POSL: A Parallel-Oriented Solver Language

Thesis for the Degree of
Doctor of Computer Science

Alejandro Reyes Amaro

Doctoral School STIM

Academic advisors:
Eric Monfroy[1], Florian Richoux[2]

[1]Department of Informatics
Faculty of Science
University of Nantes
France

[2]Department of Informatics
Faculty of Science
University of Nantes
France

Submitted: dd/mm/2016

Assessment committee:

Prof. (1)

Institution (1)

Prof. (2)

Institution (2)

Prof. (3)

Institution (3)

ISBN ??

POSL: A Parallel-Oriented Solver Language

Short abstract:
For a couple of years, all processors in modern machines are multi-core. Massively parallel architectures, so far reserved for super-computers, become now available to a broad public through hardware like the Xeon Phi or GPU cards. This architecture strategy has been commonly adopted by processor manufacturers, allowing them to stick with Moore's law. However, this new architecture implies new ways to design and implement algorithms to exploit its full potential. This is in particular true for constraint-based solvers dealing with combinatorial optimization problems. Here we propose a Parallel-Oriented Solver Language (POSL, pronounced "puzzle"), a new framework to build interconnected meta-heuristic based solvers working in parallel. The novelty of this approach lies in looking at solver as a set of components with specific goals, written in a parallel-oriented language based on operators. A major feature in POSL is the possibility to share not only information, but also behaviors, allowing solver modifications during runtime. Our framework has been designed to easily build constraint-based solvers and reduce the developing effort in the context of parallel architecture. POSL's main advantage is to allow solver designers to quickly test different heuristics and parallel communication strategies to solve combinatorial optimization problems, usually time-consuming and very complex technically, requiring a lot of engineering.

Keywords: Constraint programming, meta-heuristics, language.

Contents

Resumé:
La technologie multi-cœur et les architectures massivement parallèles sont de plus en plus accessibles à tous, à travers des matériaux comme le Xeon Phi ou les cartes GPU. Cette stratégie d'architecture a été communément adoptée par les producteurs pour faire face à la loi de Moore. Or, ces nouvelles architectures impliquent d'autres manières de concevoir et d'implémenter les algorithmes, pour exploiter complètement leur potentiel, en particulier dans le cas des solveurs de contraintes traitant de problèmes d'optimisation combinatoire. Dans cette travail nous presentons un Langage pour créer des Solveurs Orienté Parallèle (POSL pour Parallel-Oriented Solver Language, et prononcé "puzzle") : cadre permettant de construire des solveurs basés sur des méta-heuristiques interconnectées travaillant en parallèle, dans le but de résoudre des instances des CSP et de mesurer sa performance.

La nouveauté de cette approche porte sur le fait que l'on voit un solveur comme un ensemble de composants spécifiques, écris dans un langage orienté parallèle basé sur des opérateurs. Un avantage de POSL est la possibilité de partager non seulement des informations mais aussi des comportements, permettant ainsi de modifier à chaud les solveurs. POSL permets aux composants d'un solveur d'être transmis et exécutés par d'autres solveurs. Il propose également une couche supplémentaire permettant de définir dynamiquement des connexions entre solveurs.

Nous testons plusieurs stratégies de résolution, grâce au langage orienté parallèle, basé sur des opérateurs, que POSL fournis.

Preface

This thesis was submitted to the Faculty of Science, University of Nantes, as a requirement to obtain the PhD degree. The work presented was carried out in the years 2013-2016 in the laboratory LINA (*Laboratoire d'Informatique de Loire Atlantique*) as part of the Inria-TASC research team, under the supervision of Éric MONFROY and Florian Richoux, and with a French Ministry of Research's Grant. I've always enjoyed my time at Nantes where I spent about two years months. I additionally spent some days uslises

During this time I have participated in some schools schools and I have received some curses like.

Context

The evolution of computer architecture is leading us toward massively multi-core computers for tomorrow, composed of thousands of computing units. However nowadays, we do not know how to design algorithms able to manage efficiently such a computing power. In particular, this is true for combinatorial optimization algorithms, like algorithms solving constraint-based problems.

There exists several techniques for solving constraint-based problems: constraint programming, linear programming, boolean satisfaction methods, and local search methods to give an non-exhaustive list. The latter is often among the most efficient techniques to solve large size problems. Nowadays and up to our knowledge, there exists only one algorithm showing very good performances scaling to thousands of cores. However its parallel scheme does not include inter-processes cooperative communications. Moreover, the rising of more and more complex algorithms leads to an number of parameters which become intractable to manage by hand, and parallel algorithms emphasize this trend.

Thesis objectives

In this context, this Ph.D. topic has two major objectives :

a) Obj 1

Problem presentation

Optimization Problems are classical problems in Applied Mathematics and Computer Science. Their main goal is to find the best solution to a given mathematical model, which can have restrictions (constraints) or not. Variables composing the problem take their value from continuous or discrete domains. In the latter case, we are talking about of Combinatorial Optimization. Combinatorial Optimization has important applications in several fields, including machine learning, artificial intelligence, and software engineering. For example, some common problems involving combinatorial optimization are the traveling salesman problem (TSP) and the minimum spanning tree problem (MST). In some cases (problems), the main goal is only to find one solution, not the best, which is the case of the Constraint Satisfaction Problems (CSP), i.e., a solution will be a configuration of variables that complies with the constraints set. In other words: finding one feasible solution is enough.

The CSPs are one of the most trendiest in the Combinatorial Optimization field, and they find a lot of application in the industry. In theoretical terms, it means that new and more complex (also bigger) problems appear. For that reason, a lot of techniques and methods are applied to the resolution of these problems. Although many of these techniques, like meta-heuristics, have shown themselves to be effective, sometimes the real problems we want to solve are too large, i.e., the search space is huge, and in most cases too long execution time is needed to find a solution. However, the development of the super-computing has opened a new way to find solutions for these problems in a more feasible manner, reducing the search times. Therefore, in this thesis we will focus in finding new technologies and methods for the solution of CSPs through parallel computing, developing cooperative parallel algorithms and applying auto-tuning techniques to choose the proper parameters for them, which is a field not explored yet.

Thesis outline

In Part I,

Part I

Presentation

1

Introduction

The Introduction of the work is presented. We describe the target problem (the formal definition will be in the next chapter), and the approaches implemented so far to solve them. The necessity of a new approach to exploit the new era of parallelism is introduced. In this section are presented the goals of the thesis, and POSL *is introduced as a new parallel approach including others and novel features. Finally, we describe the structure of the document.*

Combinatorial Optimization has several applications in fields like machine learning, artificial intelligence, and software engineering. In some cases, the main goal is only to find a solution, like for *Constraint Satisfaction Problems (CSP)*. A solution will be an assignment of variables satisfying the constraint set. A CSP is defined by a triple $\langle X, D, C \rangle$ where $X = \{x_1, x_2, \ldots, x_n\}$ is a finite set of variables, $D = \{D_1, D_2, \ldots, D_n\}$ is the set of domains associated to each variable in X, and $C = \{c_1, c_2, \ldots, c_m\}$ is a set of constraints. Each constraint is defined over a set of variables, and specifies the possible combinations of values for these variables. A configuration $s \in D_1 \times D_2 \times \cdots \times D_n$ is a combination of values for the variables of X. We say that s is a solution of a *CSP* if and only if s satisfies all constraints $c_i \in C$.

