threads at a time (called *warps*), thus allowing group of threads in a block to use the computing resources while other threads of the same block might be waiting for information (e.g., completing a slow memory request). Threads have access to several memory levels, each with different properties in terms of speed, organization (e.g., banks that can be concurrently accessed) and capacity. Each thread stores its private variables in very fast registers (anywhere from 8K to 64K per SM); threads within a block can communicate by reading and writing a common area of memory (called *shared memory*). Communication between blocks and with the host is realized through a large *global memory* (up to several gigabytes).

The kernel, invoked by the host, is executed by the device and it is written in standard C-code. The number of running blocks (gridDim) and the number of threads of each block (blockDim) are specified in the call executing the kernel, with the following syntax:

$$\mathsf{Kernel} \lll \mathsf{gridDim},\ \mathsf{blockDim} \ggg (\mathsf{param}_1,\ \ldots,\ \mathsf{param}_n);$$

In order to perform a computation on the GPU, it is possible to move data between the host memory and the device memory. By using the specific identifier of each block (blockIdx—providing x, y coordinates of the block in the grid), its dimension (blockDim) and the identifier of each thread (threadIdx—providing x, y, z coordinates for the thread within the block), it is possible to differentiate the data accessed by each thread and the corresponding code to be executed. For example, the following code fragment shows a kernel and the corresponding call from the host. Each element of a two dimensional matrix is squared, and each thread is in charge of one element of the matrix. The matrix A is represented by a pointer in the device's global memory. CUDA provides functions (e.g., cudaMemCopy) to transfer data between the host and the device's global memory.

```
int main() {                              __global__ sqMatrix(float *Mat){
   ...                                        int i=threadIdx.x;
    dim3 thrsBlock(n,n);                      int j=threadIdx.y;
    sqMatrix<<<1,thrsBlock>>>(A);             Mat[i][j] = Mat[i][j]*Mat[i][j];
    ...                                   }
```

While it is relatively simple to develop correct CUDA programs (e.g., by incrementally modifying an existing sequential program), it is challenging to design an efficient solution. Several factors are critical in gaining performance. The SIMT model requires active threads in a warp to execute the same instruction—thus, diverging flow paths among threads may reduce the amount of actual concurrency. Memory levels have significantly different sizes (e.g., registers are in the order of dozens per thread, while shared memory is in the order of a few kilobytes per block) and access times; different cache behaviors are applied to different memory levels (e.g., constant memory is a cached read-only global memory) and various optimization techniques are used (e.g., accesses to consecutive global memory locations by contiguous threads can be *coalesced* into a single memory transaction).

4.4.1 Parallel Search on GPU

Several studies have addressed the problem of parallelizing the search process using GPU architectures obtaining different results. Naive implementations of well-known search algorithms such as *depth first search* or *breadth first search* on GPU often perform poorly [85, 106], whereas exploiting

GPU parallelism for local search strategies speed ups the search of several factors, and solves large scale problems.

In [106], the authors present a GPU implementation of the breadth first search (BFS) algorithm achieving up to 10× speed up. The algorithm considers a standard BFS but instead of exploring the queue of nodes sequentially, it explores all the nodes in the frontier in parallel (i.e., parallel exploration of each level of the search tree).

In order to avoid sequential computation due to the insertion of the nodes in the queue, the authors define two different types of queue: one queue local to each block (implemented in shared memory), and one queue shared among all blocks of the kernel (implemented in global memory). Local queues are explored in parallel by each thread, and the whole frontier is then copied on global memory by each block according to specific offsets. This strategy avoids collisions between threads of different blocks, ensuring parallel writes into global memory.

Parallel depth first search on GPU is not promising as parallel BFS. In [85], the authors report on the investigation of backtracking paradigms on GPUs. The authors propose two levels of parallalelization: (1) a *tree-level* parallelization assigning different kernel blocks to different subtrees of the search three, and (2) a *node-level* parallelization where multiple threads expand the nodes of each subtree. Despite of a double level of parallelization, the GPU implementation performs poorly w.r.t. the corresponded CPU implementation: different sub trees have different structures and, therefore, divergent computational paths, forcing threads to a sequential computation.

Local Search on GPU. Local search strategies move from solution to solution trying to maximize a criterion among a number of candidate solutions. These iterative methods can exploit the computational power of GPUs in order to perform massively parallel exploration of the search space, without being constrained to complete-search schemes.

A guideline for design and implementation of LS strategies on GPUs is presented in [107, 163]. The authors describe a general methodology for implementing local search methods on GPU: parallelism is exploited at the iteration level (i.e., parallel evaluation of neighborhoods), while CPU performs the iterative process of selecting and updating the current solution with the specific local search strategy. Three major GPU encodings for neighborhood representation are considered:

- *Binary representation*: this representation is used for binary problems (i.e., binary domains) mapping each neighborhood to a different set of GPU threads according to a given Hamming distance (the *Hamming distance* between two strings of equal length is the number of positions at which the corresponding symbols are different).
- *Discrete vector representation*: this representation extends the binary one to an alphabet of size greater than two. The mapping between GPU threads and neighborhood is similar to the one used for binary representations considering a Hamming distance of 1.
- *Permutation representation*: this representation identifies neighborhoods by a pairwise exchanging of values. Since a permutation can be identified by the two indexes of the swapped elements, threads are mapped to permutations by a bijective function $\mathbb{N} \to \mathbb{N} \times \mathbb{N}$. Therefore, each thread identifier is used to calculate the pair of values to swap within a neighborhood.

Let us observe that it is possible to identify three general parallel models for local search methods (see Figure 4.4):

- *solution-level*: parallel evaluation of a solution;
- *iteration-level*: parallel exploration of a neighborhood;
- *algorithmic-level*: parallel exploration of the search space through different local search strategies.

It is possible to combine the three models in order to obtain a higher level of parallelization, at the cost of a more complex framework and several classes of new potential software bugs (e.g., synchronization between the models).

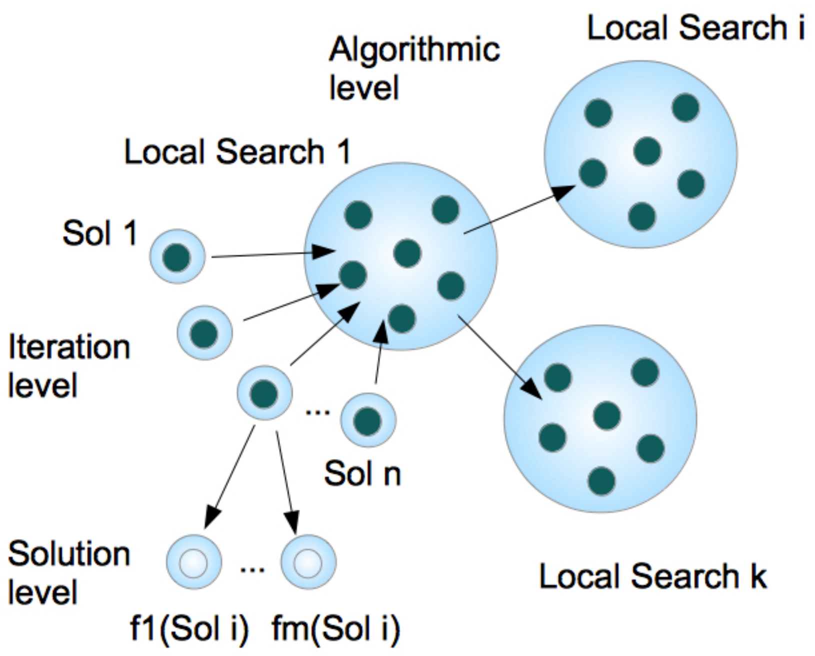


Figure 4.4: Parallel models for local search methods.

Local Search for Constraint Problems. There are few studies in literature about the use GPU architecture and constraint programming. In [7] the authors present a GPU implementation of a parallel local search strategy for solving CSPs and COPs. Parallelism is exploited at two different levels: (1) *solution level*, where multiple copies of the search process run at the same time, each associated to a different CUDA block, and (2) *iteration-level*, where the threads of each block explore the neighborhood in parallel.

Local search guides the search process considering the number of violated constraints. This value is calculated using an *error function* that takes a complete assignment of variables and returns the number of violated constraint. To reduce the error value a min-conflict heuristic is applied (see Sec. 2.6). Let us observe that it is not guaranteed that the assignment found by the iterative process satisfies all the constraints (i.e., it represents a solution). Speed ups mostly depend on the exploitation of solution level parallelism and they vary from $3\times$ to $17\times$.

5

GPU-based Propagation

This chapter presents an experimental study aimed at assessing the feasibility of parallelizing constraint propagation—with particular focus on arc-consistency—using GPUs. GPUs support the form of data parallelism (i.e., SIMT parallelism) that appears to be suitable to the type of processing required to cycle through constraints and domain values during consistency checking and propagation. We describe an implementation of a constraint solver capable of hybrid propagations (i.e., alternating CPU and GPU), and demonstrate the potential for competitiveness against sequential implementations.

The choice of SIMT parallelism has two driving motivations. First of all, it is our belief that this form of parallelism is suitable to the type of processing that constraints are subjected to during consistency checking. Second, SIMT is the style of parallelism that is natively supported by moderns GPUs which provide hundreds of computing cores at an affordable cost. Exploiting the parallelism offered by GPUs is not trivial—the cores are often significantly slower than CPU cores, they impose restrictions on branching, and provide a complex memory hierarchy with differences in speed, size, and concurrency of accesses.

In this chapter we propose a methodology to map constraints, variables, and domain elements to threads running on GPU cores, thus enabling the concurrent analysis of arc and bound-consistency and removal of inconsistent domain values. The methodology is implemented in an experimental solver, and shown to produce performance enhancements even in its simple and unoptimized form. The prototype demonstrates also the strengths and weaknesses of GPU parallelism in constraint solving.

5.1 The Framework

CSP solvers (e.g., Algorithm 5) alternate labeling, and constraint propagation to reduce the set of admissible values of the variables and possibly detect inconsistencies (see Section 2.2). Thus, at the core of a CSP solver there is a constraint propagation engine, that repeatedly propagates information based on the available constraints; its basic component s a function, from domains to domains, referred to as *propagator* [142].

Given two n-tuples of domains D_1 and D_2, we say that $D_1 \sqsubseteq D_2$ if, $\forall x \in X$, it holds that $D_1^x \subseteq D_2^x$. A *propagator* f is a monotonically decreasing function: $f(D) \sqsubseteq D$ and $f(D_1) \sqsubseteq f(D_2)$ whenever $D_1 \sqsubseteq D_2$. Each constraint $c \in C$ is implemented by a set of propagators prop(c) that operate on the m-tuple of domains of the variables in scp(c). We denote by $\mathcal{F}$ the set of all propagators considered. If $f(D) = D' \wedge D' = D$ for all $f \in \mathcal{F}$ then D' is a *fixpoint* of $\mathcal{F}$. A *propagation solver* **i-solv** for a set of propagators $\mathcal{F}$ and an initial domain D finds the greatest fixpoint of $\mathcal{F}$. **i-solv** starts its computation from a subset $F_0 \subseteq \mathcal{F}$ of propagators and the current domains.

The procedure **i-solv** (Algorithm 6) iteratively invokes the propagators until the greatest fixpoint is reached. Two general decisions have to be made in order to reach the fixpoint: *(1)* Which propagators should be executed, and *(2)* In which order they should be executed. These decisions are based on the notion of *events:* an event is a change in the domain of a variable. We distinguish five types of events: **(1)** ***failed_event***: there is a variable x such that $D'^x = \emptyset$. **(2)** ***empty_event***:

Algorithm 5 **search**(X, D, C, ℓ)

```
if $\ell > |X|$ then
    output $D$;
    return true;
end if
for all $d$ in $D^{x_\ell}$ do
    $D' \leftarrow \langle D^{x_1}, \dots, D^{x_{\ell-1}}, \{d\}, D^{x_{\ell+1}}, \dots, D^{x_{|X|}} \rangle$;
    $F_0 \leftarrow \{\mathbf{prop}(c) : c \in C \wedge x_\ell \in \mathbf{scp}(c)\}$;
    if i-solv$(F_0, D') \wedge$ search$(X, D', C, \ell + 1)$ then
        return true;
    end if
end for
return false;
```

no event happened, i.e., $D'^x = D^x$ for all variables considered. **(3)** ***sing_event***: there is a variable x such that $|D'^x| = 1$. **(4)** ***bc_event***: there is a variable x such that $\min D'^x > \min D^x$ or $\max D'^x < \max D^x$. **(5)** ***dmc_event***: there is a variable x such that $D'^x \subset D^x$. These events are used to invoke the necessary propagators only, based on the changes to the variables' domains that occurred.

Algorithm 6 **i-solv**(Q,D)

```
$D' \leftarrow D$;
while $Q \neq \emptyset$ do
    for all $f \in Q$ do
        $D'' \leftarrow f(D)$;
        if failed_event then return false; end if
        $D \leftarrow D''$;
    end for
    $Q \leftarrow$ new$(Q, D', D'')$;
end while
return true;
```

The pseudo code in Algorithm 6 is similar to the well-known *AC-3* algorithm (Section 2.2): the **while** loop (lines 2–9) propagates the constraints in the queue of propagators Q until no changes happen in the domains, i.e., D is a *fixpoint* for the propagators invoked, or some domain is empty. The procedure **new**(Q, D', D'') chooses the new propagators to be inserted in the queue, based on the changes between the original domain D' and the final domain D'' and on the propagators already in Q. As a side-effect, the procedure modifies the values of the calling domain variable in the **search** procedure.

5.2 Parallelizing the Constraint Engine

In this section we describe our approach to GPU-based execution of the **i-solv** procedure presented in Section 5.1. The corresponding pseudo-code is reported in Algorithm 7.

Our model encodes three different types of parallelism for constraint propagation. Recall that constraint propagation is monotonic, therefore the order in which the data is analyzed does not influence the result (while it might affect the number of operations performed to reach the fixpoint).

`Constraints`: Given a set C of constraints for which propagation and consistency checks are to be performed, a natural form of parallelism is to delegate the processing of each constraint $c \in C$ to a different parallel computation. In particular, it is convenient to map a block of threads (B_c) to the handling of each c, in order to exploit the various parallel GPU's SMs.

A kernel with a number of blocks of the size of the current constraint queue C is invoked. Up to 2^{32} blocks can be used on NVIDIA 2.x cards.

`Variables`: A second level of parallelism is applied to the processing of a constraint c assigned to a block B_c. Domain reductions for the variables involved in the constraint (namely $x \in scp(c)$) can be performed in a parallel fashion. In particular, each variable can be handled by a different thread that executes the domain filtering. This level of parallelization is suitable to global constraints, such as *element*, *inverse*, or *table* constraint—while it would not bring benefit to constraints that admit efficient propagation algorithms.

`CPU and GPU`: Host and device are capable of independent and parallel work, that can be synchronized by specific programming constructs. We designed a third level of parallelism for constraint propagation, by partitioning the set of propagators in two queues: one to be processed by the CPU and another one by the GPU. Constraints with efficient propagators (e.g. small scope), are executed on the host, while the others are delegated to the GPU. During the evolution of the propagation, exchanges of information between host and device ensure to reach the fixpoint faster.

Let us describe the main components of Algorithm 7.

Algorithm 7 i-solv(F_0, D)

```
1: $T \leftarrow \mathbf{max}\{|scp(c)| : c \in C\}$;
2: $\langle \mathbf{Q_{host}}, \mathbf{Q_{dev}} \rangle \leftarrow \mathbf{split}(F_0)$;
3: while $\mathbf{Q_{host}} \cup \mathbf{Q_{dev}} \neq \emptyset$ do
4:   if $\mathbf{Q_{dev}} \neq \emptyset$ then
5:     cudaMemcpy$(D_{\mathbf{dev}}, D)$;
6:     gpu_propagate$<<< |\mathbf{Q_{dev}}|, T >>>$ $(\mathbf{Q_{dev}}, D_{\mathbf{dev}})$;
7:     cudaMemcpy$(D', D_{\mathbf{dev}})$;
8:     if failed_event then return false; end if
9:   end if
10:  if $\mathbf{Q_{host}} \neq \emptyset$ then
11:    for $f \in \mathbf{Q_{host}}$ do
12:      $D'' \leftarrow$ cpu_propagate$(f, D)$;
13:      if failed_event then return false; end if
14:    end for
15:  end if
16:  $D_{\mathbf{aux}} \leftarrow D; D \leftarrow D' \cap D''$;
17:  $\langle \mathbf{Q_{host}}, \mathbf{Q_{dev}} \rangle \leftarrow \mathbf{split}(\mathbf{props}(D, D_{\mathbf{aux}}, \mathbf{Q_{host}} \cup \mathbf{Q_{dev}}))$;
18: end while
19: return true;
```

At each invocation of the **i-solv** procedure, the set of initial propagators F_0 is split between host and device by the function **split** that initializes the queues of constraints $\mathbf{Q_{host}}$ and $\mathbf{Q_{dev}}$ (host and device constraints), based on the type of constraints to be propagated in line 2. The default distribution, based uniquely on the type, can be changed by the **split** function according to two internal thresholds: **(1)** If the number of CPU-propagators is higher than a given *upper bound*, they are all moved to $\mathbf{Q_{dev}}$; **(2)** If the number of GPU-propagators is lower than a given *lower bound*, they are moved to $\mathbf{Q_{host}}$.

By varying these bounds, it is possible to force the computation completely on the CPU (huge lower bound) or completely on the GPU (upper bound = 0). These bounds are used to handle the cases where a large number of efficient propagators are assigned to the CPU, while they could take advantage of parallel propagation or, vice-versa, very few expensive propagators are assigned to the GPU, where the time required by memory transactions between host and device would likely offset the advantages of a parallel propagation. The only exception to these rules is for complex constraints (such as the *table* constraint) that are always delegated to the GPU.

Every loop iteration analyzes and modifies the propagators in $\mathbf{Q_{host}}$ and in $\mathbf{Q_{dev}}$. If $\mathbf{Q_{dev}}$ is not empty, parallel propagation is performed by invoking the kernel **gpu_propagate** (line 6), with as many blocks as the size of $\mathbf{Q_{dev}}$, and as many threads per block as the maximum scope size among all constraints. The kernel function **gpu_propagate** is sketched in Algorithm 8 and explained later. If $\mathbf{Q_{host}}$ is not empty sequential propagation is performed by invoking the function **cpu_propagate** (line 12). If both propagations succeed, the new states D' and D'', produced respectively by the GPU and the CPU, are merged (line 16) and the function **props**() determines the minimal sets of propagators that are not at their fixpoint for the domain D (line 17). The function **props**() is based on the notion of *events*. It calculates the events based on status $D_{\mathbf{aux}}$ of the previous iteration and the current status D ($evts(D, D_{\mathbf{aux}})$), and updates the queue of propagators accordingly:

$$\mathbf{props}(D, D_{\mathbf{aux}}, Q) = \{f \in \mathcal{F} \,:\, evt\_set(f) \cap evts(D, D_{\mathbf{aux}}) \neq \emptyset\} \setminus \mathtt{fix}(Q, D)$$

where the set $evt\_set(f)$ is the set of events related to the propagator f, and $\mathtt{fix}(Q, D) = \{f \in Q \,:\, f(D) = D\}$. This set of events is computed by analyzing the differences between D and $D_{\mathbf{aux}}$.

Algorithm 8 **gpu_propagate**(Q, D)

```
1: $c\_id \leftarrow Q[blockIdx]$;
2: $\mathbf{get\_propagators}[\mathbf{get\_type}(c\_id)](c\_id, D)$;
```

Let us briefly discuss Algorithm 8. This kernel invokes a propagator per block. The identifier of the block (*blockIdx*) is used as index on the queue Q to retrieve the identifier c_id of the constraint to propagate. The function **get_propagators** returns a pointer to the device function that implements the (set of) propagators for the constraint c indexed by its type **get_type**(c_id). The constraint identifier c_id is also used by the propagator to identify the scope and any parameters of the constraint to propagate. The relationships between constraints and variables (constraint graph) is stored in the device memory, to limit the information exchange between CPU and GPU. A **failed_event** is generated when there is an empty domain. If this is the case, then the propagation will fail and the **i-solve** procedure will return **false**; this will cause the search to backtrack (line 9).

The propagation on the host is similar; the kernel invocation is replaced by a *for* loop that iterates over all the propagators in $\mathbf{Q_{host}}$ (lines 12-15). Let us note that, differently from the propagation on the device, the **failed_event** is checked every time a propagator has been considered. Let us discuss some details related to the CPU and GPU implementations of these algorithms.

5.2.1 Domain representation

Domains are represented using *bit-masks* stored in k *unsigned int* 32-bit variables. More precisely, we restrict the domains of the variables to subsets of $\{0, \ldots, 32k-1\}$, therefore representing each domain D as $\sum_{i=0}^{32k-1} 2^i b_i$, where if $i \in D$ then $b_i = 1$, else $b_i = 0$. Negative numbers can be implemented using an appropriate offset value. Three extra variables are used: two for storing the domain bounds ($\min D$ and $\max D$) and one for storing the current event associated to D. We denote with $M = k + 3$ the number of variables used. For instance, for storing domains included in [0..927] we use $M = 32$ unsigned int variables.

5.2.2 State of the computation

The state of the computation at every node of the search tree is represented by a vector of $M \cdot |V|$ where M is as described above. This representation of the status reduces the total number of accesses to the global memory, since every consecutive 32 domain values are grouped together in a single *integer* value. The choice of M as a multiple of 32 *integers* allows us to take advantage of the device cache, since global memory accesses are cached and served as part of 128-byte memory transactions. Moreover, using the same array of data for both the bit-mask and the domain bounds increases the *coalesced* memory accesses, i.e., the accesses to the global memory are coalesced for contiguous locations in global memory, increasing access performance.

5.2.3 Data transfers

The memory dataflow is designed in order to optimize memory throughput. Since applications should strive to minimize data transfers between the host and the device (i.e., data transfers with low bandwidth), at each parallel propagation step we transfer the minimum information needed to represent the current state in the search tree. Namely, we copy into the global memory of the GPU the previous decisions performed in the current exploration of the search tree, and only the domains of the variables that have not been assigned yet. These domains still ensure a correct execution of the propagation algorithm, as we are interested in reducing only the domains of the variables that have not been labeled yet. In order to allow concurrent computations on the host and the device, every **cudaMemcpy** is performed as an asynchronous data transfer. A call to the CUDA function **cudaDeviceSynchronize()**, used to synchronize the host and the device, is requested only when the CPU has finished its sequential propagation.

5.2.4 MiniZinc constraints encoding

In this work we considered the finite domain constraints that are available in the *MiniZinc/FlatZinc* modeling language [120].

MiniZinc is a medium-level constraint modelling language. MiniZinc allows one to express most CSPs/COPs with relatively easy, but is low-level enough that it can be interfaced easily to different (existing) backend solvers. This is done by transforming the constraint problem expressed using MiniZinc syntax into a FlatZinc model. FlatZinc models consist of variable declaration, constraint definitions and a definition of the objective function if the problem is an optimization problem. The translation from MiniZinc to FlatZinc can be specialized by the backend solvers in order to control what form of constraints end up in. For example, MiniZinc allows the specification of global constraint by decomposition.

A MiniZinc problem specification has two parts: (1) the model, which describes the structure of a class problem; and (2) the data, which specifies one particular problem within this class. The pairing of a model with a particular data set is a *model instance.*

Example 5.2.1 *The n-Queens problem can be modelled in MiniZinc as follows:*

```
int: n;
array[1..n] of var 1..n: queens;

constraint all_different(queens);
constraint all_different([queens[i]+i | i in 1..n]) :: domain;
constraint all_different([queens[i]-i | i in 1..n]) :: domain;

solve satisfy;
```

The `all_different` constraint is defined as:

$$\texttt{all\_different}(x_1, \ldots, x_n) = \{(d_1, \ldots, d_n) \mid d_i \in D^{x_i}, d_i \neq d_j \textbf{ for } i \neq j\}$$

where $x_1, \ldots, x_n$ are variables with respective finite domains $D^{x_1}, \ldots, D^{x_n}$.

Given $n = 20$, the above MiniZinc model is then translated in the following corresponding FlatZinc model:

```
array [1..20] of var 1..20: queens;
constraint int_lin_ne([1, -1], [queens[1], queens[2]], -1);
constraint int_lin_ne([1, -1], [queens[1], queens[2]], 1);
...
```

```
constraint int_ne(queens[1], queens[2]);
constraint int_ne(queens[1], queens[3]);
...
solve satisfy;
```

Given a MiniZinc model, we translate it and produce an input for our solver in three steps:

1. We read the MiniZinc file to identify the global constraints being used;
2. We translate the model into a FlatZinc model without considering the global constraints (we use the compiler available in the MiniZinc distribution);
3. The FlatZinc translation is given as input to a parser that produces the input for the solver.

5.2.5 Propagators

We have implemented the propagators for the FlatZinc constraints plus specific propagators for some *global* constraints that take advantage of GPU parallelism. As described earlier, every propagator is implemented as a specific device function invoked by a single block. For example, let us consider an *all_different* constraint c on the variables $x_1, \ldots, x_n$, naively encoded as a quadratic number of binary $\neq$ constraints. It can be implemented by a set of n propagators $p_1, \ldots, p_n$, such that the propagator p_i takes care of the constraints $x_i \neq x_j$ where $j \neq i$ (see Algorithm 9). The propagator is typically activated for one i at a time. A sequential implementation of this propagator requires time $\mathcal{O}(n)$, while the parallel version requires $\mathcal{O}(1)$.

Algorithm 9 $p_i(c\_id, D)$

```
1: $x_i \leftarrow \mathbf{scp}(\text{c\_id})[i]$;
2: $label \leftarrow min\ D^{x_i}$; $\{min\ D^{x_i} = max\ D^{x_i}$ since $x_i$ is the current labeled variable$\}$
3: $n \leftarrow \mathbf{scp(}c\_id\mathbf{).size}()$; {Constraints information on device global memory}
4: if $threadIdx < n \wedge threadIdx \neq i$ then
5:     $temp \leftarrow \mathbf{scp}(\text{c\_id})[threadIdx]$;
6:     $D^{temp}[label] \leftarrow 0$;
7: end if
```

Some other constraints require more than one block to fully exploit the parallel computation. This is the case, for example, of the *table* constraint (see Sec. 5.3). To handle these cases, we modified Algorithm 7 in order to further split the queue $\mathbf{Q_{dev}}$ in two queues: one for constraints that are propagated using one block per propagator, and one for constraints that use more than one block. This information is stored in a lookup table, accessed any time the constraint queue must be filled with the current constraints to propagate. The solver first runs the propagators for the constraints that require one block per constraint in order to find possible inconsistencies. If no inconsistency is found, the solver runs the propagators for the constraints that require more than one block (one kernel call per constraint).

5.3 Results

We experimentally evaluated our solver using several classical benchmarks. Benchmarks are encoded in MiniZinc and compiled automatically by the solver. In particular, we compare the performance of our solver (in terms of execution time) with two state-of-the-art solvers, namely *Gecode* [62] and *JaCoP* [82]. Our solver does not include advanced search strategies at this time—therefore, for a fair comparison, we use Gecode and JaCoP with a naive "leftmost" strategy with increasing value assignment. In order to measure parallel performance, we analyze the speed-ups and limitations of the GPU version against a purely CPU execution of our code—as mentioned earlier, this can be realized by modifying the bounds used to manage the constraint queues. Thus, while the first set of comparisons gives us an idea about the baseline performance of our core solver (including an indication of the overhead introduced to support parallelism), the second set of data measures the improvements gained by using parallelism. We have aimed at creating a core solver that is efficient and competitive with the state-of-the-art in constraint solving, containing overhead to the minimum. All tests have been performed on the following hardware: the *Host* is an AMD Opteron 270, 2.01GHz, RAM 4GB, while the *Device* is an NVIDIA GeForce GTS 450, 192 cores (4MP), Processor Clock 1.566GHz, OS Linux.

5.3.1 Comparison with Gecode and JaCoP

We start by evaluating the performance of our solver w.r.t. the solvers Gecode and JaCoP on some classical benchmarks, specifically *n-Queens*, *Schur* (numbers $1, \dots, N$ in B blocks), and the *propagation stress* benchmarks (see, e.g., the MiniZinc benchmarks folder [120]). Let us remark that the *all_different* constraints is implemented in a "quadratic way" in all these problem instances—this explains the relatively slow running times for n-Queens. As expected, there are instances that better fit one solver, and other instances that better fit others (see Tables 5.1, 5.2, and 5.3—running times in seconds). We report the times (sec.) for the sequential (CPU column) and the parallel version of our solver (GPU column). For this experiment, let us focus on the GPU column (the CPU column is used in the following experiments). or a fair comparison, we modified the hybrid and adaptive recomputation parameters of Gecode. In particular we switched off *cloning* (i.e., the option that allows the solver to store search nodes during the solving process instead of recomputing them to speed up backtracking) by setting the value c_d (*commit distance*) greater than the expected depth of the search tree.

Let us observe that the solver we are proposing is, on average, comparable with the state-of-the-art.

N	CPU	GPU	Gecode	JaCoP
24	6.273	9.699	7.094	47.59
26	5.975	8.773	7.438	47.55
28	50.88	68.47	66.88	442.6
30	930.3	1278	1407	9600

Table 5.1: Comparison between **i-solv** (sequential (CPU) and parallel (GPU) versions), Gecode, and JaCoP for the *n-Queens* benchmarks

N	B	CPU	GPU	Gecode	JaCoP
40	4	88.59	84.75	19.02	2.570
41	4	92.92	90.71	19.54	2.610
42	4	97.03	95.41	20.54	2.700
43	4	108.4	98.75	21.35	2.850

Table 5.2: Comparison between **i-solv** (sequential (CPU) and parallel (GPU) versions), Gecode, and JaCoP for the *Schur* benchmark

5.3.2 Comparing GPU vs CPU

In this section we compare the GPU parallel version of the solver w.r.t. a purely sequential version. The core of the propagators are implemented in the same way (i.e., they use the same C encoding). The main drawbacks of the GPU computations are primarily related to data transfers, due to the GPU memory latency and coalesced access patterns, and to the difference between the GPU clock and the CPU clock.

k	n	m	CPU	GPU	Gecode	JaCoP
10	20	200	0.043	0.053	0.696	2.550
10	20	300	0.068	0.082	1.740	4.730
10	20	400	0.175	0.159	3.155	8.460
10	20	500	0.339	0.306	4.968	13.94

Table 5.3: Comparison between **i-solv** (sequential (CPU) and parallel (GPU) versions), Gecode, and JaCoP for the *propagation stress* benchmark.

Table 5.1, Table 5.2, and Table 5.3 show that the running times are comparable for the sequential and parallel executions. Similar considerations hold for other "small" instances. We used the upper bound (UB) parameter to move constraints from the host queue to the device queue. UB is calculated empirically, and it is automatically set by the solver in a preprocessing step, by considering the average numbers of global memory accesses w.r.t. the type of propagators involved in the model. For example, if there is an average of 3 memory accesses for each propagator, and each propagator requires $\mathcal{O}(1)$ time, then the upper bound will be set to at least 900, since each global memory access requires about 300 clock cycles. Table 5.4 shows how the UB affects the execution time on the *Golomb ruler* problem for a ruler of 20 integers. Notice that the solver with an appropriate upper bound performs better than both the CPU and the GPU without upper bound (UB = 0, all constraints propagated on device). The model comprise both $\mathcal{O}(1)$ and $\mathcal{O}(n)$ propagators.

CPU	UB = 0	UB = 100	UB = 500	UB = 1000	UB = 1500	UB = 2000
266.4	223.4	216.4	214.2	210.4	207.8	208.2

Table 5.4: Influence of the upper bound parameter on the *Golomb ruler* problem.

Significant performance improvements emerge when more complex constraints are considered. As explained in Section 5.2, the GPU is delegated to large sets of non trivial propagators. Using the CUDA framework, the CPU and the GPU can execute concurrently, since the kernels and the memory copy operations between host and device can be performed asynchronously. Let us focus on two "expensive" constraints, namely the *inverse* and the combinatorial *table* constraint.

5.3.3 The *inverse* constraint

This constraint ties two arrays of variables using the global *inverse* property. Given two lists $X = [x_1, \dots, x_n]$ and $Y = [y_1, \dots, y_n]$ of integer variables, where $D^{x_i} = D^{y_i} = [1..n]$, the constraint *inverse*(X, Y) holds iff $(\forall i \in [1..n])(\forall j \in [1..n])(x_i = j \leftrightarrow y_j = i)$. The FlatZinc implementation of this constraint uses n^2 Boolean variables and $2n^2$ *reified equality* constraints:

$$\bigwedge_{i,j} x_i = j \leftrightarrow B_{ij} \quad \wedge \quad \bigwedge_{i,j} y_j = i \leftrightarrow B_{ij}$$

The GPU version of this constraint is implemented by $2n$ propagators. Namely, n propagators are used for the "$\rightarrow$" (resp., "$\leftarrow$") direction of the constraint, considering the labeling of one variable in X (resp., in Y). Since we expand the relation $x_i = j \leftrightarrow y_j = i$ either on the left or the right side depending on the labeled variable, we do not need to explicitly use the Boolean variables B_{ij} to link the binary equality constraints. These constraints are propagated by n threads. For example, let us assume that $x_1 = 2$ after the labeling of x_1; the constraint engine invokes the propagator *inverse*(x_1, Y) where the thread whose ***threadIdx*** $= 2$ propagates the constraint $y_2 = 1$ (i.e, $B_{12} =$ **true**), while the other threads propagate the constraints $y_i \neq 1$, where $i \in \{1, 3, 4, \dots, n-1, n\}$.

Table 5.5 compares the sequential and the parallel implementations of the *inverse* constraints, by increasing the number n of variables in its scope.

For $n = 100$ there is a poor speedup, since the CPU cores are faster than the GPU cores and the instance of the problem is small. The speedup increases for bigger instances (i.e., $n > 200$)

n	CPU	GPU	Speedup
100	0.030	0.026	1.15
250	0.338	0.152	2.22
500	2.456	0.744	3.30
750	7.855	2.142	3.66

Table 5.5: Time comparison for the *inverse* constraint.

where the parallel computations offset the difference of speed between CPU and GPU cores. We have verified that the FlatZinc encoding of the *inverse* constraint is sensibly slower; for instance, if $n = 100$, the CPU takes time 3.583 seconds, while the GPU 3.334 seconds.

The *inverse* constraint is employed in several encodings, such as the *black hole* problem, and it is also used to create the dual models of problems.

5.3.4 The *table* constraint

A *table constraint* is an *extensional* constraint defined by explicitly listing (a set of n) m-tuples of values that are either allowed (*positive* table constraint) of disallowed (*negative* table constraint) for the variables in its scope. *Table* constraints arise naturally in configuration problems where they represent available combinations of options. For some applications, compatibility between resources, e.g., persons or machines, can be expressed by tables. Tabular data may also come from databases: the results of database queries are sometimes expressed as tables that have large arity.

A table constraint c represented by a $n \times m$ matrix and the *Generalized Arc Consistency* (*GAC*) [135] is maintained through propagation. Precisely, focusing on a variable $x_i \in \{x_1, \ldots, x_m\} = \mathbf{scp}(c)$ a *support* for all the values in D^{x_i} is searched. This is realized by iterating over the n allowed tuples until a valid one is found. This algorithm ensures consistency in time $\mathcal{O}(n^m)$ (a faster, but more complex, algorithm is presented in [100]).

Using the GPU, it is possible to reduce this time to (parallel) time $\mathcal{O}(1)$, by performing the GAC test as follows: we assign each row to a kernel block, and each column to a different thread within the block. For table constraints with scope size larger than 1024, we split the computation among multiple kernels. For $1 \leq i \leq n$ and $1 \leq j \leq m$, thread t_{ij} checks whether the value contained in the cell c_{ij} is valid w.r.t. the domain D_{x_j}. The domains of the variables involved in the constraint are then replaced with the (new) domains, containing only those values that still might lead to a solution, as determined by each block.

We impose a specific ordering among propagated constraints: we first propagate binary constraints and constraints that have a fast propagator, that may eventually lead to a failure; more expensive propagators are executed last.

Table 5.6 compares the times for the propagation of the *table* constraint varying the number of rows n, the number of columns m, and the size of the domains of the variables. The tables are filled with random values, where $|D|$ is the size of the domain; note that larger domains produce fewer valid tuples after the labeling of a variable involved in the constraint.

$n \times m$	$\mid D \mid$	CPU	GPU	Speedup	$n \times m$	$\mid D \mid$	CPU	GPU	Speedup
100×100	2	0.002	0.001	2.00	100×100	50	0.001	0.001	1.00
250×250	2	0.007	0.003	2.30	250×250	50	0.003	0.001	3.00
500×500	2	0.026	0.010	2.60	500×500	50	0.013	0.004	3.25

Table 5.6: Time comparison for the *table* constraint with random values.

5.3.5 Examples containing table and inverse constraints

***Three-barrels* problem.** is a planning problem, where the state of the world is represented by three barrels of wine, whose capacities are n (even number), $n/2 + 1$, and $n/2 - 1$, respectively.

At the beginning, the largest barrel is full of wine, while the other two are empty. The goal is to reach a state in which the two largest barrels contain the same amount of wine. Moreover, the only admissible action is to pour wine from one barrel to another, until the latter is full or the former is empty. We encoded this problem as a decision problem, by imposing an upper bound ℓ on the number of actions and evaluating whether the goal state can be reached in ℓ steps. In this setting, we have $3(\ell+1)$ variables, with domains $\{1,\dots,n\}$, representing the sequence of states, and ℓ variables with domains $\{0,\dots,5\}$, representing the 6 possible "pouring" actions. The labeling is done on the action variables, and ℓ table constraints tie the i^{th} state with the successor $i+1^{th}$ state. Table 5.8 (left) shows the results for the *Three-barrels* problem considering a number of actions ℓ equal to n, that was experimentally found to be the length of the shortest successful plan. The speedup is slowly increasing due to the size of the tables ($r \times 7$, with r proportional to n) and the number of valid rows at each labeling (at most 6 given the current state), that reduce the propagation time to $\mathcal{O}(r)$.

Black-hole is a card game problem derived from [64]. A MiniZinc model is also present in the benchmark folder of the MiniZinc distribution [120], using both the global constraints *inverse* and *table*. The former is used to relate card values and positions in the sequence, while the latter is used to impose matching constraints among consecutive cards. The "<" (*less than*) constraints impose an order between played cards, and are always propagated on the host. Table 5.8 (right) shows the results for the *Black-hole* game problem. Since the game is devised for 52 cards, the set of order constraints for instances 104 and 208 are artificially introduced. The table shows an increasing speedup. The GPU is faster even on small instances, since the two expensive constraints are propagated in parallel on the GPU.

Three-Barrels Problem			
n	CPU	GPU	Speedup
100	176.5	160.8	1.09
120	364.9	324.3	1.12
140	679.6	588.8	1.15

Black-hole Problem			
n. cards	CPU	GPU	Speedup
52	7.637	7.694	0.99
104	68.14	51.08	1.33
208	73.77	42.66	1.72

Table 5.7: Time comparison for the *Three-barrels* problem and the *Black-hole* game

Positive *table* constraint benchmarks The following benchmark problems are defined using only positive table constraints.[1]. They include some well-known problems, such as the *crossword* game, the *Langford* problem, several synthetic problems, and some other real-world problems, such as the *modified Renault* problem. A speedup of at least 2 is obtained in all the problem instances, showing that the use of the GPU pays off on large instances and real problems.

Instance	CPU	GPU	Speedup	Instance	CPU	GPU	Speedup
CW-m1c-lex-vg4-6	0.015	0.005	3.00	langford-2-50	44.06	15.16	2.94
CW-m1c-uk-vg16-20	1.488	0.225	6.61	ModRen_0	0.381	0.154	2.74
CW-m1c-lex-vg7-7	209.4	43.87	4.77	ModRen_49	0.317	0.117	2.74
langford-2-40	136.4	46.39	2.90	RD_k5_n10_d10_m15	0.138	0.053	2.60

Table 5.8: Positive *table* constraint benchmarks.

[1]These benchmarks can be downloaded from `http://becool.info.ucl.ac.be/resources/positive-table-constraints-benchmarks`

5.4 Summary

In this chapter, we presented a feasibility study exploring the potential for exploitation of fine-grained GPU-level parallelism from the process of constraint propagation. The investigation has been grounded in a prototype (with competitive performance with the state-of-the-art), demonstrating the potential for enhanced performance, especially in the context of complex global constraints. This is not an easy task, and the speedups proposed are in-line with results observed for parallelization of other classes of problems on GPUs.

Let us conclude with a final observation: the overall strategy for handling constraint propagation reported in Algorithm 6 is designed for efficient sequential implementation, and indeed is at the core of the state-of-the-art constraint solvers. Alternative schemes (e.g., AC-3), that can be found in several other implementations, provide a lower level of sequential performance, but they are also more amenable for GPU-level parallelization. Unfortunately, the difference in sequential performance effectively defeats the advantages gained from parallelism.

6 GPU-based Search

Constraint programming as *declarative* approach allows one to model a broad class of optimization problems with ease. These problems are often characterized by huge search spaces (e.g., [22]) and heterogeneous constraints. In this case, *incomplete* search strategies (e.g., local search) are preferred w.r.t. exact approaches that require prohibitive time to find an optimal solution.

In this chapter, we propose the design and implementation of a novel *constraint solver* that exploits parallel *Local Search* (*LS*) using GPU architectures to solve constraint optimization problems. The optimization process is performed in parallel on multiple large promising regions of the search space, with the aim of improving the quality of the current solution. The local search model pursued is a variant of *Large Neighborhood Search (LNS)* (see Section 2.6). LNS is a local search techniques characterized by an alternation of *destroy* and *repair* methods. A solution is partially destroyed and an exploration of its (large) neighborhood is performed until the solution is repaired with a new one. Each neighborhood is explored using local search strategies and the best neighborhood (i.e, the one that better improves the quality of the solution) is selected to update the variables accordingly. The use of GPUs allows us to speed-up this search process and represents an alternative way to enhance performance of constraint solvers. In this chapter we present three main contributions on the current state-of-the-art:

1. Novel design and implementation of a constraint solver performing parallel search. Unlike the traditional approaches to parallelism, we take advantage of the computational power of GPUs for solving any Constraint Optimization Problem expressed as a MiniZinc model. To the best of our knowledge this is the first general constraint solver system that uses GPU to perform parallel local search.

2. A general framework that exploits *Single-Instruction Multiple-Threads* (*SIMT*) parallelism to speed-up local search strategies. We will present six different local search strategies that can be used to explore in parallel multiple large neighborhoods. These strategies are implemented by making very localized changes in the definition of a neighborhood. Hence, the user needs only to specify the structure of a neighborhood without modifying the structure of the underlying parallel computation.

3. A hybrid method for solving constraint optimization problems that uses local search strategies on large neighborhoods of variables. Usually, large neighborhood are explored using standard CP techniques. Instead, we present an approach based on local search to find the neighborhood that improves the objective function the most among a large set of different neighborhoods.

6.1 Solver Design and Implementation

6.1.1 Structure of the Solver

The structure of the constraint solver is based on the general framework described in Chapter 5—where a GPU architecture is used to perform parallel constraint propagation within a traditional event-driven constraint propagation engine [142]. We adopt this design to compute a first feasible solution to be successively improved via LNS (an initial solution as input, if known, can be specified).

Domains and Constraints. Variable's domains are represented using *bit-masks* stored in ℓ *unsigned int* 32-bit variables (for a suitable ℓ), while the status of the computation at every node of the search tree is represented by a vector of bit-masks corresponding to the current domains of all the variables in the model (see Section 5.2).

The supported constraints correspond to the set of finite domain constraints that are available in the *MiniZinc/FlatZinc* modeling language [120]. We modify the `solve` directive of FlatZinc to specify the local search strategy to be used during the neighborhood exploration.

Variables. The solver manages two types of variables: **(1)** Standard *Finite Domain* (*FD*) variables and **(2)** *Auxiliary* (*Aux*) variables. Aux variables are introduced to represent FlatZinc intermediate variables and they are used to compute the objective function. Their domains are initially set to all allowed integer values. We denote with $x^{\mathbf{aux}}_{\mathbf{fobj}}$ the Aux variable that represents the cost of the current solution. The search is driven by assignments of values to the FD variables of the model. The value of Aux variables is assigned by constraint propagation.

Search strategy. After a solution s is found, a neighbor is computed using $\eta(s)$, by randomly selecting a set of variables to be "released" (i.e., unassigned). The use of a GPU architecture allows us to concurrently explore several of these sets $\mathcal{N}_1, \ldots, \mathcal{N}_t$, all of them randomly generated by $\eta(s)$. Let m be a fixed constant; we compute m initial assignments for the variables in the set $\mathcal{N}_i$—these are referred to as the *(LS) starting points* $SP_{i,j}$ ($i = 1, \ldots, t$ and $j = 1, \ldots, m$) and can be computed in two ways. In the first option (*random*), each $SP_{i,j}$ is obtained by randomly choosing values from the domains of the variables in $\mathcal{N}_i$. This random assignment might not produce a solution of the constraints. However, for problems with a high number of solutions, this choice can be an effective LNS starting point. In the second option (*CP*), a random choice is performed only for the first variable in $\mathcal{N}_i$; this choice is followed by constraint propagation, in order to reduce the domains of other variables; in turn, a random choice is made for the second variable, using its reduced domain, and so on. If this process leads to a solution, then such solution is used as a starting point $SP_{i,j}$. Otherwise a new attempt is done. It is of course possible to implement other heuristics for the choices of the variables and their values (e.g., *first-fail*, *most-constrained*). If the process leads to failure for a given number of consecutive attempts, only the already computed $SP_{i,j}$ (if any) are considered.

GPU mapping. A total of $128 \cdot k$ ($1 \leq k \leq 8$) threads (a block) are associated to each $SP_{i,j}$ belonging to the correspondent set $\mathcal{N}_i$. These threads will perform LS starting from $SP_{i,j}$ (Fig. 6.1). The value of k depends on the architecture and it is used to split the computation within each starting points, as described in what follows.

When all the threads end their computations—according to a given LS algorithm (see Sec. 6.2)—we select among all of them the solution σ that optimizes the value $x^{\mathbf{aux}}_{\mathbf{fobj}}$ among all solutions $\sigma_{i,j}$ computed. This solution is compared with the previous one and, in case, σ is stored as the new best solution found so far.

This process is repeated for h *Iterative Improving* (*II*) steps, each restarting from the best found so far, but changing the randomly generated subsets of variables $\mathcal{N}_i$. After h IIs, the process restarts from the initial solution and is repeated for s *restarts* or until a given time-out limit is reached. In the end, the best solution found during the whole search process is restored.

Example 6.1.1 *The directives:*

```
lns( 50, 2, 4, 10, Gibbs, 100, 600 );
solve minimize fobj;
```

written in the model cause the solver to select $t = 2$ *subsets* $\mathcal{N}_i$*, each containing* 50% *of the whole*

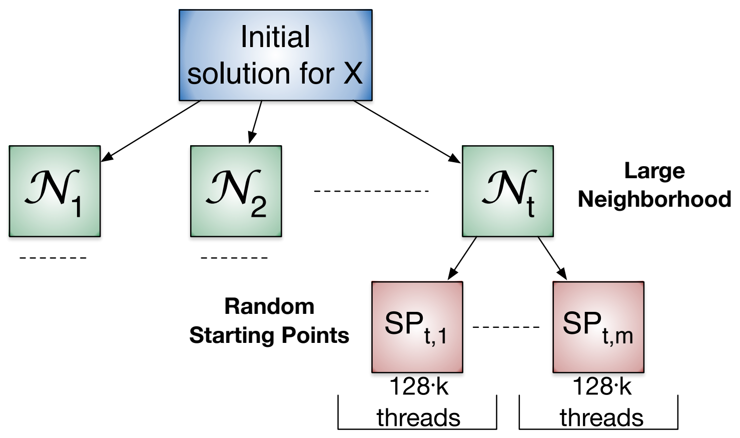


Figure 6.1: Parallel exploration of subsets $\mathcal{N}_i$ of variables. A LS strategy explores the space of $\mathcal{N}_i$ in parallel from different starting points SP_{i_j}.

set of variables X (randomly chosen), with $m = 4$ *SP per subset,* $s = 100$ *restarts, and a time-out limit of* 600 *sec. The solver tries to improve the value of* x^{aux}_{fobj} *in* $h = 10$ *II using* Gibbs *sampling as LS strategy—see Sect. 6.2.*

6.1.2 Exploiting GPU Parallelism.

Let us describe more in detail how we divide the workload among parallel blocks, i.e, the mapping between the subsets of variables $\mathcal{N}_i$ and CUDA blocks. The complete set of constraints, including those involving the auxiliary variable, and the initial domains are static entities; these are communicated to the GPU once at the beginning of the computation. We refer to the *status* as the current content of the domains of the variables—in particular, an assigned variable has a singleton domain. As soon as the solver finds a feasible solution, we copy the status into the global memory of the device, as well as the t subsets of variables representing the neighborhoods to explore. The CPU is in charge to launch the sequence of kernels with the proper number of blocks and threads. In what follows we focus on a single II step since the process remains the same for each restart s (the CPU stores the best solution found among all restarts). At each iterative improving step r, $0 \leq r \leq h$, the CPU launches the kernel K^r_1 with $t \cdot m$ blocks, where each block is assigned its own $SP_{i,j}$. Each block contains $128k$ threads, depending on the architecture (e.g., $k = 4$ is usually a good choice, see Paragraph 6.1.1). A kernel of type K_1 starts a local search optimization process from each starting point in order to explore different parts of the search tree at the same time. The current global status will be updated w.r.t. the best neighborhood selected among all.

After the kernel K^r_1 has been launched by the host, the control goes back immediately to the CPU which calls a second kernel K^r_2 that will start the computation on GPU as soon as K^r_1 has finished. This kernel is in charge of performing a parallel reduction on the array of costs computed by K^r_1. It can be the case that in some blocks, the LS strategy implemented is unable to find a solution; in this case the corresponding value is set to $\pm\infty$ (according to minimization/maximization). Moreover, K^r_2 updates the status with the new assignment σ of values for the variables in the subsets $\mathcal{N}^r_i$ that has led to the best improvement of $x^{\mathbf{aux}}_{\mathbf{fobj}}$.

At each II, r is incremented. If $r \leq h$ then the CPU will select t new subsets of variables $\mathcal{N}^{r+1}_i$ for the following cycle. Also this operation is performed asynchronously w.r.t. the GPU, i.e., the new subsets of variables are copied to the global memory of the device by a call to an asynchronous `cudaMemcpy` instruction. As a technical note, the array containing the new subsets is allocated on the host using the so-called *pinned* (i.e., host-locked) memory that is necessary in order to perform asynchronous copies between host and device.

When the time limit is reached or $r > h$, host and device are synchronized by a synchronous copy of the current status from the GPU to the CPU (Fig. 6.2). If the time limit is not reached and another restart has to be performed, the current solution is stored (if it improves the current objective value), the objective function is relaxed, and the whole process is repeated.

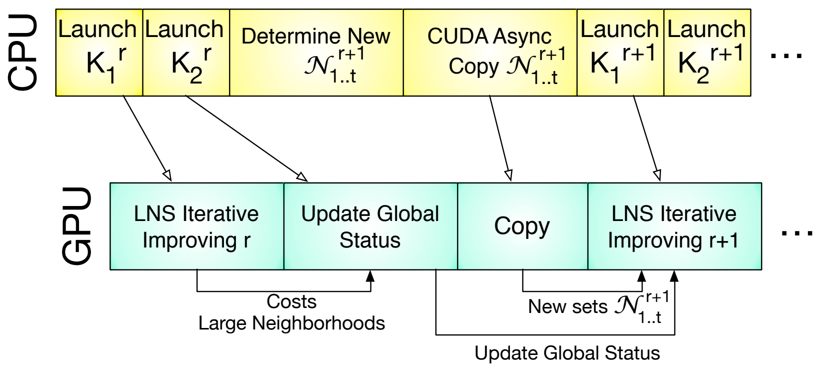

CPU
Launch K_1^r
Launch K_2^r
Determine New $\mathcal{N}_{1..t}^{r+1}$
CUDA Async Copy $\mathcal{N}_{1..t}^{r+1}$
Launch K_1^{r+1}
Launch K_2^{r+1}
GPU
LNS Iterative Improving r
Update Global Status
Copy
LNS Iterative Improving r+1
Costs Large Neighborhoods
New sets $\mathcal{N}_{1..t}^{r+1}$
Update Global Status

Figure 6.2: Concurrent computations between host and device.

A portion of the global memory of the GPU is reserved to store the status, the array representing the sets $\mathcal{N}$, and an array of size $(1 + |\mathcal{N}|) \cdot t \cdot m$ of 32 bits unsigned integer, to store the assignment and the correspondent cost for each starting point.

As anticipated above, an additional level of parallelism is exploited using the threads belonging to the same block focused on the LS part (kernel K_1^r). Precisely, K_1^r is launched with $128k$ threads (i.e., $4k$ warps) per block. We use parallelism at the level of warp to avoid divergent computational branches for threads belonging to the same warp. Divergent branches do not fit into the SIMT model, and cause a decrease of the real parallelism achieved by the GPU.

First, all of the threads are used to speed-up the copy of the current status from the global to the shared memory, and to restore the domains of the Aux variables. The queue of constraints to be propagated is partitioned among warps, according to the kind of variables involved: **(1)** FD variables only, **(2)** FD variables and *one* Aux variable, **(3)** two or more Aux variables, and **(4)** $x_{\mathbf{fobj}}^{\mathbf{aux}}$. Since the process starts with $SP_{i,j}$, the constraints of type **(1)** are only used to check consistency when random option for SP is used. This is done using the first two warps (i.e., threads $0 \ldots 64k - 1$). Observe that the use of a thread per constraint might lead to divergent computations, when threads have to check consistency of different constraints. As soon as a warp finds an inconsistent assignment, it sets the value of the $x_{\mathbf{fobj}}^{\mathbf{aux}}$ variable to $\pm\infty$ in the shared memory, as well as a global flag to inform the other threads to exit. Constraints of type **(2)** propagate information to the unique Aux variable involved. This can be done in parallel by the other two warps (i.e., threads $64k \ldots 128k - 1$).

If no failure has been found, all threads are synchronized in order to be ready to propagate constraints of type **(3)**. This propagation phase requires some sequential analysis of a queue of constraints and a fixpoint computation. To reduce the numbers of scans of this queue, we use the following heuristic: we sort the queue in order to consider first constraints that involve variables that are also present in constraints of type **(2)**, and only later constraints that involve only Aux variables. The idea is that Aux variables that are present in constraints of type **(2)** are already assigned after their propagation and can propagate to the other Aux variables. We experimentally observed that this heuristic reduces the number of scans to a single in most of the benchmarks. We use all warps to propagate this type of constraints. In practice, we divide the queue in $4k$ chunks, and we loop on these chunks until all variables are ground or an inconsistent assignment is detected. Finally, threads are synchronized and the value of the variable $x_{\mathbf{fobj}}^{\mathbf{aux}}$ is computed propagating the last type of constraints. (Fig. 6.3).

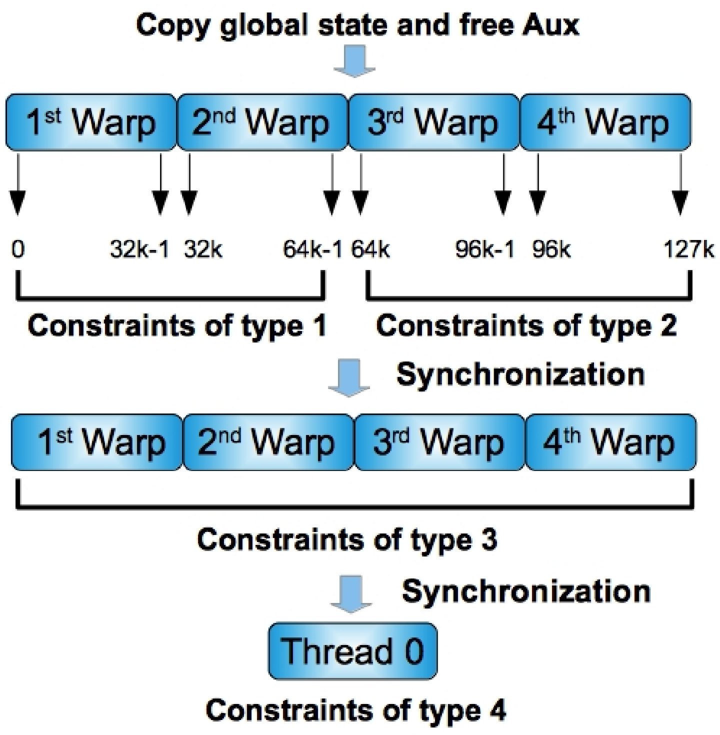


Figure 6.3: Thread partition within a block.

6.1.3 Some Technical Details

Since the whole process is repeated several times, some FD variables and Aux variables need to be released. This process is done exploiting CUDA parallelism, as well. In our experiments we set $k = 4$, and hence we use 512 threads per blocks for splitting the constraints. The splitting is parametric w.r.t. k A greater (or lower) number of threads is, of course, possible since kernel invocations and splitting are parametric w.r.t. the value k. The reason behind 512 depends on the specific GPU we are using and the number of SMs available. In particular, a larger number of threads would require more resources on the device, leading to a slower context switch between on blocks. Experiments allowed us to observe that for our hardware 512 threads is a good compromise between parallelism and resources allocated to each block. However, this is a compile-time parameter that can be changed for other platforms.

We also introduce an additional level of parallelism based on the size of the domains—suitable to support some of the LS strategies discussed in Sect. 6.2 (e.g., *ICM*). These strategies may explore the whole domain of a FD variable in order to select the best assignment. This exploration can be done in parallel, by assigning $64k$ threads to the first half of the domain and $64k$ threads to the second half (i.e., the queues of constrains will be spit in $64k$ chunks instead of $128k$).

The design presented so far does not depend on the local search strategy adopted, as long as it guarantees that each variable is assigned a value. We also require that the status does not exceed 49KB, since this is a typical limit for the shared memory of a block in the current GPUs. If the size of the problem is greater than this threshold, we copy chunks of status into the local memory according to the variables involved in the current queue of constraints to propagate.

6.2 Local Search Strategies

We have implemented six LS strategies for testing our framework. These strategies lead from a solution s to s' by repeatedly applying η on the set $\mathcal{N}$ of variables that can be re-assigned. After the action, constraints consistency is checked and $x^{\mathbf{aux}}_{\mathbf{fobj}}$ is computed. New strategies can be added as long as they implement a function η starting from s and from a subset of variables $\mathcal{N}$. We stress that the primary purpose of the LS presented in this section is to show how these methods can take advantage of the underlying parallel framework, more than the quality of the results they produce. Ad-hoc LS strategies should be implemented based on the problem to solve.

1. The *Random Labeling (RL)* strategy randomly assigns to the variables of $\mathcal{N}$ values drawn from their domains. Each thread is assigned to a variable in $\mathcal{N}$ and chooses a random value invoking the random number generator provided by CUDA. This strategy might be effective when we consider many sets $\mathcal{N}$, and the COP is not very constrained. It can be repeated a number p of times.

2. The *Random Permutation (RP)* strategy performs a random permutation (e.g., using Knuth's shuffling algorithm) of the values assigned to the variables in $\mathcal{N}$ in s and updates the values according. We use only one thread to perform the (sequential) random permutation of values, using the random number generator provided by CUDA. All threads are then synchronized in order to check the consistency of the (random) permutation. It can be used on problems where the domains of the variable are identical (e.g., *TSP*). It can be repeated p times.

3. The *Two-exchange permutation (2P)* strategy swaps the values of one pair of variables in $\mathcal{N}$ (using one thread). The neighborhood size is $n = \frac{|\mathcal{N}|(|\mathcal{N}|+1)}{2}$, and we force the number m of starting points to be $\leq n$.

4. The *Gibbs Sampling (GS)* strategy [17] is a simple *Markov Chain Monte Carlo* algorithm commonly used to solve the maximum a-posteriori estimation problem. We use it for COPs in the following way. Let ν be the current value of $x^{\mathbf{aux}}_{\mathbf{fobj}}$. The function f is defined as follows: *for each* variable x in $\mathcal{N}$, we choose a *random* candidate $d \in D^x \setminus \{s(x)\}$; then we determine the new value ν' of $x^{\mathbf{aux}}_{\mathbf{fobj}}$, and accept or reject the candidate d with probability $\frac{\nu'}{\nu}$. We use one thread for selecting the random candidate while all threads are used to propagate constraints and then check consistency. This process is repeated for p samplings steps; for p large enough, the process converges to the a local optimum for the large neighborhood.

5. The *Iterated Conditional Mode (ICM)* [17] can be seen as a greedy approximation of Gibbs sampling. The idea is to consider one variable $x \in \mathcal{N}$ at the time, and evaluate the cost of the solution for all the assignments of x satisfying the constraints, keeping all the other variables fixed. Then x is assigned with the value that minimize (maximize) the costs. To speed-up this process, all values for x are evaluated in parallel, splitting the domain of D^x between two groups of $2k$ warps each.

6. The *Complete Exploration (CE)* enumerates all the possible combination of values of the variables in $\mathcal{N}$. Given an enumeration $\vec{d_1}, \dots, \vec{d_e}$ of these values, each $\vec{d_i}$ is assigned to a block i, and the corresponding cost function is evaluated. The assignment with the best solution is returned. This method can be adopted when the product of the size of domain's variables of $\mathcal{N}$ is not huge.

6.3 Experiments

We implemented CPU and GPU versions of the LNS-based solver called *CPU/GPU-LNS* respectively. We first compare the two versions of the solver. Then, we compare the GPU-LNS against a pure CP approach in JaCoP [82], and a LNS implementation in *OscaR* [128]. We run our experiments on a CPU AMD Opteron (TM), 2.3GHz, 132 GB memory, Linux 3.7.10-1.16-desktop x86_64, and GPU GeForce GTX TITAN, 14 SMs, 875MHz, 6 GB global memory, CUDA 5.0 with compute capability 3.5. In this Thesis we report only the most significant results. The interested reader can visit `http://clp.dimi.uniud.it/sw/cp-on-gpu/` for a more extensive set of tests and benchmarks. In all tables t ($|\mathcal{N}|$) denotes the number (size) of large neighbors, m the number of *SP* per neighbor, times are reported in seconds, and best results are **boldfaced**.

6.3.1 CPU vs GPU: solving CSPs

We compared CPU and GPU on randomly generated CSPs defined by $\neq$ constraints between pairs of variables. We used this benchmark to test the performance of GPU-LNS on finding feasible starting points SP (see option CP, Sect. 6.1). Therefore, we used a random-generated instance of the graph coloring problem in order to avoid any particular structure on the constraint graph that could affect the choice of some particular neighborhoods.

Table 6.1 reports the results in seconds for a CSP consisting of 70 variables and 200 constraints. In these experiments, SP are generated considering one variable at a time, assigning it randomly with a value in its domain and subsequently propagating constraints to reduce domains of the other variables. When the number of $SP_{i,j}$ increases, speedups of one order of magnitude w.r.t. the sequential implementation are obtained. A high number of parallel tasks compensate both the different speed of the GPU cores w.r.t. the CPU cores and the memory latency of the device memory.

$\lvert\mathcal{N}\rvert$	t	m	CPU-LNS(s)	GPU-LNS(s)	Speedup
20	1	1	**0.216**	0.218	0.99
20	50	50	1.842	**0.379**	4.86
20	100	100	6.932	**0.802**	8.64
30	1	1	**0.216**	0.218	0.99
30	50	50	2.460	**0.377**	6.52
30	100	100	8.683	**0.820**	10.58

Table 6.1: CPU vs GPU: solving CSP

6.3.2 CPU vs GPU: evaluating LS strategies

CPU and GPU solvers have been compared considering the LS strategies of Sect 6.2. As benchmark we considered a *Mod*ified version of the *k-Coloring Problem* (*MKCP*). The goal is to maximize the difference of colors between adjacent nodes, i.e. $\max \sum_{(i,j)\in E} |x_i - x_j|$, where x_i (x_j) represents the color of the nodes i (j), provided pairs of adjacent nodes are constrained to be different. Here we report the results concerning on one instance (`1-Insertions_4.col` from `http://www.cs.hbg.psu.edu/txn131/graphcoloring.html` Other tests are available on-line.) of a graph with 67 nodes and 232 edges, that requires 4423 Aux variables and 4655 constraints. The initial solution (value 2098 with domains of size 30) is found by a *leftmost* strategy with increasing value assignment (this time has not been considered in the table). Since in this experiments our goal is just to compare CPU and GPU times, we run tests with the same pseudo-random sequence, $h = 10$ and $s = 0$. Results are reported in Table 6.2. For the LS and RP we considered $p = 5$ repetitions (for the RP strategy we slightly modified the model transforming the coloring benchmark into a permutation problem). Better speedups are observed for larger neighborhoods and in particular for the RL method and the CE method which are the less demanding strategies (GPU cores receive simple but numerous task to execute). On the other hand, the higher speedups are obtained by the CE strategy. Using CE we considered only one neighborhood reducing its size to 2, 3 and varying the domains's size from 10 to 30.

6.3.3 Comparison with standard CP

In this section we evaluate the performance of the GPU-LNS solver on some Minizinc benchmarks, comparing its results against the solutions found by the state-of-the-art CP solver JaCoP. We present results on medium-size problems which are neither too hard to be solved with standard CP techniques nor too small to make a local search strategy useless.

We considered the following four problems:

LS	$\lvert\mathcal{N}\rvert$	t	m	Max	CPU-LNS(s)	GPU-LNS(s)	Speedup
RL	20	1	1	22828	0.206	0.359	0.57
RL	20	50	50	28676	9.470	0.603	15.70
RL	20	100	100	**29084**	35.22	1.143	30.81
RL	30	1	1	20980	0.218	0.258	0.84
RL	30	50	50	27382	7.733	0.615	12.57
RL	30	100	100	29028	43.24	1.394	**31.01**
RP	20	1	1	15902	0.046	0.069	0.66
RP	20	50	50	17586	13.59	4.154	3.27
RP	20	100	100	**17709**	53.32	16.28	**3.27**
RP	30	1	1	16489	0.045	0.068	0.66
RP	30	50	50	17375	13.49	4.187	3.22
RP	30	100	100	17527	53.88	16.46	3.27
2P	10	1	1	15073	0.151	0.062	2.43
2P	10	20	20	16541	1.231	0.381	3.23
2P	10	50	50	16636	2.839	0.832	**3.41**
2P	20	1	1	15083	0.285	0.119	2.39
2P	20	20	20	16628	4.597	1.351	3.40
2P	20	50	50	**16646**	11.11	3.267	3.40
GS	10	1	1	26486	0.546	1.910	0.28
GS	10	10	10	29308	28.09	12.15	2.31
GS	10	50	50	**30810**	724.2	279.6	2.59
GS	30	1	1	24984	1.053	4.880	0.21
GS	30	10	10	27722	78.59	33.84	2.32
GS	30	50	50	28546	1982	747.92	**2.65**
ICM	5	1	1	31718	0.644	1.637	0.39
ICM	5	10	10	32204	32.23	7.650	4.21
ICM	5	20	20	32296	120.8	26.50	4.55
ICM	20	1	1	31948	0.993	2.522	0.39
ICM	20	10	10	32202	25.55	4.636	5.51
ICM	20	20	100	**32384**	92.68	13.26	**6.98**
CE	2	1	100	8004	0.692	0.324	2.13
CE	3	1	1000	9060	3.932	0.829	4.74
CE	2	1	400	17812	2.673	0.279	9.58
CE	3	1	8000	20020	43.26	1.298	33.32
CE	2	1	900	24474	3.444	0.817	4.21
CE	3	1	27000	**29262**	83.06	2.159	**38.47**

Table 6.2: MKCP benchmark using six LS strategies (maximization)

System	Benchmark	First Sol	Best Sol(sd)	Time(s)
JaCoP	Transportation	6699	6640	600
JaCoP	TSP	10098	6307	600
JaCoP	Knapsack	7366	15547	600
JaCoP	Coins_grid	20302	19478	600
GPU-LNS	Transporation	7600	**5332** (56)	57.89
GPU-LNS	TSP	13078	**6140** (423)	206.7
GPU-LNS	Knapsack	0	**48219** (82)	6.353
GPU-LNS	Coins_grid	20302	**16910** (0)	600

Table 6.3: Minizinc benchmarks (minimizazion problems, save Knapsack).

System	q	First Sol	Best Sol (sd)	Time(s)
OscaR	15	79586	9086 (0)	63.09
OscaR	32	430	254 (0)	126.2
OscaR	64	300	212 (0)	1083
GPU-LNS	15	83270	**0** (0)	0.242
GPU-LNS	32	368	**199.6** (9.66)	1.125
GPU-LNS	64	254	**121.6** (2.87)	2.764

Table 6.4: Quadratic Assignment Problem (minimization)

1. The *Trasportation* problem, with only 12 variables but the optimal solution is hard to find using CP. The heuristics used for JaCoP is the `first_fail`, `indomain_min`, while for GPU-LNS we used the RL method. We used $t = 100$ neighborhoods of size 3, $m = 100$ SP each, and $h = 500$.

2. The *Travelling Salesman Problem* with 240 cities and some flow constraints. The heuristics used for JaCoP is the same as above, RP strategy is used in GPU-LNS with $p = 1$. We use $t = 100$ neighborhood of size 40, $m = 100$, and $h = 5000$.

3. The *Knapsack* problem. We considered instances of 100 items.[1] The strategy adopted in JaCoP is `input_order`, `indomain_random`, while for GPU-LNS we used the RL search strategy, with $t = 50$ neighborhoods of 20 variables, $m = 50$, and $h = 5000$.

4. The *Coins_grid* problem. We considered this problem to test our solver on a *highly* constrained problem. For this benchmark we sightly modified the LS strategy: first we set $\eta(s) = s$, then we used CP (option 2) to generate random SPs. The strategy adopted in JaCoP is `most_constrained`, `indomain_max`, while for GPU-LNS we used the RL search strategy, with $t = 300$ neighborhoods of 20 variables, $m = 150$, and $h = 50000$.

Models and corresponding description are available at `http://www.hakank.org/minizinc/`. Table 6.3 reports the first solution value, the best solution found (within 10 min) and the (average on 20 runs for GPU-LNS) running times. For GPU-LNS the standard deviation of the best solution is reported. Results show effectiveness of the GPU-LNS solver.

6.3.4 Comparison with Standard LNS

We compare GPU-LNS against a standard implementation of a LNS in *OscaR*. OscaR is a Java toolkit that provides libraries for modelling and solving COP using Constraint Based Local Search [78]. We compare the two solvers on a standard benchmark used to test LNS strategies, namely the *Quadratic Assignment Problem* (*QAP*). The description of the problem and the model used for OscaR are available at `https://bitbucket.org/oscarlib/oscar/wiki/lns`. We use three different datasets (small / medium / large sizes). OscaR is run using adaptive LNS with Restart techniques. For each instance we try different combinations of restarts and adaptive settings; Results for the best combination are reported in Table 7.1, as well as GPU-LNS results with the RP strategy, $h = 10$, $t = 50$ neighborhood of size 20, and $m = 50$. For both systems results are averaged on 20 runs and standard deviation of best results is reported. Standard deviations of best solutions are reported. The GPU-LNS version of the solver outperforms OscaR (this is mainly due to the fact that GPU-LNS considers 2500 neighborhoods at a time). We also try to compare GPU-LNS against OscaR on the *Coins_problem* benchmark. We started both the LNSs from the same initial solution found by OscaR (i.e., 123460), and we used the same setting described above for GPU-LNS. Both system reached the time-out limit with an objective value of 25036 for GPU-LNS, and 123262 for OscaR.

[1]An hard instance has been generated using the generator that can be found at `http://www.diku.dk/~pisinger/generator`.

The presented results show speedups that increase with the size of the problems. However, it is not always easy to estimate the quality of the parallelization and, in particular, to obtain linear speedup w.r.t. GPU cores. A simple formula such as $\mathsf{number\_of\_cores} \cdot \mathsf{GPU\_speed}/\mathsf{CPU\_speed}$ returns an unreachable upper bound, since the GPU architecture, bank conflicts, memory speed, GPU-CPU transfer time are major bottlenecks. These factors must be considered and different parameters (e.g., the number of threads per block) must be tuned according to the available architecture.

6.4 Summary

In this chapter, we presented the design of a constraint solver that uses GPU computation to perform both parallel constraint propagation and parallel search. Large neighborhoods are explored using LS techniques with the goal of improving the current solution evaluating a large set of neighborhoods at a time. The choice of local search strategies is twofold: first, incomplete but fast methods are usually preferred for optimization problems where the search space is very large but not highly constrained. Second, with very few changes, the parallel framework adopted for a local search method can be easily generalized to be suitable for many different local search strategies, requiring minimal parameter tuning. Our experimental results show that the solver implemented on GPU outperforms its sequential version. Good results are also obtained by comparing the solver against standard CP and LNS. Moreover, we showed that many LS strategies can be encoded on our framework by changing few parameters, without worrying about how it is actually performed the underlying parallel computation.

7 GPU & DCOPs

In this chapter we turn our attention to *Distribute Constraint Optimization Problems* (*DCOP*s) to show that the use of GPU computation can lead to speed ups even in a distributed environment.

In particular, we introduce a general framework that uses *Markov Chain Monte Carlo* (MCMC) sampling algorithms to solve DCOPs, exploring the use of GPUs to parallelize the sampling process. This framework, which we call *GPU-based Distributed MCMC* (DMCMC), emulates the computation and communication operations of DPOP [130], except that the computation of utilities is done using MCMC sampling algorithms on GPUs. We demonstrate the generality of this framework using the Gibbs [63] and Metropolis-Hastings [17] algorithms, two commonly used MCMC algorithms, on meeting scheduling and smart grid network problem domains.

In Section 4.3 we presented some notions about DCOPs. In what follows we review some complementary background about this formalism extending this formalism to the aspects needed to fully understand the the algorithms presented in this chapter.

7.1 Background

7.1.1 DCOPs

A *Distributed Constraint Optimization Problem* (*DCOP*) [130, 176] is defined by $\langle \mathcal{X}, \mathcal{D}, \mathcal{F}, \mathcal{A}, \alpha \rangle$, where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of *variables*; $\mathcal{D} = \{D_1, \ldots, D_n\}$ is a set of finite *domains*, where D_i is the domain of variable x_i; $\mathcal{F} = \{f_1, \ldots, f_m\}$ is a set of *utility functions* (also called *constraints*), where each k-ary utility function $f_i : D_{i_1} \times D_{i_2} \times \ldots \times D_{i_k} \mapsto \mathbb{R} \cup \{-\infty, +\infty\}$ specifies the utility of each combination of values of variables in its *scope* (i.e., $x_{i_1}, \ldots, x_{i_k}$); $\mathcal{A} = \{a_1, \ldots, a_p\}$ is a set of *agents* and $\alpha : \mathcal{X} \to \mathcal{A}$ maps each variable to one agent. Let us observe that $\mathcal{F}$ corresponds to the set of constraints C presented in Section 2.2.

We use the notations S_i to refer to the set of variables in the *scope* of function f_i, L_i to refer to the set of (*local*) variables owned by agent a_i, and B_i to refer to the set of (*boundary*) variables owned by agent a_i that share constraints with variables owned by another agent. A solution is a value assignment for a subset of variables. Its utility is the sum of the evaluations of all utility functions on it. A solution is *complete* iff it is a value assignment for all variables. The goal is to find a utility-maximal complete solution.

A *constraint graph* visualizes a DCOP instance, where nodes in the graph correspond to variables in the DCOP and edges connect pairs of variables in the scope of the same utility function. A *DFS pseudo-tree* arrangement has the same nodes and edges as the constraint graph and satisfies two conditions: (i) there is a subset of edges, called *tree edges*, that form a rooted tree, and (ii) two variables in the scope of the same utility function appear in the same branch of the tree. The other edges are called *backedges*. Tree edges connect parent-child nodes, while backedges connect a node with its pseudo-parents and its pseudo-children.