There exist many different techniques to solve such problems, mainly classified into two categories: tree-search based algorithms exploring the full search space and meta-heuristics applying some stochastic moves. Although many of these techniques have been shown to be effective, sometimes the search space of problems we want to solve is huge, and in most cases it makes the problem intractable, in particular for tree-search based algorithms. However, the development of computer architecture is leading us toward massively multi/many–core computers. These architectures unlock new algorithmic possibilities to tackle problems sequential algorithms cannot handle.

As result of such developments, parallel algorithms have opened new ways to solve constraint problems. Adaptive Search [1] is an efficient method reaching linear speed-ups on hundreds and even thousands of cores (depending of the problem), using an independent multi-walk local search parallel scheme. Munera et al [2] present another implementation of Adaptive Search using communication between search engines, showing the efficiency of cooperative multi-walks strategies. *Meta–S* is an implementation of a theoretical framework proposed in [3], which allows to tackle problems, through the cooperation of arbitrary domain–specific constraint solvers. All these results use a multi-walk parallel approach and show the robustness and efficiency of this parallel scheme. Although, they all concluded there is room for improvements.

It is well-known software programming is a very time-consuming activity. This is even more the case while developing a parallel software, where debugging is an extremely difficult task. POSL's main goal is to propose to CSP solver designers/programmers a parallel framework to quickly build parallel prototypes, speeding-up the design process.

In this thesis we present POSL, a framework for easily building many and different cooperating solvers based on coupling four fundamental and independent components: *computation modules*, *communication modules*, the *abstract solvers* and *communication strategies*. Recently, the hybridization approach leads to very good results in constraint satisfaction. For

that reason, since the solver's component can be combined, our framework is designed to execute in parallel sets of different solvers, with and without communication.

POSL provides, through a simple operator-based language, a way to create *abstract solvers*, combining already defined *modules* (*computation modules* and *communication modules*). A similar idea was proposed in [4] without communication, introducing an evolutionary approach that uses a simple composition operator to automatically discover new local search heuristics for SAT and to visualize them as combinations of a set of building blocks. In the last phase of the coding process with POSL, solvers can be connected each others, depending on the structure of their *communication modules*, and this way, they can share not only information, but also their behavior, by sharing their *computation modules*. This approach makes the solvers able to evolve during the execution.

2

State of the art

This chapter presents an overview to the state of the art of Combinatorial Optimization Problems and different approaches to tackle them. We introduce in Section 2.1 the definition of a Combinatorial Optimization Problem and the its link with Constraint Satisfaction Problems (CSP), where we concentrate our main efforts, and we give some examples. The basic techniques used to solve these problems are introduced, like Constraint Propagation (2.2), meta– and hyper–heuristic methods (Sections 2.3 and 2.4). We also present some advanced techniques like hybridization in Section 2.5, parallel computing in Section 2.6, and Solvers cooperation in Section 2.7. Finally we present ıparameter setting techniques in Section 2.8.

Contents

This chapter presents an overview to the state of the art of *Combinatorial Optimization Problems* and different approaches to tackle them. In Section 2.1 the definition of a *Constraint Satisfaction Problem* (*CSP*), emphasizing in the concept of *Constraint Satisfaction Problems*, where we concentrate our main efforts. Constraint propagation techniques are deterministic methods to attack these kind of problems (presented in Section 2.2), but in some cases they are incapable to solve them (they are mostly used to reduce the problem's search space or to prove it unsatisfiable). For that reason, the model presented in this thesis is based on *meta-heuristic* methods (Section 2.3). The *Hybridization* approach combines different techniques in the same solution strategy, so the progresses in this field are exposed in Section 2.5.

The evolution of computer architecture is leading us toward massively multi-core computers for tomorrow, composed of thousands of computing units. A parallel model to solve *CSPs* is the core of this work, and its advances, as well as those obtained in the field of *cooperation between solvers*, are presented in Sections 2.6 and 2.7 respectively. Finally, this chapter presents in Section 2.8 an overview of the progresses in the field of *parameter settings*.

2.1 Combinatorial Optimization

An *Optimization Problem* consists in finding the best solution among all possible ones, subject or not, to a set of constraints, depending on it is a restricted or an unrestricted problem. The suitable values for the involved variables belong to a set called *domain*. When this domain contains only discrete values, we are facing a *Combinatorial Optimization Problem*, and its goal is to find the best possible solution satisfying a global criterion, named *objective function*. *Resource Allocations* [5], *Task Scheduling* [6], *Master-keying* [7], *Traveling Salesman*, *Knapsack Problem*, among others, are well-known examples of *Combinatorial Optimization Problems* [8].

Sometimes, the main goal is not to find the best solution, but finding one feasible solution. This is the case of *Constraint Satisfaction Problems*. Formally, we present the definition of a *CSP* (sometimes also called *Constraint Network*).

Definition 1 (Constraint Satisfaction Problem) *A Constraint Satisfaction Problem (CSP, denoted by $\mathcal{P}$) is a triple $\langle X, D, C \rangle$, where:*

- $X = \{X_1, \ldots, X_n\}$ *is finite a set of variables,*
- $D = \{D_1, \ldots, D_n\}$ *is the set of associated domains. Each domain D_i specifies the set of possible values to the variable X_i.*

- $C = \{c_1, \ldots, c_m\}$ *is a set of constraints. Each constraint is defined involving a set of variables, and specifies the possible combinations of values for these variables.*

In *CSPs*, a *configuration* $s \in D_1 \times D_2 \times \cdots \times D_n$ is a combination of values for the variables in X. Following we define the concept of solution, which is in other words, a configuration satisfying all the constraints $c_i \in C$.

Definition 2 (Solution of a CSP) *Given a CSP* $\mathcal{P} = \langle X, D, C \rangle$ *and a configuration* $S \in D_1 \times D_2 \times \cdots \times D_n$ *we say that it is a solution if and only if:*

$$c_i(S) \text{ is true } \forall c_i \in C$$

The set of all solutions of $\mathcal{P}$ *is denoted by* $Sol(\mathcal{P})$

This field, also called *Constraint Programing* is a famous research topic developed by the field of artificial intelligence in the middle of the 70's, and a programming paradigm since the end of the 80's. A *CSP* can be considered as a special case of *Combinatorial Optimization Problems*, where the objective function is to reduce to the minimum the number of violated constraints in the model. A solution is then obtained when the number of violated constraints reach the value zero. We focus our work in solving this particular case of problems.

CSPs find a lot of "real-world" applications in the industry. In practice, these problems are intractable for classical constraint programming approaches, like *tree search-based solvers* or *backtracking-based solvers* [9], exploring the whole solution space, wish is huge. For that reason, these kinds of problems are mostly tackled by *meta-heuristic methods* or hybrid approaches, like *Monte Carlo Tree Search* methods, which combine precision (tree search) with randomness (meta-heuristic) showing good results in artificial intelligence for games [10, 11].

2.2 Constraint propagation

Constraint propagation techniques are methods used to modify a *Constraint Satisfaction Problem* in order to reduce its variables domains, and turning the problem into one that is equivalent, but usually easier to solve [12]. The main goal is to choose one (or some) constraint(s) and analyzing *local consistency*, which means trying to find values in the variables domain which make constraint unsatisfiable, in order to remove them from the domain. The applied procedure to reduce the variable domains is called *reduction function*,

and it is applied until a new, "smaller" and easier to solve is obtained, and it can not be further reduced: a *fixed point*.