We use the notations C_i to refer to the set of child agents of agent a_i in the pseudo-tree, P_i the parent agent of agent a_i in the pseudo-tree, and *sep*(a_i) to refer to the *separator* of agent a_i, which is the set of variables owned by its ancestor agents that are constrained with variables owned by the agent or by its descendant agents. Figure 7.1(a) shows the constraint graph of an example DCOP

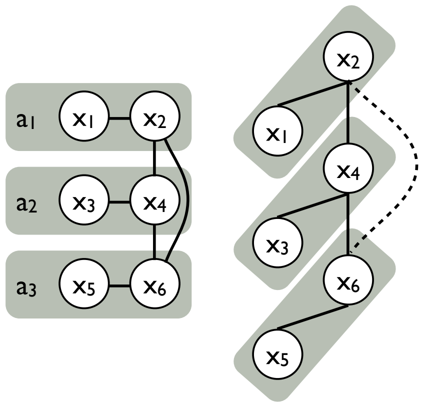


Figure 7.1: Example DCOP - Constraint graph (left) and Pseudo-Tree (right).

x_i	x_j	Utilities
0	0	20
0	1	8
1	0	10
1	1	3

Table 7.1: Example DCOP - Utilities.

with 3 agents a_1, a_2, and a_3, where $L_1 = \{x_1, x_2\}$, $L_2 = \{x_3, x_4\}$, $L_3 = \{x_5, x_6\}$, $B_1 = \{x_2\}$, $B_2 = \{x_4\}$, and $B_3 = \{x_6\}$. Each variable can be assigned the values 0 or 1. Figure 7.1(b) shows one possible pseudo-tree (the dotted line is a *backedge*). Figure 7.1(c) shows the utilities of all 6 utility functions.

7.1.2 DPOP

The *Distributed Pseudo-tree Optimization Procedure* (DPOP) [130] is a complete DCOP algorithm with the following three phases:

- **Pseudo-tree Generation Phase:** DPOP calls existing distributed pseudo-tree construction methods [72] to build a pseudo-tree.

- **UTIL Propagation Phase:** Each agent, starting from the leaves of the pseudo-tree, computes the optimal sum of utilities in its subtree for each value combination of variables in its separator. The agent does so by summing the utilities of its constraints with the variables in its separator and the utilities in the UTIL messages received from its child agents, and then projecting out its own variables by optimizing over them. In our example problem, agent a_3 computes the optimal utility for each value combination of variables x_2 and x_4 (see Table 7.2(a)), and sends the utilities to its parent agent a_2 in a UTIL message. Agent a_2 then computes the optimal utility for each value of the variable x_2 (see Table 7.2(b)), and sends the utilities to its parent agent a_1 in a UTIL message. Finally, agent a_1 computes the optimal utility of the entire problem (see Table 7.2(c)).

- **VALUE Propagation Phase:** Each agent, starting from the root of the pseudo-tree, determines the optimal value for its variables. The root agent does so by choosing the values of its variables from its UTIL computations.

x_2	x_4	Utilities
0	0	max(60,24,50,19) = 60
0	1	max(50,19,40,14) = 50
1	0	max(50,19,40,14) = 50
1	1	max(40,14,30, 9) = 40

(a) Computations of a_3

x_2	Utilities
0	max(100,66,90,61) = 100
1	max(80,51,70,46) = 80

(b) Computations of a_2

Utilities
max(120, 110) = 120

(c) Computations of a_1

Table 7.2: Example UTIL Phase Computations

Algorithm 10 SAMPLING($\mathbf{z}$)

```
1: $\mathbf{z}^{(0)} \leftarrow$ INITIALIZE($\mathbf{z}$)
2: for $t = 1$ to $T$ do
3:   if use METROPOLIS-HASTING sampling then
4:     $\mathbf{z}^* \leftarrow$ SAMPLE($q(\mathbf{z}^* \mid \mathbf{z}^{(t-1)})$)
5:   else if use GIBBS sampling then
6:     for $i = 1$ to $n$ do
7:       $z_i^t \leftarrow$ SAMPLE($\frac{1}{Z_\pi}\tilde{\pi}(z_i \mid z_1^t, \dots, z_{i-1}^t, z_{i+1}^{t-1}, \dots, z_n^{t-1})$)
8:     end for
9:   end if
10:  $\mathbf{z}^{(t)} \leftarrow \begin{cases} \mathbf{z}^* & \textbf{with } p = \min(1, \frac{\tilde{\pi}(\mathbf{z}^*)q(\mathbf{z}^{(t-1)},\mathbf{z}^*)}{\tilde{\pi}(\mathbf{z}^{(t-1)})q(\mathbf{z}^*,\mathbf{z}^{(t-1)})}) \\ \mathbf{z}^{(t-1)} & \textbf{with } 1-p \end{cases}$
11: end for
```

7.1.3 MCMC Sampling

Markov Chain Monte Carlo (MCMC) sampling algorithms are commonly used to solve the *Maximum A Posteriori* (*MAP*) estimation problem. [123] shows that DCOPs can be mapped to MAP estimation problems by mapping DCOP variables to *random variables*, and the cost functions to potential function. Therefore, finding the most probable assignment to all variables in a MAP estimation problem is equivalent to finding the complete assignment that maximizes the objective function of the DCOP (see B). Thus, MCMC algorithms can be used to solve DCOPs as well by sampling on the corresponding MAP problem. Let us describe two commonly used MCMC algorithm—Gibbs [63] and Metropolis-Hastings [17].

Suppose we have a joint probability distribution $\pi(\mathbf{z})$ over n variables, with $\mathbf{z} = z_1, z_2, \dots, z_n$, which we would like to approximate. Moreover, as it is often the case, suppose that it is easy to evaluate $\pi(\mathbf{z})$ for any given $\mathbf{z}$ up to some normalizing constant Z_π, such that: $\pi(\mathbf{z}) = \frac{1}{Z_\pi}\tilde{\pi}(\mathbf{z})$, where $\tilde{\pi}(\mathbf{z})$ can be easily computed but Z_π may be unknown or hard to evaluate. In order to draw the samples $\mathbf{z}$ to be fed to $\tilde{\pi}(\cdot)$, we use a *proposal distribution* $q(\mathbf{z} \mid \mathbf{z}^{(\tau)})$, from which we can easily generate samples, each depending on the current state $\mathbf{z}^{(\tau)}$ of the process. The latter can be interpreted as saying that when the process is in the state $\mathbf{z}^{(\tau)}$, we can generate a new state $\mathbf{z}$ from $q(\mathbf{z} \mid \mathbf{z}^{(\tau)})$. The proposal distribution is thus used to generate a sequence of samples $\mathbf{z}^{(1)}, \mathbf{z}^{(2)}, \dots$, which forms a Markov chain.

Algorithm 10 shows the pseudo-code of the *Metropolis-Hastings* algorithm. It first initializes $\mathbf{z}^{(0)}$ to any arbitrary value the variables $z_1, \dots, z_n$ (line 1). Then, it iteratively generates a candidate $\mathbf{z}^*$ for $\mathbf{z}^{(t)}$ by sampling from the proposal distribution $q(\mathbf{z}^* \mid \mathbf{z}^{(t-1)})$ (line 3). The candidate sample is then accepted with probability p defined in line 4. If the candidate sample is accepted, then $\mathbf{z}^{(t)} = \mathbf{z}^*$, otherwise $\mathbf{z}^{(t-1)}$ is left unchanged. This process continues for a fixed number of iterations (i.e., parameter T in Alg. 10).

The *Gibbs* sampling algorithm is a special case of the Metropolis-Hastings algorithm (lines 6-8). Additionally, let us observe that Gibbs requires the computation of the normalizing constant Z_π while Metropolis-Hasting does not, as the calculation of the proposal distribution does not require that information. This is desirable when the computation of the normalizing constant becomes prohibitive (e.g., with increasing problem dimensionality).

Algorithm 11 DMCMC(R, T)

```
1: Generate pseudo-tree
2: GPU-Initialize( )
3: $\langle M_i^1, U_i^1\rangle, \ldots, \langle M_i^R, U_i^R\rangle \leftarrow$ GPU-MCMC-Sample($R, T$)
4: $UTIL_{a_i} \leftarrow$ Get-Best-Sample($\langle M_i^1, U_i^1\rangle, \ldots, \langle M_i^R, U_i^R\rangle$)
5: if $C_i = \emptyset$ then
6:     $UTIL_{a_i} \leftarrow$ CalcUtils( )
7:     Send $UTIL$ message ($a_i$, $UTIL_{a_i}$) to $P_i$
8: end if
9: Activate UTILMessageHandler($\cdot$)
10: Activate VALUEMessageHandler($\cdot$)
```

Algorithm 12 VALUEMessageHandler(a_k, $VALUE_{a_k}$)

```
1: $VALUE_{a_i} \leftarrow VALUE_{a_k}$
2: for $x_i^j \in L_i$ do
3:     $d_i^{j*} \leftarrow$ ChooseBestValue($VALUE_{a_i}$)
4: end for
5: for $a_c \in C_i$ do
6:     $VALUE_{a_i} \leftarrow \{(x_i^j, d_i^{j*}) \mid x_i^j \in sep(a_c)\} \cup \{(x_k, d_k^*) \in VALUE_{a_k} \mid x_k \in sep(a_c)\}$
7:     Send $VALUE$ message ($a_i$, $VALUE_{a_i}$) to $a_c$
8: end for
```

Algorithm 13 UTILMessageHandler(a_k, $UTIL_{a_k}$)

```
1: Store $UTIL_{a_k}$
2: if received UTIL message from each child $a_c \in C_i$ then
3:     $UTIL_{a_i} \leftarrow$ CalcUtils( )
4:     if $P_i = NULL$ then
5:         for $x_i^j \in L_i$ do
6:             $d_i^{j*} \leftarrow$ ChooseBestValue($\emptyset$)
7:         end for
8:         for $a_c \in C_i$ do
9:             $VALUE_{a_i} \leftarrow \{(x_i^j, d_i^{j*}) \mid x_i^j \in sep(a_c)\}$
10:            Send $VALUE$ message ($a_i$, $VALUE_{a_i}$) to $a_c$
11:        end for
12:    else
13:        Send $UTIL$ message ($a_i$, $UTIL_{a_i}$) to $P_i$
14:    end if
15: end if
```

7.2 GPU-based Distributed MCMC Framework

Let us now describe the *GPU-based Distributed MCMC* (DMCMC) framework, which extends centralized MCMC sampling algorithms and DPOP. At a high level, its operations are similar to

Algorithm 14 CalcUtils( )

```
1: $UTIL_{sep} \leftarrow$ utilities for all value comb. of
2: $x_i \in B_i \cup sep(a_i)$
3: $UTIL_{a_i} \leftarrow$ JOIN($UTIL_{a_i}, UTIL_{sep}, UTIL_{a_c}$)
4: for all $a_c \in C_i$ do
5:     $UTIL_{a_i} \leftarrow$ PROJECT($a_i, UTIL_{a_i}$)
6: end for
7: return $UTIL_{a_i}$
```

Algorithm 15 GPU-MCMC-Sample(T, R)

```
1: $\langle \mathbf{z}, \mathbf{z}^*, [q, Z_\pi], G_i \rangle \leftarrow$ ASSIGNSHAREDMEM()
2: $r_{id} \leftarrow$ the thread's row index of $M_i$
3: $\mathbf{z} \overset{|L_i|}{\leftleftarrows} M_i[r_{id}]$
4: $\langle \mathbf{z}^*, util^* \rangle \leftarrow \langle \mathbf{z}, \sum_{f_j \in G_i} f_j(\mathbf{z}_{|S_j}) \rangle$
5: for $t = 1$ To $T$ do
6:     $\mathbf{z} \overset{k}{\leftleftarrows}$ SAMPLE($q(\mathbf{z} \mid \mathbf{z}^{(t-1)})$) w/ prob. $\min\{1, \frac{\tilde{\pi}(\mathbf{z})}{\tilde{\pi}(\mathbf{z}^{(t-1)})}\}$
7:     $util \leftarrow \sum_{f_j \in G_i} f_j(\mathbf{z}_{|S_j})$
8:     if $util > util^*$ then
9:         $ma\langle \mathbf{z}^*, util^* \rangle \leftarrow \langle \mathbf{z}, util \rangle$
10:    end if
11: end for
12: $\langle M_i^R[r_{id}], U_i^R[r_{id}] \rangle \leftarrow \langle \mathbf{z}^*, util^* \rangle$
```

the operations of DPOP except that the computation of the utility tables sent by agents during the UTIL phase is done by sampling with GPUs. Notice that the computation of each row in a utility table is independent of the computation in the other rows. Thus, DMCMC exploits this independence and samples the utility in each row in parallel.

Algorithm 11 shows the pseudocode of DMCMC for an agent a_i. It takes as inputs R, the number of sampling runs to perform from different initial value assignments, and T, the number of sampling trials. Like DPOP, DMCMC also exhibits three phases. The first phase is identical to that of DPOP (line 1). In the second phase:

- Each agent a_i calls GPU-INITIALIZE() to set up the GPU kernel specifics (e.g., number of threads and amount of shared memory to be assigned to each block, and to initialize the data structures on the GPU device memory) (line 2). The GPU kernel settings are decided according to the shared memory requirements and the number of registers used by the successive function call, so to maximize the number of blocks that can run in parallel.

- Each agent a_i, *in parallel*, calls GPU-MCMC-SAMPLE() which performs the local MCMC sampling process to compute the best utility and the corresponding solution (value assignments for all non-boundary local variables $x_i^j \in L_i \setminus B_i$) for each combination of values of the boundary variables $x_i^k \in B_i$ (line 3). This computation process is done via sampling with GPUs and the results are hence transferred from the device to the host (line 4).

- Each agent a_i computes the utilities for the constraints between its variables and its separator, joins them with the sampled utilities (line 6), and sends them to its parent (line 7). The agent repeats this process each time it receives a UTIL message from a child (Alg. 12, line 7 - Alg. 13, line 10).

By the end of the second phase (Alg. 13, line 3), like in DPOP, the root agent knows the overall utility for each combination of values of its variables $x_i^j \in B_i$. It chooses its best value combination that results in the maximum utility (Alg. 13, line 6), and starts the third phase by sending to each child agent a_c the values of variables $x_i^j \in sep(a_c)$ that are in the separator of the child (Alg. 13, lines 8-11). The *MessageHandlers* of lines 9-10 of Alg. 11 are activated for any new incoming message.

7.2.1 GPU Data Structures

In order to fully capitalize on the parallel computational power of GPUs, the data structures need to be designed in such a way to limit the amount of information exchanged between the CPU host and the GPU devices. Each DMCMC agent stores all the information it needs in its local variables in the global memory of the GPU devices. This allows each agent running on a GPU device to communicate with the CPU host only once, which is at the end of the sampling process, to transfer the results. Each agent a_i maintains the following information:

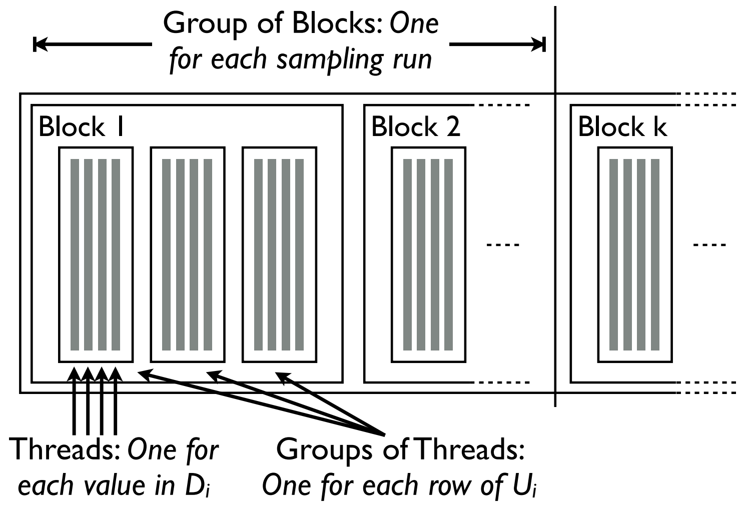
- its *local* variables $L_i \subseteq \mathcal{X}$;
- its *boundary* variables $B_i \subseteq L_i$;
- the domains of its local variables, D_i (for the sake of simplicity we assume these domains to have uniform size);
- a matrix M_i of size $|D_i|^{|B_i|} \times |L_i|$, where the j-th row is associated with the j-th permutation of the boundary variable values, in lexicographic order, and the k-th column is associated with the k-th variable in L_i. The matrix columns associated to the local variables in L_i are initialized with random value assignments in $[0, D_i - 1]$. At the end of the sampling process it contains the converged domain values of the local variables for each value combination of the boundary variables;
- a vector U_i of size $|D_i|^{|B_i|}$, which stores the utilities of the solutions in M_i;
- The *local constraint graph* G_i, which includes the local variables L_i and constraints between local variables.

All the data stored on the GPU devices is organized in mono-dimensional arrays, so as to facilitate *coalesced memory accesses*. The set of local variables L_i are ordered, for convenience, in lexicographic order and so that the boundary variables B_i are listed first.

7.2.2 Local Sampling Process

The GPU-MCMC-Sample procedure of line 3 is the core of the local sampling algorithm, and can be performed by any MCMC sampling method. It executes T sampling trials for the subset of non-boundary local variables $L_i \setminus B_i$ of agent a_i. Since the MCMC sampling procedure is stochastic, we can run R parallel sampling processes with different initial value assignments and take the best utility and corresponding solution across all runs. Each parallel run is executed by a *group of blocks*. Independent operations within each sample are also exploited in parallel using *groups of threads* within each block. For example, the proposal distribution adopted by Gibbs is computed using $|D_i|$ parallel *threads*. Figure 7.2 illustrates the different parallelizations performed by the GPU-MCMC-Sample process with Gibbs.

The general GPU-MCMC-Sample procedure is shown in lines 1-12 of Algorithm 15 and we use the symbols $\leftarrow$ and $\overset{k}{\leftleftarrows}$ to denote sequential (single thread) and parallel (k threads) operations, respectively. We also denote with n the size of the state $\mathbf{z}$ being sampled, where $n = |L_i| - |B_i|$. The function takes as inputs the number of desired sampling trials T and the number of parallel sampling runs R. It first assigns the shared memory allocated to the arrays $\mathbf{z}$ and $\mathbf{z}^*$, which are used to store the current and best sample of value assignments for all local variables, respectively; the local constraint graph G_i; and, if the MCMC sampling algorithm requires computing the normalization constant of the proposal distribution explicitly, the array q and Z_π, which are used


Group of Blocks: *One for each sampling run*
Block 1
Block 2
Block k
Threads: *One for each value in D_i*
Groups of Threads: *One for each row of U_i*
Figure 7.2: Parallelization Illustration

to store the probabilities for each value of the non-boundary local variables and the normalization constant, respectively (line 1).

Each thread identifies its row index r_{id} of the matrix M_i, initializes its sample with the values stored in $M_i[r_{id}]$, calculates the utility for that sample, and stores the initial sample and utility as the best sample and utility found so far (lines 2-4). It then runs T sampling trials, where in each trial, it samples a new state $\mathbf{z}$ from a proposal distribution $q(\mathbf{z} \mid \mathbf{z}^{(t-1)})$ and updates that state according to the accept/reject probabilities described in the MCMC background (line 6).

The proposal distribution q and the accept/reject probabilities depend on the choice of MCMC algorithm. We now describe them for Metropolis-Hasting and Gibbs.

- **Metropolis-Hastings**: The proposal distribution that we adopt is a multivariate normal distribution $q \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, with $\boldsymbol{\mu}$ being a n-dimensional vector of mean values, where each component $\mu_j^{(t)}$ has the value of the corresponding component in the previous sample $z_j^{(t-1)}$ and $\boldsymbol{\Sigma}$ is the covariance matrix defined with the only non-zero elements being their diagonal ones and set to be all equal to $\sqrt{D_i}$. We compute the proposal distribution q using n parallel threads. The proposal distribution for Metropolis-Hastings is symmetric and, thus, the accept/reject probabilities are simplified as shown in line 6.

- **Gibbs**: Gibbs sequentially iterates through all the non-boundary local variable $x_k \in L_i \setminus B_i$ and computes in parallel the probability $q[d_{id}]$ of each value d_{id} according to the equation:

$$q(x_k = d_{id} \mid x_l \in L_i \setminus \{x_k\}) = \frac{1}{Z_\pi} \exp \sum_{f_j \in G_i} f_j(\mathbf{z}_{|S_j})$$

  where $\mathbf{z}_{|S_j}$ is the set of value assignments for the variables in the scope S_j of constraint f_j and Z_π is the normalizing constant. We compute q using $|D_i|$ parallel threads.

To ensure that the procedure returns the best sample found, we verify whether there is an improvement on the best utility (lines 7-9). At the end of the sampling trials, it stores its best sample and utility in the r_{id}-th row in the matrix M_i and vector U_i, respectively (line 40).

7.2.3 Theoretical Properties

In what follows we present some theoretical properties regarding the convergence of the sampling process described above. Background notions about *Finite Markov Chains* and the proofs of the following properties can be found in Appendix B

Algorithm 16 CUDA Gibbs proposal distribution calculation()

```
1: $d_{id} \leftarrow$ the thread's value index of $D_i$
2: for $k = |B_i|$ To $|L_i| - 1$ do
3:     $q[d_{id}] \overset{|D_i|}{\leftleftarrows} \exp\left[\sum_{f_j \in G_i} f_j(\mathbf{z}_{|S_j})\right]$
4:     $Z_\pi \leftarrow \sum_{i=0}^{|D_i|-1} q[i]$
5:     $q[d_{id}] \overset{|D_i|}{\leftleftarrows} q[d_{id}] \cdot \frac{1}{Z_\pi}$
6:     $\mathbf{z} \leftarrow$ SAMPLE$(q(\mathbf{z} \mid \mathbf{z}^{(t-1)}))$
7: end for
```

Property 1 *If a Markov chain with given initial state $\mathbf{z}^0 = s_0$ is* irreducible *and* aperiodic*, then the chain converges to a unique stationary distribution π given enough steps.*

Definition 7.2.1 *For an agent a_i, the top α_i-percentile solutions S_{α_i} is a set of solutions for the local variables L_i that are no worse than any solution in the supplementary set $D_i \backslash S_{\alpha_i}$, and $\frac{|S_{\alpha_i}|}{|D_i|} = \alpha_i$. Given a list of agents $a_1 \ldots a_m$, the top $\bar{\alpha}$-percentile solutions $S_{\bar{\alpha}}$ is defined as $S_{\bar{\alpha}} = S_{\alpha_1} \times \ldots \times S_{\alpha_m}$.*

Property 2 *After $N_i = \frac{1}{\alpha_i \epsilon_i}$ number of samples with an MCMC, the probability that the best solution found thus far $\mathbf{z}_{N_i}$ is in the top α_i for an agent a_i is at least $1 - \epsilon_i$:*

$$P_{\mathrm{T}}\left(\mathbf{z}_{N_i} \in S_{\alpha_i} \mid N_i = \frac{1}{\alpha_i \cdot \epsilon_i}\right) \geq 1 - \epsilon_i.$$

Theorem 7.2.2 *Given m agents $a_1, \ldots, a_m \in \mathcal{A}$, and a number of samples $N_i = \frac{1}{\alpha_i \cdot \epsilon_i}$ $(i = 1, \ldots, m)$, the probability that the best complete solution found thus far $\mathbf{z}_{\mathbf{N}}$ is in the top $\bar{\alpha}$-percentile is greater than or equal to $\prod_{i=1}^{m}(1 - \epsilon_i)$, where $\mathbf{N} = \bigwedge_{i=1}^{m} N_i$. In other words,*

$$P_{\mathrm{T}}\left(\mathbf{z}_{\mathbf{N}} \in S_{\bar{\alpha}} \mid \mathbf{N}\right) \geq \prod_{i=1}^{m}(1 - \epsilon_i).$$

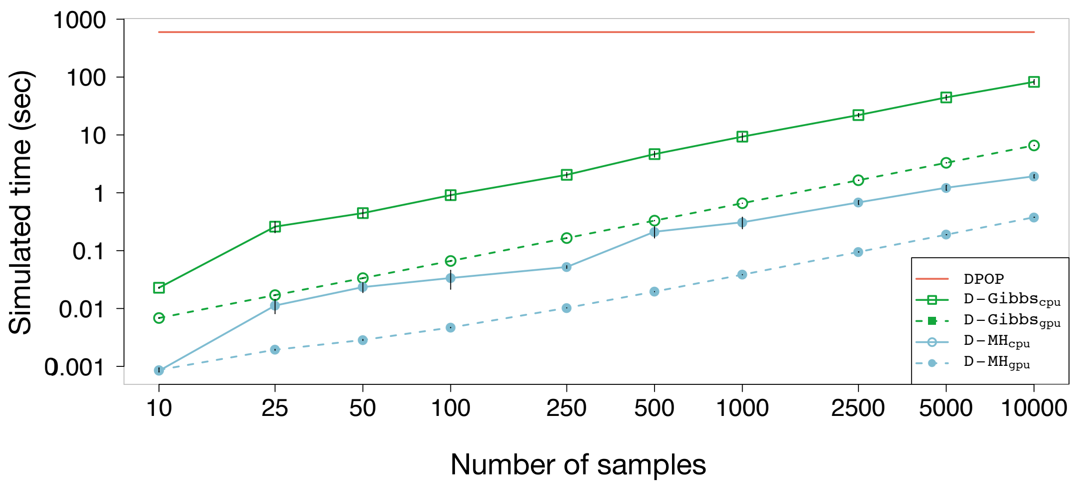
Property 3 *The number of messages required by DMCMC is linear in the size of the agents.*

Property 4 *The memory requirement of each DMCMC agent is exponential in the induced width of the problem.*

7.3 Experimental Results

We consider CPU and GPU versions of the DMCMC framework with Gibbs (D-Gibbs) and Metropolis-Hastings (D-MH) as the MCMC sampling algorithms. The CPU versions sample in sequence while the GPU versions sample in parallel with GPUs. We compare them against DPOP [130] (an optimal/complete algorithm), MGM and MGM2 [110] (sub-optimal/incomplete algorithms). Let us observe that we did not compare against Distributed Gibbs as the authors' implementation does not handle hard constraints, and we do not compare against DUCT as no public implementation is available. We use publicly-available implementations of these algorithms, which are implemented in the FRODO framework [99]. We run our experiments on a machine with a 2.4GHz CPU, a 14-multiprocessor 448-core 1.15GHz GPU, and 32GB of RAM. We measure runtime using the simulated time metric [161] and perform evaluations on meeting scheduling and smart grid network problems.

Meeting Scheduling: In these problems, meetings need to be scheduled between members of a hierarchical organization, taking restrictions in their availability as well as their priorities into account [111]. Figures 7.3 and 7.4 show the average (a, c) and the median (b, d) results for 100



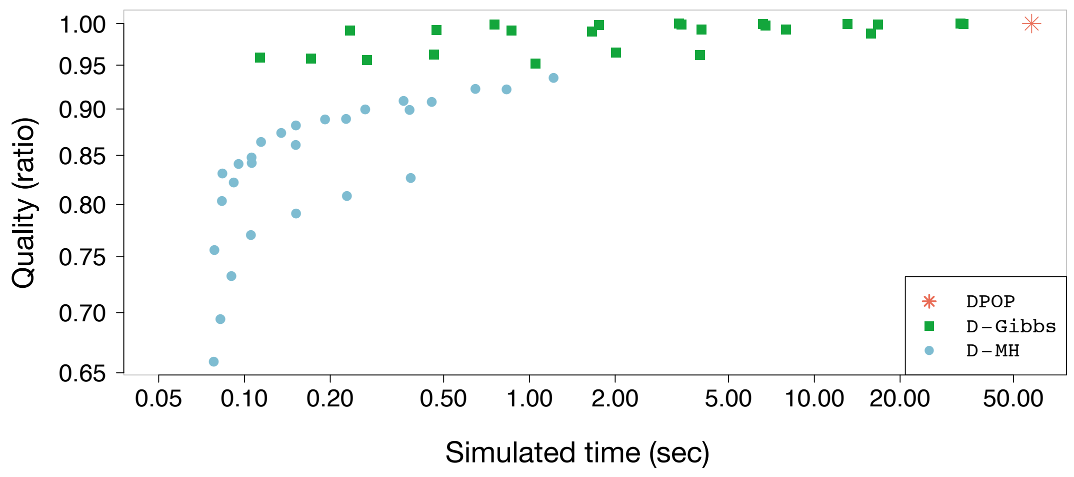


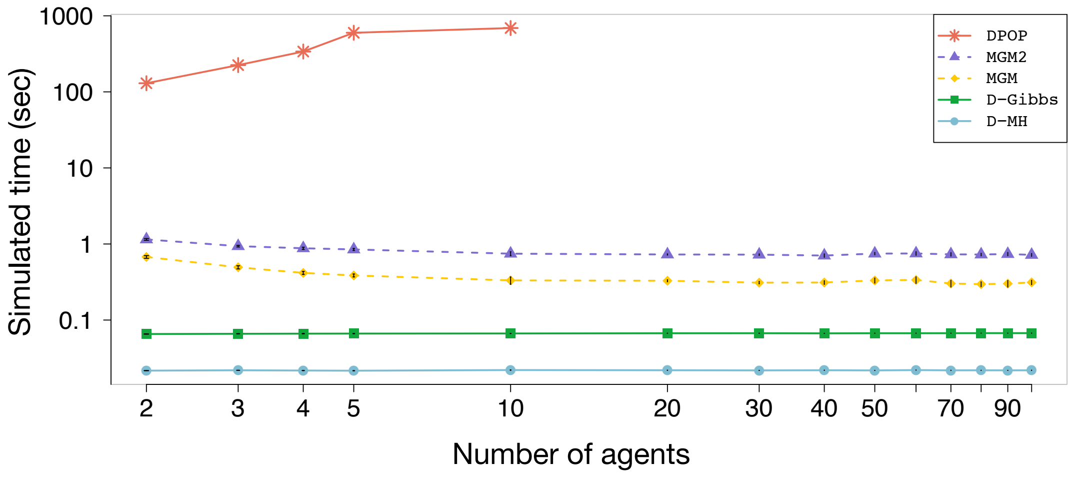
Figure 7.3: Experimental Results: Meeting Scheduling. (a) - top, (b) - bottom.

runs, together with the standard deviations (vertical bars) of problem instances with a variable number of agents and fixing each agent's number of variables to 10, the domain size of its variables to 12, its local constraint graph density to 0.7, and its number of boundary variables to 1.

We first compare the performance of the CPU and GPU DMCMC algorithms on an instance of the meeting scheduling problem with 5 agents. Figure 7.3 (a) shows the runtimes of the CPU (solid line) and GPU (dotted line) versions of DMCMC together with DPOP (solid horizontal line). The results shows that there is a clear benefit to parallelizing the sampling operations with GPUs, exhibiting more than one order of magnitude speed up. In the rest of the experiments, we show the GPU version only.

Figure 7.3 (b) shows the tradeoff between quality and runtime for the D-Gibbs and D-MH for a range of initial parameters $R = \{1, 10, 50, 100\}$ and $T = \{100, 250, 500, 1000, 5000, 10000\}$. The prediction quality increases with increasing R and T. D-Gibbs is slower than D-MH, as it requires computing normalization constants, which is computationally expensive even when parallelized. However, D-Gibbs finds better solutions.

Finally, we evaluate the algorithms in 14 benchmarks where we vary the number of agents $|\mathcal{A}|$ from 2 to 100. We set $S = 100$ and $R = 10$ for D-Gibbs and $S = 500$ and $R = 100$ for D-MH. Figures 7.4 (c) and (d) show the runtime and solution qualities, respectively. DPOP ran



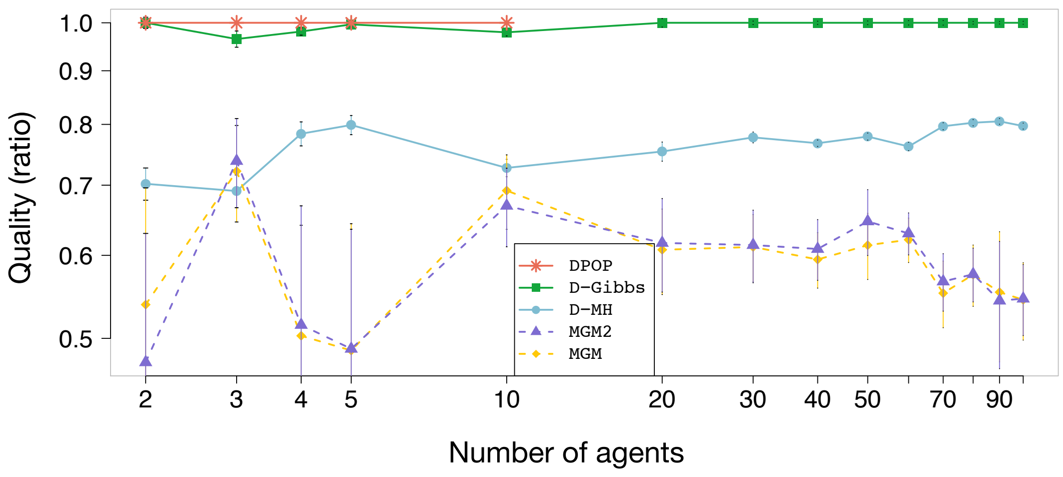


Figure 7.4: Experimental Results: Meeting Scheduling. (c) - top, (d) - bottom.

out of memory for problems with more than 10 agents. The DMCMC algorithms are up to 2 order of magnitude faster than MGM and MGM2 and can find better solutions, demonstrating the strength of sampling-based approaches over incomplete search algorithms. The results are statistically significant (p-values $< 10^{-10}$ for all parameter configurations).

Alg.	$\|\mathcal{A}\| = 100$		$\|\mathcal{A}\| = 250$		$\|\mathcal{A}\| = 500$	
D-MH	0.025	(0.01%)	0.026	(0.02%)	0.031	(0.00%)
D-Gibbs	1.387	(1.72%)	1.285	(1.72%)	1.318	(1.71%)
DPOP	15.58	(0.00%)	59.06	(0.00%)	70.01	(0.00%)

Table 7.3: Experimental Results: Smart Grid Networks

Grid Networks. We generate smart grid network problems [69] with clustered scale-free graph topologies, where each cluster has a few high density nodes. We generated 3 problem instances where we vary the number of agents $|\mathcal{A}| = \{100, 250, 500\}$ and the number of local variables of each agent depends on the number of neighboring agents. We fix the domain sizes to 11 and the maximum constraint arity to 5. Table 7.3 reports the simulated runtimes (in seconds) and the error in solution quality (in parenthesis). These results show that the DMCMC algorithms can find close-to-optimal solutions significantly faster than DPOP. We omit MGM and MGM2 as they always found unsatisfactory solutions due to the large number of hard constraints in the problem.

7.4 Summary

Motivated by two recent developments—(*i*) the recent introduction of sampling-based DCOP algorithms, which have been shown to outperform existing incomplete DCOP algorithms, and (*ii*) the advances in Graphical Processing Units (GPUs)—we take the first step towards harnessing the power of parallel computation of GPUs to solve DCOPs. In this dissertation, we introduce the GPU-based Distributed MCMC framework, which decomposes a DCOP into independent subproblems that can each be sampled in parallel by GPUs. Our experimental results show that it can find reasonably good solutions up to one order of magnitude faster than MGM and MGM2. These results demonstrate the potential for using GPUs to scale up DCOP algorithms, which is exciting as GPUs provide access to hundreds of computing cores at a very affordable cost.

III

Parallel Constraint Solving: Case Study

8 The Protein Structure Prediction Problem on GPU

This chapter realizes the main results presented so far in this dissertation into a prototype parallel solver for a hard combinatorial real-world problem, namely the *Protein Structure Prediction (PSP)* problem. The proposed approach relies on a *Multi-Agent System (MAS)* perspective, where concurrent agents explore the folding of different parts of a protein. The strength of the approach lies in the agents' ability to apply different types of knowledge, expressed in the form of *declarative constraints*, to prune the search space of folding alternatives. We demonstrate the suitability of a GPU approach to implement such MAS infrastructure, with significant performance improvements over the sequential implementation and other methods.

8.1 Introduction

In this chapter, we tackle the PSP problem using a perspective that builds on the methodologies inherited from the *Multi-Agent Systems (MAS)* domain. The proposed MAS approach is used to concurrently explore and then assemble foldings of local segments of the protein. Distinct agents are in charge of retrieving, filtering, and coordinating local information about parts of a protein, aiming to reach a global consensus. Relationships among substructures are described and exploited in terms of *constraints*—where a constraint is a high-level and declarative specification of required mutual relationships among entities. In our case, the proposed constraints deal with spatial relationships among parts of proteins being configured and assembled. A strength of constraint-based methods is their *elaboration tolerance,* that allows the incremental addition of new knowledge about the protein (e.g., properties of the amino acids, knowledge about specific substructures) without the need of redesigning the solving mechanisms. Thus, any new knowledge about a protein can be readily integrated and used to prune the space of potential conformations. Furthermore, constraint-based methods offer the power of *propagation* of any decision made during construction of a protein conformation, immediately removing infeasible branches of search space as each decision is performed. Indeed, the interest towards constraint-based methods for structural bioinformatics has grown in recent years—the reader is referred to [11, 37] for recent surveys.

In what follows, we present a solver that performs a constraint-based local search, distributed among several agents. The computation proceeds until a local minimum is found. Experimental results confirm that the local minimum reached captures with good precision the actual shape of the protein being studied. Agents use GPU cores to explore large portions of the search space and to propagate constraints on the ensemble of structures produced by each agent. The solver is capable of achieving excellent performance and demonstrates speedups over a sequential solver on a large pool of benchmarks.

8.2 Problem Formalization

Given a primary sequence $\vec{a} = a_1 \cdots a_n$ of a protein of length n (each a_i is a symbol representing an amino acid), we model the PSP problem as a COP $\mathcal{Q} = (\langle X, D, C \rangle, E)$, where X, D, C, E are described in the following subsections.

8.2.1 Variables and domains

$X = \mathcal{X} \cup \mathcal{P}$ is a set of finite domain variables, where $\mathcal{X} = \{x_1, y_1, \ldots, x_n, y_n\}$, and $\mathcal{P} = \{p_1, \ldots, p_{15n}\}$ (hence, $m = |X| = 17n$). The variables x_i and y_i (for $1 \leq i \leq n$) are associated to the torsional angles ϕ and ψ of the i^{th} amino acid, respectively.

The variables $p_{5(i-1)+3t+1}, p_{5(i-1)+3t+2}, p_{5(i-1)+3t+3}$ ($i = 1, \ldots, n$ and $t = 0, \ldots, 4$) are associated to the x, y, z coordinates of the t^{th} atom of the i^{th} amino acid a_i. More precisely, if $t = 0$ then it is an N-atom, referred as n_i; if $t = 1$ then it is the C_α-atom, referred as C_{α_i}; if $t = 2$ then it is a C-atom, referred as c_i; if $t = 3$ then it is an O-atom, referred as o_i; and if $t = 4$ then it is a H-atom, referred as h_i. We will denote the above list of 15 coordinates as $\vec{\mathcal{P}}_i = \langle n_i, C_{\alpha_i}, c_i, o_i, h_i \rangle$ and we refer to these variables as *point* variables.