Chaotic Iterations is a technique, that comes from numerical analysis and adapted for computer science needs, used for computing limits of iterations of finite sets of functions [13, 14]. In [15, 16] a formalization of constraint propagation is proposed through *chaotic iterations*. In [17], a coordination-based chaotic iteration algorithm for constraint propagation is proposed. It is a scalable, flexible and generic framework for constraint propagation using coordination languages, not requiring special modeling of *CSPs*. We can find an implementation of this algorithm in DICE (Distributed Constraint Environment) [18] using the MANIFOLD coordination language. Coordination services implement existing protocols for constraint propagation, termination detection and splitting of *CSPs*. DICE combines these protocols with support for parallel search and the grouping of closely related components into cooperating solvers.

MANIFOLD is a strongly-typed, block-structured, event-driven language for managing events, dynamically changing interconnections among sets of independent, concurrent and cooperative processes. A MANIFOLD application consists of a number of processes running on a heterogeneous network. Processes in the same application may be written in different programming languages. MANIFOLD has been successfully used in a broad range of applications [19].

In [20] is proposed an implementation of constraint propagation by composition of reductions. It is a general algorithmic approach to tackle strategies that can be dynamically tuned with respect to the current state of constraint propagation, using composition operators. A composition operator models a sub–sequence of an iteration, in which the ordering of application of reduction functions is described by means of combinators for sequential, parallel or fixed–point computation, integrating smoothly the strategies to the model. This general framework provides a good level of abstraction for designing an object-oriented architecture of constraint propagation. Composition can be handled by the *Composite Design Pattern* [21], supporting inheritance between elementary and compound reduction functions. The propagation mechanism uses the *Observer (Listener) Design Pattern* [22], that makes the connection between domain modifications and re–invocation of reduction functions (event-based relations between objects); and the generic algorithm has been implemented using the *Strategy Design Pattern* [23], that allows to parametrize parts of algorithms.

A propagation engine prototype with a *Domain Specific Language* (DSL) was implemented in [24]. It is a solver–independent language able to configure constraint propagations at the modeling stage. The main contributions are a DSL to ease configure constraint propagation engines, and the exploitation of the basic properties of DSL in order to ensure both completeness and correctness of the produced propagation engine, like: i) *Solver*

independent description: The DSL does not rely on specific solver requirements (but assuming that solvers provide full access to variable and propagator properties), ii) *Expressivity*: The DSL covers commonly used data structures and characteristics, iii) *Extensibility*: New attributes can be introduced to make group definition more concise. New collections and iterators can provide new propagation schemes, iv) *Unique propagation*: The top-bottom left-right evaluation of the DSL ensures that each arc is only represented once in the propagation engine.

Some characteristics are required to fully benefit from the DSL. Due to their positive impact on efficiency, modern constraint solvers already implement these techniques: i) Propagators are discriminated thanks to their priority (deciding which propagator to run next): lighter propagators (in the complexity sense) are executed before heavier ones. ii) A controller propagator is attached to each group of propagators. iii) Open access to variable and propagator properties: for instance, variable cardinality, propagator arity or propagator priority.

To be more flexible and more accurate, they assume that all arcs from the current *CSP*, are explicitly accessible. This is achieved by explicitly representing all of them and associating them with *watched literals* [25] (controlling the behavior of variable–value pairs to trigger propagation) or *advisors* [26] (Method for supporting incremental propagation in propagator–centered setting). *Advisors* in [26] are used to modify propagator state and to decide whether a propagator must be propagated or "scheduled". They also present a concrete implementation of the DSL based on *Choco* [27] (an open source java constraint programming library) and an extension of the *MiniZinc*, a simple but expressive constraint programming modeling language which is suitable for modeling problems for a range of solvers. It is the most used language for codding *CSP*s [28].

However, we can not solve some *CSPs* only applying constraint propagation techniques. It is necessary to combine them with other methods.

2.3 Meta-heuristic methods

Meta-heuristic Methods are non problem-specific techniques that efficiently explore the search space in order to find the solution, and can often find them with less computational effort than iterative methods, so an effective way to face the *CSPs*. Their algorithms are approximate and usually non-deterministic.

A *Meta-heuristic Method* is formally defined as an iterative generation process which guides a subordinate heuristic by combining smartly different concepts for *exploring* (also called

diversification, is guiding the search process through a much larger portion of the search space with the hope of finding promising solutions that are not yet visited) and *exploiting* (also called *intensification*, is guiding the search process into a limited (but promising) region of the search space with the hope of improving a promising already found solution) the search space (the finite set of candidate solutions or configurations) [29], avoiding getting trapped in lost areas of the search space (local minimums). Sometimes they may make use of domain-specific knowledge in the form of heuristics that are controlled by the upper level strategy. Nowadays more advanced meta-heuristics use search experience to guide the search [30].

They are often nature-inspired and are divided in two groups [31]:

a) Single Solution Based: more exploitation oriented, intensifying the search in some specific areas. (We will focus our attention on this first group)

b) Population Based: more exploration oriented, identifying areas of the search space where there are (or where there could be) the best solutions.

2.3.1 Single Solution Based Meta-heuristic

Methods of the first group are also called *trajectory methods*, and they are based on choosing a solution taking into account some criterion (usually random), and they move from a solution to his *neighbor*, following a trajectory into the search space. They can be seen as an intelligent extension of *Local Search Methods* [31]. Local Search Methods are the most widely used approaches to solve *Combinatorial Optimization Problems* because they often produces high–quality solutions in reasonable time.

Simulated Annealing (SA) [32] is one of the first algorithms with an explicit strategy to scape from local minima. Is a method inspired by the annealing technique used by the metallurgists to obtain a "well ordered" solid state of minimal energy. Its main feature is to allow moves resulting in solutions of worse quality than the current solution, in order to scape from local minima, under certain probability, which is decreased during the search process [30]. In [33] is presented a SA algorithm (TTSA) for the Traveling Tournament Problem (TPP) that explores both feasible and infeasible schedules that includes advanced techniques such as strategic oscillation to balance the time spent in the feasible and infeasible regions, by varying the penalty for violations; and reheats (increasing the temperature again) to balance the exploration of the feasible and infeasible regions and to escape local minima.

Tabu Search (TS) [34], is among the most used meta-heuristics for *Combinatorial Optimization Problems*. It explicitly maintain a history of the search, as a short term memory keeping

track of the most recently visited solutions, to scape from local minima, to avoid cycles, and to deeply explore the search space. A TB meta-heuristic guides the search on the approach presented in [35] to solve instances of the *Social Golfers* problem.

The idea of *Guided Local Search* (GLS) [36] is to dynamically change the objective function to help the search to gradually scape from local minima, by changing the search landscape. The set of solutions and the neighborhood are fixed, while the objective function is dynamically changed with the aim of making the current local optimum less attractive [30]. In [37] an implementation of a GLS is used to solve the satisfiability (SAT) problem, which is a special case of a *CSP* where the variables take booleans values an the constraints are disjunctions (logical OR) of literals (variables or theirs negations).

The *Variable Neighborhood Search* (VNS) is another meta-heuristic that systematically changes the size of neighborhood during the search process. These neighborhoods can be arbitrarily chosen, but often a sequence $|\mathcal{N}_1| < |\mathcal{N}_2| < \cdots < |\mathcal{N}_{k_{max}}|$ of neighborhoods with increasing cardinality is defined. The choice of neighborhoods of increasing cardinality yields a progressive diversification of the search [38, 30]. In [39] is introduced a *generalized Variable Neighborhood Search* for *Combinatorial Optimization Problems*, and in [40] is presented a model combining integer programming and VNS for *Constrained Nurse Rostering* problems.

One meta-heuristic that can be efficiently implement on parallel processors is *Greedy Randomized Adaptive Search Procedures* (GRASP). GRASP is an iterative randomized sampling technique in which each iteration provides a solution to the target problem at hand through two phases (constructive and search) within each iteration: the first smartly constructs an initial solution via an adaptive randomized greedy function, and the second applies a local search procedure to the constructed solution in to find an improvement [41]. GRASP does not make any smart use of the history of the search process. It only stores the problem instance and the best found solution. That is why GRASP is often outperformed by other meta-heuristics [30]. However, in [42] some extensions like alternative solution construction mechanisms and techniques to speed up the search are presented.

Galinier et al. present in [43] a general approach for solving constraint based problems by local search. In this work, authors present the concept of *penalty functions*, that we pick up in order to write a *CSP* as an *Unrestricted Optimization Problem* (UOP). This formulation was useful in this thesis for modeling the tackled benchmarks. In this formulation, the *objective function* of this new problem must be such that its set of optimal solutions is equal to the solution set of the original (associated) *CSP*.

Definition 3 (Local penalty function) *Let a* ***CSP*** $\mathcal{P}\langle X, D, C\rangle$ *and a configuration* S

be. We define the operator **local penalty function** *as follow:*

$$\omega_i : D(X) \times 2^{D(X)} \rightarrow \mathbb{R}^+ \cup 0 \text{ where:}$$
$$\omega_i(S, c_i) = \begin{cases} 0 & \text{if } c_i(S) \text{ is true} \\ k \in \mathbb{R}^+ & \text{if not} \end{cases}$$

This penalty function defines the cost of a configuration with respect to a given constraint. In consequence, we define the *global penalty function*, to define the cost of a configuration with respect to all constraint on a *CSP*:

Definition 4 (Global penalty function) *Let a* ***CSP*** $\mathcal{P}\langle X, D, C\rangle$ *and a configuration* S*. We define the operator* **global penalty function** *as follows:*

$$\Omega : D(X) \times 2^{D(X)} \rightarrow \mathbb{R}^+ \cup 0 \text{ where:}$$
$$\Omega(S, C) = \sum_{i=1}^{m} \omega_i(S, c_i)$$

We can now formulate a *Constraint Satisfaction Problem* as an *UOP*:

Definition 5 (CSP's Associated Unrestricted Optimization Problem) *Given a* ***CSP*** $\mathcal{P}\langle X, D, C\rangle$ *we define its* **associated Unrestricted Optimization Problem** $\mathcal{P}_{opt}\langle X, D, f\rangle$ *as follows:*

$$\min_X f(X, C)$$
Where: $f(X, C) \equiv \Omega(X, C)$ *is the objective function to be minimized over the variable* X

It is important to note that a given S is optimum if and only if $f(S, C) = 0$, which means that S satisfies all the constrains in the original *CSP* $\mathcal{P}$.

Adaptive Search is also another efficient algorithm based *local search method*, that takes advantage of the structure of the problem in terms of constraints and variables. It uses also the concept of *penalty function* and relies on iterative repair, based on this information, seeking to reduce the *error* (a projected cost of a variable, as a measure of how responsible is the variable in the cost of a configuration) on the worse variable so far. It computes the penalty function of each constraint, then combines for each variable the *errors* of all constraints in which it appears. This allows to chose the variable with the maximal *error* will be chosen as a "culprit" and thus its value will be modified for the next iteration with the best value, that is, the value for which the total error in the next configuration is minimal [1, 44, 45]. In [46] Munera et al. based their solution method in *Adaptive Search* to solve the *Stable Marriage with Incomplete List and Ties* problem [47], a natural variant of the *Stable Marriage Problem* [48].

Michel and Van Hentenryck [49] propose a constraint-based, object-oriented, architecture to reduce the development time of local search algorithms significantly. The architecture consists of two main components: a declarative component which models the application in terms of constraints and functions, and a search component which specifies the meta-heuristic. Its main feature is to allow practitioners to focus on high-level modeling issues and to relieve them from many tedious and error-prone aspects of local search. The architecture is illustrated using COMET, an optimization platform that provides a Java-like programming language to work with constraint and objective functions (a high level constraint programming) [50, 51], that supports the local search architecture with a number of novel concepts, abstractions, and control structures.

2.3.2 Population Based Meta-heuristic

Also there exist heuristic methods based on populations. These methods do not work with a single solution, but with a set of solutions named *population*. In this other group we can find the *Evolutionary Algorithms*. This is the general definition to name the algorithms inspired by the "Darwin's principle", that says that only the best adapted individuals will survive. They involve *operators* to handle the population to guide it through the search process. The evolutionary algorithm's operators are another branch of study, because they have to be selected properly according to the specific problem, due to they will play an important roll in the algorithm behavior [52].

Probably the most popular in this group are the *Genetic Algorithms* [53], and theirs operators are based on the simulation of the genetic variation process to achieve individuals (solutions in this case) more adapted; and the *Ant Colony* algorithms [54], that simulate the behavior of an ant swarm to find the shortest path from the food source to the nest.

2.4 Hyper-heuristic Methods

Hyper-heuristics are automated methodologies for selecting or generating heuristics to solve hard computational problems [55]. This can be achieved with a learning mechanism that evaluates the quality of the algorithm solutions, in order to become general enough to solve new instances of a given problem. *Hyper-heuristics* are related with the *Algorithm Selection Problem*, so they establish a close relationship between a problem instance, the algorithm to solve it and its performance [56].

Hyper-heuristic frameworks are also known as Algorithm-Portfolio-based frameworks, and their goal is predicting the running time of algorithms using statistical regression. Then the fastest predicted algorithm is used to solved the problem until a suitable solution is found or a time-out is reached [57]. In [58] is presented a *Simple Neighborhood-based Algorithm Portfolio* written in *Python* (Snappy), a very recent framework. Its aim is to provide a tool that can improve its own performances through on-line learning. Instead of using the traditional off-line training step, a neighborhood search predicts the performance of the algorithms.

HYPERION^2 [59] is a Java^{TM} framework for meta– and hyper– heuristics which allows the analysis of the trace taken by an algorithm and its constituent components through the search space, built with the principles of interoperability, generality and efficiency. The main goals of HYPERION^2 are:

a) Promoting interoperability via component interfaces,

b) Allowing rapid prototyping of meta– and hyper– heuristics, with the potential to use the same source code in either case,

c) Providing generic templates for a variety of local search and evolutionary computation algorithms,

d) Making easier the construction of novel meta- and hyper- heuristics by hybridization (via interface interoperability) or extension (subtype polymorphism),

e) *Only pay for what you use* – a design philosophy that attempts to ensure that generality doesn't necessarily imply inefficiency.

hMod is inspired by the previous frameworks, and using a new object-oriented architecture, encodes the core of the hyper-heuristic in several modules, referred as algorithm containers. *hMod* directs the programmer to define the heuristic using two separate XML files; one for the heuristic selection process and another one for the acceptance criteria [60].