D^{x_i} and D^{y_i} will store sets of angles retrieved from a database of proteins using statistical information (we use DASSD [141]). The domains for all the point variables are initially set all equal to the range $[-500000..500000]$, that has been experimentally proved to be large enough to accommodate all proteins tested in our benchmarks.

8.2.2 Constraints

C is a finite set of constraints over X. These constraints describe geometric properties that the final structure must satisfy to be a physically admissible structure. There is a strong correlation between $\mathcal{X}$ and $\mathcal{P}$, allowing us to infer the first from the second and vice versa. Keeping both types of variables explicit allows the programmer to easily add specific angle or spatial constraints. In particular, assignments of angle variables in $\mathcal{X}$ identify structures that are represented by ground points in $\mathcal{P}$. On the other hand, structures obtained by constraint propagation over variables in $\mathcal{P}$ identify a unique sequence of pairs of angles ϕ and ψ for the variables in $\mathcal{X}$.

Let us introduce the most relevant constraints used to encode the PSP problem.

The `table` Constraint

Values for variables in $\mathcal{X}$ are retrieved from a statistical database. These values are given as pairs $\langle \phi, \psi \rangle$. Therefore, for each $i = 1, \ldots, n$, we consider a *table* constraint that associates $\langle x_i, y_i \rangle$ to the possible admissible pairs for those angles. As a result, during the search, the assignment of x_i and y_i is done simultaneously according to the table.

The `alldistant` Constraint

This constraint has been originally introduced by [40]. Given a list of $3k$ variables $\mathcal{P} = (p_1, \ldots, p_{3k})$, and a list of k positive values $\vec{d}$, the constraint `alldistant`$(\mathcal{P}, \vec{d})$ imposes a distance relation between each pair of 3D points identified by the variables in $\mathcal{P}$, i.e., $\forall i \in \{1, \ldots, k-1\}$ and $\forall j \in \{i+1, \ldots, k\}$:

$$\| \underbrace{(p_{(i-1)3+1}, p_{(i-1)3+2}, p_{(i-1)3+3})}_{A} - \underbrace{(p_{(j-1)3+1}, p_{(j-1)3+2}, p_{(j-1)3+3})}_{B} \| \geq d_i + d_j$$

where $\| \cdot \|$ is the Euclidian norm. d_i and d_j can be seen as the radii of two spheres centered in the points A and B, respectively. The constraint states that these two spheres cannot intersect.

We use this constraint to model the fact that the various atoms have a minimum distance each other. In particular, the value d_i is chosen as the radius of a sphere containing the atom of coordinates $(p_{(i-1)3+1}, p_{(i-1)3+2}, p_{(i-1)3+3})$. An additional `alldistant` constraint, that considers only the C_α atoms of the amino acids, can also be added, imposing a minimum distance that depends on the the radii of sphere that contain their respective amino acids. These radii are computed from an average analysis in a database of known proteins.

The Single Angle (sang) Constraint

Given the list $\mathcal{X}$ of $2n$ FD variables and the list $\mathcal{P}$ of $15n$ point variables, and given $i \in 2, \ldots, n$, the *single angle* constraint $\texttt{sang}(i, \mathcal{X}, \mathcal{P})$ imposes a relation between the lists of points of the 15 variables $\vec{\mathcal{P}}_{i-1}$ and the subsequent 15 variables $\vec{\mathcal{P}}_i$ so as to satisfy:

$$\vec{\mathcal{P}}_i \in \{\mathbf{Rot}(\vec{\mathcal{P}}_{i-1}, \phi, \psi) \,:\, \langle \phi, \psi \rangle \in D^{x_i} \times D^{y_i}\}$$

where $\mathbf{Rot}(\cdot)$ is the roto-translation matrix needed to properly align the amino acid structure described by the angles $\langle \phi, \psi \rangle$ with the position of the previously placed atoms in $\vec{\mathcal{P}}_{i-1}$ (see Fig. 8.1). We use this constraint to model the relative positions of consecutive tuples of atoms related to amino acids $i-1$ and i according to the angles selected.

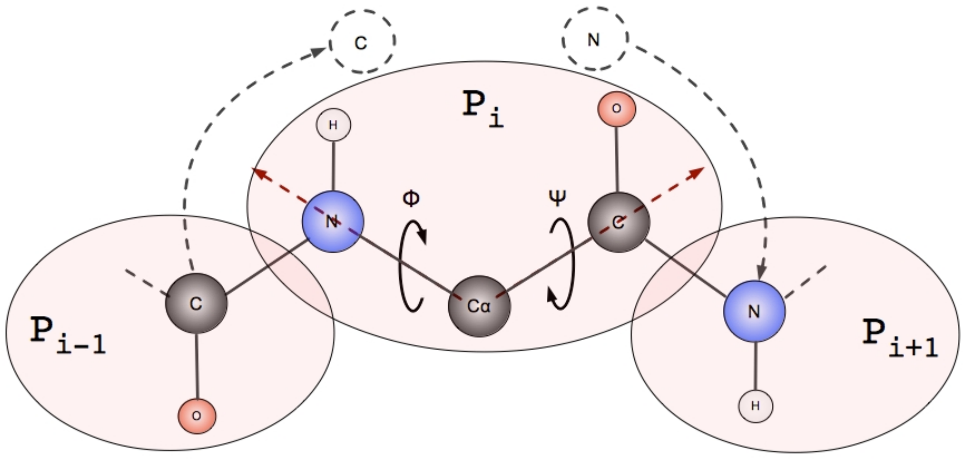


Figure 8.1: Rotation of the vector C_α–N (angle ϕ) or the vector C_α–C (angle ψ)

8.2.3 The Cost Function E

Since our aim is to find the tertiary structure that minimizes the free energy of the protein, we used as cost function a protein *energy function* already used in literature (e.g., the one adopted in [36]), composed of three components, described below:

1) *Hydrogen component:* Hydrogen bonding potentials are calculated from pairs of atoms N–H of amino acid i (n_i, h_i) and an O atom of another amino acid j (o_j), that are located within a certain distance threshold; an auxiliary statistical table $\texttt{tab}_h$ is used [118]. This contribution is calculated by a function $\texttt{Hydro}(X)$ defined as follows:

$$\texttt{Hydro}(X) = \sum_{i=1}^{n} \sum_{j=1, j\neq i}^{n} hc(X, i, j)$$

where $hc(\cdot)$ returns the energy potential of one hydrogen bond:

$$hc(X,i,j) = \begin{cases} \texttt{tab}_h(\delta(X,i,j), \Theta(X,i,j), \Psi(X,i,j), \Gamma(X,i,j)) & \text{if } 1.75 \leq \delta(X,i,j) \leq 2.60 \\ 0 & \text{otherwise} \end{cases}$$

where: $\delta(X,i,j) = \|h_i - o_j\|$, $\Theta(X,i,j)$ is the bond angle $\widehat{o_j\, h_i\, n_i}$, $\Psi(X,i,j)$ is the bond angle $\widehat{C_{\alpha_j}\, o_j\, h_i}$, and $\Gamma(X,i,j)$ is the torsional angle identified by $n_i, h_i, C_{\alpha_j}, o_j$.

2) *Contact component:* it calculates the contribution of the contact of each pair of centroids of the side chain using the statistical table of contact energies $\texttt{tab}_c$ [16]. This component considers a threshold distance equal to the sum of the Van der Walls radii of the side chains of the amino acids involved (the van der Waals radius of an atom is the radius of an imaginary hard sphere which can be used to model atoms) If the distance is greater than such threshold, the potential decreases quadratically. The contact component $\texttt{Cont}(X, \vec{a})$ is defined as follows:

$$\texttt{Cont}(X, \vec{a}) = \sum_{i=2}^{n-1} \sum_{j=i+1}^{n-1} \mathit{contact}(\gamma(X, i, a_i), \gamma(X, j, a_j), a_i, a_j)$$

where $\vec{a}$ is the sequence of amino acids and the function $\gamma(X, i, a)$ returns the position of the side chain centroid of the amino acid i, which is dependent on the type of the amino acid a and the points $C_{\alpha_{i-1}}, C_{\alpha_i}, C_{\alpha_{i+1}}$. Let us observe that the first and the last centroids of the structure are not taken into account since the tails of the protein do not contribute significantly to the energy value. The $\mathit{contact}(\cdot)$ function returns the contact potential computed in [16], retrieved by the indexing function $\texttt{tab}_c$:

$$\mathit{contact}(p, q, a, a') = \begin{cases} \texttt{tab}_c(a, a') & \text{If } \|p - q\| \leq VdW(a, a') \\ \texttt{tab}_c(a, a') \frac{VdW(a,a')^2}{\|p-q\|^2} & \text{Otherwise} \end{cases}$$

where $VdW(a, a')$ is the sum of the Van der Walls radii of amino acids a, a'.

3) *Correlation component*: Let Λ_i be the torsional angle determined by the C_α atoms C_{α_i}, $C_{\alpha_{i+1}}$, $C_{\alpha_{i+2}}$, $C_{\alpha_{i+3}}$. Two statistically computed tables $\texttt{tab}_{t_1}$ and $\texttt{tab}_{t_2}$ are are used as auxiliary functions. The first one retrieves information from one torsional angle Λ_i parametrized by the type of the amino acids a_{i+1} and a_{i+2}. The second table retrieves information from the pair of torsional angles Λ_{i-1} and Λ_i independently of the amino acids' types. The correlation component is computed as follows (see [56] for the physical backgrounds of this component):

$$\texttt{Corr}(X, \vec{a}) = \sum_{i=2}^{n-4} \texttt{tab}_{t_1}(\Lambda_i, a_{i+1}, a_{i+2}) + \texttt{tab}_{t_2}(\Lambda_{i-1}, \Lambda_i)$$

Finally, we set:

$$E(X, \vec{a}) = w_1 \texttt{Hydro}(X) + w_2 \texttt{Cont}(X, \vec{a}) + w_3 \texttt{Corr}(X, \vec{a})$$

Experimental training on the *top500* dataset (trying to reduce the standard deviation of the energies calculated w.r.t. each component) produces the following values: $w_1 = 8, w_2 = 0.1, w_3 = 7$ for predicting α/β structures, and $w_1 = 8, w_2 = 22, w_3 = 7$ for predicting turns and loops. All auxiliary tables used are available in `http://www.cs.nmsu.edu/fiasco/`.

The literature on precise energy functions for assessing protein foldings is extensive (see, e.g., [74, 148, 138]). Our approach is completely parametric w.r.t. the energy function.

8.3 A Multi-Agent System Architecture

A *Multi-Agent System (MAS)* is a system composed of several agents in an environment, each with a certain degree of autonomy, and collaborating with other agents to solve a common problem [168]. These agents have only a local and partial view of the global system, which is too complex to be handled by a single agent. The basic structure of an agent is shown in Figure 8.2 (left). Usually, an agent receives information from the external environment, it processes the information, and it uses a predefined strategy to determine which action to perform—the action has the potential to affect the environment, by making changes to it.

In this work, we use a multi-level MAS to compute the folding process. We define four types of agents: **(1)** A *Supervisor* agent, which is the highest level agent in the system, and it coordinates all of the other agents, **(2)** *Structure* agents, and **(3)** *Coordinator* agents, which are associated to the secondary structure elements of the target protein; both the structure and coordinator

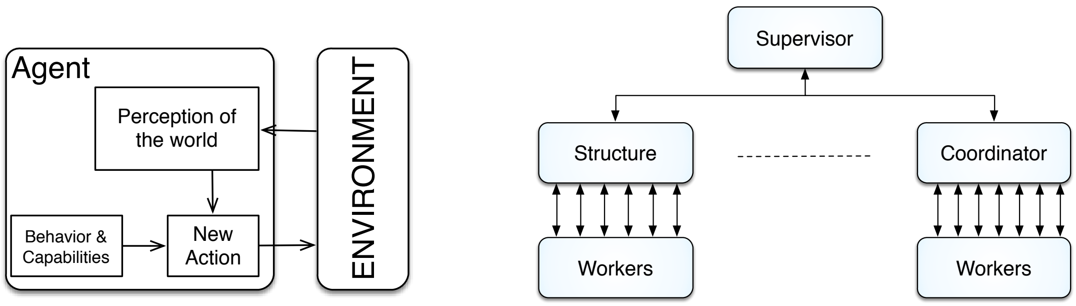


Figure 8.2: Basic structure of an agent and Multi-Agent System architecture

agents implement their own `search` methods; **(4)** *Worker* agents, which are associated to the $\mathcal{X}$ variables of the constraint model. The worker agents are in charge of propagating constraints and labeling the associated variables and, therefore, they are implemented as device kernel functions. The communication between agents passes through the *Supervisor* agent, since it is the only agent that has complete knowledge about the folding process during the complete search phase. The Supervisor agent as well as the Structure and Coordinator agents are implemented on the host. Figure 8.2 (right) shows the architecture of the multi-agent system with the four types of agents. Let us consider a target protein of length n with s secondary structures elements involving p amino acids. s structure agents will be activated and coordinated by the supervisor agent. The supervisor agent will also activate one coordinator agent to take care of the remaining $s-1$ internal loops/turns and of the initial and final tails. More than one coordinator agents (e.g., one for each loop or turn) could be specified by the user to the supervisor agent as input parameter. n worker agents are created, one for each amino acid a_i, assigned as auxiliary agents for the corresponding structure/coordinator agent. At each instant of the computation either p (if we are computing secondary structures elements) or $n-p$ (if we are coordinating structures looking for loops and turns) workers will run in parallel.

8.3.1 The Supervisor Agent

The *Supervisor* agent is the intermediary between the external world (e.g., user, external knowledge) and the other agents. It represents the highest level of abstraction in the framework and it holds global spatial information about the protein structure during the complete folding process. Its main task is to assign different sub-sequences of the primary sequence to the immediately underlying types of agents—the *coordinator* and the *structure* agents—and to supervise them, in order to guide the entire folding process towards a stable global configuration.

It is also responsible for creating the set of *worker* agents associated to the coordinator agents and to the structure agents. Moreover, it imposes both the search strategies and the constraints to be propagated on the other agents.

The supervisor agent sets a priority order among the agents. First, each "highly constrained" secondary structure (α-helix/β-sheet) is computed by the structure agents; afterwards, the coordinator agents are invoked to model the whole tertiary structure by moving *loops* or *turns*. In this work, we will restrict to a single coordinator agent. The supervisor agent must ensure also that the structures determined by each structure agent can be combined in order to obtain the global structure that can be effectively folded by the coordinator agent. To discard symmetric equivalent solutions, the secondary structure $a_i \cdots a_j$ with the lowest i among those computed is deterministically placed in the 3D space. The positions of all other points are obtained starting from it, as soon as the angles are assigned to variables x, y.

The location in the primary sequence and the type of the secondary structure elements is done by the supervisor agent using some of the capabilities of the secondary structure prediction algorithm *JNet* [30]. We remark that only the location is delegated to external knowledge, while the actual folding process is performed by the structure agents.

8.3.2 The Worker Agent

Worker agents are the lowest-level agents in the system. Each worker agent is associated either to a coordinator agent or to a structure agent and takes care of a particular amino acid a_i. In particular, it assigns all admissible values to the pair of variables x_i, y_i and executes consequent constraint propagation possibly rejecting inconsistent assignments. Precisely, propagation is performed to the portion of the constraint problem delegated to the structure/coordinator agent associated with it. A set of admissible structures (i.e., a set of structures that satisfy all the constraints the agent can see) is returned to the agent at the higher level.

This separation between high-level agents and worker agents allows us to not worry about how effectively the propagation is performed (e.g., choosing between CPU and GPU), but to solely invoke the propagation function of the worker agent from the structure or the coordinator agent, and choosing the best labeling according to an appropriate criterion and the type of constraint imposed on the variable associated to the worker agent.

8.3.3 The Structure Agent

This agent models secondary structure elements, such as α-helices and β-sheets. There is a structure agent for each secondary structure element of the target protein. Thus, each structure agent works exclusively on a specific part of the overall structure—i.e., it solves a folding problem on a limited subsequence of amino acids. It is not the task of the structure agent to maintains relations between the assigned subsequence and the rest of the structure (this will be handled by the coordinator agents).

The agent implements a *coordinate-wise gradient descent* method (e.g., see [17]). We call this search strategy *Iterated Conditional Mode* (*ICM*), implemented by Algorithm 18. This search strategy iterates to “greedily” refine a first feasible solution, as described in the next sections. This *greedy* strategy is suitable to secondary structure elements, since α-helices and β-sheets are “highly constrained” structures presenting a strong energy correlation between pairs of atoms; this implies a strong gap between geometrically stable structures and unfolded structures. Let us recall that our solution uses an independent system for secondary structure prediction to suggest type and locations of secondary structures.

8.3.4 The Coordinator Agent

The *coordinator agent* folds the protein by determining loops and turns that connect the secondary structure elements. x_i, y_i variables in already computed helices and sheets from the structure agents are already assigned; their related Cartesian variables $\mathcal{P}_i$ are instead unassigned: this allows to “move” structures as rigid objects. Since loops and turns are, in general, poorly structured, thus generating an intractable search space, the folding process is driven by a sampling of the search space. We compute the energy values of a set of structures to obtain more discriminant energy values. Coordinator agents adopt a search strategy that generalizes the one used by the structure agents and it is summarized in Algorithm 18 of Sect. 8.4.1. In particular, two sub-strategies have been implemented: a variant of the *Gibbs* [17] sampling search strategy, and a *Monte Carlo* search strategy (see Sections 8.4.2–8.4.3).

8.4 General search schema

The structure and coordinator agents execute a large neighborhood search on a COP. The general schema of the LNS is implemented by Algorithm 17. The inputs provided to the algorithm are: the list $\mathcal{X}$ of $2n$ variables for angles, the list $\mathcal{P}$ of $15n$ point variables, and a list of *atom sizes* $\vec{d}$. Let us observe that each agent receives only a part of the whole protein, thus n is, in this case, not the number of amino acids of the whole protein, but only of a sub-part of it.

The first step of the algorithm (lines 3–7) is to post all the constraints required for the resolution of the problem—i.e., the `table` constraints that relate pairs of angles for variables x_i, y_i, the `sang` constraints that link point variables and corresponding FD variables, and an `alldistant` constraint over the set of point variables. Let us observe that this step is done only once, before the search begins.

The next step is to compute a first solution for the set of constraints (line 8)—this is a rather trivial task, e.g., by determining a solution that connects the most "straight" fragments; this allows us to satisfy the `alldistant` constraint in a trivial manner. The energy of this initial solution is computed in line 9.

Algorithm 17 Search($\mathcal{X}, \mathcal{P}, \vec{d}$)

```
 1: - Constraints:
 2: $n \leftarrow |\mathcal{X}|/2$;
 3: post table $\langle\phi, \psi\rangle$ constraint on $|\mathcal{X}|$;
 4: post constraint: alldistant($\mathcal{P}, \vec{d}$);
 5: for $i \leftarrow 2$ to $n$ do
 6:   post constraint: sang($i, \mathcal{X}, \mathcal{P}$);
 7: end for
 8: $\mathcal{S} \leftarrow$ first_solution($\mathcal{X}, \mathcal{P}$);
 9: current_energy $\leftarrow$ compute_energy($\mathcal{S}$); {Other parameters are omitted}
10: - Search:
11: if (Agent = Structure) then
12:   repeat
13:     best_energy $\leftarrow$ current_energy;
14:     ($\mathcal{S}$, current_energy) $\leftarrow$ icm($\mathcal{X}, \mathcal{P}$);
15:   until (current_energy $\geq$ best_energy)
16: else if (Agent = Coordinator) $\wedge$ (Search = Montecarlo) then
17:   repeat
18:     best_energy $\leftarrow$ current_energy;
19:     ($\mathcal{S}$, current_energy) $\leftarrow$ mc($\mathcal{X}, \mathcal{P}$);
20:   until (current_energy $\geq$ best_energy for $k$ consecutive times)
21: else if (Agent = Coordinator) $\wedge$ (Search = Gibbs) then
22:   $\mathcal{S}^* \leftarrow \mathcal{S}$;
23:   best_energy $\leftarrow$ current_energy;
24:   for $i \leftarrow 1$ to $m$ do
25:     mc($\mathcal{X}, \mathcal{P}$); {Prepare a new starting point ($\mathcal{X}, \mathcal{P}$)}
26:     for $t \leftarrow 1$ to $n\_samples$ do
27:       ($\mathcal{S}$, current_energy) $\leftarrow$ gibbs($\mathcal{X}, \mathcal{P}$);
28:       if current_energy < best_energy then
29:         best_energy $\leftarrow$ current_energy
30:         $\mathcal{S}^* \leftarrow \mathcal{S}$;
31:       end if
32:     end for
33:   end for
34:   $\mathcal{S} \leftarrow \mathcal{S}^*$;
35: end if
36: return $\mathcal{S}$;
```

The search phase starts at line 10. The code is slightly different in the case of structure agent and the case of coordination agent.

8.4.1 ICM

The search strategy adopted for the Structure agents is described by the loop in lines 12–15, that invokes the ICM algorithm (Algorithm 18) and it terminates when the energy value cannot be further improved.

Algorithm 18 **icm**($\mathcal{X}, \mathcal{P}$)

```
1: for $i \leftarrow 1$ to $n$ do
2:     $wrk \leftarrow$ get_worker_agent($i$);
3:     $\mathcal{X} \leftarrow$ choose_best_label($wrk, \mathcal{X}, \mathcal{P}$); {Label $x_i, y_i$ again}
4:     $\mathcal{S} \leftarrow$ compute_structure[$\mathcal{X}$]; {Structure Updating}
5:     current_energy $\leftarrow$ compute_energy($\mathcal{S}$);
6: end for
7: return ($\mathcal{S}$, current_energy)
```

The ICM search strategy takes one pair of variables x_i, y_i at a time, it evaluates the energy for all their possible assignments, and they are assigned to the value that returns lowest energy. This task is performed by the worker agent i which runs the **choose_best_label** function (line 3). This function implements the strategy for selecting a new solution and it is performed in parallel on the GPU. Precisely, the variables x_i, y_i are assigned with all the elements (pair of angles) in their domains that satisfy the `table` constraint—variables $\vec{\mathcal{P}}_i$ are deterministically assigned by propagating the `sang` constraint. For each new assignment the variables x_i, y_i and the variables $\vec{\mathcal{P}}_i, \vec{\mathcal{P}}_{i+1}, \dots, \vec{\mathcal{P}}_{|X|}$ are uninstantiated. All the resulting structures consistent with the `alldistant` constraint are energetically evaluated, and the one providing the minimum value is kept in $\mathcal{X}$.

Observe that the starting configuration is in the set of structures scanned by the ICM algorithm and, therefore, a solution is ensured to exists and the energy cannot increase. Although, in principle, two structures with the same minimum energetic value might emerge in the above step, the fine-grained energy model employed makes this situation highly unlikely; as a result, in practice the algorithm converges deterministically towards a local minimum. In the case in which two equivalent structures emerge, the first one encountered is chosen.

8.4.2 Monte Carlo

The Monte Carlo search strategy is implemented by the loop in lines 17–20 of Algorithm 17. Termination is forced after k consecutive loop iterations without any energy improvement, where k is a user-selectable parameter. The search space is sampled by the function **mc** described by the Algorithm 19.

Algorithm 19 **mc**($\mathcal{X}, \mathcal{P}$)

```
1: for $i \leftarrow 1$ to $n$ do
2:     $wrk \leftarrow$ get_worker_agent($i$);
3:     $\mathcal{X} \leftarrow$ choose_best_label_random($wrk, \mathcal{X}, \mathcal{P}$); {Label $x_i, y_i$ again}
4:     $(t_{2i-1}, t_{2i}) \leftarrow (x_i, y_i)$;
5: end for
6: if (solution_check($\vec{t}$) = true) then
7:     $\mathcal{X} \leftarrow \vec{t}$ {Update consistently}
8: else
9:     $\mathcal{X} \leftarrow$ assignment_from($\mathcal{S}$); {Retrieve previous value (*)}
10: end if
11: $\mathcal{S} \leftarrow$ compute_structure[$\mathcal{X}$]; {Structure Updating}
12: current_energy $\leftarrow$ compute_energy($\mathcal{S}$); {End of Coordination Agent}
13: return ($\mathcal{S}$, current_energy)
```

The for loop scans through all the variables associated to the agent and it focuses on one pair of variables, x_i, y_i, at a time to determine a better structure. All the variables (related to loops and turns) assigned to the coordinator are uninstantiated. The worker agent implements the **choose_best_label_random** function (line 3). If x_i, y_i are not allocated to the coordinator agent, then the procedure halts immediately, leaving their values unchanged; otherwise, every admissible assignment to the pair x_i, y_i is attempted (for each attempt all the variables assigned to the coordinator and all the variables $\mathcal{P}$ are uninstantiated). A fixed number of random assignments (samples) for the other variables x_j, y_j is attempted for each assignment of x_i, y_i. The values of x_i, y_i that is part of the sample returning the best energetic value is retained. In our experiments, we set the number of samples to 1,000. As before, the values to be assigned to the point variables $\mathcal{P}$ are obtained by propagation. The values t_{2i-1}, t_{2i} for x_i, y_i that allows us to obtain the best (i.e., minimal) energetic value are kept (lines 3–4). The list $\vec{t} = (t_1, \ldots, t_{2n})$ of these values constitutes a new possible solution. If this assignment (and the subsequent propagation to $\mathcal{P}$ driven by the `sang` constraint) satisfies the `alldistant` constraint (line 6—**solution_check**), the energy is evaluated and updated (line 7), otherwise, a new iteration of the loop in lines 17–20 of Algorithm 17 is started.

Variants of this method include also the possibility of worsening solutions with some fixed probability or with a probability decreasing over time (i.e., as in simulated annealing). The changes to the code to achieve these variants are minimal—these will be explored as future work. Moreover, if the test in line 6 fails, then some small changes in the solution will be attempted before deciding to restart from the previous solution. In particular, the algorithm considers other s solutions (default $s = 2$) sorted in ascending order of energy value before performing a new iteration of the loop in lines 17–20 (Algorithm 17). If there are any admissible assignment, the first one is used as new possible solution.

8.4.3 Gibbs sampling

Gibbs sampling mixes the ICM search strategy with the Monte Carlo sampling as follows. It uses two "meta" parameters: m (the number of *starting points*) that controls the number of iterations in lines 24–33 and *n_samples* (the number of sampling steps) that controls the number of iterations in lines 26–32. The coordinator agent invokes the function **gibbs** (line 27 of Algorithm 17) to sample one variable at a time based on the current assignment of the other variables. Each starting point is a *valid* random assignment of values to variables computed by invoking the **mc** function (see, Alg. 19). Algorithm 20 tries to improve the initial random point as follows.

Initially, the current (random) structure is computed and energetically evaluated (lines 2–3). Then, for each pairs of variables x_i, y_i, the corresponding worker agent invokes the function **choose_label_random** to randomly select a pairs of elements from their domains that satisfies the `table` constraint. If the selected assignment (and the subsequent propagation to $\mathcal{P}$ driven by the `sang` constraint) satisfies the `alldistant` constraint (line 6—**solution_check**), the new energy is evaluated on an auxiliary structure $\mathcal{S}^*$ (lines 7–8). The random choice is then accepted with a probability value q that depends on the ratio between the previous energy value and the current one (lines 6–14). If the random assignment does not satisfy the `alldistant` constraint the previous assignment is retrieved from $\mathcal{S}$ (line 15).

At each step of the Gibbs sampling algorithm the assignment is updated according to a *Metropolis* sampling process (i.e., a random assignment is accepted with probability that depends on the previous energy value—lines 9–10). Variants of this method includes also the possibility of changing the acceptance ratio in line 9 by multiplying the energy values by a *temperature factor* that varies over time. Moreover, the set of starting points can be partitioned in subsets of equal size, each represented by a different temperature factor.

8.4.4 The LNS general schema

Let us show how Algorithm 17, together with its auxiliary algorithms, implements a LNS schema. After one solution is computed (line 8 of Algorithm 17), a loop looking for improving solutions is

Algorithm 20 **gibbs**$(\mathcal{X}, \mathcal{P})$

```
1: for $i \leftarrow 1$ to $n$ do
2:   $\mathcal{S} \leftarrow$ compute_structure$[\mathcal{X}]$;
3:   current_energy $\leftarrow$ compute_energy$(\mathcal{S})$;
4:   $wrk \leftarrow$ get_worker_agent$(i)$;
5:   $\mathcal{X} \leftarrow$ choose_label_random$(wrk, \mathcal{X}, \mathcal{P})$; {Label $x_i, y_i$ again}
6:   if (Solution_check( $\mathcal{X}$ ) = true) then
7:     $\mathcal{S}^* \leftarrow$ compute_structure$[\mathcal{X}]$; {Auxiliary structure updating}
8:     current_energy$^* \leftarrow$ compute_energy$(\mathcal{S}^*)$;
9:     $q \leftarrow \mathbf{min}\left(1, \frac{\exp(-\text{current\_energy}^*)}{\exp(-\text{current\_energy})}\right)$;
10:    $r \leftarrow$ rand();
11:    if $r \leq q$ then
12:      $\mathcal{X} \leftarrow$ assignment_from$(\mathcal{S})$; {Reject the new state}
13:    end if
14:  else
15:    $\mathcal{X} \leftarrow$ assignment_from$(\mathcal{S})$; {Retrieve previous value (*)}
16:  end if
17: end for
18: $\mathcal{S} \leftarrow$ compute_structure$[\mathcal{X}]$;
19: current_energy $\leftarrow$ compute_energy$(\mathcal{S})$;
20: return ($\mathcal{S}$, current_energy)
```

entered.

In the case of the structure agent, the auxiliary procedure 18 generates sequentially n "neighborhoods", each obtained by releasing the assignments of the 17 variables corresponding to amino-acid i; neighbors are the set of assignments for such variables that satisfy all the constraints.

In the case of the coordination agent, we have two options: with Monte Carlo and with Gibbs. As far as Monte Carlo is concerned, the situation is similar to the structure agent case, with the difference that we do not optimize the neighbor at every step, but simply obtain an optimum of a sample of a subset of the neighbors. With a large number of samples (e.g. 1,000) this guarantees good results. As far as Gibbs is concerned, the situation is again similar: an improving solution (if any) is selected.

In this framework, GPU is used to speedup the exploration of large neighborhoods. The use of GPUs architectures is not new for speeding up LNS strategies. For example, a guideline for design and implementation of LNS strategies on GPUs is presented in [107].

8.5 Some Implementation Details

We implemented, in C++, a constraint solver that exploits parallelism on GPUs to explore the search space, following the previously described multi-level MAS model. As anticipated, locations of secondary structures are computed by a secondary structure prediction algorithm based on neural networks and sequence similarity alignments—specifically, the JNet application (`http://www.compbio.dundee.ac.uk/www-jpred/advanced.html`). This descriptions can be provided by other secondary structure alignments tools (e.g., PSIPRED, see `http://bioinf.cs.ucl.ac.uk/psipred/`) or by the user. In this section, we provide some implementation details about this solver.

8.5.1 Constraints

The `table` constraint is simply used for pairs of simultaneous assignments to variables x_i and y_i. Let us make some observations regarding how the other constraints can be efficiently handled on a GPU.

The `alldistant` constraint is checked on the whole set of point variables representing a protein structure. Using a sequential algorithm, the test of consistency of this constraint on a list of 3D coordinates of length n requires time $\mathcal{O}(n^2)$. In our implementation, we map this consistency check and constraint propagation to distinct cores of the GPU—by enabling each core to serve as representative of a different quintuple of atoms, that defines each amino acid of the structure. This allows us to reduce the complexity to $\mathcal{O}(n)$.

The propagation of the `sang` constraint relies on a kernel that is invoked with n threads, if the constraint is imposed on a list of $15n$ point variables. Each thread deals with a different quintuple of point variables, reducing the computation from $\mathcal{O}(n)$ to $\mathcal{O}(1)$ time. Further parallelism is obtained by calling the subroutine by all threads generating viable candidates within the implementation of the ICM search procedure.

8.5.2 Energy

The implementation of the energy function used in this work (see Sect. 8.2.3) as a CUDA kernel requires the introduction of two levels of parallelism:

1. Given a set of admissible structures, the energy value of each structure is calculated in parallel by a number of blocks equal to the size of the set, and

2. For a given structure, each energy field is calculated in parallel by a thread within the block.

To obtain a linear time computation for the *contact* and the *hydrogen* potentials, we adopt the same strategy used for the `alldistant` constraint: if we consider a structure of length n, then we use n threads for both the *contact* potential and the *hydrogen bond* potential, while we use a single thread for the *correlation* potential. The total energy value is calculated in $\mathcal{O}(n)$ time. Further parallelism is obtained by calling the subroutine by all threads generating viable candidates.

8.5.3 Subroutines of the Search Algorithms

Subroutines are executed with references to an amino acid $i = 1, \dots, n$. Let k be the number of pairs that can be assigned to variables x_i, y_i. Then the function **choose_best_label** is implemented by a CUDA kernel invoked with with a number of blocks equal to k and a number of threads equal to n (the average number of the pairs of angles allowed is $k = 107$). Each kernel block $b = 1, \dots, k$ considers an assignment $\langle \phi_b, \psi_b \rangle$ and each thread j computes the alignment for the $j^{\mathbf{th}}$ atoms $\vec{\mathcal{P}}_j$. Let us observe that the alignment of the atoms $\vec{\mathcal{P}}_j$ w.r.t. the previously placed atoms $\vec{\mathcal{P}}_{j-1}$ is deterministic, provided the positions of at least one list of points $\vec{\mathcal{P}}_l$, for $1 \leq l \leq n$ (see Sect. 8.2.2). Hence, the roto-translation matrices $\mathbf{Rot}(\cdot)$ can be computed independently by each thread once any 15-tuple of point $\vec{\mathcal{P}}_l$ variables is assigned. The cost of this subroutine is then $\mathcal{O}(1)$ time instead of $\mathcal{O}(kn)$ time of a sequential implementation.

A similar idea is applied for the subroutine **choose_best_label_random**. Let k be the number of pairs that can be assigned to variables x_i, y_i. Then a kernel CUDA is invoked a number of blocks $m \geq k$ equal to the number of random samplings to be performed. Let us observe that the average number of the pairs of angles allowed is 389. We have tested values of k from 500 to 3000 and we have observed that the best behavior for the system is for values for k close to 1000. The first k blocks performs exactly the same computation as the **choose_best_label** function—thus, simply exploring the impact of modifying only the angles associated to the x_i, y_i variables. The remaining $m - k$ blocks are subdivided in k groups, each corresponding to a different possible assignment of the variables x_i, y_i; these $m - k$ blocks are in charge of the actual random sampling. Each one of these blocks starts with the initial configuration of the variables x_i, y_i—determined by which of the k groups the block belongs to. The block then continues by determining the random assignments for all the other pairs of variables x_j, y_j, with $i \neq j$ (among those variables allocated to the coordinator agent), as described in Sect. 18. The roto-translation matrices $\mathbf{Rot}(\cdot)$ differ every time the alignment for a new list of point variables $\vec{\mathcal{P}}_{j+1}$ is computed based on the previous list of point variables $\vec{\mathcal{P}}_j$. The block makes use of two threads to speed-up such computation. In particular, we use one thread to perform the sequential alignment on the lists of point variables $\vec{\mathcal{P}}_1, \ldots, \vec{\mathcal{P}}_{j-1}$, while the second thread performs the sequential alignment on the lists of point variables $\vec{\mathcal{P}}_{j+1}, \ldots, \vec{\mathcal{P}}_n$, according to the two angles $\langle \phi, \psi \rangle$ randomly selected for the variables x_j, y_j. This subroutine runs in $\mathcal{O}(n)$ time using this organization in blocks and threads, instead of the $\mathcal{O}(k \cdot v \cdot m)$ required by a sequential implementation, where v is the number of variables assigned to the coordinator agent, and m is the number of random samplings.

As mentioned earlier, given a set of instantiated point variables, calculated by either the **choose_best_label** function or the **choose_best_label_random** function, the energy values of the corresponding structures are calculated in parallel on the GPU. Once a list of energy values (for each of the structures determined by the **choose_best_label/choose_best_label_random**) has been determined, we delegate the computation of the minimal energy to the host—using a simple linear scan of the list of energy values of the various structures. We do not use a logarithmic reduction to compute the minimal energy, since the cost of invoking a kernel CUDA plus the costs of the memory transactions between host and device would exceed the time required by the sequential scan.

As far as the Gibbs sampling strategy is concerned, the `for` of lines 24–33 of Alg. 17 is delegated to a CUDA kernel invoked with m blocks, where each block is assigned to a different starting point. Starting points are computed as described above by the **choose_best_label_random** function considering $k = 1$ and forcing the random assignment of the variables associated to the agent (i.e., it will be performed the same computation performed by one of the blocks $\geq (m - k)$ of the **choose_best_label_random** function). The for-loop that iterates on the number of samples (Alg. 17—line 26-32) and the for-loop that iterates on the number of worker agents (Alg. 20—lines 1-17) are kept as sequential loops due to the correlation of consecutive samples (i.e., consecutive samples define a Markov chain). Energy values are calculated in parallel on the GPU at each iteration of the two nested loops on all the set of (updated) starting points.

8.5.4 General details about CUDA

We present some details related to the implementation of the CUDA kernels described in the previous sections. In particular, due to the features of the architectural model of CUDA, we must consider three main aspects that can affect the performance of the parallel computations: **(1)** The maximum number of threads per block; **(2)** The maximum number of threads that can be physically executed in parallel on each processor of the GPU (also known as the *warp* size); **(3)** The information stored on the device memory and the copies of data to and from the host memory.

Let us consider a (typical) hardware where the maximum number of threads per block is limited to 1024, and the size of a *warp* is 32. The first restriction could potentially limit the maximum size of the target protein to 1024, when we use one thread per amino acid (e.g., to implement the `alldistant` constraint). To avoid such restriction we can split the computation in multiple executions of the same kernel. For each invocation we simply consider a different window of 1024 consecutive amino acids, until we cover the whole protein. However, protein typical size is less than 1024, so this further stage is normally not needed.