Evolving evolutionary algorithms are specialized hyper-heuristic method which attempt to readjust an evolutionary algorithm to the problem needs. An Evolutionary Algorithm (EA) discover the rules and knowledge, to find the best algorithm to solve a problem. In [61] is used linear genetic programming and multi-expression genetic programming, to optimize the EA solving unimodal mathematical functions and another EA adjusts the sequence of genetic and reproductive operators. A solution consists of a new evolutionary algorithm capable of outperforming genetic algorithms when solving a specific class of unimodal test functions.

2.5 Hybridization

The *Hybridization* approach is the one who combine different approaches into the same solution strategy, and recently, it leads to very good results in the constraint satisfaction field. For example, constraint propagation may find a solution to a problem, but they can fail even if the problem is satisfiable, because of its local nature. At each step, the value of a small number of variables are changed, with the overall aim of increasing the number of constraints satisfied by this assignment, and applying other techniques to avoid local solutions, for example adding a stochastic component to choose variables to affect. Integrations of global search (complete search) with local search have been developed, leading to hybrid algorithms.

Hooker J.N. presents in [62] some ideas to illustrate the common structure present in exact and heuristic methods, to encourage the exchange of algorithmic techniques between them. The goal of this approach is to design solution methods ables to smoothly transform its strategy from exhaustive to non-exhaustive search as the problem become more complex.

In [63] a taxonomy of hybrid optimization algorithms is presented in an attempt to provide a mechanism to allow qualitative comparison of hybrid optimization algorithms, combining meta-heuristics with other optimization algorithms from mathematical programming, machine learning and constraint programming.

Monfroy et al. present in [64, 65] a general hybridization framework is proposed to combine complete constraints resolution techniques with meta-heuristic optimization methods in order to reduce the problem through domain reduction functions, ensuring not loosing solutions. Other interesting ideas are TEMPLAR, a framework to generate algorithms changing predefined components using hyper-heuristics methods [66]; and *ParadisEO*, a framework to design parallel and distributed hybrid meta-heuristics showing very good results [67], including a broad range of reusable features to easily design evolutionary algorithms and local search methods.

Another technique has been developed, the called *autonomous search*, based on the supervised or controlled learning. This systems improve their functioning while they solve problems, either modifying their internal components to take advantage of the opportunities in the search space, or to adequately chose the solver to use (*portfolio point of view*) [68].

In [69] is proposed another portfolio-based technique, *time splitting*, to solve optimization problems. Given a problem P and a schedule $Sch = [(\Sigma_1, t_1), \ldots, (\Sigma_n, t_n)]$ of n solvers, the corresponding time-split solver is defined as a particular solver such that:

a) runs Σ_1 on P for a period of time t_1,

b) then, for $i = 1, \ldots, n-1$, runs Σ_{i+1} on P for a period of time t_{i+1} exploiting or not the best solution found by the previous solver Σ_i t_i units of time.

In [70] is proposed a tool (`xcsp2mzn`) for converting problem instances from the XCSP[i] format [71] to MiniZinc that is a simple but expressive constraint programming modeling language which is suitable for modeling problems for a range of solvers. It is the most used language for codding *CSPs* [28]. The second contribution of this work is the development of `mzn2feat` a tool to extract static and dynamic features from the MiniZinc representation, with the help of the Gecode[ii] [72] interpreter, and allows a better and more accurate selection of the solvers to be used according to the instances to solve. Some results are showed proposing that the performances that can be obtained using these features are competitive with state of the art on *CSP* portfolio techniques.

2.6 Parallel computing

Parallel computing is a way to solve problems using some calculus resources at the same time. It is a powerful alternative to solve problems which would require too much time by using the traditional ways, i.e., sequential algorithms [73]. That is why this field is in constant development and it is the topic where we will put most of our effort.

For a couple of years, all processors in modern machines are multi-core. Massively parallel architectures, previously expensive and so far reserved for super–computers, become now a trend available to a broad public through hardware like the Xeon Phi or GPU cards. The power delivered by massively parallel architectures allow us to treat faster these problems [74]. However this architectural evolution is a non-sense if algorithms do not evolve at the same time: the development and the implementation of algorithms should take this into account and tackling the problems with very different methods, changing the sequential reasoning of researchers in Computer Science [75, 76]. We can find in [77] a survey of the different parallel programming models and available tools, emphasizing on their suitability for high-performance computing.

Falcou propose in [78] a programming model: *Parallel Algorithmic Skeletons* (along with a C++ implementation called Quaff to make parallel application development easier. Writing efficient code for parallel machines is less trivial, as it usually involves dealing with low-level

[i] Is a XML-like language for coding *CSPs*. Is not more used than MiniZinc but although it was mainly used as the standard in the *International Constraint Solver Competition* (ended in 2009), the *ICSC* dataset is for sure the biggest dataset of CSP instances existing today

[ii] Is an efficient open source environment for developing constraint-based system and applications.

APIs such as OpenMP, message-passing interfaces (MPI), among others. However, years of experience have shown that using those frameworks is difficult and error-prone. Usually many undesired behaviors (like deadlocks) make parallel software development very slow compared to the classic, sequential approach. In that sense, this model is a high-order pattern to hide all low-level, architecture or framework dependent code from the user, and provides a decent level of organization. QUAFF is a skeleton-based parallel programming library, which has demonstrated its efficiency and expressiveness solving some application from computer vision, that relies on C++ template meta-programming to reduce the overhead traditionally associated with object-oriented implementations of such libraries: the code generation is done at compile time.

Some results have been obtained on this field. The contribution in terms of hardware has been crucial, achieving powerful technologies to perform large–scale calculations. But the development of the techniques and algorithms to solve problems in parallel is also visible, focusing the main efforts in three fundamentals concepts:

a) Problem subdivision,

b) Scalability and

c) Inter-process communication.

In a preliminary review of literature on parallel constraint solving [79], addressing the literature in constraints on exploitation of parallel systems for constraint solving, is starting first by looking at the justification for the multi-core architecture. The direction it is most likely to take and limiting factors on performance such as *Amdahl*'s law, and then reviewing recent literature on parallel constraint programming. For their part, they group the paper into four areas: i) parallelizing the search process, ii) parallel and distributed arc-consistency, iii) multi-agent and cooperative search and iv) combined parallel search and parallel consistency.

The issue of sub-dividing a given problem in some smaller sub-problems is sometimes not easy to address. Even when we can do it, the time needed by each process to solve its own part of the problem is rarely balanced. In [80] are proposed some techniques to tackle this problem, taking into account that sometimes, the more can be sub-divided a problem, the more balanced will be the execution times of the process. In [81] is presented an comparison between Transposition-table Driven Scheduling (TDS) and a parallel implementation of a best-first search strategy (Hash Distributed A*), that uses the standard approach of of *Work Stealing* for partitioning the search space. This technique is based on maintaining a local work queue, (provided by a *root process* through hash-based distribution that assign an unique processor to each work) accessible to other process that "steal" work from if they become unoccupied. The same approach is used in [82] to evaluate *Zobrist Hashing*, an

efficient hash function designed for table games like chess and Go, to mitigate communication overheads.