The second restriction is important when we allow threads of the same warp to diverge to different computational branches. Since kernel instructions are issued not to each thread, but to each warp, to solve the divergences the compiler of CUDA generates code that will run sequentially one branch after the other, causing a delay in the execution of the entire warp. We solved this problem by splitting the parallel computation of different parts of the kernel code among warps of 32 threads. For example, considering the energy function instead of $2n + 2$ threads per block, we invoke the kernel with $2m + 64$ threads per block, where $m = \lceil \frac{n}{32} \rceil$. Hence, $2m$ threads are used to compute the *contact* potential and the *hydrogen bond* potential, and 64 threads are used to compute the *correlation* and *torsional* potential.

The final aspect regards the optimization of the memory usage in order to achieve maximum memory throughput. CUDA has different types of memory spaces. Each thread block has access to a small amount of a fast *shared* memory within the scope of the block. In turn, all threads have access to the same *global* memory. Global memory is slower than shared memory but it can store more data. Since applications should strive to minimize data transfers between the host and the device (i.e., data transfers with low bandwidth), we reserve in the global memory an array of size equal to the maximum number of structures expected for the sampling of the coordinator agent multiplied the size of a structure. Moreover, we reserve an array of Boolean values for the admissible structures and an array of doubles for the energy values. Each kernel receives the number of structures to be considered and the pointers to the arrays in the global memory, in order to properly overwrite them considering only the memory area affected by the kernel function. The memory transfers to and from the CPU are made at each labeling step by copying the array of structures produced by the propagation of constraints and the array of the energy values. Again, only the elements affected by the computation of the kernel are transferred into the host memory.

To optimize the computation on the device, we store all the structures to rotate, the structures on which to perform the consistency checks for the `alldistant` constraint, and those on which to calculate the energy values in the shared memory of the GPU—paying particular care to not exceed the maximum size available for the device in use. The shared memory can be a limitation when we manage a large number of structures or very long proteins. Nevertheless, this is not a problem if we consider that the size of our domains is about 300, and the proteins are typically 200-300 amino acids long. These numbers are compatible with the characteristics of CUDA (e.g., maximum number of threads per block), which makes this architecture particularly efficient on our model.

8.6 Results

We report some experimental results obtained from the GPU implementation of the multi-agent system (briefly, GMAS). We run our experiments on a CPU AMD Opteron 2.3GHz, 132 GB memory, Linux 3.7.10-1.16-desktop x86 64, and GPU GeForce GTX TITAN, 14 SMs, 875MHz, 6 GB global memory, CUDA 5.0 with compute capability 3.5. To evaluate the speedup gained from exploiting parallelism on the GPU, we implement a sequential version of the multi-agent system (briefly, CMAS). Since coordinator agents use randomized search strategies, we report the results averaged over 20 runs per protein, as well as their standard deviation (sd). Computational times are reported in second. To assess the quality of our predictions we use the Root Mean Square Deviation (RMSD) as an indicator of how close is the predicted structure w.r.t. the correspondent (known) structure deposited in the protein data bank (the lower the better). Energy and RMSD values are averaged on the sets of results produced by both the CPU and the GPU implementations. “SUp” denotes Speedup in tables.

For following experiments we considered two different benchmark set of proteins selected from the Protein Structure Databank [15]. The first benchmark set, *BS1*, is composed by 27 proteins divided in 9 sets based on their length. In particular for $x = 1, \ldots, 9$ we we randomly selected an α protein, a β protein, and an $\alpha\beta$ protein of length $n \in \{10x + 1, \ldots, 10(x + 1)\}$.

The second benchmark set, *BS2*, is composed by 12 proteins of length that ranges from 125 to 200 residues, split into 4 subsets (where $n = 125, 150, 175, 200$). For each subset we randomly selected an α protein, a β protein, and an $\alpha\beta$ protein. The symbol *“*”* marks some proteins’ *ID*s that represents difficult targets due to their *supersecondary structure* conformation. A *supersecondary* structure is a compact three-dimensional protein structure of several adjacent elements of secondary structure that is smaller than a protein domain or a subunit. For both benchmark sets we will consider a structure agent per secondary structure element and one coordinator agent (default options).

In all CMAS/GMAS experiments we set a global timeout of 5000 seconds. This situation is reported as “5000 (-)” in the tables. Let us observe that in these cases, the best solution found in the time allowed is returned.

In Section 8.6.4 we analyze the system on a long protein as a case of study; for this test we used more coordinator agents. In Section 8.6.3 for the comparison with I-TASSER and FIASCO we considered the benchmark sets presented in the publications concerning those tools. Globally the GMAS tool has been tested on 65 proteins. The CPU and the GPU versions of the multi-agent system, the set of proteins, the input files, and the best results computed by the MAS tool can be found at `http://www.cs.nmsu.edu/fiasco/`.

8.6.1 GPU vs. CPU

Structure agents and secondary structure elements: In Table 8.1 and 8.2 we compare GMAS and CMAS w.r.t. the times required by Structure agents to fold α-helices and β-sheets for the benchmark sets BS1 and BS2, respectively. SS denotes the total length of the secondary structures in the protein of length n. Let us observe that Structure agents use the ICM algorithm to fold secondary structures.

Protein ID	Type	SS/n	CPU	GPU	SUp
2CZP	α	11/15	0.065 (0.001)	0.011 (0.0)	5.9
1LE0	β	6/12	0.006 (0.001)	0.010 (0.0)	0.6
2H2D	$\alpha\beta$	9/18	0.003 (0.0)	0.007 (0.0)	0.4
2L5R	α	20/23	0.367 (0.001)	0.060 (0.0)	6.1
1E0N	β	13/27	0.103 (0.001)	0.037 (0.0)	2.7
1YYP	$\alpha\beta$	12/31	0.046 (0.001)	0.019 (0.0)	2.4
1ZDD	α	25/35	0.675 (0.001)	0.062 (0.0)	10.8
1E0L	β	12/37	0.054 (0.001)	0.035 (0.0)	1.5
1PNH	$\alpha\beta$	19/31	0.181 (0.003)	0.061 (0.0)	2.9
2K9D	α	33/44	0.973 (0.001)	0.075 (0.0)	12.9
1YWI	β	16/41	0.125 (0.001)	0.050 (0.0)	2.5
1HYM	$\alpha\beta$	18/45	0.575 (0.001)	0.072 (0.0)	7.9
1YZM	α	41/51	2.729 (0.001)	0.149 (0.0)	18.3
2CRT	β	27/60	1.014 (0.001)	0.176 (0.0)	5.7
2HBB	$\alpha\beta$	28/51	1.101 (0.011)	0.069 (0.0)	15.9
1AIL	α	59/69	6.986 (0.005)	0.395 (0.005)	17.6
1PWT	β	31/61	2.117 (0.001)	0.141 (0.0)	15.0
2IGD	$\alpha\beta$	40/61	4.267 (0.021)	0.345 (0.002)	12.3
1OF9	α	53/77	4.816 (0.010)	0.242 (0.002)	19.9
1SPK	β	27/72	0.994 (0.001)	0.111 (0.0)	8.9
1VIG	$\alpha\beta$	42/71	3.022 (0.001)	0.213 (0.002)	14.1
1I11	α	44/81	2.707 (0.001)	0.202 (0.0)	13.4
1TEN	β	48/87	7.258 (0.036)	0.556 (0.004)	13.0
1DCJ	$\alpha\beta$	43/81	2.943 (0.002)	0.204 (0.0)	14.4
1JHG	α	82/100	16.12 (0.003)	0.557 (0.003)	28.9
1WHM	β	38/92	7.309 (0.002)	0.342 (0.006)	21.3
2CJO	$\alpha\beta$	41/97	7.955 (0.001)	0.296 (0.005)	26.8

Table 8.1: CPU vs GPU: Secondary Structure predictions for the set BS1.

Protein ID	Type	SS/n	CPU	GPU	SUp
1A0B	α	98/125	40.15 (0.103)	1.223 (0.007)	32.8
1H10	β	56/125	24.12 (0.083)	0.730 (0.003)	33.0
1F98	$\alpha\beta$	74/125	18.96 (0.010)	0.574 (0.002)	33.0
2CJ5	α	116/150	71.78 (0.021)	1.155 (0.012)	62.1
1STB	β	72/150	36.46 (0.040)	0.748 (0.005)	48.7
1LEO	$\alpha\beta$	89/150	39.15 (0.157)	0.763 (0.001)	51.3
1BGD	α	113/175	80.73 (0.283)	1.311 (0.005)	61.5
1FNL	β	92/175	48.24 (0.024)	1.073 (0.008)	44.9
1T8A	$\alpha\beta$	130/175	95.44 (0.017)	1.592 (0.006)	59.9
1IB1	α	165/200	113.4 (0.108)	2.851 (0.002)	39.7
2GH2	β	100/200	63.60 (0.115)	1.264 (0.008)	50.3
1RR9	$\alpha\beta$	114/200	86.50 (0.023)	1.147 (0.005)	75.4

Table 8.2: CPU vs GPU: Secondary Structure predictions for the set BS2.

The speedups increase as SS increases. In particular, the higher speedups in most sets are obtained on Structure agents associated to proteins of type α. This is due to the highly constrained structure of the helices where the (rather greedy) search strategy reaches a local minima without iterating many times on the set of variables associated to the agent. β-sheets are less constrained structures and they require more iterations to converge to the local minima. For the GMAS implementation point of view, more iterations leads to more information exchanged between host and device (e.g., memory copies) resulting in a decrease in performance.

AB-Initio Prediction using Monte Carlo: Ab-Initio prediction is performed by the Supervisor agent by managing Structure agents and Coordinator agents. In Table 8.3 we report the comparison between CMAS and GMAS considering one Coordinator agent using the Monte Carlo search strategy on the benchmark set BS1. Running time are inclusive of secondary structure prediction (by structure agents), while the external computation using JNet of the possible segments where looking for secondary structures is not considered. For each of the following experiments regarding Monte Carlo we considered a sample set of 1000 structures, a number of improving steps $k = 4$ (parameters found experimentally considering a trade-off between time and quality of predictions) and a time-out limit of 10800 seconds (3 hours). Speedups range from 4.5 to 17.3. For each protein

ID	n	CPU (sd)	GPU (sd)	SUp	RMSD (sd)	Energy (sd)	Best
2CZP	15	0.559 (0.128)	0.046 (0.010)	12.1	0.8 (0.0)	-490.5 (2.55)	0.8
1LE0	12	1.923 (0.176)	0.337 (0.020)	5.7	1.2 (0.2)	-344.2 (23.50)	0.5
2H2D	18	1.815 (0.118)	0.275 (0.073)	6.6	1.2 (0.0)	-263.3 (6.76)	1.2
2L5R	23	1.603 (0.222)	0.188 (0.028)	8.5	1.4 (0.0)	-1482.54 (0.00)	1.4
1E0N	27	63.80 (14.180)	10.73 (1.463)	5.9	2.4 (0.6)	-1945.00 (41.17)	1.5
1YYP	22	8.479 (2.262)	1.473 (0.482)	5.7	1.7 (0.3)	-1205.24 (31.23)	1.2
1ZDD	35	39.40 (7.801)	4.343 (1.263)	9.0	1.1 (0.2)	-2472.0 (27.79)	0.9
1E0L	37	20.76 (3.230)	1.816 (0.257)	11.4	2.7 (0.6)	-959.366 (70.40)	1.9
1PNH	31	24.15 (3.037)	2.865 (0.301)	8.4	2.7 (0.3)	-2611.61(40.20)	2.3
2K9D	44	87.46 (10.81)	13.67 (5.427)	6.3	1.5 (0.4)	-6162.405 (150.29)	0.9
1YWI	41	32.23 (4.690)	3.523 (0.959)	9.1	3.4 (0.3)	-1295.636 (14.75)	2.7
1HYM	45	81.10 (5.193)	9.309 (1.033)	8.7	3.5 (0.2)	-2258.224 (71.90)	3.0
1YZM	51	32.45 (4.361)	1.871 (0.279)	17.3	2.6 (0.4)	-4963.852 (63.70)	1.6
2CRT	60	1213 (166.45)	163.2 (20.70)	7.4	3.7 (0.6)	-9514.056 (271.25)	2.6
2HBB*	51	476.3 (37.86)	64.42 (6.383)	7.3	3.6 (0.5)	-6175.155 (217.69)	3.0
1AIL	69	88.10 (15.37)	6.546 (0.593)	13.4	3.2 (0.6)	-10443.04 (251.59)	2.3
1PWT	61	512.8 (45.63)	66.61 (12.23)	7.6	4.1 (0.6)	-7983.511 (238.57)	3.2
2IGD*	61	681.4 (50.13)	92.79 (12.55)	7.3	6.2 (0.8)	-7928.462 (218.74)	5.0
1OF9*	77	750.6 (82.15)	133.9 (32.54)	5.6	4.3 (0.7)	-17063.19 (480.68)	3.3
1SPK	72	1343 (149.27)	174.4 (28.00)	7.7	4.1 (0.4)	-8457.154 (334.07)	3.6
1VIG	71	977.1 (107.20)	164.8 (34.63)	5.9	4.9 (0.7)	-9295.76 (296.07)	3.7
1I11	81	296.7 (23.22)	38.70 (10.75)	5.6	4.4 (0.4)	-7852.049 (251.77)	3.8
1TEN*	87	2995 (466.99)	1014 (86.20)	2.9	5.2 (0.8)	-13955.71 (686.41)	4.1
1DCJ*	81	1590 (139.66)	310.7 (43.92)	5.1	4.5 (0.4)	-15398.04 (606.98)	3.7
1JHG	100	1168 (123.86)	187.0 (33.59)	6.2	5.7 (1.0)	-23687.56 (360.45)	4.0
1WHM	92	2796 (414.802)	611.5 (63.32)	4.5	4.5 (0.7)	-17220.98 (525.289)	3.3
2CJO*	97	9195 (547.22)	1988 (162.6)	4.6	4.6 (0.2)	-15785.77 (318.45)	4.2

Table 8.3: Time (sec.), quality, and energy values averaged on 20 runs for the set of proteins BS1. Coordinator agent uses *Monte Carlo* algorithm.

we also report the best result found in 20 runs in terms of RMSD (column "Best"). Let us observe that RMSD values are rather low: for proteins of length 100 we are in an average of 5Å.

AB-Initio Prediction using Gibbs: Coordinator agents can use the Gibbs sampling algorithm to model loops and turns and hence to fold the entire protein. In terms of time, the Gibbs sampling algorithm has a more stable behavior than Monte Carlo since the former runs for a fixed number of iterations, while the latter runs until a local minimum (or a given time-out) is reached. Computation of secondary structures are performed by structure agents as described in the previous experiment. In Table 8.4 we report the comparisons between CMAS and GMAS using the Gibbs sampling strategy for the Coordinator agent, considering a number of samples ***n_samples* = 50**, and $m = 1000$ starting points. The number of sampling steps has been found experimentally and it has been chosen as a good compromise between computational time and quality of predictions, as discussed in what follows.

Protein ID	*n.*	CPU (sd)	GPU (sd)	SUp	RMSD (sd)	Energy (sd)	Best
2CZP	15	1.243 (0.001)	0.219 (0.002)	5.6	0.8 (0.0)	-490.461 (1.965)	0.8
1LE0	12	2.674 (0.001)	0.557 (0.009)	4.8	1.4 (0.1)	-308.881 (4.903)	1.3
2H2D	18	2.210 (0.018)	0.513 (0.009)	4.3	1.5 (0.3)	235.4091 (7.062)	1.1
2L5R	23	5.009 (0.154)	0.356 (0.006)	14.0	1.4 (0.0)	-1482.54 (0.0)	1.4
1E0N	27	69.69 (0.167)	3.379 (0.025)	20.6	2.7 (0.6)	-1724.791 (41.558)	1.7
1YYP	22	19.12 (0.14)	1.450 (0.009)	13.1	1.8 (0.4)	-1203.736 (11.868)	1.2
1ZDD	35	57.64 (0.09)	2.639 (0.013)	21.8	1.1 (0.1)	-2470.327 (26.436)	0.9
1E0L	37	27.06 (0.459)	1.895 (0.014)	14.2	2.5 (0.4)	-894.690 (55.465)	2.0
1PNH	31	33.99 (0.104)	1.793 (0.020)	18.9	2.3 (0.1)	-2538.936 (25.54)	2.1
2K9D	44	144.0 (1.282)	4.687 (0.020)	30.7	1.7 (0.7)	-5883.502 (102.288)	0.9
1YWI	41	41.48 (0.221)	2.251 (0.011)	18.4	3.4 (0.3)	-1168.549 (13.060)	2.9
1HYM	45	74.58 (1.250)	3.478 (0.049)	21.4	3.5 (0.5)	-2037.445 (39.651)	2.8
1YZM	51	56.04 (0.719)	1.800 (0.015)	31.3	2.8 (0.2)	-4993.436 (27.578)	2.4
2CRT	60	665.7 (10.23)	17.93 (0.320)	37.1	4.0 (0.7)	-8289.048 (166.152)	2.7
2HBB*	51	372.6 (1.628)	11.68 (0.116)	31.9	4.0 (0.2)	-5309.193 (101.03)	3.7
1AIL	69	156.0 (3.019)	3.889 (0.017)	40.1	3.5 (0.3)	-11275.98 (190.78)	3.2
1PWT	61	474.4 (0.345)	12.70 (0.611)	37.3	4.7 (0.6)	-6942.941 (146.546)	3.6
2IGD*	61	550.4 (2.870)	14.15 (0.231)	38.8	5.1 (0.9)	-7029.251 (86.943)	4.4
1OF9*	77	780.5 (10.87)	18.50 (0.429)	42.1	4.0 (0.8)	-14739.77 (335.616)	2.5
1SPK	72	700.3 (1.521)	18.42 (0.353)	38.0	4.6 (0.5)	7071.708 (80.60)	4.0
1VIG	71	750.0 (2.161)	23.00 (0.288)	32.6	5.1 (1.0)	-8015.638 (130.803)	3.4
1I11	81	433.1 (1.244)	10.11 (0.122)	42.8	3.9 (0.6)	-7544.959 (114.85)	2.8
1TEN*	87	1841 (2.742)	50.93 (1.611)	36.1	5.5 (0.5)	-12576.83 (413.913)	4.6
1DCJ*	81	1021 (1.246)	26.18 (1.077)	38.9	5.2 (0.6)	-13061.12 (174.81)	3.9
1JHG	100	1462 (1.344)	32.63 (1.255)	44.8	4.9 (1.3)	-21332.2 (258.721)	3.3
1WHM	92	1035 (1.281)	37.19 (0.262)	27.8	4.9 (0.6)	-13953 (237.5671)	4.3
2CJO*	97	2024 (2.149)	56.41 (1.274)	35.8	5.2 (0.6)	-12964.17 (170.607)	4.3

Table 8.4: Time (sec.), quality, and energy values averaged on 20 runs for the set of proteins BS1. Coordinator agent uses *Gibbs* sampling algorithm.

The standard deviations of running times are smaller than those obtained using Monte Carlo. Speedups are higher than in the previous experiments; this is due to the fact that Gibbs algorithm updates the initial set of starting points in parallel considering one structure at a time for each sampling step, while Monte Carlo produces a new sampling set for each variable associated to the agent at each iteration step.

Time vs. Number of Samplings: The time needed to fold the structures and their quality are two factors that are strongly correlated to the number of samples when using the Gibbs sampling search strategy. Although it is quite difficult to relate the upper bound on the number of samples with an upper bound on the quality of the results (e.g., see [122]), it is easy to study how the computational time varies w.r.t. the number of samples and the length of the proteins. In particular, since the coordinator agent invokes the `gibbs` function for a fixed number of times (see Alg. 17, lines 26–35), the computational time varies linearly w.r.t. the number of samples. We report this analysis in Figure 8.3 for the computationally most demanding protein of each subset, varying the number of samples from 1 to 50, and comparing GMAS with CMAS.

In order to study how the quality of the solution changes w.r.t. the number of samples we selected the longer protein of the benchmark set (i.e., 2CJO, with $n = 97$ but with 48 amino acids delegated to the Coordinator agent) and we varied the number of samples from 1 to 50. Table 8.5 reports the results in terms of RMSD and standard deviation w.r.t. the number of samples.

Our choice of setting 50 as upper bound on the number of sampling for the benchmarks set BS1 is motivated by the two following observations: (1) for $n \geq 30$ the improvements on the quality of

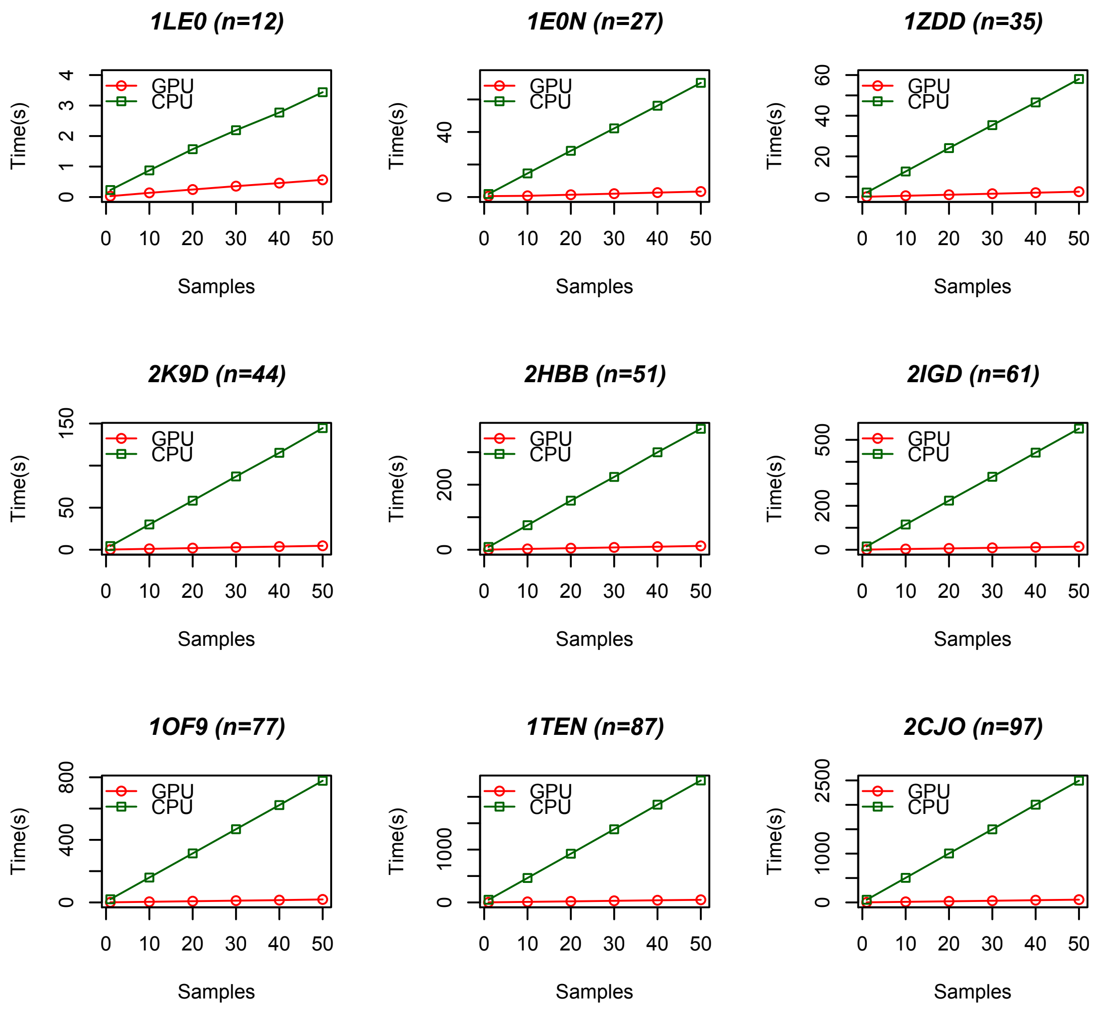


Figure 8.3: Time vs number of samples for the Gibbs sampling algorithm

Protein ID	Number of Samples					
	1	10	20	30	40	50
2CJO	6.3 (0.9)	5.8 (0.2)	5.5 (0.4)	5.4 (0.6)	5.4 (0.5)	5.2 (0.6)

Table 8.5: Quality w.r.t. number of samples for the Gibbs sampling strategy.

the structures are relatively small, and (2) 2CJO can be implicitly considered as a representative protein for all the other targets in BS1, since it has the larger set of amino acids assigned to a coordinator agent (51 residues). Therefore, we have conjectured that this value is suitable for the whole set. Other, not reported, experiments have confirmed this hypothesis. Let us conclude this analysis by observing that the observed linearity guarantees that using less than 50 sampling steps would preserve the speedups.

Quality evaluation: using RMSD as Objective Function: The quality of the predicted structure strongly depends on the energy function adopted in the model. The energy function can be changed as a black box and without affecting the overall structure of the system. In this test, we evaluated the differences in terms of RMSD between the Gibbs sampling and the Monte Carlo algorithm using the RMSD w.r.t. the native known structure as objective function for both the structure and the coordinator agents. Of course this function cannot be used for still unknown proteins; however we experiment it in order to see if our tool is, in principle, able to compute the native structure if the "real" protein energy function would be know. Table 8.6 shows the results in terms of average and best RMSD computed for the benchmark set is BS1. Best results are reported in **boldfont**.

Protein ID	n	Gibbs (sd)	Best	MC (sd)	Best
2CZP	15	1.0 (0.0)	**1.0**	1.0 (0.0)	**1.0**
1LE0	12	1.0 (0.0)	1.0	0.9 (0.9)	**0.9**
2H2D	18	0.5 (0.0)	**0.5**	1.2 (0.0)	1.2
2L5R	23	1.4 (0.0)	1.4	1.1 (0.1)	**1.0**
1E0N	27	1.3 (0.2)	1.0	0.9 (0.0)	**0.8**
1YYP	22	0.9 (0.0)	0.9	1.5 (0.1)	**0.8**
1ZDD	35	1.6 (0.1)	1.5	1.5 (0.1)	**1.4**
1E0L	37	3.4 (0.1)	3.1	3.5 (0.3)	**2.8**
1PNH	31	2.8 (0.4)	**2.1**	2.9 (0.2)	2.7
2K9D	44	2.0 (0.3)	**1.5**	1.9 (0.1)	1.7
1YWI	41	2.5 (0.4)	**1.9**	2.5 (0.2)	2.2
1HYM	45	2.4 (0.3)	**1.6**	2.4 (0.2)	2.1
1YZM	51	1.5 (0.0)	1.5	1.5 (0.1)	**1.4**
2CRT	60	4.3 (0.3)	**3.9**	4.5 (0.2)	4.1
2HBB*	51	2.6 (0.3)	2.1	1.8 (0.3)	**1.4**
1AIL	69	2.0 (0.3)	**1.5**	2.8 (0.7)	1.7
1PWT	61	3.5 (0.3)	2.9	2.7 (0.4)	**2.1**
2IGD*	61	3.0 (0.2)	**2.6**	2.9 (0.3)	**2.6**
1OF9*	77	2.2 (0.3)	2.1	1.7 (0.1)	**1.5**
1SPK	72	4.3 (0.5)	3.8	4.2 (0.3)	**3.7**
1VIG	71	5.0 (0.5)	**4.2**	5.8 (0.4)	4.9
1I11	81	2.9 (0.4)	2.3	2.4 (0.1)	**2.2**
1TEN*	87	6.3 (0.4)	**5.8**	6.5 (0.4)	**5.8**
1DCJ*	81	4.6 (0.7)	**3.8**	4.3 (0.3)	**3.8**
1JHG	100	5.7 (0.5)	5.1	5.7 (0.6)	**4.3**
1WHM	92	5.2 (0.9)	**4.1**	4.8 (0.2)	4.5
2CJO*	97	4.8 (0.5)	4.1	4.2 (0.4)	**3.5**

Table 8.6: Quality evaluation: RMSD as objective function.

As expected, the average quality of the results is better than using the original energy function, since both search strategies try to minimize the spatial difference between the native structure and the target. Nevertheless, RMSD values are not close to zero since the variables' domains are created to be used for predicting unknown structures. Therefore, they do not necessary contain the true pair of angles of the amino acids of each target.

Let us observe that an objective function based on the RMSD value can be adopted for new search strategies. For example, minimization of the RMSD distance between (parts of) the target and a corresponding set of templates can be useful in template-based modeling techniques [52].

Adding constraints while preserving speedup: In this section we show that the system is highly modular and that we can introduce additional geometric constraints that allows to reduce the search space while preserving the speedup.

We consider the case of modeling the side-chain of each amino acid. This can be done by defining a new constraint that relates the position of each C_α atom with the group R defined on it. The `centroid (CG)` constraint enforces a relation among four triples of real variables $\vec{p}_1, \vec{p}_2, \vec{p}_3$, and $\vec{p}_4$. This relation establishes the value to assign to the variables $\vec{p}_4$ representing the coordinates of the side chain defined on the carbon atom C_{α_i}, given the bend angle formed by the carbon atoms $\vec{p}_1 \mapsto C_{\alpha_{i-1}}, \vec{p}_2 \mapsto C_{\alpha_i}$, and $\vec{p}_3 \mapsto C_{\alpha_{i+1}}$, and the average C_{α_i}–side chain distance [55]. The coordinates of $\vec{p}_1, \vec{p}_2$, and $\vec{p}_3$ are already present in our modeling, while the coordinates of $\vec{p}_4$ are considered only here. Moreover, this constraint checks the minimum distance between side chains and all the other atoms of the structure in order to avoid steric clashes, as in the case of the `alldistant` constraint. Note that, using a sequential algorithm, it is possible to check the consistency of this constraint for a given assignment of values to the variables in P in time $\mathcal{O}(n^2)$. We adopt the same strategy used for the `alldistant` constraint to obtain $\mathcal{O}(n)$ time in the parallel implementation.

In Table 8.7 (resp., 8.8) we report the times for CMAS and GMAS on the protein set BS1 for the Monte Carlo (reps., Gibbs sampling strategies), using the `alldistant` constraint and the `CG` constraint.

Protein ID	n	CPU (sd)	GPU (sd)	SUp	RMSD (sd)	Energy (sd)	Best
2CZP	15	0.760 (0.158)	0.075 (0.017)	10.1	0.8 (0.0)	-487.009 (0.0)	0.8
1LE0	12	2.225 (0.247	0.489 (0.047)	4.5	1.4 (0.1)	-310.5714 (15.03)	1.3
2H2D	18	2.112 (0.340)	0.425 (0.042)	4.9	1.3 (0.2)	-258.0266 (4.655)	1.1
2L5R	23	2.072 (0.525)	0.281 (0.109)	7.3	1.4 (0.0)	-1482.54 (0.0)	1.4
1E0N	27	72.46 (9.620)	14.18 (2.458)	5.1	2.3 (0.4)	-1723.076 (49.3655)	1.7
1YYP	22	10.94 (2.793)	1.695 (0.114)	6.4	1.4 (0.2)	-1190.033 (36.27)	1.2
1ZDD	35	45.799 (7.169)	5.284 (1.378)	8.6	1.7 (0.2)	-2272.059 (55.53058)	1.5
1E0L	37	24.85 (3.285)	4.493 (0.271)	5.5	2.6 (0.6)	-882.369 (37.0802)	1.8
1PNH	31	27.25 (4.602)	3.370 (0.387)	8.0	3.4 (0.6)	-2518.261 (52.135)	2.1
2K9D	44	95.42 (9.547)	12.68 (1.798)	7.5	3.0 (1.3)	-5258.283 (114.113)	1.4
1YWI	41	29.07 (5.566)	3.940 (1.688)	7.3	3.1 (0.5)	-1152.215 (40.8021)	2.2
1HYM	45	97.96 (18.16)	10.29 (0.964)	9.5	3.5 (0.3)	-2113.299 (85.986)	2.9
1YZM	51	36.69 (10.11)	2.406 (0.330)	15.2	2.7 (0.3)	-4393.806 (39.639)	2.4
2CRT	60	1280 (110.4)	187.5 (22.14)	6.8	4.0 (0.3)	-8346.851 (264.8791)	3.5
2HBB*	51	562.2 (91.53)	67.66 (12.38)	8.3	3.9 (0.8)	-5471.265 (67.6792)	2.8
1AIL	69	132.5 (9.236)	7.575 (0.653)	17.4	3.8 (0.8)	-8880.202 (227.2679)	2.7
1PWT	61	554.5 (40.66)	67.75 (4.649)	8.1	4.1 (0.6)	-7034.326 (178.9961)	3.0
2IGD*	61	717.7 (135.3)	82.09 (12.19)	8.7	5.3 (1.0)	-7126.439 (224.9547)	4.1
1OF9*	77	768.2 (100.4)	140.0 (33.79)	5.4	3.5 (0.7)	-14429.41 (342.4358)	2.6
1SPK	72	1317 (103.3)	175.7 (9.452)	7.4	4.2 (0.5)	-7189.066 (264.136)	3.4
1VIG	71	1176 (184.4)	160.3 (34.90)	7.3	4.7 (0.9)	-8118.327 (181.7243)	3.3
1I11	81	322.7 (18.45)	49.91 (5.924)	6.4	4.0 (0.7)	-6870.417 (151.4574)	3.1
1TEN*	87	5330 (564.8)	1128 (148.1)	4.7	6.0 (0.8)	-13505.96 (204.369)	5.1
1DCJ*	81	1623 (209.8)	292.0 (52.30)	5.5	5.2 (0.7)	-12866.59 (445.7158)	4.4
1JHG	100	1417 (101.7)	179.8 (28.56)	7.8	6.1 (0.9)	-20293.91 (404.9333)	4.8
1WHM	92	2829 (250.4)	968.9 (189.5)	2.9	5.1 (0.8)	-14676.57 (484.0556)	3.8
2CJO*	97	8248 (748.3)	1967 (262.8)	4.1	5.4 (0.8)	-13916.12 (395.8187)	4.4

Table 8.7: Time (sec.), quality, and energy values averaged on 20 runs for some proteins of different length and type. Coordinator agents use *Monte Carlo* algorithm to explore conformations. CG constraint has been enabled.

8.6.2 Longer Proteins

After experimented the speedup of GMAS w.r.t. CMAS, in this section we show the results of GMAS on the set of larger proteins BS2. Table 8.9 shows the results in terms of time, RMSD, energy, and best RMSD when using the Gibbs algorithm for the Coordinator agent (1 Coordinator agent, 50 sampling steps). Times vary from 40.4 to 376.6 seconds, while the best RMSD values are always under 8.0Å. In Table 8.10 the CG constraint is added to the encoding. Quality of results

Protein ID	n	CPU (sd)	GPU (sd)	SUp	RMSD (sd)	Energy (sd)	Best
2CZP	15	2.006 (0.001)	0.314 (0.029)	6.3	0.8 (0.0)	-491.458 (1.022)	0.8
1LE0	12	4.114 (0.007)	0.806 (0.077)	5.1	1.4 (0.1)	-300.8306 (6.789)	1.3
2H2D	18	3.300 (0.039)	0.758 (0.060)	4.3	1.7 (0.7)	-244.392 (7.862)	0.9
2L5R	23	6.787 (0.070)	0.521 (0.047)	13.0	1.4 (0.0)	-1482.54 (0.000)	1.4
1E0N	27	93.61 (10.66)	4.964 (0.118)	18.8	3.0 (0.6)	-1517.871 (28.7422)	2.3
1YYP	22	25.71 (0.062)	2.100 (0.099)	12.2	1.7 (0.5)	-1183.739 (9.5)	1.1
1ZDD	35	81.41 (0.500)	3.384 (0.065)	24.0	1.6 (0.1)	-2378.678 (61.27)	1.5
1E0L	37	39.52 (0.782)	2.692 (0.017)	14.6	2.8 (0.6)	-849.1835 (35.66)	2.0
1PNH	31	47.02 (0.205)	2.451 (0.034)	19.1	2.8 (0.4)	-2440.618 (55.42)	2.1
2K9D	44	186.9 (15.24)	5.311 (0.069)	35.1	2.0 (0.4)	-5191.226 (247.499)	1.6
1YWI	41	52.96 (0.211)	3.203 (0.090)	16.5	3.0 (0.5)	-1072.859 (16.79)	2.3
1HYM	45	110.7 (0.198)	4.646 (0.093)	23.8	3.9 (0.7)	-1905.429 (62.30)	2.9
1YZM	51	70.85 (0.770)	2.575 (0.007)	27.5	2.6 (0.1)	-4533.8 (46.809)	2.5
2CRT	60	636.3 (4.732)	18.29 (0.331)	34.7	3.8 (0.6)	-7252.04 (247.893)	3.1
2HBB*	51	578.2 (4.624)	13.77 (0.208)	41.9	4.3 (0.4)	-4818.446 (54.610)	3.6
1AIL	69	226.8 (0.356)	6.403 (0.651)	35.4	3.7 (0.7)	-9842.159 (227.2839)	2.4
1PWT	61	524.7 (173.7)	11.60 (1.573)	45.2	4.2 (0.7)	-6173.182 (109.042)	3.3
2IGD*	61	361.7 (29.61)	16.94 (0.347)	21.3	5.4 (0.9)	-6185.173 (151.216)	4.0
1OF9*	77	1077 (23.35)	18.18 (0.615)	59.2	4.8 (0.9)	-12970 (212.057)	3.0
1SPK	72	554.2 (28.9)	15.08 (0.597)	36.7	4.9 (0.5)	-6151.49 (128.4445)	4.1
1VIG	71	644.6 (19.86)	22.70 (0.765)	28.3	5.0 (0.6)	-6995.402 (223.483)	3.9
1I11	81	461.9 (106.3)	13.56 (0.177)	34.0	4.0 (0.7)	-6747.406 (94.80)	2.8
1TEN*	87	3348 (15.58)	62.29 (0.354)	53.7	6.3 (0.9)	-11030.24 (311.69)	4.8
1DCJ*	81	1065 (392.9)	19.51 (1.196)	54.5	5.9 (0.9)	-11330.29 (271.17)	3.5
1JHG	100	1556 (63.96)	15.80 (0.347)	98.4	5.2 (0.8)	-19074.76 (579.9539)	4.0
1WHM	92	1879 (0.431)	30.18 (1.693)	62.2	5.2 (1.0)	-12364.63 (199.9923)	3.7
2CJO*	97	1795 (148.7)	45.30 (0.742)	39.6	5.8 (0.9)	-11197.55 (188.3018)	4.7

Table 8.8: Time (sec.), quality, and energy values averaged on 20 runs for some proteins of different length and type. Coordinator agents use *Gibbs* sampling algorithm to explore conformations. CG constraint has been enabled.