In [83] is presented a study of the impact of space-partitioning techniques on the performance of parallel local search algorithms to tackle the *k-medoids* clustering problem. Using a parallel local search, this work aims to improve the scalability of the sequential algorithm, which is measured in terms of the quality of the solution within the same time with respect to the sequential algorithm. Two main techniques are presented for domain partitioning: first, *space-filling curves*, used to reduce any N-dimensional representation into a one-dimension space (this technique is also widely used in the nearest-neighbor-finding problem [84]); and second, *k-Means* algorithm, one of the most popular clustering algorithms [85].

In [86] is proposed a mechanism to create sub-*CSPs* (whose union contains all the solutions of the original *CSP*) by splitting the domain of the variables though communication between processes. The contribution of this work is explained in details in section 2.7.

Related to the search process, we can find two main approaches. First, the *single walk* approach, in which all the processes try to follow the same path towards the solution, solving their corresponding part of the problem, with or without cooperation (communication). The other is known as *multi walk*, and it proposes the execution of various independent processes to find the solution. Each process applies its own strategies (portfolio approach) or simply explores different places inside the search space. Although this approach may seem too trivial and not so smart, it is fair to say that it is in fashion due to the good obtained results using it [1].

Scalability is the ability of a system to handle the increasing growth of workload. A system which has improved over time its performance after adding work resources, and it is capable of do it proportionally is called *scalable*. The increase has not been only in terms of calculus resources, but also in the amount of sub-problems coming from the sub-division of the original problem. The more we can divide a problem into smaller sub-problems, the faster we can solve it [87]. *Adaptive Search* is a good example of local search method that can scale up to a larger number of cores, e.g., a few hundreds or even thousands [1]. For this algorithm, an implementation of a cooperative multi-walks strategy has been published in [2]. In this framework, the processes are grouped in teams to achieve search intensification, which cooperate with others teams through a head node (process) to achieve search diversification. Using an adaptation of this method, Munera et al. propose a parallel solution strategy able to solve hard instances of *Stable Marriage with Incomplete List and Ties Problem* quickly. In [88] is presented a combination of this method with an *Extremal Optimization* procedure: a nature-inspired general-purpose meta-heuristic [89].

A lot of studies have been published around this topic. A parallel solver for numerical *CSPs* is presented in [90] showing good results scaling on a number of cores. In [91], an estimation

of the speed-up (a performance measure of a parallel algorithm) through statistical analysis of its sequential algorithm is presented. This is a very interesting result because it a way to have a rough idea of the resources needed to solve a given problem in parallel.

Another issue to treat is the interprocess communication. Many times a close collaboration between process is required, in order to achieve the solution. But the first inconvenient is the slowness of the communication process. Some work have achieved to identify what information is viable to share. One example is [92] where an idea to include low-level reasoning components in the SAT problems resolution is proposed. This approach allow us to perform the clause-shearing, controlling the exchange between any pair of process.

In [2] is presented a new paradigm that includes cooperation between processes, in order to improve the independent multi-walk approach. In that case, cooperative search methods add a communication mechanism to the independent walk strategy, to share or exchange information between solver instances during the search process. This proposed framework is oriented towards distributed architectures based on clusters of nodes, with the notion of *teams* running on nodes and controlling several search engines (*explorers*) running on cores, and the idea that all teams are distributed and thus have limited inter–node communication. The communication between teams ensures diversification, while the communication between explorers is needed for intensification. This framework is oriented towards distributed architectures based on clusters of nodes, where teams are mapped to nodes and explorers run on cores. This framework was developed using the *X10 programming language*, which is a novel language for parallel processing developed by IBM Research, giving more flexibility than traditional approaches, e.g., MPI communication package.

In [3] have been presented an implementation of the meta-solver framework which coordinates the cooperative work of arbitrary pluggable constraint solvers. This approach intents to integrate arbitrary, new or pre–existing constraint solvers, to form a system capable of solving complex mixed–domain constraint problems. The existing increased cooperation overhead is reduced through problem-specific cooperative solving strategies.

In [93] is proposed the first *Deterministic Parallel DPLL* (A complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulas in conjunctive normal form) engine. The experimental results show that their approach preserves the performance of the parallel portfolio approach while ensuring full reproducibility of the results. Parallel exploration of the search space, defines a controlled environment based on a total ordering of solvers interactions through synchronization barriers. To maximize efficiency, information exchange (conflict-clauses) and check for termination are performed on a regular basis. The frequency of these exchanges greatly influences the performance of the solver. The paper explores the trade off between frequent synchronizing which allows

the fast integration of foreign conflict–clauses at the cost of more synchronizing steps, and infrequent synchronizing at the cost of delayed foreign conflict–clauses integration.

Considering the problem of parallelizing restarted back–track search, in [94] was developed a simple technique for parallelizing restarted search deterministically and it demonstrates experimentally that it can achieve near–linear speed–ups in practice, when the number of processors is constant and the number of restarts grows to infinity. They propose the following: Each parallel search process has its own local copy of a scheduling class which assigns restarts and their respective fail–limits to processors. This scheduling class computes the next *Luby* restart fail–limit and adds it to the processor that has the lowest number of accumulated fails so far, following an *earliest–start–time–first strategy*. Like this, the schedule is filled and each process can infer which is the next fail–limit that it needs to run based on the processor it is running on – without communication. Overhead is negligible in practice since the scheduling itself runs extremely fast compared to CP search, and communication is limited to informing the other processes when a solution has been found.

In [95], were explored the two well–known principles of diversification and intensification in portfolio–based parallel SAT solving. To study their trade–off, they define two roles for the computational units. Some of them classified as *Masters* perform an original search strategy, ensuring diversification. The remaining units, classified as *Slaves* are there to intensify their master's strategy. There are some important questions to be answered: i) what information should be given to a slave in order to intensify a given search effort?, ii) how often, a subordinated unit has to receive such information? and iii) the question of finding the number of subordinated units and their connections with the search efforts? The results lead to an original intensification strategy which outperforms the best parallel SAT solver *ManySAT*, and solves some open SAT instances.

Multi-objective optimization problems involve more than one objective function to be optimized simultaneously. Usually these problems do not have an unique optimal solution because the exist a trade-off between one objective function and the others. For that reason, in a multi-objective optimization problem, the concept of Pareto optimal points is used. A Pareto optimal point is a solution that improving one objective function value, implies the deterioration of at least one of the other objective function. A collection of Pareto optimal points defines a Pareto front. In [96], is proposed a new search method, called *Multi-Objective Embarrassingly Parallel Search* (MO–EPS) to solve multi-objective optimization problems, based on: i) Embarrassingly Parallel Search (EPS): where the initial problem is split into a number of independent sub-problems, by partitioning the domain of decision variables [80, 97]; and ii) Multi-Objective optimization adding cuts (MO–AC): an algorithm that transforms the multi-objective optimization problem into a feasibility one, searches a feasible solution and then the search is continued adding constraints to the problem until either the problem becomes infeasible or the search space gets entirely explored [98].

A component-based constraint solver in parallel is proposed in [99]. In this work, a parallel solver coordinates autonomous instances of a sequential constraint solver, which is used as a software component. The component solvers achieve load balancing of tree search through a time-out mechanism. It is implemented a specific mode of solver cooperation that aims at reducing the turn-around time of constraint solving through parallelization of tree search. The main idea is to try to solve a *CSP* before a time-out. If it can not find a solution, the algorithm defines a set of disjoint sub-problems to be distributed among a set of solvers running in parallel. The goal of the time-out mechanism is to provide an implicit load balancing: when a solver is busy, and there are no subproblems available, another solver produces new sub-problems when its time-out elapses.