Protein ID	Time (sd)	RMSD (sd)	Energy (sd)	Best RMSD
1A0B	40.44 (0.99)	5.2 (1.1)	-23592.9 (1007.557)	3.5
1H10*	69.18 (3.278)	7.6 (1.0)	-14041.22 (1010.967)	6.9
1F98*	84.37 (3.410)	6.9 (0.9)	-21683.44 (760.408)	6.0
2CJ5	100.6 (4.546)	6.4 (0.8)	-35302.44 (1062.648)	5.2
1STB*	118.0 (5.452)	7.5 (1.1)	-19563.05 (910.6381)	5.6
1LEO	101.6 (8.208)	6.7 (0.9)	-32627.28 (909.583)	5.4
1BGD*	135.5 (7.360)	8.1 (1.5)	-47544.61 (1441.902)	5.2
1FNL*	376.6 (26.34)	8.4 (0.7)	-40839.9 (1424.246)	7.2
1T8A	214.8 (12.56)	7.9 (1.0)	-44482.65 (1107.889)	6.3
1IB1	208.7 (10.80)	8.4 (1.0)	-50007.91 (1047.775)	6.4
2GH2*	84.75 (0.851)	9.8 (1.5)	-38321.71 (1283.365)	7.5
1RR9*	298.8 (17.93)	8.4 (1.1)	-45074.02 (1959.957)	7.0

Table 8.9: Longer proteins $(125, 150, 175, 200)$: time and quality evaluation using Gibbs sampling.

are slightly decreased due to the poor approximation of the side chain with a single atom, while computational times are reduced since more structures have been pruned.

Protein ID	Time (sd)	RMSD (sd)	Energy (sd)	Best RMSD
1A0B	32.33 (1.424)	7.8 (1.3)	-21292.37 (708.0099)	5.6
1H10*	49.76 (2.648)	7.3 (1.1)	-13354.67 (184.2466)	5.0
1F98*	51.56 (2.795)	7.1 (1.0)	-19554.71 (751.1157)	5.9
2CJ5	75.33 (7.619)	6.5 (1.2)	-30636.26 (218.7721)	5.6
1STB*	58.00 (3.399)	7.2 (1.5)	-17567.47 (862.4277)	4.5
1LEO	60.32 (5.187)	7.3 (0.8)	-28359.66 (658.0844)	6.1
1BGD*	83.57 (9.395)	9.0 (1.2)	-42451.74 (942.5947)	7.8
1FNL*	101.9 (15.99)	10.5 (1.7)	-34714.2 (2204.668)	7.9
1T8A	113.6 (4.960)	8.8 (1.7)	-39357.61 (862.8464)	5.5
1IB1	112.6 (10.07)	9.1 (1.2)	-42539.18 (1545.187)	7.3
2GH2*	118.5 (20.11)	12.4 (1.2)	-33937.24 (2317.069)	11.1
1RR9*	144.3 (18.84)	8.8 (1.2)	-38644.9 (1474.511)	6.9

Table 8.10: Longer proteins $(125, 150, 175, 200)$: time and quality evaluation using Gibbs sampling with the CG constraint enabled.

Protein ID	Time (sd)	RMSD (sd)	Energy (sd)	Best RMSD
1A0B	285.9 (61.19)	4.1 (0.4)	-25588.58 (717.2202)	2.7
1H10*	1693 (300.3)	5.8 (0.6)	-18914.5 (459.4581)	5.1
1F98*	2987 (598.4)	5.8 (0.9)	-26607.36 (721.812)	4.7
2CJ5	1583 (230.9)	7.0 (1.4)	-41436.0 (1558.731)	5.5
1STB*	5392 (625.9)	6.5 (0.2)	-25676.32 (1207.134)	6.2
1LEO	5145 (609.2)	5.8 (0.3)	-40749.62 (1038.819)	5.4
1BGD*	5000 (-)	6.2 (1.3)	-38185.23 (1199.362)	5.1
1FNL*	5000 (-)	6.0 (0.4)	-45450.2 (1014.597)	5.7
1T8A	5000 (-)	4.7 (0.7)	-39630.5 (922.132)	4.2
1IB1	4436 (543.1)	3.4 (0.2)	-43938.1 (907.0643)	3.1
2GH2*	5000 (-)	7.2 (0.7)	-49686.63 (503.952)	6.7
1RR9*	5000 (-)	5.8 (1.2)	-45133.57 (640.239)	4.6

Table 8.11: Longer proteins $(125, 150, 175, 200)$: time and quality evaluation using the Monte Carlo search strategy.

Protein ID	Time (sd)	RMSD (sd)	Energy (sd)	Best RMSD
1A0B	240.6 (44.56)	7.1 (1.8)	-22661.39 (242.603)	5.1
1H10*	1659 (247.8)	6.4 (0.5)	-13641.52 (5259.107)	5.9
1F98*	2287 (240.3)	6.3 (0.8)	-22783.95 (530.0175)	5.2
2CJ5	1547 (149.6)	13.5 (2.9)	-35356.96 (660.8399)	8.6
1STB*	4891 (372.8)	7.0 (0.6)	-21351.42 (440.864)	6.1
1LEO	4125 (402.6)	6.1 (0.9)	-33733.4 (164.6203)	5.2
1BGD*		6.6 (1.1)	-40615.83 (870.601)	5.4
1FNL*	5000 (-)	6.3 (1.0)	-44714.47 (382.754)	5.1
1T8A		4.6 (0.5)	-38881.8 (997.4146)	4.1
1IB1	3625 (538.4)	4.8 (0.3)	-43085.2 (934.407)	4.5
2GH2*	5000 (-)	9.2 (0.4)	-47081.13 (1051.74)	9.5
1RR9*	5000 (-)	6.3 (1.0)	-40543.97 (1206.53)	5.6

Table 8.12: Longer proteins $(125, 150, 175, 200)$: time and quality evaluation using the Monte Carlo search strategy and CG constraint.

8.6.3 Comparison with other Systems

Comparison with Rosetta: We compared the quality of the proposed tool in terms of RMSD against the state-of-the-art *Rosetta* tool, initially presented in [153] and continuously evolved since then. For each protein of benchmark set SB1 and SB2 we built the dictionary for 3 and 9 amino acid long peptides previous sequence alignment using the PSIPRED online server (`http://bioinf.cs.ucl.ac.uk/psipred`). We followed the examples in the Rosetta distribution. Let us observe that our tool uses e a generic database of angles whereas Rosetta uses a database of fragments created based on the target sequence. To be as fair as possible in comparing the tools we run them considering their default settings. Moreover, we run the Rosetta tool on a host machine equipped with 8 processors Intel(R) Core(TM) i7-2600 CPU, 3.40GHz.

Table 8.13 show the results in terms of RMSD and time as given by Rosetta. For each protein we report also the RMSD corresponding to the best structure found by Rosetta and the RMSD corresponding to best structure found by GMAS among all the previous experiments using Monte Carlo (MC) and Gibbs sampling as well as the corresponding times. Best results are reported in **boldfont**.

		Rosetta			**GMAS**			
Protein ID	*n*	RMSD (sd)	Time (sd)	Best	MC	Time (sd)	Gibbs	Time (sd)
2CZP	15	0.4 (0.1)	2.700 (0.483)	**0.2**	0.8	0.046 (0.010)	0.8	0.219 (0.002)
1LE0	12	0.7 (0.2)	2.600 (0.516)	**0.3**	0.5	0.337 (0.020)	1.3	0.557 (0.009)
2H2D	18	1.3 (0.3)	3.428 (0.534)	**0.9**	1.1	0.425 (0.042)	**0.9**	0.758 (0.060)
2L5R	23	1.1 (0.0)	3.500 (0.547)	**1.1**	1.4	0.188 (0.028)	1.4	0.356 (0.006)
1E0N	27	2.7 (0.4)	7.100 (1.523)	2.0	**1.5**	10.73 (1.463)	1.7	3.379 (0.025)
1YYP	22	2.5 (0.4)	7.857 (0.690)	2.1	1.2	1.473 (0.482)	**1.1**	2.100 (0.099)
1ZDD	35	1.3 (0.4)	11.00 (0.942)	**0.8**	0.9	4.343 (1.263)	0.9	2.639 (0.013)
1E0L	37	3.5 (0.6)	11.70 (2.540)	2.3	**1.8**	4.493 (0.271)	2.0	1.895 (0.014)
1PNH	31	3.2 (0.8)	11.20 (1.549)	**1.8**	2.1	3.370 (0.387)	2.1	1.793 (0.020)
2K9D	44	2.8 (1.3)	12.30 (2.057)	1.6	**0.9**	13.67 (5.427)	**0.9**	4.687 (0.020)
1YWI	41	2.5 (0.4)	9.222 (1.715)	2.8	**2.2**	3.940 (1.688)	2.3	3.203 (0.090)
1HYM	45	4.2 (0.9)	15.10 (0.567)	3.1	2.9	10.29 (0.964)	**2.8**	3.478 (0.049)
1YZM	51	0.9 (0.4)	10.75 (1.035)	**0.5**	1.6	1.871 (0.279)	2.4	1.800 (0.015)
2CRT	60	4.5 (0.8)	20.75 (1.035)	3.5	**2.6**	163.2 (20.70)	2.7	17.93 (0.320)
2HBB	51	3.4 (0.7)	17.11 (1.452)	**2.2**	2.8	67.66 (12.38)	3.6	13.77 (0.208)
1AIL	69	4.1 (1.4)	26.33 (0.707)	**1.7**	2.3	6.546 (0.593)	2.4	6.403 (0.651)
1PWT	61	3.9 (0.7)	23.12 (0.640)	**2.8**	3.0	67.75 (4.649)	3.3	11.60 (1.573)
2IGD	61	5.6 (0.6)	20.75 (3.615)	4.7	4.1	82.09 (12.19)	**4.0**	16.94 (0.347)
1OF9	77	11.5 (0.1)	18.33 (1.032)	10.9	2.6	140.0 (33.79)	**2.5**	18.50 (0.429)
1SPK	72	4.6 (0.6)	25.12 (1.246)	3.7	**3.4**	175.7 (9.452)	4.0	18.42 (0.353)
1VIG	71	5.5 (1.2)	25.25 (1.281)	4.1	**3.3**	160.3 (34.90)	3.4	23.00 (0.288)
1I11	81	4.8 (1.1)	25.00 (1.322)	2.9	3.1	49.91 (5.924)	**2.8**	10.11 (0.122)
1TEN	87	6.8 (0.6)	34.50 (1.511)	5.7	**4.1**	1014 (86.20)	4.6	50.93 (1.611)
1DCJ	81	5.3 (0.9)	30.87 (1.807)	4.0	3.7	310.7 (43.92)	**3.5**	19.51 (1.196)
1JHG	100	5.0 (0.7)	38.25 (6.273)	3.8	4.0	187.0 (33.59)	**3.3**	32.63 (1.255)
1WHM	92	6.4 (0.5)	35.00 (1.772)	5.8	**3.3**	611.5 (63.32)	3.7	30.18 (1.693)
2CJO	97	4.7 (0.8)	38.62 (2.924)	**3.5**	4.2	1988 (162.6)	4.3	56.41 (1.274)
1A0B	125	6.0 (1.4)	45.25 (2.964)	4.4	**2.7**	285.9 (61.19)	3.5	40.44 (0.99)
1H10	125	7.1 (1.4)	47.50 (5.830)	5.4	5.1	1693 (300.3)	**5.0**	49.76 (2.648)
1F98	125	5.8 (1.0)	55.00 (3.681)	**4.5**	4.7	2987 (598.4)	5.9	51.56 (2.795)
2CJ5	150	7.1 (1.9)	62.77 (5.540)	**4.8**	5.5	285.9 (61.19)	5.2	100.6 (4.546)
1STB	150	6.3 (0.7)	62.80 (5.329)	4.8	6.1	4891 (372.8)	**4.5**	58.00 (3.399)
1LEO	150	6.5 (1.0)	69.50 (5.380)	**4.9**	5.2	4125 (402.6)	5.4	101.6 (8.208)
1BGD	175	7.8 (1.2)	59.10 (9.085)	5.8	**5.1**	5000 (-)	5.2	135.5 (7.360)
1FNL	175	6.8 (1.2)	60.70 (4.056)	**5.1**	**5.1**	5000 (-)	7.2	376.6 (26.34)
1T8A	175	6.8 (1.2)	86.40 (6.380)	5.0	**4.1**	5000 (-)	5.5	113.6 (4.960)
1IB1	200	7.8 (1.5)	129.0 (9.297)	6.4	**3.1**	4436 (543.1)	**6.4**	208.7 (10.80)
2GH2	200	8.0 (1.0)	101.9 (6.297)	6.9	**6.7**	5000 (-)	7.5	84.75 (0.851)
1RR9	200	7.1 (1.8)	86.40 (12.33)	4.8	**4.6**	5000 (-)	6.9	144.3 (18.84)

Table 8.13: Quality evaluation: best results of GMAS systems against Rosetta.

The quality of the structures predicted by GMAS is in line with the results of Rosetta. On the other hand, Rosetta tends to be faster on longer proteins (e.g., ≥ 150 amino acids). This is due to several factors such as a better energy function that can lead sooner to a local minimum, the computation of the fragments database for each target, and the various heuristics encoded in that tool during the years.

Comparison with I-TASSER: We compared GMAS with another state-of-the-art protein structure prediction system, namely the Iterative TASSER (*I-TASSER*) tool [181, 136]. I-TASSER is a threading-based system that builds protein structures from primary sequences considering already known homologous proteins. Table 8.14 presents the comparison between I-TASSER and GMAS. For comparing I-TASSER with GMAS we considered the complete benchmark set of 16 proteins presented in [169] (benchmark 1). For each protein we report the best result found by I-TASSER (as reported in their work), as well as the best results found by GMAS using Monte Carlo without the GC constraint and Gibbs with the CG constraints, that previous experiments proved to be the best combinations. The running time of I-TASSER computations are those reported in the 2007 paper. However, we have re-launched some of them with our machine obtaining similar (slightly worse) results with their timeout of five hours. The RMSD results shows that our tool (considering our best result with the two options) has results comparable to I-TASSER (and obtained in shorter time).

		I-TASSER	**GMAS**			
			MC		Gibbs	
ID	n	RMSD	RMSD	Time (sd)	RMSD	Time (sd)
1B72_A	49	3.1	**2.4**	20.56 (3.863)	3.1	7.437 (0.081)
1SHF_A	59	**1.7**	3.0	144.2 (11.63)	3.7	17.92 (0.096)
1TIF	59	7.0	3.9	45.11 (4.230)	**3.6**	9.350 (0.665)
2REB_2	60	4.7	4.1	17.78 (3.393)	**2.4**	7.023 (0.018)
1R69	61	**1.9**	3.5	124.5 (15.06)	3.5	15.58 (0.164)
1CSP	67	**2.1**	4.4	248.0 (46.72)	4.8	22.26 (1.124)
1DI2_A	69	**2.3**	4.8	41.23 (5.382)	5.4	13.09 (0.273)
1N0U_A4	69	4.4	4.7	128.3 (16.03)	**3.7**	19.63 (0.950)
1MLA_2	70	**2.7**	3.6	61.75 (14.83)	4.6	16.95 (0.241)
1AF7	72	4.2	**3.4**	60.39 (7.870)	**3.4**	14.92 (0.109)
1OGW_A	72	**1.1**	4.3	485.1 (66.16)	3.5	30.40 (0.518)
1DCJ_A	73	10.0	**3.7**	267.5 (42.02)	3.9	24.40 (1.267)
1DTJ_A	74	**1.7**	4.3	247.8 (23.99)	3.1	25.41 (0.172)
1O2F_B	77	5.2	3.6	228.5 (16.91)	**3.4**	22.82 (1.262)
1MKY_A3	81	4.5	**3.8**	545.7 (88.16)	4.9	35.20 (0.551)
1TIG	88	**4.4**	4.7	196.4 (30.92)	5.6	31.73 (0.301)

Table 8.14: Quality evaluation: best results of GMAS systems against I-TASSER.

Comparison with FIASCO: We compared GMAS with another CP-based tool FIASCO ([19] —Table 8.15). FIASCO is a C++-based constraint solver targeted at modeling a general class of protein structure studies that relies on fragment assembly techniques. The benchmark set used for these experiments is the same used in [19]. For GMAS we report the best results among 20 runs for each combination using Monte Carlo and Gibbs without the CG constraints, that previous experiments proved to be the best combinations. Let us observe that GMAS is a clear winner in this case.

		FIASCO		**GMAS**			
				MC		Gibbs	
ID	n	RMSD	Time	RMSD	Time (sd)	RMSD	Time (sd)
1ZDD	35	2.0	685.2	**0.9**	4.343 (1.263)	**0.9**	2.639 (0.013)
2GP8	40	6.2	376.8	**1.3**	1.916 (0.226)	**1.3**	2.070 (0.180)
2K9D	44	2.5	513.0	**0.9**	13.67 (5.427)	**0.9**	4.687 (0.020)
1ENH	54	8.2	1900	2.3	13.75 (5.731)	**1.3**	10.86 (1.301)
2IGD	61	10.5	1588	4.1	82.09 (12.19)	**4.0**	16.94 (0.347)
1SN1	63	5.5	889.2	4.1	47.35 (10.41)	**3.0**	46.40 (9.726)
1AIL	69	4.5	267.6	**2.3**	6.546 (0.593)	2.4	6.403 (0.651)
1B4R	79	6.1	504.6	4.1	462.1 (97.58)	**4.0**	479.8 (47.01)
1JHG	100	4.5	270.0	4.0	187.0 (33.59)	**3.3**	32.63 (1.255)

Table 8.15: Quality evaluation: best results of GMAS systems against FIASCO.

8.6.4 A Case of Study: 3BHI

In this section we consider a specific long protein as representative for a case of study to assess the capabilities of our solver on "hard" proteins. We selected the protein *3BHI* with $n = 276$, and its secondary structure contains 6 α-helices, 3 β-sheets, and 7 turns. First of all we compute the offsets of the secondary structure elements on the primary sequence using JNet that will produce a text file containing the desired alignment:

SSGIHVALVTGGNKGIGLAIVRDLCRLFSGDVVLTARDV . . .
———EEEE————HHHHHHHHHHHHH——EEEEE——H . . .

JNet requires few seconds to generate the alignment and hence it does not affect the overall computational time. However, this input file might be also generated by other tools or on-line servers (e.g., `http://bioinf.cs.ucl.ac.uk/psipred/`). Once the alignment has been generated we run the solver specifying the Gibbs algorithm as search strategy for the Coordinator agents, with 150 sampling steps, and the file containing the alignment as input file. We used the Gibbs option in the coordinator that proved to be the faster for long proteins in our previous experiments. We increased the number of samples to 150 since we experimentally observed a convergence of the quality of the solutions after 150 samples, running the system considering 50, 100, 150 and 200 samples. Results are reported in Table 8.16 - `Default` row.

Experiment	Time (sd)	RMSD (sd)	Energy (sd)	Best RMSD
`Default`	3885 (178.4)	12.15 (2.0)	-83854.05 (3060.509)	10.8
`Default-CG`	908.2 (60.83)	11.40 (1.4)	-74065.95 (2979.682)	10.1
`Multi-Coordinators`	1317 (2.146)	10.175 (2.1)	-86866.9 (4447.796)	8.3
`Multi-Coordinators-CG`	939.8 (3.024)	11.17 (1.2)	-76466.71 (5154.601)	9.8

Table 8.16: Case of Study: 3BHI (276 amino acids)

Then we have introduced the CG constraint: `Default-CG` row shows that time is reduced from 3885 to 908.2 seconds. Let us observe that also the quality of the solution is improved from 12.15Å to 11.40Å although the energy value is increased. This means that the energy function and, in particular, the weights of the energy components are not completely precise.

To further improve the prediction we used more than one coordinator agent, in particular one for each sequence of amino acids between a pair of consecutive secondary structures. Results are reported in column `Multi-Coordinators` column. With these new settings we improved the quality of the predictions (i.e., the structure is more compact since loops and turns are better simulated) and we reduced the computational time (since coordinator agents are associated to few variables). Finally we use both these facilities and results are reported in `Multi-Coordinators-CG` row. The time is further reduced but the average RMSD is increased due to the side chain centroids that forbid structures that are too compact.

8.6.5 Comparing different GPUs

In this section, we compare three different GPUs with different computational capabilities in order to evaluate how much the architecture affects execution times. We report the details of the various hardware used (CPU and GPU). (1) is the hardware used in previous experiments: TITAN: *Host* AMD Opteron Processor 6376, 2.3GHz, *Device* GeForce GTX TITAN, 2688 cores@875MHz (14SM); (2) Quadro: *Host* Intel Core i7-3770, 3.4GHz, *Device* Quadro 600, 96 cores@640 MHz (2SM); (3) Tesla: *Host* Intel Xeon, 2.4GHz, *Device* Tesla C2075, 448 cores@1.15 Ghz (14SM).

Table 8.17 compare these three different architectures on three different predictions using the MC algorithm and Gibbs sampling considering default settings, without the CG constraint. We report running time and speed-ups in any of the three machines. It emerges clearly that the approach benefits from better GPUs either in the running time or in the speed-up. Let us observe that we still have speedups also for the Quadro device that is the less powerful graphic card among the three devices, whit 2 SM and 96 cores (whereas the host has 3.4GHz w.r.t. 2.3 GHz of TITAN host and Tesla host). Speedups are in the order of $2 \sim 3$ when considering the Gibbs sampling algorithm that has been proven in the previous experiments to be the more effective for longer proteins.

Protein ID	n	**Time**									
		TITAN			Quadro			Tesla			
		CMAS (sd)	GMAS (sd)	SUp	CMAS (sd)	GMAS (sd)	SUp	CMAS (sd)	GMAS (sd)	SUp	
1E0N	27	63.80	10.73 (1.463)	5.9	59.92 (5.050)	152.3 (27.07)	0.3	87.92 (12.24)	23.98 (4.681)	3.6	
2HBB	51	476.3	64.42 (6.383)	7.3	329.9 (45.69)	873.6 (146.9)	0.3	523.2 (52.55)	163.0 (19.51)	3.2	
1JHG	100	1168	187.0 (33.59)	6.2	1175 (82.02)	1719 (414.7)	0.6	1414 (137.3)	270.2 (76.11)	5.2	Monte Carlo
1E0N	27	69.69	3.379 (0.025)	20.6	67.80 (0.037)	33.16 (0.458)	2.0	74.89 (0.147)	7.362 (0.073)	10.1	
2HBB	51	372.6	11.68 (0.116)	31.9	405.8 (1.574)	150.9 (2.008)	2.6	433.3 (0.800)	26.72 (0.246)	16.2	
1JHG	100	1462	32.63 (1.255)	44.8	1602 (11.15)	433.8 (12.02)	3.6	1674 (0.957)	71.96 (2.411)	23.2	Gibbs

Table 8.17: Comparison between different GPUs - Monte Carlo and Gibbs sampling.

8.7 Summary

In this chapter we presented a novel perspective for addressing the Protein Structure Prediction Problem. We used a declarative approach for ab-initio simulation, implementing a multi agent system. Moreover, we used a GPU architecture to efficiently explore the search space and to propagate constraints. The results are remarkable; the use of GPU allows us to obtain speedups up to 75. The system can fold proteins of small-medium length with a low computational time (in the order of minutes) and quality of the results comparable with the state-of-the-art systems.

9 Conclusions

In this dissertation we presented a feasibility study exploring the potential of fine-grained GPU-level parallelism in the context of constraint solving. We addressed several aspects regarding the constraint solving process and how to speed up a constraint engine by using GPGPU computation. This study has resulted in the development of several solver prototypes, e.g., FIASCO (see Section 3) or the parallel multi-agent system (see Section 8). We showed that the constraint solving process can be effectively improved by the use of GPU architectures, obtaining remarkable speed ups, especially on hard combinatorial real-world problems.

The dissertation is divided in three main parts; in what follows we recap the main results of each part and we suggest some possible extensions as future work.

9.1 Constraint Programming: Definitions and Challenging Applications

In the first part of this dissertation we introduced the constraint programming paradigm. We presented some definitions and general notions about constraint satisfaction and optimization problems, constraint propagation, and search strategies. This introductory notions have been realized in a prototype constraint solver targeted at modeling a general class of protein structure studies. We considered a real-world problem–the protein structure prediction problem–to assess the strengths and the weaknesses of constraint-based technologies on hard combinatorial problems. The purpose of this part of the dissertation was to enter in detail on the aspects that characterize the solving process, in particular, on the aspects that could potentially take advantage from a parallel implementation. From a practical point of view, we presented the *FIASCO* (*Fragment-based Interactive Assembly for protein Structure prediction with COnstraints*) resolution engine, an efficient C++-based constraint solver. We presented a novel constraint (joined-multibody) to model model rigid bodies connected by joints, with constrained degrees of freedom in the 3D space. We presented a polynomial time approximated filtering algorithm of the joined-multibody constraint, that exploits the geometrical features of the rigid bodies. In particular, the filtering algorithm is combined with search heuristics that can produce a pool of admissible solutions that are uniformly sampled. This allows for a direct control of the quality and number of solutions. The filtering algorithm is based on a 3D clustering procedure that is able to cope with a high variability of rigid bodies, while preserving the computational cost. The tests showed how the parameters of the constraint are able to control effectively the quality and computational cost of the search.

9.1.1 Future work

As future work, from the applications side, we plan to explore the protein loop closure problem, with the use of specific databases and scoring functions. For the close problem of protein flexibility, we plan to use FIASCO solver to generate the conformational space of long scale movements for nuclear receptors. Finally, we plan to use FIASCO in the general context of protein structure prediction with the combination of local search methods and protein-ligand spatial constraints (e.g., implementing the *Large Neighborhood Search* technique). From the constraint side, we plan to integrate the JM filtering algorithm with other distance constraints, in order to generate more accurate clusters; we plan to integrate spatial constraints inferred from bounds on energy terms (e.g., the favorable contributions provided by pairing secondary structure elements translate into energy bounds and distance constraints). We plan to investigate the use of multiple JM constraints to model super-secondary structures placement, which are useful to capture important functional and structural protein features. The latter can be thought of as imposing several spatial path preferences to a given chain of points. Finally, we intend to integrate the constraint solver with a visual interface to make it easily available to Biologist and other practitioners.

9.2 Parallel Constraint Solving

In the second part of this dissertation we moved our attention to parallel constraint solving. We introduced some literature review on parallel constraint consistency (e.g., ParAC-3) and parallel search, with particular attention to parallel local search strategies that have been proved effective in solving hard combinatorial problems. We then focus on the two main aspects regarding parallel constraint solving, namely GPU-based propagation and GPU-based search. GPU-based propagation regards the parallelization of a constraint engine using GPU architectures. We presented the structure of a constraint solver capable of hybrid propagation (i.e., alternating CPU and GPU) within a sequential exploration of the search space. We showed that SIMT parallelism is suitable to the type of processing that constraints are subject to during consistency checking. In particular, parallel constraint propagation on GPU is effective in the context of complex global constraints (e.g., *table* constraint, see Chapter 5).

In GPU-based search we proposed the design and implementation of a novel constraint solver that exploits parallel Local Search (LS) using GPGPU architectures to solve constraint optimization problems. The optimization process is performed in parallel on multiple large promising regions of the search space, with the aim of improving the quality of the current solution. Large neighborhoods are explored using LS techniques with the goal of improving the current solution evaluating a large set of neighborhoods at a time. The choice of local search strategies was twofold: first, incomplete but fast methods are usually preferred for optimization problems where the search space is very large but not highly constrained. Second, with very few changes, the parallel framework adopted for a local search method can be easily generalized to be suitable for many different local search strategies, requiring minimal parameter tuning. Experimental results showed that the solver implemented on GPU outperforms its sequential version. Good results were also obtained by comparing the solver against standard CP and LNS. Moreover, we showed that many LS strategies can be encoded on our framework by changing few parameters, without worrying about how it is actually performed the underlying parallel computation.

9.2.1 Future work

As future work, we plain to exploit a deeper integration within LNS and constraint propagation. The framework should be general enough to allow the user to combine kernels in order to design any search strategy, in a transparent way w.r.t. the underlying parallel computation. Combining kernels to define different (local/complete) search strategies should be done using a declarative approach, i.e., we plan to extend the MiniZinc language to support the above features.

9.3 Parallel Constraint Solving: Case Study

In the third part of this dissertation we considered a real-world problem as case of study for investigating the effective potential of GPGPU computation in constraint programming. In particular, we address the Protein Structure Prediction Problem presented in the first part of this dissertation through a declarative approach for ab-initio simulation, implementing a multi agent system. Differently from the sequential version of the solver FIASCO, we used a GPU architecture to efficiently explore the search space and to propagate constraints. The results are remarkable; the use of GPU allows us to obtain speedups up to 75. The system can fold proteins of small-medium length with a low computational time (in the order of minutes) and quality of the results comparable with the state-of-the-art systems.

9.3.1 Future work

As future work, we plan to improve the search strategies of the agents. In particular, we shall try to make use of dynamic priorities between agents. These priorities are set by the supervisor agent, and dynamically modified on the base of the current partial results computed by the structure and the coordinator agents. Moreover, the prediction of the area where secondary structures might occur, currently delegated to the external tool JNet, will be incorporated in the system (as initial module of the supervisor agent).

An interesting direction is the study of an implementation of the multi-agent system on a multi-GPU environment. Different GPUs can be assigned to different structure and coordinator agents, which can exchange information during the whole folding process. The entire search phase is governed by the supervisor agent which assigns jobs to and retrieves results from processes running on different GPUs. We are also developing a visual interface that allows the use of the tool outside the community of computer scientists.

A

JM Constraint: Complexity Analysis

A.1 Complexity Analysis

JM constraint. The problem of establishing *consistency*—i.e., existence of a solution—of JM constraints is NP-complete. To prove this fact, we start from the NP-completeness of the consistency problem of the constraint *Self-Avoiding-Walk* (*SAW-constraint*) in a discrete lattice, proved in [40]. In particular, we will use the 3D cubic lattice, where the problem can be stated as follows. Let $\vec{X} = X_1, \ldots, X_n$ be a list of variables. Each variable has a finite domain $\mathsf{D}^{(}X_i) \subseteq \mathbb{Z}^3$. $\sigma : \vec{X} \longrightarrow \mathbb{Z}^3$ is a solution of the SAW constraint if:

- For all $i = 1, \ldots, n$: $\sigma(X_i) \in \mathsf{D}^{(}X_i)$,
- For all $i = 1, \ldots, n-1$: $\|\sigma(X_i) - \sigma(X_{i+1})\| = 1$,
- For all $i, j = 1, \ldots, n$, $i < j$, it holds that $\|\sigma(X_i) - \sigma(X_j)\| \geq 1$.

As emerges from the proof in [40], the problem is NP complete even if the domains of $\mathsf{D}^{(}X_1)$ and $\mathsf{D}^{(}X_2)$ are singleton sets. Without loss of generality we can concentrate on SAW problems where $\mathsf{D}^{(}X_1) = \{(0,0,0)\}$ and $\mathsf{D}^{(}X_2) = \{(0,1,0)\}$—the other cases can be reduced to this one using a roto-translation.

Theorem A.1.1 *The consistency problem for the JM constraint is NP-complete.*

Proof. [sketch] The proof of membership in NP is trivial; given a tentative solution, it is easy to test it in polynomial time—the most complex task is building the rotation matrixes.

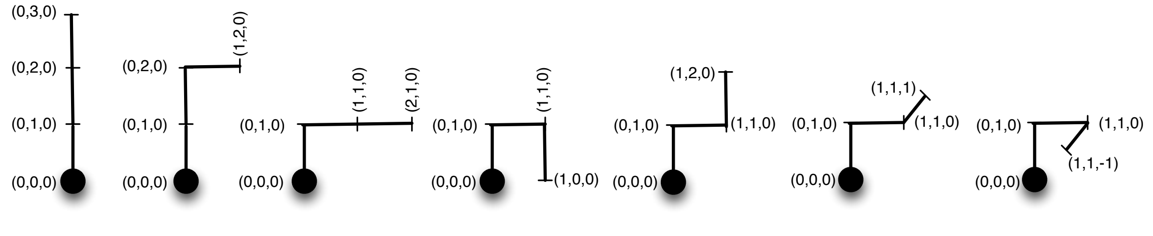


Figure A.1: Rigid blocks for SAW (from left to right, block 1, ..., 7)

To prove completeness, let us reduce SAW, with the further hypothesis on $\mathsf{D}^{(}X_1)$ and $\mathsf{D}^{(}X_2)$ to JM. Let us consider an instance $A = \langle \vec{X}, \mathsf{D}^{(}\vec{X})\rangle$ of SAW with n ($n > 3$) variables. We define a equi-satisfiable instance $B = \langle \vec{S}, \vec{V}, \vec{\mathcal{A}}, \vec{\mathcal{E}}, \delta\rangle$ of JM as follows. Let us choose $\vec{V} = V_1, \ldots, V_{n-3}$. We select the sets S_i of rigid blocks of the multi-body to be all identical, and consisting of all the (non overlapping) fragments of three contiguous unitary segments of length 1, starting from $(0,0,0)$ and with bends of 0 or 90 degrees. A filtering using symmetries is made and the blocks are indicated in Fig. A.1. For all $i = 1, \ldots, n-3$ we assign the following sets of 3D points to the end-effectors:

$$\mathcal{E}_{3i-2} = \mathsf{D}^{(}X_{i+1}) \cap \mathbb{Z}^3, \quad \mathcal{E}_{3i-1} = \mathsf{D}^{(}X_{i+2}) \cap \mathbb{Z}^3, \quad \mathcal{E}_{3i} = \mathsf{D}^{(}X_{i+3}) \cap \mathbb{Z}^3$$

Moreover, let $\mathcal{A}_1 = \{(0,0,0)\}, \mathcal{A}_2 = \{(0,1,0)\}, \mathcal{A}_3 = \{(0,2,0),(1,1,0)\}$. Observe that all these sets are subsets of $\mathbb{Z}^3$ and therefore points of the same lattice of the SAW problem. Assigning $\delta = 1$, the reduction is complete. It is immediate to check that SAWs in the 3D lattice and the solutions of the JM constraint defined via reduction are the same 3D polygonal chain. $\square$

Compatible JM constraint. We introduce the notion of *compatible multi-body*, i.e., a multi-body where for all pairs of fragments $B, B' \in \mathcal{B} = \bigcup_{i=1}^{n} S_i$ and for all the front- and end- anchors points $\vec{p}, \vec{q} \in \{\mathsf{front}(B), \mathsf{front}(B'), \mathsf{end}(B), \mathsf{end}(B')\}$ it holds that $\vec{p} \frown \vec{q}$. A JM constraint is said to be *compatible* if the sequence of bodies $S_1, \dots, S_n$ is a compatible multi-body. Here we prove the NP-completeness of the compatible-JM constraint where we do not make use of joints made by collinear points. The proof of NP hardness is a reduction from the Hamiltonian Circuit (HC, for short) problem on particular graphs (special planar graphs—SPG, for short) and it is organized in two steps. First we show how to encode a SPG G in a discrete cubic lattice, then we define domains and variables for a compatible-JM constraint, such that there is a solution of the HC problem on the graph *iff* there is a solution for the compatible-JM constraint.

A *special planar graph* $G = (N, E)$ is a planar graph composed by *loops* and *paths.* For example, Figure A.2 represents a special planar graph with 4 loops and 9 paths. Each loop contains only nodes of degree 3 (e.g., the nodes A, B, C, D in loop 1 of Fig. A.2) while each path has length 2 and connects nodes that belong to distinct loops. Since we look for an HC, we focus on graphs

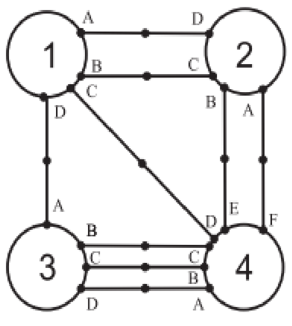


Figure A.2: A special planar graph G. There are four loops with nodes of degree 3 and 9 paths of length 2, connecting nodes that belong to distinct loops.

containing only loops with an even numbers of nodes. Moreover, we assume that G contains at least 2 loops, otherwise the HC problem is trivially true.

A three dimensional cubic lattice is a graph of the form (P, A) such that $P \subseteq \mathbb{N}^3$ and $A = \{\{(x_1,y_1,z_1),(x_2,y_2,z_2)\} \subseteq P : |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2| = 1\}$. We use auxiliary graphs, embedded in a three dimensional cubic lattice and referred to as *cubic graphs*, in order to map SPGs into cubic lattices.

As a first step, given a SPG $G = (N, E)$, we map it into a cubic graph $G' = (N', E')$. Let us consider a loop l_i of G whose size is n_i and which consists of the nodes $h_1, \dots, h_{n_i}$. We replace l_i in G' with an equivalent gadget (i.e., a subgraph of G') which contains a number of nodes proportional to n_i. The core of the gadget has a symmetrical structure and is made of a loop of $7n_i$ nodes. In particular, there are n_i *starting* nodes, connected through paths to the nodes $p_1, \dots, p_{n_i}$. These nodes correspond to the loop nodes $h_1, \dots, h_{n_i}$ and they are called *output* nodes of the gadget. Figure A.3 represents a gadget corresponding to a loop of size 4: starting nodes are represented using empty circles, while p_1, p_2, p_3, and p_4 are the output nodes of the gadget.