2.7 Solvers cooperation

The interaction between solvers exchanging some information is called *solver cooperation* and it is very popular in this field due to their good results. Its main goal is to improve some kind of limitations or inefficiency imposed by the use of unique solver. In practice, each solver runs in a computation unit, i.e. thread or processor. The cooperation is performed through inter–process communication, by using different methods: *signals*, asynchronous notifications between processes in order to notify an event occurrence; *semaphore*, an abstract data type for controlling access, by multiple processes, to a common resource; *shared memory*, a memory simultaneously accessible by multiple processes; *message passing*, allowing multiple programs to communicate using messages; among others.

Kishimoto et al. present in [100] a parallelization of the an algorithm A* (Hash Distributed A*) for *optimal sequential planning* [101], exploiting distributed memory computers clusters, to extract significant speedups from the hardware. In classical planning solving, both the memory and the CPU requirements are main causes of performance bottlenecks, so parallel algorithms have the potential to provide required resources to solve changeling instances. In [81], authors study scalable parallel best-first search algorithms, using MPI, a paradigm of *Message Passing Interface* that allows parallelization, not only in distributed memory based architectures, but also in shared memory based architectures and mixed environments (clusters of multi-core machines) [102].

In [103] is presented a paradigm that enables the user to properly separate computation strategies from the search phases in solver cooperations. The cooperation must be supervised by the user, through *cooperation strategy language*, which defines the solver interactions in order to find the desired result.

In [92], an idea to include low-level reasoning components in the SAT problems resolution is proposed, dynamically adjusting the size of shared clauses to reduce the possible blow up in communication. [103] presents a paradigm that enables the user to properly separate strategies combining solver applications in order to find the desired result, from the way the search space is explored.

Meta–S is an implementation of a theoretical framework proposed in [3], which allows to tackle problems, through the cooperation of arbitrary domain–specific constraint solvers. *Meta–S* [3] is a practical implementation and extension of a theoretical framework, which allows the user to attack problems requiring the cooperation of arbitrary domain–specific constraint solvers. Through its modular structure and its extensible strategy specification language it also serves as a test–bed for generic and problem–specific (meta–)solving strategies, which are employed to minimize the incurred cooperation overhead. Treating the employed solvers as black boxes, the meta–solver takes constraints from a global pool and propagates them to the individual solvers, which are in return requested to provide newly gained information (i.e. constraints) back to the meta–solver, through variable projections. The major advantage of this approach lies in the ability to integrate arbitrary, new or pre–existing constraint solvers, to form a system that is capable of solving complex mixed–domain constraint problems, at the price of increased cooperation overhead. This overhead can however be reduced through more intelligent and/or problem–specific cooperative solving strategies. HYPERION [59] is an already mentioned framework for meta– and hyper–heuristics built with the principle of interoperability, generality by providing generic templates for a variety of local search and evolutionary computation algorithms and efficiency, allowing rapid prototyping with the possibility of reusing source code.

Arbab and Monfory propose in [86] a technique to guide the search by splitting the domain of variables. A *Master* process builds the network of variables and domain reduction functions, and sends this informations to the worker agents. They workers concentrate their efforts on only one sub-CSP and the *Master* collects solutions. The main advantage is that by changing only the search agent, different kinds of search can be performed. The coordination process is managing using the MANIFOLD coordination language [19].

2.8 Parameter setting techniques

Most of these methods to tackle combinatorial problems, involve a number of parameters that govern their behavior, and they need to be well adjusted, and most of the times they depend on the nature of the specific problem, so they require a previous analysis to study their behavior [104]. That is way another branch of the investigation arises: *parameter*

tuning. It is also known as a meta optimization problem, because the main goal is to find the best solution (parameter configuration) for a program, which will try to find the best solution for some problem as well. In order to measure the quality of some found parameter setting for a program (solver), one of these criteria are taken into consideration: the speed of the run or the quality of the found solution for the problem that it solves.

The are tow classes to classify these methods:

a) Off-line tunning: Also known just as parameter tuning, were parameters are computed before the run.

b) On-line tunning: Also known as parameter control, were parameters are adjusted during the run, and

2.8.1 Off-line tunning

The technique of parameter tuning or off-line tunning, is used to computed the best parameter configuration for an algorithm before the run (solving a given instance of a problem), to obtain the best performance. Most of algorithms are very sensible to their parameters. This is the case of Evolutionary Algorithms (EA), were some parameters define the behavior of the algorithm. In [105] is presented a study of methods to tune these algorithms.

In [106] is presented *EVOCA*, a tool which allows meta-heuristics designers to obtain good results searching a good parameter configuration with no too much effort, by using the tool during the iterative design process. Holger H. Hoos highlights in [107] the efficacy of the technique named *racing procedure*, that is based on choosing a set of model problems and adjusting the parameters through a certain number of solver runs, discarding configurations that show a behavior substantially worse than the best already obtained so far.

ParamsILS (version 2.3) is a tool for parameter optimization for parametrized algorithms, which uses powerful stochastic local search methods and it has been applied with success in many combinatorial problems in order to find the best parameter configuration [108]. It is an open source program written in *Ruby*, and the public source include some examples and a detailed and complete User Guide with a compact explanation about how to use it with a specific solver [109].

Revac is a method based on information theory to measure parameter relevance, that calibrates the parameters of EAs in a robust way. Instead of estimating the performance of an EA for different parameter values, the method estimates the expected performance when parameter values are chosen from a given probability density distribution C. The method iteratively refines the probability distribution C over possible parameter sets, and starting

with a uniform distribution C_0 over the initial parameter space $\mathcal{X}$, the method gives a higher and higher probability to regions of $\mathcal{X}$ that increase the expected performance of the target EA [110]. In [111] is presented a case study demonstrating that using the REVAC the "world champion" EA (the winner of the CEC-2005 competition) can be improved with few effort.

Another technique was successfully used to tune automatically parameters for EAs, through a model based on a *case-based reasoning* system. It attempts to imitate the human behavior in solving problems: look in the memory how we have solved a similar problem [112] .

2.8.2 On-line tunning

Although parameter tunning shows to be an effective way to adjust parameters to sensibles algorithms, in some problems the optimal parameter settings may be different for various phases of the search process. This is the main motivation to use on-line tuning techniques to find the best parameter setting, also called *Parameter Control Techniques*. Parameter control techniques are further divided into i) *deterministic parameter control*, where the value of a strategy parameter is altered by some deterministic rule, ignoring any feedback; ii) *adaptive parameter control*, which continually update their parameters using feedback from the population or the search, and this feedback is used to determine the direction or magnitude of the parameter changes; and iii) *self-adaptive parameter control*, which assign different parameters to each individual, Here the parameters to be adapted are coded into the chromosomes that undergo mutation and recombination, but these parameters are coded into the chromosomes that undergo mutation and recombination [113].

Differential Evolution (DE) algorithm has been demonstrated to be an efficient, effective and robust optimization method. However, its performance is very sensitive to the parameters setting, and this dependency changes from problem to problem. The selection of proper parameters for a particular optimization problem is a quite complicate subject, especially in the multi-objective optimization field. This is the reason why many researchers are motivated to develop techniques to set the parameters automatically.