Let us show how to arrange starting nodes, output nodes, and the paths connecting them. Each starting node is separated from the next one by exactly 6 nodes using two different templates (Fig. A.3 (right)):

- The first template (T_1) is used to connect two starting nodes on the upper and lower side of the gadget. The relative coordinates of this template are: $(0,0,0), (0,1,0), (2,1,0),$

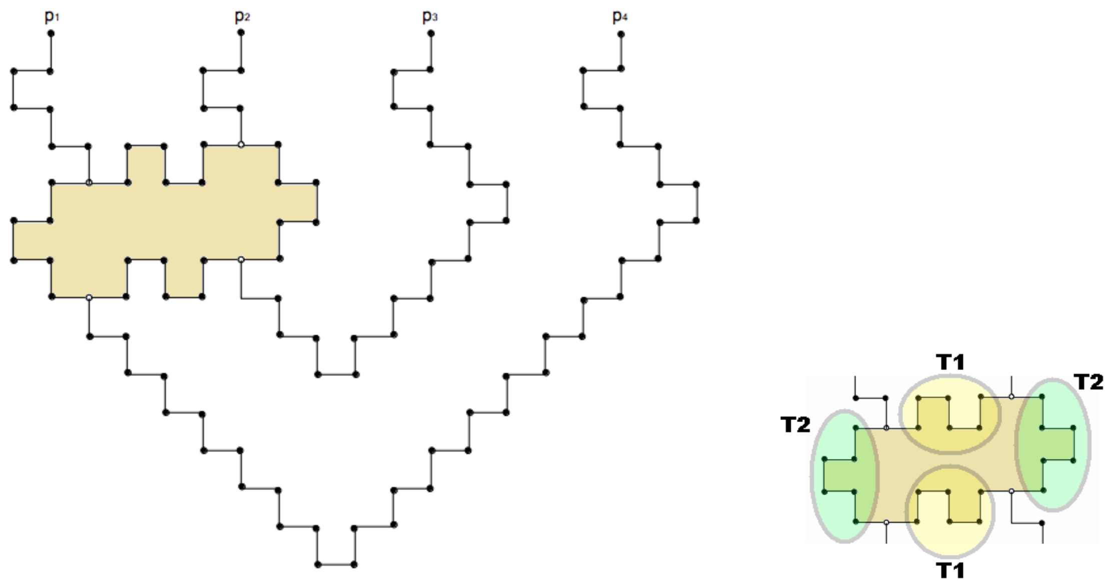


Figure A.3: Gadget for a loop of size 4. *Starting* nodes are represented by empty circles. p_1, p_2, p_3, p_4 are the *output* nodes. The two templates T_1 and T_2 are emphasized on the right

$(2, 0, 0), (3, 0, 0), (3, 1, 0)$

- The second template (T_2) is used to connect the left and right side of the gadget. The relative coordinates of this template are: $(0, 0, 0), (0, -1, 0), (-1, -1, 0), (-1, -2, 0), (0, -2, 0), (0, -3, 0)$

The paths from the starting nodes to the output nodes of a gadget are built in such a way to create an angle of 90 degrees between each pair of consecutive edges (see Fig. A.3). Let us observe that each path leading to the output nodes is at a minimum distance of 5 from each other, including the output nodes. These distances must be taken into account when we identify a HC on the nodes of G'—determined as the result of an overlapping of rigid blocks obtained as a solution of the compatible-JM constraint. Rigid blocks correspond to sequences of nodes of G' that must be placed on the cubic lattice considering proper distances between them, as shown in the remainder of this proof.

To complete the graph G' let us fix one of the dimensions of the cube, for example $z = 0$, and let us work on the resulting bi-dimensional plane. We consider an enumeration of the loops $l_1, \dots, l_k$ of G. Hence, we align the gadgets on the plane (x, y) according to such ordering and setting a distance of 6 between the rightmost output node of the gadget l_j and the leftmost output node of the gadget l_{j+1}, for $1 \leq j \leq k$, as shown in Fig. A.4. The output nodes of the gadgets will have $y = 0$, while all other nodes of the gadget will have $y > 0$. Note that with such arrangement there is a minimum distance of 3 for each pair of consecutive gadgets.

Now we show how to connect the gadgets in G' in order to mimic the topology of G. Consider, in a lexicographical order, the loop pairs $\langle l_a, l_b \rangle$, $a < b$, that are connected by edges in E. We wish to create copies of the output nodes of l_a and l_b in a separate plane (to avoid intersection of edges) and recreate on this plane the connection structure of G.

Let us illustrate the process for all the pairs $\langle l_1, l_{b_1} \rangle, \dots, \langle l_1, l_{b_h} \rangle$ such that there are edges between loop 1 and loop b_i in G. In the plane i, we add copies of the "relevant" output nodes for loop 1 and b_i and we connect these copies to the output nodes of the relative gadget with a square spiral path of side 1. For example, if an output node of loop 1 or loop b_i is at coordinates $(x, y, 0)$, and such node is part of an edge connecting these two loops, then a copy of such node will be created at coordinates $(x, y + 1, i)$ together with the intermediate nodes and edges connecting them at coordinates $(x, y, 1)$, $(x - 1, y, 1)$, $(x - 1, y + 1, 1)$, $(x, y + 1, 1)$, $(x, y + 1, 2), \dots, (x, y + 1, i)$

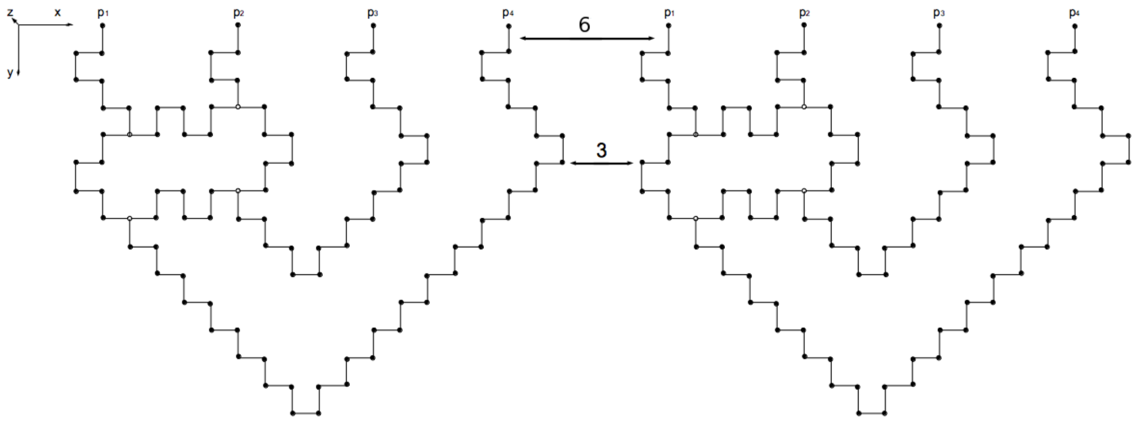


Figure A.4: Two aligned gadgets.

(Fig. A.5). Note that the “last” square created on level i for the output node of the loop 1 should be symmetrical w.r.t the last square created on level i for the output node of the loop b_i, i.e, the missing edge of the last square should be on the right side for the loop 1 and on the left side for the loop b_i. The edges between the nodes of distinct loops of G are simulated by paths in the plane i.

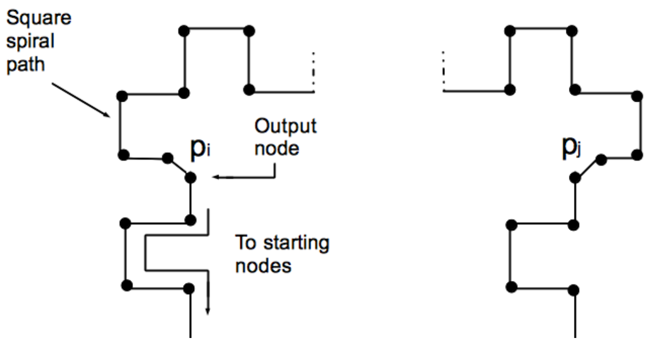


Figure A.5: Square spiral paths used to connect output nodes with their copies on different levels.

Similarly to the core paths of a gadget, every pair—but possibly the last—of adjacent edges form an angle of 90 degrees. The only difference here is that if a path starts at coordinates (x, y, z), then its nodes can be set only on the coordinates y and $y + 1$. Moreover, the last pair of edges must be considered. Due to the topology of the cubic lattice, when there is an even distance (e.g., 6) between the output nodes of the loop 1 and the output nodes of the loop b_i, we use a special gadget formed by a square of side 1 (Fig. A.6) in order to connect them. Let us call this gadget a *ml* gadget (*merge-loops* gadget).

Furthermore, every pair of adjacent paths in the same plane must be placed at a distance of (at least) 3 from each other, and every path must be at a distance of 4 from every square spiral path of the same gadget in the same level, as shown in the example of Figure A.7. Note that since G is planar, we can connect loops using *non-intersecting* paths in plane i.

The above process is repeated for all the other pairs of connected loops (incrementing the plane levels). It is immediate to prove that the size of G' is polynomial w.r.t. the number of nodes of G.

Since G' maintains the topology of G, it holds that a solution for the HC problem on G can be mimicked in G'. Note also that G' can be seen as a copy of G where the edges are “stretched” (by adding new nodes of degree 2). We will refer to the sequences of edges connecting different loops on G' as *l2l* (*loop-to-loop* paths).

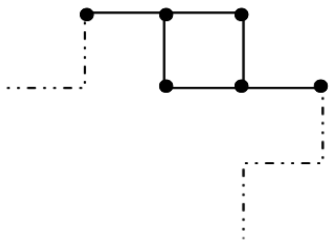


Figure A.6: The *ml* (*merge-loops*) gadget.

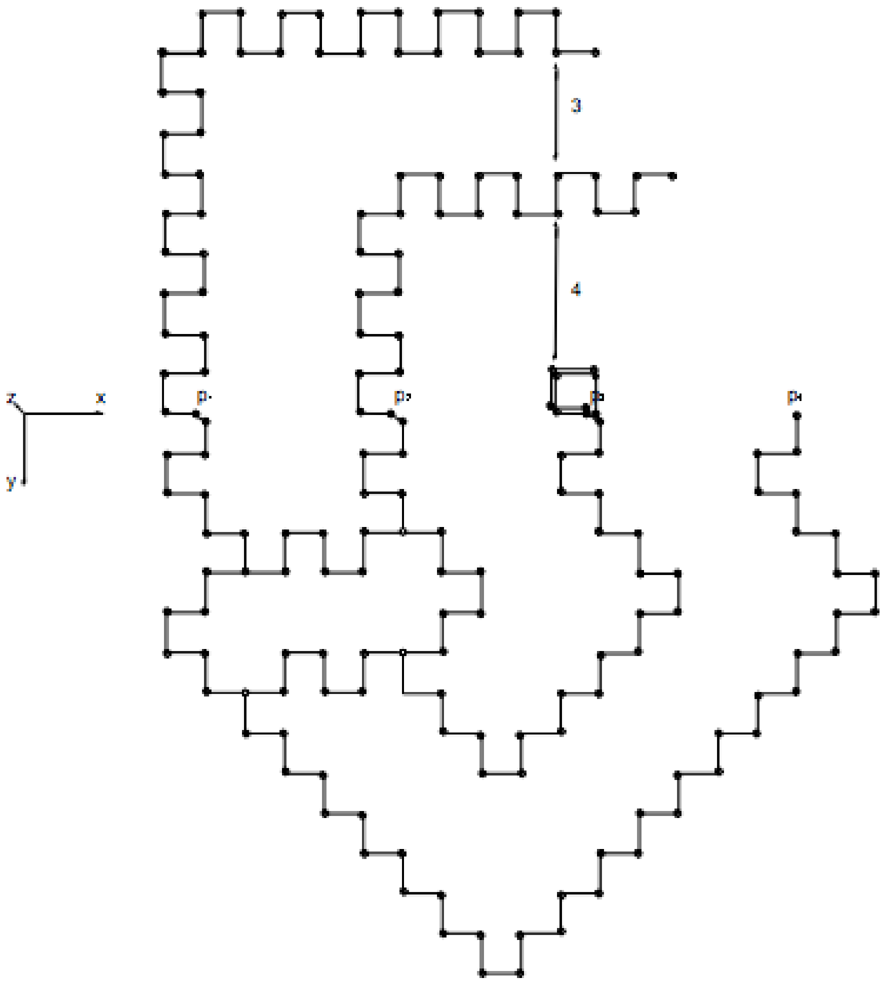


Figure A.7: Minimum distance between paths connecting different loops.

We can finally state the following theorem:

Theorem The consistency problem for the compatible JM constraint is NP-complete.
Proof. The proof of membership in NP is trivial; given a tentative solution, it is easy to test it in polynomial time—the most complex task is building the rotation matrixes.

To prove completeness, let us reduce the HC problem to an equi-satisfiable instance $B = \langle \vec{S}, \vec{V}, \vec{\mathcal{A}}, \vec{\mathcal{E}}, \delta \rangle$ of the compatible-JM constraint, and show that there exists a solution of the HC problem *iff* there is a solution for B.

We select the sets S_i of rigid blocks of the multi-body to be all identical, and consisting of 6 contiguous and non-colliding unitary segments of length 1 with additional constraints:

- The first three points of each rigid block are set on the coordinates $(0, 0, 0)$, $(0, 2, 0)$, $(2, 2, 0)$, respectively.
- It is possible to have bends of 0 or 90 degrees on the 3^{th}, 4^{th}, and 5^{th} point of each rigid block.
- There must be a fixed bend of 90 degree on the 6^{th} point of each rigid block.

These constraints allow us to handle rigid blocks where there is the same overlapping plane defined both on the anchors and the end-effectors of each rigid block. Note that the anchors and end-effectors of each rigid block are defined by 3 non collinear points, as stated by the definition of compatible-JM constraint. In particular, the constraints require the anchors of each rigid block K_i to form an angle of 90 degree between the first two segments of K_i. The same angle is also defined by the end-effectors of K_i (i.e., it is present between its last two segments). Considering the unitary length of each segment it is easy to see that the overlapping planes are the same among all the possible rigid blocks (Fig. A.8(a)).

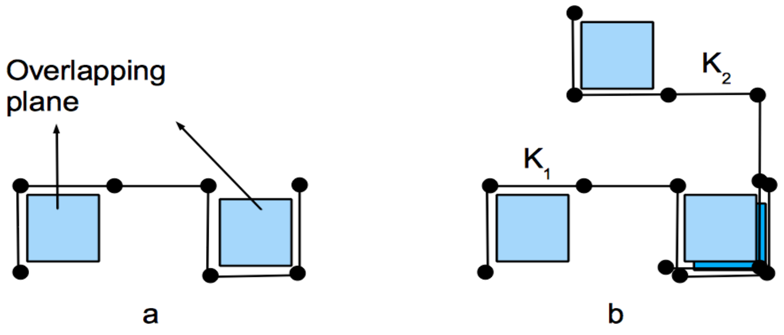


Figure A.8: Overlapping planes defined on rigid blocks (a). Two overlapped rigid blocks: K_2 on K_1 (b).

It follows that, given two rigid blocks K_1 and K_2 defined as above, it is always possible to overlap K_2 on K_1 (or vice-versa) by roto-translation of K_2 (K_1) w.r.t. K_1 (K_2) (Fig. A.8(b)). This fact was taken into account in the embedding of G into G'. In particular, G' has an angle of 90 degree between every pair of adjacent edges, allowing us to define an overlapping plane for a rigid block from every node of G' w.r.t. its two adjacent nodes. Roughly speaking, it is possible to overlap a rigid block on every 3 consecutive points of G'. The 38 different *templates* of possible rigid blocks are shown in Figure A.9 and Figure A.10 (rigid blocks defined on the bi-dimensional plane (x, y), Fig. A.9 and rigid blocks defined on the 3D plane (x, y, z), Fig. A.10).

Now, we define the set of variables $\vec{V} = V_1, \ldots, V_t$ in such a way that it is possible to decompose the graph G' in t overlapping rigid blocks. The basic idea is to find a decomposition of an HC on G' in rigid blocks which represent a solution for an instance of the compatible JM-constraint. Note that the number t of variables must be in relation with both the dimension of G' and the length of a rigid block. In particular, we must take care of the fact that since each rigid block has length 7, and it overlaps the last 3 points of the previous rigid block, t must correspond to a multiple of 4 (without considering the first fragment of the sequence of the overlapping fragments).

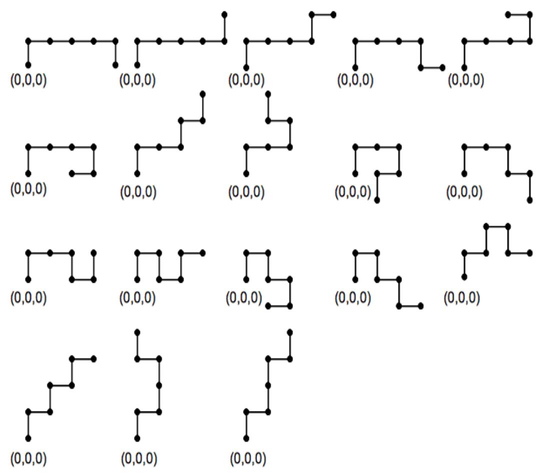


Figure A.9: 18 rigid blocks defined on the bi-dimensional plane (x, y). These templates are used as elements domains for the compatible JM-constraint.

Let $G = (N, E)$ a special planar graph and $G' = (N', E')$ the corresponding graph generated as described previously. Let $m = |N'|$, L be the global number of nodes in loops of G divided by 2, R the number of gadgets ml and $n = m - 6L - R - 4$. Without loss of generality, we can consider graphs where $n > 10$. First we define the values $l \in \{1, 2\}$ and $x \in \{0, 1, 2, 3\}$ as the result of the following system of equations, which take into account that n might not be a multiple of 4:

$$\begin{cases} (n - 7 - x) \bmod 4 & = 0, \\ l & = \begin{cases} 1 & \textbf{if} \quad x = 0 \\ 2 & \textbf{if} \quad x \neq 0. \end{cases} \end{cases}$$

Then, we calculate t as:

$$t = (n - 7 - x)/4 + l.$$

To complete the definition of the instance of JM, let us fix two consecutive nodes α and ζ of degree 2 in a l2l, as depicted in Figure A.11. Moreover, with a slight abuse of notation let us denote each node in N' with its 3D position and with $\zeta - (+)i$ the coordinates of the closest i^{th} point to $\zeta(\alpha)$ that precedes(follow) both ζ and α (Fig. A.11).
Then, for all $i = 1, \ldots, t - 1$ we assign the sets of points of G' to the end-effectors:

$$\begin{aligned} \mathcal{E}_{7i-2} &= N' \cap \mathbb{Z}^3, \\ \mathcal{E}_{7i-1} &= N' \cap \mathbb{Z}^3, \\ \mathcal{E}_{7i} &= N' \cap \mathbb{Z}^3. \end{aligned}$$

Instead, for the last end-effectors, we assign the following sets:

- if $x = 0$:

$$\begin{aligned} \mathcal{E}_{7t-2} &= \{\zeta - 5\} \cap \mathbb{Z}^3, \\ \mathcal{E}_{7t-1} &= \{\zeta - 4\} \cap \mathbb{Z}^3, \\ \mathcal{E}_{7t} &= \{\zeta - 3\} \cap \mathbb{Z}^3, \end{aligned}$$

- if $x \neq 0$:

$$\begin{aligned} \mathcal{E}_{7t-2} &= \{\zeta - 2\} \cap \mathbb{Z}^3, \\ \mathcal{E}_{7t-1} &= \{\zeta - 1\} \cap \mathbb{Z}^3, \\ \mathcal{E}_{7t} &= \{\zeta\} \cap \mathbb{Z}^3. \end{aligned}$$

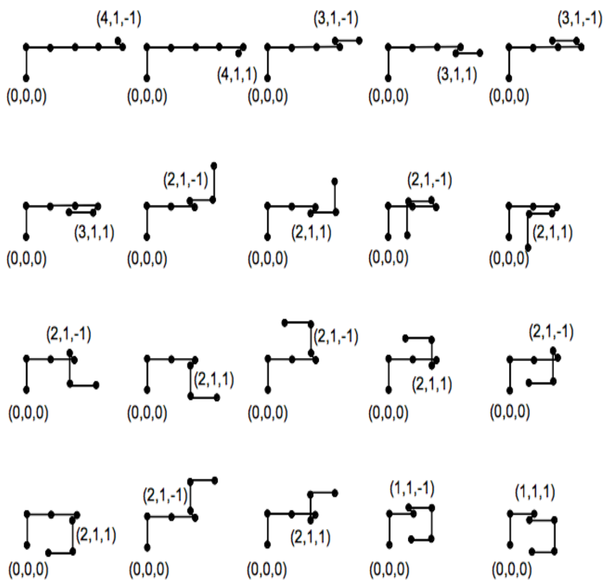


Figure A.10: 18 rigid blocks defined on 3D plane (x, y, z). These templates are used as elements domains for the compatible JM-constraint.

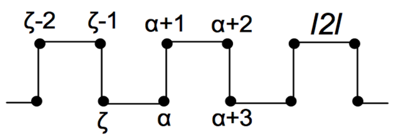


Figure A.11: α and ζ nodes in a *l2l*

For the anchors, let

$$\begin{aligned}\mathcal{A}_1 &= \{\alpha\} \cap \mathbb{Z}^3, \\ \mathcal{A}_2 &= \{\alpha + 1\} \cap \mathbb{Z}^3, \\ \mathcal{A}_3 &= \{\alpha + 2\} \cap \mathbb{Z}^3.\end{aligned}$$

Observe that all these sets are subsets of $\mathbb{Z}^3$ and therefore points of the same lattice of G'. We need to prove that with the last end-effector of V_t we can only reach either the point ζ or $\zeta - 3$. This is trivial, considering that between $\zeta - 3$ and α there are 3 nodes in G', and the number t of variables V_i depends on the module of 4. Assigning $\delta = 1$, the instance of the JM constraint is complete.

Now, we prove that there is an HC on G if and only if there is a solution σ of the instance B of the compatible JM-constraint. Let us assume that G has an HC. The same cycle can be mimicked on the extended graph G'. All nodes in 12ls are traversed by this path. Moreover, consider the Figure A.12 from left to right. Any Hamiltonian Cycle traverses the loop in a way similar to the one depicted (Fig. A.12a). In particular, there is a corresponding cycle traversing the loop in G' (Fig. A.12b). However it is not Hamiltonian but 6 half of the nodes of the loop cannot be traversed by that path, and for each gadget *ml* we left out exactly one node. Then we have exactly $n = m - 6L - R$ nodes traversed by the HC path. Choosing the nodes α and ζ at a distance of 1 from each other we can define an assignment σ of the compatible-JM constraint that decompose G' in t different fragments to assign to each variable in $\vec{V}$ (remember that we can overlap a rigid block on every 3 consecutive points of G'). These fragments belong to the domains of templates, each fragment is overlapped with the previous one and the minimal distance $\delta = 1$ holds. Hence σ

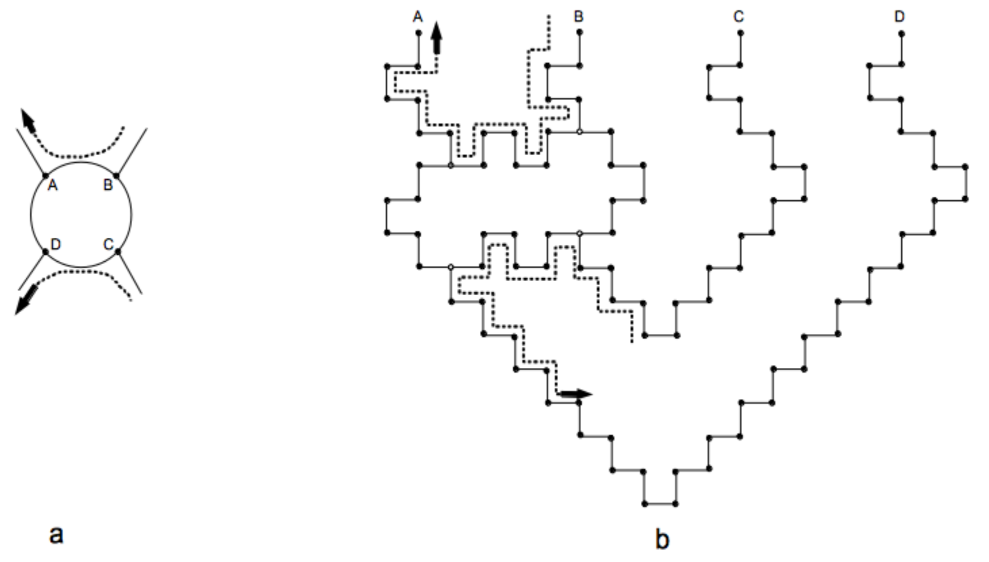


Figure A.12: Loops in G and G'; Observe that the Hamiltonian paths in G cannot touch half of the extra nodes added in loops in G'

is a solution for B.

On the other hand, let σ be a solution for the compatible JM-constraint $B = \langle \vec{S}, \vec{V}, \vec{\mathcal{A}}, \vec{\mathcal{E}}, \delta \rangle$ defined on G'. We show that this solution represents an HC on G. First of all, we observe that with σ we cannot overlap points. This is given by the δ constraint of JM (i.e., the paths defined by the overlapping rigid blocks assigned to each variable by σ are only *self-avoiding walks* on G'). Then we observe that the solution σ defines a path S of length $m - 6L - R - 4$ by overlapping rigid blocks (i.e., fragments of length 6) starting from $\alpha, \alpha + 1, \alpha + 2$ and ending in $\zeta - 2, \zeta - 1, \zeta$. Moreover, each rigid block has only the points of G' as possible end-effectors (corresponding to the anchors of the following rigid block) and, hence, S must be defined on G'. We show now that S defines an HC on G' and, in particular, an HC on G. In order to do this, we must prove that the solution σ assigns rigid blocks to the variables $V_1, \dots, V_t$ avoiding to reach other "zones" of the cubic lattice using paths not defined on graph G'. This could happen only when σ chooses a rigid block where the 4^{th} point of such rigid block is not set over G', though the $5^{th}, 6^{th}$, and 7^{th} must be, as depicted in the example of Fig. A.13.

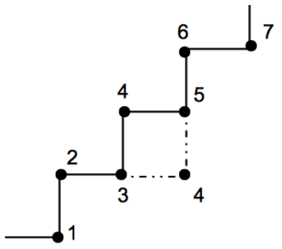
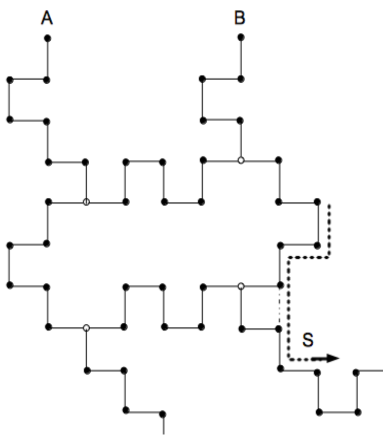


Figure A.13: Example of a case where the 4^{th} point of rigid block doesn't match the relative point in G'.

In particular, since G' each path is separated from its neighbors paths by a distance of of at least 3, it is easy to see that if a rigid block k has the 4^{th} point not set over G', then k must "re-enter" on G' with its $5^{th}, 6^{th}$, and 7^{th} points, in order to satisfy $\mathcal{E}_{7k-2}, \mathcal{E}_{7k-1}, \mathcal{E}_{7k}$. Note that there are cases where a rigid block k can take a different path in G' and left out more than 1 node w.r.t. G' (as show in the example of Figure A.14 - left). In such cases, there are less than t rigid blocks used to define the path on G' and we have a contradiction.



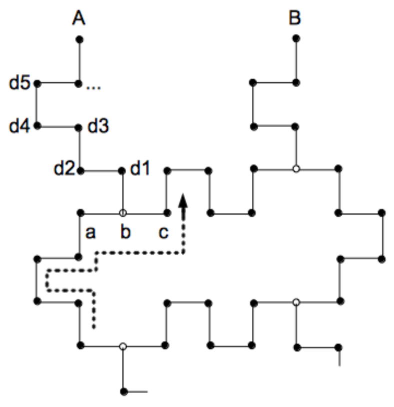


Figure A.14: Example of a case where the rigid block leave out 2 nodes of G' (left), and one of the cases s.t. the path S defined by σ enters and exits loops in G' (right).

Now, we consider the following 3 cases about how the path S defined by σ enters and exits each loop in G':

1. If S enters and exists all the loops as in Figure A.12(b), then it will leave out $m-6L-R-4$ nodes and it is easy to see that S corresponds to an Hamiltonian Path in G starting in α and ending in ζ. It is sufficient to add the edges $(\zeta, \zeta+1), \ldots, (\alpha-1, \alpha)$ to find the cycle which corresponds to an Hamiltonian Cycle in G.

2. Since α and ζ are in the same l2l it is impossible that S starts in α and ends in ζ without entering any loops.

3. It remains to analyze the case in which S traverses a loop in a way different from that of point 1. Assuming that such a path exists, we will nd a contradiction with its length $m-6L-R-4$. In this case, there is at least one l2l with at least 20 nodes left out from a path traversing one loop. Note that every l2l has always length ≥ 20 (e.g., considering level 1 and the l2l connecting the closets output nodes of adjacent gadgets). Let $d_1, \ldots, d_{20}$ be the first 20 nodes of the l2l left out. These twenty nodes cannot be crossed by S. As a matter of fact, if one of the 20 nodes are reached by S, there is no way to go back to ζ without repeatedly visiting the same nodes and have a contradiction with δ of the solution σ. On the other hand, inside the loop S visits both the points a and c adjacent to the entering point b of the analyzed l2l. With respect to a path S of the form dealt with in point 1, S visits one additional point in the loop but looses twenty points outside the loop (Fig. A.14 - right). This happens for every loop and for every l2l excluded by S. Thus, more than $n = m-6L-R-4$ points of G' are left out, and these points can not be retrieved by other points on the cubic lattice but the points on G'. This is a contradiction with the number t of variables in $\vec{V}$ defined on n.

The proposed reduction is polynomial time (and implementable using just for-loops bounded by the size of G and if/then/else, hence is logspace) thus the consistency problem of the compatible JM-constraint is NP-complete. □

B

A Distributed MCMC Framework for Solving Distributed Constraint Optimization Problems with GPUs

B.1 Introduction

This document is organized as follow: we first give some background on Markov Chains and define the objective of Markov Chain Monte Carlo (MCMC) algorithms to our purpose. We then introduce general properties that need to be satisfied by Markov chains to guarantee convergence to a stationary distribution. We provide a mapping from a *Maximum a Posteriori* (MAP) estimation problem to a *Distributed Constraint Optimization Problem* (DCOP) using general assumptions from a broad class of MCMC algorithms. Finally, we introduce theoretical properties to relate DCOP solution quality to MCMC sampling strategies, and provide a complexity analysis for the DMCMC requirements. Through this document we adopt the same notation as the one used in the main paper.

Suppose we have a joint probability distribution $\pi(\mathbf{z})$ over n variables, with $\mathbf{z} = z_1, \dots, z_n$, and $z_i \in \mathbb{R}$, which we are interested to approximate. Sampling algorithms are often used to examine posterior distributions as they provide ways of generating samples with the property that the empirical distribution of the samples approximate the posterior distribution π. It is not often the case that one can sample directly from the posterior distribution obtaining an i.i.d. sample from π. When sampling directly from the posterior distribution is difficult, due to the high dimensionality or because computing the posterior may be computationally intense, one can use a proposal distribution q which approximates the posterior π up to some normalizing constant, and performs a dependent sample, such as the sample path of a Markov chain. MCMC algorithms generate a sample path from a Markov chain that has π as its stationary distribution

B.2 Background: Finite Markov Chains

Let $\mathbf{Z} = (\mathbf{z}^0, \mathbf{z}^1, \dots, \mathbf{z}^t, \dots)$, with $\mathbf{z}^t \in \mathbf{D} \subseteq \mathbb{R}$ be a *Markov chain* with finite state space $\mathbf{S} = \{s_1, s_2, \dots, s_L\}$ and a $L \times L$ *transition matrix* T whose entries are all non-negative and such that for each state $s_i \in \mathbf{S}$, $\sum_{s_j \in \mathbf{S}} T_{ij} = 1$, which defines the probability of transiting from one state to another as

$$P(\mathbf{z}^{t+1} = s_j \mid \mathbf{z}^t = s_i) = T_{ij}.$$

We denote with T^m the probability of moving from a state $\mathbf{z^0}$ to a state $\mathbf{z}^m$ in m time steps.

We now introduce the structural properties required for a Markov Chain to guarantee convergence to the posterior distribution π.

Definition B.2.1 (Irreducibility) *A Markov chain is said to be* irreducible *if it is possible to reach any state to any other using only transitions of positive probability. Formally,*

$$\forall s_i, s_j \in \mathbf{S}, \exists m < \infty \,.\, P(\mathbf{z}^{t+m} = s_j \mid \mathbf{z}^t = s_i)$$

for a given instance t.

Definition B.2.2 (Periodicy) *A state $s_i \in \mathbf{S}$ has a period k if any return of the chain in it is possible with multiple of k time steps. The period of a state is defined as*

$$k = \boldsymbol{gcd}\{t : P(\mathbf{z}^t = s_i \mid \mathbf{z}^0 = s_i) > 0\}$$

where gcd is the greatest common divisor. A state is said to be aperiodic if $k = 1$, that is, visits of the Markov chain to such state can occur at irregular times: $P(\mathbf{z}^t = s_i \mid \mathbf{z}^0 = s_i) > 0$. A Markov chain is said to be aperiodic *if every state in* $\mathbf{S}$ *is aperiodic.*

Note that for an irreducible Markov chain, if at least one state is aperiodic, then the whole Markov chain is aperiodic.

Definition B.2.3 (Reaching Time) *The* reaching time τ_s *of a state $s \in \mathbf{S}$ is the first (positive) time at which a chain visits that state. Formally,*

$$\tau_s := \min\{t \geq 1 \mid \mathbf{z}^t = s\}$$

Property 5 *For any states s_i and s_j of an irreducible Markov chain, the expected first return time for a state s_j from a state s_i occurs in a finite amount of steps, that is*

$$\mathbf{E}_{s_j}(\tau_{s_i}) < \infty.$$

Property 6 *Given a Markov chain defined in a finite state space $\mathbf{S}$, with transition matrix P, and for a given initial state of the chain $\mathbf{z}^0 = s_0$, if P is irreducible and aperiodic, then*

$$\exists t < \infty, \forall m \geq t : s_0 \, T^m = \pi$$

and π is unique. Moreover, for all $s \in \mathbf{S}$, $\pi(s) > 0$ and

$$\pi(s) = \frac{1}{\mathbf{E}_{\mathbf{s}}(\tau_s)}.$$

That is, given enough time steps the chain converges to a unique stationary distribution π.

B.2.1 MAP to DCOP Mapping

MCMC algorithms can be used to solve the *Maximum a Posteriori* (MAP) probability estimation problem—a mode of the posterior distribution—once the probability distribution has converged to its stationary point. Recently, [123] has shown that DCOPs can be mapped to MAP estimation problems. Thus, MCMC algorithms can be used to solve DCOPs as well. In order for this document to be self contained, we show how to extend this mapping to the general case of multivariable DCOP functions.

Consider a MAP problem on a *Markov Random Field* (MRF). An MRF is a set of random variables having the *Markov property*—the conditional probability distribution of future states of the process do not depends on other states other than the current one—and it can be described by an undirected graph (V, E). Formally an MRF is defined by

- a set of random variables $\mathbf{z} = \{z_i \mid \forall i \in V\}$, where each random variable z_i is defined over a finite domain D_i. Each random variable z_i is visualized through a node $i \in V$.
- A set of potential functions $\theta = \{\theta_i(z_k) \mid z_k \in C_i\}$, where C_i refers to a set of nodes of V denoting a clique which includes node i.

Let the joint probability distribution $\pi(z_k = d_k : z_k \in C_i)$ be defined as $\exp[\theta_i(z_k \mid z_k = d_k \in C_i)]$. For ease of presentation we denote as $\pi(z_k : z_k \in C_i)$ the joint probability of the random variables $z_k \in C_i$ and mean $\pi(z_k = d_k : z_k \in C_i)$.

A full-joint distribution of $\mathbf{z}$ has the probability:

$$\pi(\mathbf{z}) = \frac{1}{Z} \prod_{C_i \in \mathbf{C}} \exp\left[\theta_i(z_k : z_k \in C_i)\right] \tag{B.1}$$

$$= \frac{1}{Z} \exp\left[\sum_{C_i \in \mathbf{C}} \theta_i(z_k : z_k \in C_i)\right] \tag{B.2}$$

where $\mathbf{C}$ is the set of all cliques in (V, E) and Z is the normalizing constant for the density. The objective of a MAP estimation problem is to find the mode of $\pi(\mathbf{z})$, which is equivalent to find a complete assignment $\mathbf{z}$ that maximizes the function:

$$F(\mathbf{z}) = \sum_{C_i \in \mathbf{C}} \theta_i(z_k : z_k \in C_i)$$

which is also objective of a DCOP, where each potential function θ_i correspond to a utility function f_i and the associated clique C_i to the scope of the function f_i.

Therefore, if T is an MCMC sampling method that constructs a Markov chain with stationary distribution π to solve the associated MAP estimation problem, then, we can use the complete solution $\mathbf{z}$ returned to solve the corresponding DCOP.

Notice that sufficient conditions for T to converge to π are irreducibility and aperiodicity. The *Gibbs* and *Metropolis-Hastings* sampling algorithms exhibit extremely weak sufficient conditions to guarantee convergence [17]. Namely, the Gibbs proposal distribution needs to ensure lower semi-contiguity at 0 and be locally bounded, while for the Metropolis Hasting, it is sufficient that the domain of the definition of the proposal distribution q coincide with that of π.

B.3 Theoretical Properties

Below, we provide bounds on convergence rates for DMCMC algorithms based on MCMC sampling. Throughout this section, we assume that the Markov chain $(\mathbf{z}^0, \mathbf{z}^1, \ldots)$ under discussion has finite state space $\mathbf{S}$, a transition matrix T that is irreducible and aperiodic, and has a stationary distribution the posterior π.

Property 7 *The expected number of samples $\tau_{\mathbf{z}^*}$ for a MCMC algorithm to get an optimal solution $\mathbf{z}^*$ is*

$$E_{\mathbf{z}^*}(\tau_{\mathbf{z}^*}) = \frac{1}{\pi(\mathbf{z}^*)}$$

This property is a direct consequence of Property 6.

Theorem B.3.1 *The expected number of samples to find an optimal solution $\mathbf{z}^*$ with an MCMC sampling algorithm T is no greater than with a uniform sampling algorithm. In other words,*

$$P_T(\mathbf{z}^*) \geq P_{\text{uni}}(\mathbf{z}^*)$$

Theorem B.3.1 is introduced by [123] and can be generalized to any MCMC sampling algorithm that is irreducible and aperiodic as convergence is guaranteed in a finite number of time steps.

Definition B.3.2 (Top α_i-Percentile Solutions) *For an agent a_i the* top α_i-percentile solutions *S_{α_i} is a set containing solutions for the local variables L_i that are no worse than any solution in the supplementary set $D_i \setminus S_{\alpha_i}$, and $\frac{|S_{\alpha_i}|}{|D_i|} = \alpha_i$. Given a list of agents $a_1, \ldots, a_m$, the top $\bar{\alpha}$-percentile solutions $S_{\bar{\alpha}}$ is defined as $S_{\bar{\alpha}} = S_{\alpha_1} \times \ldots \times S_{\alpha_m}$.*

Property 8 *After $N_i = \frac{1}{\alpha_i \epsilon_i}$ number of samples with an MCMC sampling algorithm T, the probability that the best solution found thus far $\mathbf{z}_{N_i}$ is in the top α_i for an agent a_i is at least $1 - \epsilon_i$:*

$$P_{\mathrm{T}}\left(\mathbf{z}_{N_i} \in S_{\alpha_i} \mid N_i = \frac{1}{\alpha_i \cdot \epsilon_i}\right) \geq 1 - \epsilon_i.$$

This property is a direct extension of Theorem 2 introduced in [123].

Theorem B.3.3 *Given m agents $a_1, \ldots, a_m \in \mathcal{A}$, and a number of samples $N_i = \frac{1}{\alpha_i \cdot \epsilon_i}$ $(i = 1, \ldots, m)$, the probability that the best complete solution found thus far $\mathbf{z}_{\mathbf{N}}$ is in the top $\bar{\alpha}$-percentile is greater than or equal to $\prod_{i=1}^{m}(1 - \epsilon_i)$, where $\mathbf{N} = \bigwedge_{i=1}^{m} N_i$. In other words,*

$$P_{\mathrm{T}}\left(\mathbf{z}_{\mathbf{N}} \in S_{\bar{\alpha}} \mid \mathbf{N}\right) \geq \prod_{i=1}^{m}(1 - \epsilon_i).$$

Proof. Let $\mathbf{z}_N$ denote the best solution found so far in the process resolution and $\mathbf{z}_{N_i}$ denote the best partial assignment over the variables held by agent a_i found after N_i samples. Let $\mathbf{S}_i$ be a random variable describing wether $\mathbf{z}_{N_i} \in S_{\alpha_i}$. Thus:

$$\begin{aligned}
&P_{\mathrm{T}}(\mathbf{z}_{\mathbf{N}} \in S_{\bar{\alpha}} \mid \mathbf{N}) && \text{(B.3a)}\\
&\quad = P_{\mathrm{T}}(\mathbf{z}_{\mathbf{N}} \in S_{\bar{\alpha}} \mid \mathbf{N}_1, \ldots, \mathbf{N}_m) && \text{(B.3b)}\\
&\quad = P_{\mathrm{T}}(\mathbf{z}_{\mathbf{N}} \in S_{\alpha_1} \times \ldots \times S_{\alpha_m} \mid \mathbf{N}_1, \ldots, \mathbf{N}_m) && \text{(B.3c)}\\
&\quad = P_{\mathrm{T}}(\mathbf{S}_1, \ldots, \mathbf{S}_m \mid \mathbf{B}_1, \ldots, \mathbf{B}_m, \mathbf{N}_1, \ldots, \mathbf{N}_m) && \text{(B.3d)}
\end{aligned}$$

where each $\mathbf{B}_i$ $(i{=}1, \ldots, m)$ is a random variable describing a particular value assignment associated to the boundary variables B_i for the agent a_i. They are introduced to relate each of the $\mathbf{z}_{\mathbf{N_i}}$ to each other, which are sampled independently.

Since the values sampled in the local variable of a_i are dependent only of the values of the boundary values B_i, it follows that $\mathbf{S}_i$ is conditionally dependent of $\mathbf{B}_i$ but conditionally independent of all other $\mathbf{B}_j$, with $j \neq i$:

$$S_i \perp\!\!\!\perp B_j \mid B_i$$

for all $j = 1 \ldots m$ and $j \neq i$. Noticing that, given random variables a, b, c, whenever $a \perp\!\!\!\perp b \mid c$ we can write: $P(a \mid b, c) = P(a \mid c)$, and that $P(a, b \mid c) = P(a \mid b, c)$, it follows that Equation (B.3d) can be rewritten as:

$$\begin{aligned}
&P_{\mathrm{T}}(\mathbf{S}_1 \mid \mathbf{B}_1, \mathbf{N}_1) \cdot \ldots \cdot P_{\mathrm{T}}(\mathbf{S}_m \mid \mathbf{B}_m, \mathbf{N}_m)\\
&= P_{\mathrm{T}}(\mathbf{z}_{\mathbf{N_1}} \in S_{\alpha_1} \mid \mathbf{B}, \mathbf{N}) \cdot \ldots \cdot P_{\mathrm{T}}(\mathbf{z}_{\mathbf{N_m}} \in S_{\alpha_m} \mid \mathbf{B}, \mathbf{N}) && \text{(B.4a)}\\
&\geq (1 - \epsilon_1) \cdot \ldots \cdot (1 - \epsilon_m) && \text{(B.4b)}\\
&= \prod_{i=1}^{m}(1 - \epsilon_i). && \text{(B.4c)}
\end{aligned}$$

for any of the assignments of the variables in B_i, as the utility functions involving variables in the boundary of any two agents are solved optimally. □

Property 9 (Number of Messages) *The number of messages that DMCMC needs is linear in the size of the agents. There are $|\mathcal{A}| - 1$ UTIL messages (one through each tree-edge) and $|\mathcal{A}| - 1$ VALUE messages. The DFS construction, like in DPOP, also produces a linear number of messages (usually it requires $2|\mathcal{A}|$ messages). Thus, the total number of messages required is $O(|\mathcal{A}|)$.*

Note that, unlike DPOP, no message exchange is required to solve the constraints defined over the scope of the local variables each agent, which is achieved via local sampling.

Property 10 (Space Requirements) *Each agent $a_i \in \mathcal{A}$ needs to store its own utilities and the corresponding solution (value assignment for all non-boundary local variables $x_i^j \in L_i \setminus B_i$) for each combination of values of the boundary variables $x_i^k \in B_i$, thus requiring $O(|D_i|^{|B_i|})$ space. Moreover during the UTIL propagation phase, each agent a_i stores the UTIL messages of each of its children $a_c \in C_i$, which also sends messages of size $O(|D_i|^{|B_c|})$. Joint and projection operations can be performed efficiently within $O(|D_i|^{N_{S_i}-|B_i|})$ space, where N_{S_i} is the number of variables in the separator set of a_i which is involved in a constraint with some variable in B_i. Thus the memory complexity of each agent is exponential in the* induced width*—the maximum number of boundary variables of the parent of an agent involved in a constraint with the boundary variable of the agent itself.*

Exponential size messages do not represent necessary a limitation. One can bound the maximum message size and serialize big messages by letting the back-edge handlers ask explicitly for solutions and utilities for a subset one of their values sequentially. Moreover, one could reduce the exponential memory requirement at cost of sacrificing completeness, and propagating solutions for a bounded set of value combinations instead of all combination of values of the boundary variables.

150B. A Distributed MCMC Framework for Solving Distributed Constraint Optimization Problems with GPUs

Bibliography

[1] E. Aarts and Jan K. Lenstra, editors. *Local Search in Combinatorial Optimization.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1997.

[2] I. Al-Bluwi, T. Simeon, and J. Cortes. Motion Planning Algorithms for Molecular Simulations: A Survey. *Computer Science Review*, 6(4):125–143, 2012.

[3] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell.* Garland Science, 2007.

[4] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[5] C. B. Anfinsen. Principles that Govern the Folding of Protein Chains. *Science*, 181:223–230, 1973.

[6] K. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2009.

[7] Alejandro Arbelaez and Philippe Codognet. A GPU implementation of parallel constraint-based local search. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 648–655, 2014.

[8] R. Backofen and S. Will. A Constraint-Based Approach to Fast and Exact Structure Prediction in 3-Dimensional Protein Models. *Constraints*, 11(1):5–30, 2006.

[9] R. Backofen, S. Will, and Erich Bornberg-Bauer. Application of Constraint Programming Techniques for Structure Prediction of Lattice Proteins with Extended Alphabet. *Bioinformatics*, 15(3):234–242, 1999.

[10] D. Baker and A. Sali. Protein Structure Prediction and Structual Genomics. *Science*, 294:93–96, 2001.

[11] P. Barahona and L. Krippahl. Constraint Programming in Structural Bioinformatics. *Constraints*, 13(1-2):3–20, 2008.

[12] R. Bartak. *On-line Guide to Constraint Programming.* `http://kti.mff.cuni.cz/~bartak/constraints/`, 1998.

[13] M. Ben-David, O. Noivirt-Brik, A. Paz, J. Prilusky, J. L. Sussman, and Y. Levy. Assessment of CASP8 Structure Predictions for Template Free Targets. *Proteins*, 77:50–65, 2009.

[14] W. Bennett and R. Huber. Structural and Functional Aspects of Domain Motions in Proteins. *Crit. Rev. Biochem.*, 15:291–384, 1984.

[15] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The Protein Data Bank. *Nucleic Acids Research*, 28:235–242, 2000. `http://www.rcsb.org/pdb/`.

[16] 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.

[17] Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer-Verlag New York, Inc., 2006.

[18] S. Cahill, M. Cahill, and K. Cahill. On the Kinematics of Protein Folding. *Journal of Computational Chemistry*, 24(11):1364–1370, 2003.

[19] F. Campeotto, A. Dal Palù, A. Dovier, F. Fioretto, and Pontelli E. A constraint solver for flexible protein model. *J. Artif. Intell. Res. (JAIR)*, 48:953–1000, 2013.

[20] F. Campeotto, A. Dal Palù, A. Dovier, F. Fioretto, and E. Pontelli. A Filtering Technique for Fragment Assembly-Based Proteins Loop Modeling with Constraints. In Michela Milano, editor, *CP*, volume 7514 of *Lecture Notes in Computer Science*, pages 850–866. Springer, 2012.

[21] A.A. Canutescu and R.L. Dunbrack. Cyclic coordinate descent: a robotics algorithm for protein loop closure. *Protein Sci*, 12:963–972, 2003.

[22] A. Caprara, M. Fischetti, P. Toth, D. Vigo, and P. Guida. Algorithms for railway crew management. *Mathematical Programming*, 79(1-3):125–141, 1997.

[23] A. M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, and M. G. Wallace. ECLiPSe: An Introduction. Technical Report IC-Parc 03–1, IC-Parc, Imperial College London, 2003.

[24] G. Chelvanayagam, L. Knecht, T. Jenny, S.A. Benner, and G.H. Gonnet. A Combinatorial Distance-Constraint Approach to Predicting Protein Tertiary Models from Known Secondary Structure. *Folding and Design*, 3:149–160, 1998.

[25] Choco Team. Choco: an Open Source Java Constraint Programming Library. In *Workshop on Open-Source Software for Integer and Constraint Programming*, 2008. Available from `http://www.emn.fr/z-info/choco-solver/`.

[26] Y. Choi and C. M. Deane. FREAD Revisited: Accurate Loop Structure Prediction Using a Database Search Algorithm. *Proteins*, 78(6):1431–40, May 2010.

[27] C. R. Christian and K. Kuchcinski. Parallel consistency in constraint programming. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2009, Las Vegas, Nevada, USA, July 13-17, 2009, 2 Volumes*, pages 638–644, 2009.

[28] C. Clementi. Coarse-grained Models of Protein Folding: Toy Models or Predictive Tools? *Curr Opin Struct Biol*, 18:10–15, 2008.

[29] W.F. Clocksin and H. Alshawi. A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors. *New Generation Computing*, 5:361–376, 1988.

[30] C. Cole, J. D. Barber, and G. J. Barton. The Jpred 3 secondary structure prediction server. *Nucleic Acids Research*, 36(Web-Server-Issue):197–201, 2008.

[31] F. Corblin, L. Trilling, and E. Fanchon. Constraint Logic Programming for Modeling a Biological System Described by a Logical Network. In *Workshop on Constraint-Based Methods for Bioinformatics*, 2005.

[32] J. Cortes and I. Al-Bluwi. A Robotics Apporach to Enhance Conformational Sampling of Proteins. In *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, volume 4, pages 1177–1186. ASME, 2012.

[33] P. Crescenzi, D. Goldman, C. Papadimitriou, A. Piccolboni, and M. Yannakakis. On the Complexity of Protein Folding. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing*, pages 597–603. ACM Press, 1998.

[34] N. Cross. *The Automated Architect.* Pion Limited, 1977.

[35] A Dal Palù, A. Dovier, and F. Fogolari. Constraint Logic Programming Approach to Protein Structure Prediction. *BMC Bioinformatics*, 5(186), 2004.

[36] A. Dal Palù, A. Dovier, F. Fogolari, and E. Pontelli. Exploring protein fragment assembly using CLP. In Toby Walsh, editor, *IJCAI*, pages 2590–2595. IJCAI/AAAI, 2011.

[37] A. Dal Palù, A. Dovier, F. Fogolari, and E. Pontelli. Protein Structure Analysis with Constraint Programming. In *Computational Approaches to Nuclear Receptors*, chapter 3, pages 40–59. The Royal Society of Chemistry, 2012.

[38] A. Dal Palù, A. Dovier, and E. Pontelli. A New Constraint Solver for 3D Lattices and Its Application to the Protein Folding Problem. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 48–63. Springer Verlag, 2005.

[39] A. Dal Palù, A. Dovier, and E. Pontelli. A Constraint Solver for Discrete Lattices, its Parallelization, and Application to Protein Structure Prediction. *Software Practice and Experience*, 37(13):1405–1449, 2007.

[40] A. Dal Palù, A. Dovier, and E. Pontelli. Computing Approximate Solutions of the Protein Structure Determination Problem using Global Constraints on Discrete Crystal Lattices. *International Journal of Data Mining and Bioinformatics*, 4(1):1–20, 2010.

[41] A. Dal Palù, F. Spyrakis, and P. Cozzini. A New Approach for Investigating Protein Flexibility Based on Constraint Logic Programming: The First Application in the Case of the Estrogen Receptor. *European Journal of Medicinal Chemistry*, 49:127–140, 2012.

[42] Alessandro Dal Palù, Agostino Dovier, Federico Fogolari, and Enrico Pontelli. CLP-based Protein Fragment Assembly. *Theory and Practice of Logic Programming*, 10(4-6):709–724, 2010.

[43] C.M. Deane and T.L. Blundell. CODA. A Combined Algorithm for Predicting the Structurally Variable Regions of Protein Models. *Protein Sci*, 10:599–612, 2001.

[44] J Debartolo, G. Hocky, M Wilde, J. Xu, K.F. Freed, and T.R. Sosnick. Protein Structure Prediction Enhanced with Evolutionary Diversity: SPEED. *Protein Science*, 19(3):520–534, 2010.

[45] Rina Dechter. *Constraint Processing.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[46] M. S. Delgado and C. F. Parmeter. Embarrassingly easy embarrassingly parallel processing in r: Implementation and reproducibility. Working Papers 2013-06, University of Miami, Department of Economics, 2013.

[47] I. Dotú, M. Cebrián, P. Van Hentenryck, and P. Clote. On Lattice Protein Structure Prediction Revisited. *IEEE/ACM Trans. Comput. Biology Bioinform*, 8(6):1620–1632, 2011.

[48] R.L. Dunbrack. Rotamer Libraries in the 21st Century. *Curr. Opin. Struct. Biol.*, 12(4):431–440, 2002.

[49] E. Erdem. Applications of Answer Set Programming in Phylogenetic Systematics. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, pages 415–431. Springer Verlag, 2011.

[50] E. Erdem and F. Ture. Efficient Haplotype Inference with Answer Set Programming. In *National Conference on Artificial Intelligence (AAAI)*, pages 436–441. AAAI/MIT Press, 2008.

[51] A.K. Felts, E. Gallicchio, D. Chekmarev, K.A. Paris, R.A. Friesner, and R.M. Levy. Prediction of Protein Loop Conformations using AGBNP Implicit Solvent Model and Torsion Angle Sampling. *J Chem Theory Comput*, 4:855–868, 2008.

[52] A. Fiser. Template-based protein structure modeling. In *Computational Biology*, pages 73–94. Springer, 2010.

[53] A. Fiser, R.K.G. Do, and A. Sali. Modeling of Loops in Protein Structures. *Protein Sci*, 9:1753–1773, 2000.

[54] F Fogolari, A. Corazza, P. Viglino, and G. Esposito. Fast Structure Similarity Searches among Protein Models: Efficient Clustering of Protein Fragments. *Algorithms for Molecular Biology*, 7:16, 2012.

[55] F. Fogolari, G. Esposito, P. Viglino, and S. Cattarinussi. Modeling of Polypeptide Chains as C_α Chains, C_α Chains with C_β, and C_α Chains with Ellipsoidal Lateral Chains. *Biophysical Journal*, 70:1183–1197, 1996.

[56] F. Fogolari, L. Pieri, A. Dovier, L. Bortolussi, G. Giugliarelli, A. Corazza, G. Esposito, and P. Viglino. Scoring Predictive Models using a Reduced Representation of Proteins: Model and Energy Definition. *BMC Structural Biology*, 7(15):1–17, 2007.

[57] Y. Fujitsuka, G. Chikenji, and S. Takada. SimFold Energy Function for De Novo Protein Structure Prediction: Consensus with Rosetta. *Proteins*, 62:381–398, 2006.

[58] M.J. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP Languages. *ACM Transactions on Programming Languages and Systems*, 22(2):269–339, March 2000.

[59] J.G. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1979. AAI7925014.

[60] S. Gay, F. Fages, T. Martinez, and S. Soliman. A Constraint Program for Subgraph Epimorphisms with Application to Identifying Model Reductions in Systems Biology. In *Workshop on Constraint-Based Methods for Bioinformatics*, 2011.

[61] M. Gebser, T. Schaub, S. Thiele, and P. Veber. Detecting Inconsistencies in Large Biological Networks with Answer Set Programming. *Theory and Practice of Logic Programming*, 11(2-3):323–360, 2011.

[62] Gecode Team. Gecode: Generic Constraint Development Environment. Available from `http://www.gecode.org`, 2013.

[63] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.

[64] I. Gent, C. Jefferson, I. Lynce, I Miguel, P. Nightingale, B. Smith, and A. Tarim. Search in the patience game "black hole". *AI Communications*, 20:211–226, 2007.

[65] Fred Glover. Tabu search and adaptive memory programing advances, applications and challenges. In *Interfaces in Computer Science and Operations Research*, pages 1–75. Kluwer, 1996.

[66] N. Go and H.A. Scheraga. Ring Closure and Local Conformational Deformations of Chain Molecules. *Macromolecules*, 3:178–187, 1970.

[67] A. Graca, J. Marques-Silva, I. Lynce, and A. Oliveira. Haplotype Inference with Pseudo-Boolean Optimization. *Annals of OR*, 184(1):137–162, 2011.

[68] T. Guns, H. Sun, K. Marchal, and S. Nijssen. Cis-regulatory Module Detection Using Constraint Programming. In *IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 363–368, 2010.

[69] Saurabh Gupta, Palak Jain, William Yeoh, Satish Ranade, and Enrico Pontelli. Solving customer-driven microgrid optimization problems as DCOPs. In *Proceedings of the Distributed Constraint Reasoning Workshop*, pages 45–59, 2013.

[70] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[71] Y. Hamadi. Optimal Distributed Arc Consistency. *Constraints*, 7(3-4), 2002.

[72] Youssef Hamadi, Christian Bessière, and Joël Quinqueton. Distributed intelligent backtracking. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 219–223, 1998.

[73] J. Handl, J. Knowles, R. Vernon, D. Baker, and S.C. Lovell. The Dual Role of Fragments in Fragment-Assembly Methods for De Novo Protein Structure Prediction. *Proteins: Structure, Function and Bioinformatics*, 80(2):490–504, 2012.

[74] M.H. Hao and H.A. Scheraga. Designing potential energy functions for protein folding. *Curr. Opin. Struct. Biol.*, 9:184–188, 1999.

[75] R.S. Hartenberg and J. Denavit. A Kinematic Notation for Lower Pair Mechanisms Based on Matrices. *Journal of Applied Mechanics*, 77:215–221, 1995.

[76] J.A. Hegler, J. Lätzer, A. Shehu, C. Clementi, and P.G." Wolynes. Restriction Versus Guidance in Protein Structure Prediction. *Proc Natl Acad Sci U.S.A.*, 106(36):15302–15307, 2009.

[77] Jim Held, Jerry Bautista, and Sean Koehl. White paper from a few cores to many: A tera-scale computing research review, 2006.

[78] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search.* The MIT Press, 2009.

[79] M. D. Hill and A. J. Smith. Experimental Evaluation of On-Chip Microprocessor Cache Memories. In *11th Annual International Symposium on Computer Architecture*, pages 158–166. IEEE Computer Society, 1984.

[80] ILOG. *ILOG Solver*, 4.0 edition, 1997.

[81] M.P. Jacobson, D.L. Pincus, C.S. Rapp, T.J.F. Day, B. Honig, D.E. Shaw, and R.A. Friesner. A Hierarchical Approach to All-atom Protein Loop Prediction. *Proteins*, 55:351–367, 2004.

[82] JaCoP Team. JaCoP web page, visited November 2012, 2012. Available from `http://www.jacop.eu`.

[83] M. Jamroz and A. Kolinski. Modeling of Loops in Proteins: a Multi-method Approach. *BMC Struct. Biol.*, 10(5), 2010.

[84] R. Jauch, H.C. Yeo, P. R. Kolatkar, and N. D. Clarke. Assessment of CASP7 Structure Predictions for Template Free Targets. *Proteins*, 69:57–67, 2007.

[85] John Jenkins, Isha Arkatkar, John D. Owens, Alok Choudhary, and Nagiza F. Samatova. Lessons learned from exploring the backtracking paradigm on the gpu. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 425–437, Berlin, Heidelberg, 2011. Springer-Verlag.

[86] D. Jones. Predicting Novel Protein Folds by using FRAGFOLD. *Proteins*, 45:127–132, 2006.

[87] K. Karplus, R. Karchin, J. Draper, J. Casper, Y. Mandel-Gutfreund, M. Diekhans, and R. Hughey Source. Combining local structure, fold-recognition, and new fold methods for protein structure prediction. *Proteins*, 53(6):491–497, 2003.

[88] M. Karplus and E. Shakhnovich. Protein Folding: Theoretical Studies of Thermodynamics and Dynamics. In *Protein Folding*, pages 127–195. WH Freeman, 1992.

[89] David E. Kim, Ben Blum, Philip Bradley, and David Baker. Sampling Bottlenecks in De novo Protein Structure Prediction. *Journal of Molecular Biology*, 393(1):249 – 260, 2009.

[90] L. Kinch, S. Yong Shi, Q. Cong, H. Cheng, Y. Liao, and N. V. Grishin. CASP9 assessment of free modeling target predictions. *Proteins*, 79:59–73, 2011.

[91] S. Kirillova, J. Cortes, A. Stefaniu, and T. Simeon. An NMA-Guided Path Planning Approach for Computing Large-Amplitude Conformational Changes in Proteins. *Proteins: Structure, Function, and Bioinformatics*, 70(1):131–143, 2008.

[92] R. Kolodny, L. Guibas, M. Levitt, and P. Koehl. Inverse Kinematics in Biology: The Protein Loop Closure Problem. *The International Journal of Robotics Research*, 24(2-3):151–163, 2005.

[93] L. Krippahl and P. Barahona. Psico: Solving Protein Structures with Constraint Programming and Optimization. *Constraints*, 7(4-3):317–331, 2002.

[94] L. Krippahl and P. Barahona. Applying Constraint Programming to Rigid Body Protein Docking. In *Principles and Practice of Constraint Programming*, pages 373–387. Springer Verlag, 2005.

[95] Ludwig Krippahl and Pedro Barahona. Applying Constraint Programming to Protein Structure Determination. In *Principles and Practice of Constraint Programming*, pages 289–302. Springer Verlag, 1999.

[96] A. Larhlimi and A. Bockmayr. A New Constraint-Based Description of the Steady-State Flux Cone of Metabolic Networks. *Discrete Applied Mathematics*, 157(10):2257–2266, 2009.

[97] S.M. LaValle. *Planning Algorithms.* Cambridge University Press, 2006.

[98] T. Lazaridis, G. Archontis, and M. Karplus. Enthalpic Contribution to Protein Stability: Atom-Based Calculations and Statistical Mechanics. *Adv. Protein Chem.*, 47:231–306, 1995.

[99] T. Léauté, B. Ottens, and R. Szymanek. FRODO 2.0: An open-source framework for distributed constraint optimization. In *Proceedings of the Distributed Constraint Reasoning Workshop*, pages 160–164, 2009.

[100] Christophe Lecoutre. Optimization of simple tabular reduction for table constraints. In *In Proceedings of CP08*, pages 128–143, 2008.

[101] J. Lee, S.Y. Kim, K. Joo, I. Kim, and J. Lee. Prediction of Protein Tertiary Structure using Profesy, a Novel Method Based on Fragment Assembly and Conformational Space Annealing. *Proteins*, 56(4):704–714, 2004.

[102] J. Lee, D. Lee, H. Park, E.A. Coutsias, and C. Seok. Protein Loop Modeling by Using Fragment Assembly and Analytical Loop Closure. *Proteins*, 78(16):3428–3436, 2010.

[103] C. Likitvivatanavong, Y. Zhang, J. Bowen, and E. C. Freuder. Arc consistency in mac: A new perspective. In *In proceedings of CPAI'04 workshop held with CP'04*, pages 93–107. Springer-Verlag, 2004.

[104] P. Liu, F. Zhu, D.N. Rassokhin, and D.K. Agrafiotis. A Self-organizing Algorithm for Modeling Protein Loops. *PLOS Comput Biol*, 5(8), 2009.

[105] S. Lovell, I. Davis, W. Arendall, P. de Bakker, J. Word, M. Prisant, J. Richardson, and D. Richardson. Structure Validation by C_α Geometry: ϕ, ψ and C_β Deviation. *Proteins*, 50:437–450, 2003.

[106] L. Luo, M. Wong, and W. Hwu. An effective gpu implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[107] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Large Neighborhood Local Search Optimization on Graphics Processing Units. In *Workshop on Large-Scale Parallel Processing (LSPP) in Conjunction with the International Parallel & Distributed Processing Symposium (IPDPS)*, Atlanta, États-Unis, 2010.

[108] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[109] Alan K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artif. Intell.*, 25(1):65–74, January 1985.

[110] Rajiv Maheswaran, Jonathan Pearce, and Milind Tambe. Distributed algorithms for DCOP: A graphical game-based approach. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 432–439, 2004.

[111] Rajiv Maheswaran, Milind Tambe, Emma Bowring, Jonathan Pearce, and Pradeep Varakantham. Taking DCOP to the real world: Efficient complete solutions for distributed event scheduling. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 310–317, 2004.

[112] M. Mann and A. Dal Palù. Lattice Model Refinement of Protein Structures. In *Workshop on Constraint-Based Methods for Bioinformatics*, 2010.

[113] L. Michel, A. See, and P. Van Hentenryck. Distributed constraint-based local search, 2006.

[114] C. Micheletti, F. Seno, and A. Maritan. Recurrent oligomers in proteins: an optimal scheme reconciling accurate and concise backbone representations in automated folding and design studies. *proteins*, 40(4):662–674, 2000.

[115] Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161:149–180, 2006.

[116] M. Moll, D. Schwarz, and L.E. Kavraki. *Roadmap Methods for Protein Folding.* Humana Press, 2007.

[117] K. Molloy, S. Saleh, and A. Shehu. Probabilistic Search and Energy Guidance for Biased Decoy Sampling in Ab-Initio Protein Structure Prediction. *IEEE/ACM Trans. Comput. Biology Bioinform*, PrePrint, 2013.

[118] A.V. Morozov and T. Kortemme. Potential functions for hydrogen bonds in protein structure prediction and design. *Advances in Protein Chemistry*, 4(72):1–38, 2005.

[119] B. A. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer New York, 1988.

[120] N. Nethercote, P. J Stuckey, R. Becket, S. Brand, G. J Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *Principles and Practice of Constraint Programming–CP 2007*, pages 529–543. Springer, 2007.

[121] A. Neumaier. Molecular Modeling of Proteins and Mathematical Prediction of Protein Structure. *SIAM Review*, 39:407–460, 1997.

[122] D. T. Nguyen, W. Yeoh, and H. C. Lau. Stochastic dominance in stochastic dcops for risk-sensitive applications. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 257–264, 2012.

[123] Duc Thien Nguyen, William Yeoh, and Hoong Chuin Lau. Distributed Gibbs: A memory-bounded sampling-based DCOP algorithm. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 167–174, 2013.

[124] T. Nguyen and Y. Deville. A Distributed Arc Consistency Algorithm. *Science of Computer Programming*, 30(1-2), 1998.

[125] N.J. Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, Springer, 1982.

[126] K. Noonan, D. O'Brien, and J. Snoeyink. Protein Backbone Motion by Inverse Kinematics. *International Journal of Robotics Research*, 24(11):971–982, 2005.

[127] Brian S. Olson, Kevin Molloy, and Amarda Shehu. In Search of the Protein Native State with a Probabilistic Sampling Approach. *J. Bioinformatics and Computational Biology*, 9(3):383–398, 2011.

[128] OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.

[129] L. Perron. Search procedures and parallelism in constraint programming. In *In Proc. of CP-99*, pages 346–360. Springer Verlag, 1999.

[130] Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1413–1420, 2005.

[131] S. Raman, R. Vernon, J. Thompson, M. Tyka, R. Sadreyev, J. Pei, D. Kim, E. Kellogg, F. DiMaio, O. Lange, L. Kinch, W. Sheffler, B.H. Kim, R. Das, N. V. Grishin, and Baker D. Structure Prediction for CASP8 with All-atom Refinement using Rosetta. *Proteins*, 77(Suppl. 9):89–99, 2009.

[132] C. S. Rapp and R. A. Friesner. Prediction of Loop Geometries using a Generalized Born Model of Solvation Effects. *Proteins*, 35:173–183, 1999.

[133] O. Ray, T. Soh, and K. Inoue. Analyzing Pathways Using ASP-Based Approaches. In *Algebraic and Numeric Biology, 4th International Conference*, pages 167–183. Springer Verlag, 2010.

[134] C.C. Rolf and K. Krzysztof. Combining parallel search and parallel consistency in constraint programming. In *International Conference on Principles and Practice of Constraint Programming: TRICS workshop*, September 2010.

[135] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming.* Elsevier, 2006.

[136] A. Roy, A. Kucukural, and Y. Zhang. I-TASSER: a unified platform for automated protein structure and function prediction. *Nature Protocols*, 5:725–738, 2010.

[137] S.D. Rufino, L.E. Donate, L.H.J. Canard, and T.L. Blundell. Predicting the Conformational Class of Short and Medium Size Loops Connecting Regular Secondary Structures: Application to Comparative Modeling. *J. Mol. Biol.*, 267:352–367, 1997.

[138] D. Rykunov and A. Fiser. New Statistical Potential for Quality Assessment of Protein Models and a Survey of Energy Functions. *BMC Bioinformatics*, 11:128, 2010.

[139] A. Samal and T. Henderson. Parallel Consistent Labeling Algorithms. *International Journal of Parallel Programming*, 16(5):341–364, 1987.

[140] J. Sanders and E. Kandrot. *CUDA by Example. An Introduction to General-Purpose GPU Programming*. Addison Wesley, 2010.

[141] D. Saravanan, D. G. Nalaka, B. Savitri, and S. Heiko. Dihedral angle and secondary structure database of short amino acid fragments. *Bioinformation*, 1:78–80, 2006.

[142] C. Schulte. Parallel Search Made Simple. In N. Beldiceanu et al., editor, *Proceedings of Techniques for Implementing Constraint Programming Systems, Post-conference workshop of CP 2000*, TRA9/00, pages 41–57, University of Singapore, 2000.

[143] Christian Schulte. Programming constraint inference engines. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 519–533. Springer-Verlag, 1997.

[144] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer Berlin Heidelberg, 1998.

[145] A. Shehu. An Ab-Initio Tree-Based Exploration to Enhance Sampling of Low-Energy Protein Conformations. In *Proceedings of Robotics: Science and Systems V*, 2009.

[146] A. Shehu. *Conformational Search for the Protein Native State*, pages 431–452. John Wiley & Sons. Inc., 2010.

[147] A. Shehu and L.E. Kavraki. Modeling Structures and Motions of Loops in Protein Molecules. *Entropy*, 14:252–290, 2012.

[148] M. Shen and A. Sali. Statistical Potential for Assessment and Prediction of Protein Structures. *Protein Sci*, 15:2507–2524, 2006.

[149] E.S.C. Shih and M-J. Hwang. On the Use of Distance Constraints in Protein-Protein Docking Computations. *Proteins: Structure, Function, and Bioinformatics*, 80(1):194–205, 2011.

[150] A. Shmygelska and H.H. Hoos. An Ant Colony Optimisation Algorithm for the 2D and 3D Hydrophobic Polar Protein Folding Problem. *BMC Bioinformatics*, 6:30–52, 2005.

[151] A. Shmygelska and M. Levitt. Generalized Ensemble Methods for De Novo Structure Prediction. *Proceedings of the National Academy of Science (USA)*, 106(5):1415–1420, 2009.

[152] D. Simoncini, F. Berenger, R. Shrestha, and K.Y.J. Zhang. A Probabilistic Fragment-Based Protein Structure Prediction Algorithm. *PLOS One*, 7(7):e38799, 2012.

[153] K.T. Simons, R. Bonneau, I. Ruczinski, and D. Baker. Ab initio protein structure prediction of CASP III targets using ROSETTA. *Proteins*, 3:171–176, 1999.

[154] K.T. Simons, C. Kooperberg, E. Huang, and D. Baker. Assembly of Protein Tertiary Structures from Fragments with Similar Local Sequences using Simulated Annealing and Bayesian Scoring Functions. *J. Mol. Biol.*, 268:209–225, 1997.

[155] J. Skolnick, J. Fetrow, and A. Kolinski. Structural Genomics and its Importance for Gene Function Analysis. *Nat. Biotechnology*, 18(3):283–287, 2000.

[156] C.S. Soto, M. Fasnacht, J. Zhu, L. Forrest, and B. Honig. Loop Modeling: Sampling, Filtering, and Scoring. *Proteins: Structure, Function, and Bioinformatics*, 70:834–843, 2008.

[157] V.J. Spassov, P.K. Flook, and L. Yan. LOOPER: A Molecular Mechanics-based Algorithm for Protein Loop Prediction. *Protein Eng*, 21:91–100, 2008.

[158] P. F. Stadler. Correlation in landscapes of combinatorial optimization problems. *EPL (Europhysics Letters)*, 20(6):479, 1992.

[159] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[160] Thomas Sttzle. *Local search algorithms for combinatorial problems - analysis, improvements, and new applications.*, volume 220 of *DISKI*. Infix, 1999.

[161] Evan Sultanik, Robert Lass, and William Regli. DCOPolis: a framework for simulating and deploying distributed constraint reasoning algorithms. In *Proceedings of the Distributed Constraint Reasoning Workshop*, 2007.

[162] Swedish Institute for Computer Science. SICStus Prolog Home Page. `http://www.sics.se/sicstus/`, 2012.

[163] El-Ghazali Talbi. *Metaheuristics - From Design to Implementation.* Wiley, 2009.

[164] P. Thebault, S. de Givry, T. Schiex, and C. Gaspin. Combining Constraint Processing and Pattern Matching to Describe and Locate Structured Motifs in Genomic Sequences. In *Fifth Workshop on Modeling and Solving Problems with Constraints*, pages 53–60, 2005.

[165] Y.T. Tsai, Y.P. Huang, C.T. Yu, and C.L. Lu. MuSiC: A Tool for Multiple Sequence Alignment with Constraints. *Bioinformatics*, 20(14):2309–2311, 2004.

[166] Univ. des Saarlandes, Sweedish Institute of Computer Science, and Univ. Catholique de Louvain. The Mozart Programming System. `www.mozart-oz.org`.

[167] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[168] M. Wooldridge. *An Introduction to Multi Agent Systems.* John Wiley and Sons, 2002.

[169] Sitao Wu, Jeffrey Skolnick, and Yang Zhang. Ab initio modeling of small proteins by iterative tasser simulations. *BMC Biology*, 5(1):17, 2007.

[170] Z. Xiang, C.S. Soto, and B. Honig. Evaluating Conformal Energies: The Colony Energy and its Application to the Problem of Loop Prediction. *PNAS*, 99:7432–7437, 2002.

[171] D. Xu and Y. Zhang. Ab Initio Protein Structure Assembly Using Continuous Structure Fragments and Optimized Knowledge-based Force Field. *Proteins*, 80(7):1715–1735, 2012.

[172] R. Yang. Multiple Protein/DNA Sequence Alignment with Constraints. In *International Conference on Practical Applications of Constraint Programming*, 1998.

[173] R. Yap. Parametric Sequence Alignment with Constraints. *Constraints*, 6:157–172, 2001.

[174] R. Yap and H. Chuan. A Constraint Logic Programming Framework for Constructing DNA Restriction Maps. *Artificial Intelligence in Medicine*, 5(5):447–464, 1993.

[175] William Yeoh, Ariel Felner, and Sven Koenig. Bnb-adopt: An asynchronous branch-and-bound DCOP algorithm. *CoRR*, abs/1401.3490, 2014.

[176] William Yeoh and Makoto Yokoo. Distributed problem solving. *AI Magazine*, 33(3):53–65, 2012.

[177] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998.

[178] K. Yue and K.A. Dill. Constraint Based Assembly of Tertiary Protein Structures from Secondary Structure Elements. *Proteins Science*, 9(10):1935–1946, 2000.

[179] M. Zhang and L. Kavraki. A New Method for Fast and Accurate Derivation of Molecular Conformations. *Journal of Chemical Information and Computer Sciences*, 42(1):64–70, 2002.

[180] Y. Zhang and K. Hauser. Unbiased, Scalable Sampling of Protein Loop Conformations from Probabilistic Priors. *BMC Structural Biology*, (to appear `http://www.indiana.edu/~motion/slikmc/papers/BMC_Zhang.pdf`), 2013.

[181] Yang Zhang. I-TASSER server for protein 3D structure prediction. *BMC Bioinformatics*, 9(40), 2008.

[182] H. Zhou and Y. Zhou. Distance-scaled, Finite Ideal-gas Reference State Improves Structure-derived Potentials of Mean Force for Structure Selection and Stability Prediction. *Protein Sci*, 11:2714–2726, 2002.