Liu et al. propose in [114] an adaptive approach which uses fuzzy logic controllers to guide the search parameters, with the novelty of changing the mutation control parameter and the crossover during the optimization process. A self-adaptive DE (SaDE) algorithm is proposed in [115], where both trial vector generation strategies and their associated control parameter values are gradually adjusted by learning from the way they have generated their previous promising solutions, eliminating this way the time-consuming exhaustive search for the most suitable parameter setting. This algorithm has been generalized to multi-objective realm, with objective-wise learning strategies (OW-MOSaDE) [116].

Drozdik et al. present in [117] a study of various approaches to find out if one can find an inherently better one in terms of performance and whether the parameter control mechanisms can find favorable parameters in problems which can be successfully optimized only with a limited set of parameters. They focused in the most important parameters: 1) the *scaling factor*, which controls the structure of new invidious; and 2) the *crossover probability*.

Meta-GAs [118] is a genetic self-adapting algorithm, adjusting genetic operators of genetic algorithms. In this paper the authors propose an approach of moving towards a Genetic Algorithm that does not require a fixed and predefined parameter setting, because it evolves during the run.

3

Prior works leading to POSL

In this chapter are presented Prior works leading to POSL*. In Section 3.1 we present a brief work where we applied the problem subdivision approach to solve the k-medoids problem in parallel, as a first attempt aiming for the right direction in order to find the proper approach. Finally we present in Section 3.2 a study applying the* ParamILS *tool in order to find the optimum parameter configuration to Adaptive Search solver.*

Contents

3.1 Domain Split

Usually, when we want to solve some problem using parallelism, our first thought is either partitioning the problem into a set of sub-problems, or divide the search space. In both cases, the idea is to solve a set of problems, all of them smaller and easer than the original one, and combining all the solution to obtain the solution of the original problem. In [83] a study of the impact of space-partitioning techniques on the performance of parallel local search algorithms to tackle the *k-medoids* clustering problem is presented. The authors use a parallel local search, in order to improve the scalability of the sequential algorithm, which is measured in terms of the quality of the solution within the same time with respect to the sequential algorithm. In this work two main techniques for domain partitioning are presented: *space-filling curves*, used to reduce any N-dimensional representation into a one-dimension space; and *k-Means* algorithm. We found that the methods used for domain partitioning do not take into account the number of clients associated to each new sub-domain. This results in an unbalanced distributions of workload phenomenon. That is the reason why our goal was designing some ideas to tackle the same problem. [i]

We propose Algorithm 1, which represents the backbone of our idea. This algorithm takes a set of $\mathbb{R}^2$ *points* (representing the locations of the clients) and returns a partition of size K. Such a set of points is called a *domain* and the partition a *sub-domain*. At each intermediate step i, we have a set (list) of sub-domains. The algorithm takes the most populated one, splits it into two (or four, depending on the strategy) new sub-domains and includes them in the list. The stop condition will depends on the fallowed approach (see below).

Algorithm 1: Domain_Split

input : U: Set of client locations
output: Q = $\{Q_i\}_{i=1...K}$: K subsets of U

```
1 A ← U
2 Q.Insert(A)
3 repeat
4 |   A ← Q.GetNext() /* It also removes the returned element */
5 |   [a1, a2] ← Split(A)
6 |   Q.Insert(a1, a2)
7 until <some condition>
```

First of all, we make clear some details of the algorithm exposed above:

- `Insert(...)` Inserts a set (or two) into the data structure.

[i] This work falls within the framework of the Ulysses project between France and Ireland

- `GetNext()` Returns the next sub-set to be divided, tacking into account the *split strategy* (see below).
- `Split(...)` Returns two sub-domains as a subdivision of the given domain (parameter).

In the next sub-sections we answer two main question arising at this point:

a) How to split the each sub-domain? (<*Split*> function on line 5) It refers to, given a set of points (locations), how to decide which of them will be included into one sub-domain and which of them into the other.

b) How much to split each sub-domain? (<*some condition*> on line 3) It refers to decide when to stop splitting the domain.

3.1.1 Domain Splitting. General point of view

In order to split the domain, we can think in some approaches, taking into account the number of available cores and the number of metro-nodes we want to place in the system. In the article of A. Arbelaez and L. Quesada a domain split taking into account the number of available cores for parallel calculus is proposed. In our approach we intend to extend this idea keeping in mind also the number of metro-nodes to allocate. Following, we propose three variants to face the problem:

a) **one metro-node per core**: In this variant we can assign one metro-node to each core, and in this case, splitting the domain in K sub-domains (K is the number of cores). It means that the algorithm will compute the best position for a metro-node in a current sub-domain.

- In this case we only have to replace the line 3 by something like: **for** $i \leftarrow 1$ **to** N **do** ..., where N is the number of metro-nodes.
- The ideal scenario here is when $N = K$ which is not probable at all. So we only should study the case when they are different
- In that case, we need to distribute efficiently the metro-nodes into the domain subdivisions, but here, one possible scenario arise: it can be happen that, depending on the followed domain-split strategy, we were trying to allocate a metro-node into an area with a few clients. This produce a very local point of view of the problem. That is the reason why we propose the following *second variant*.

b) **Incomplete partition**: To split a sub-domain if it can generate sub-sub-domains containing at least C clients. It means, for example, if there exist in list a sub-domain with 8 clients, and the number C is fixed in $C = 5$, this sub-domain can not be divided, because it will generate for sure a sub-sub-domain with less that 5 clients. In this case more than one metro-node would be allocated in some of those specific sub-domains, because we will have more metro-nodes than sub-domains in our model.

- Of course, using this variant we can find situations described in the first variant. In that case we should proceed consequently.

c) **Combination**: This variant is just a combination of the two previous variants. Then, we will be working with two parameters:

- $C \rightarrow$ Lower bound of clients for new sub-domains. It means that a sub-domain can be divided iff the new produced sub-domains will contains more than C clients.
- $M \rightarrow$ Lower bound of metro-nodes to be allocated in a sub-domain. In this case we will split the domain while it can be ensure that at least M metro-nodes will be assigned to each sub-domain.

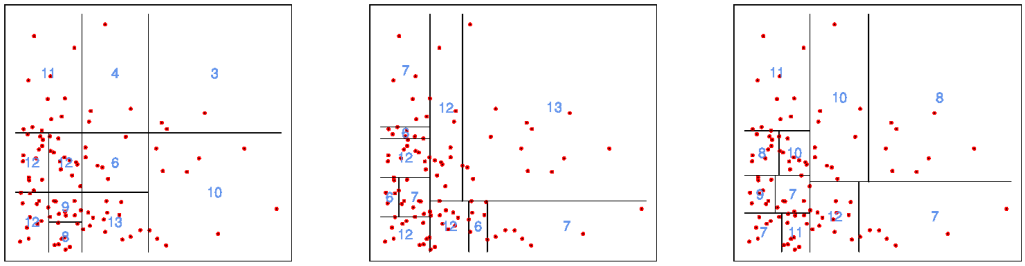
3.1.2 Split strategies

In this sub-section we first discuss three ideas to split the domain. They take a sub-domain and produce other two, dividing the space vertically or horizontally, depending on the shape of the current sub–domain. In other section we expose another strategy to follow, in which the subdivision of a sub-domain produces four sub-domains.

In all of them the number of clients in each sub-domain is taken into account, but in different ways. A common feature between them is that, at the moment of splitting a given sub-domain, it will be done in a perpendicular way (either to the x–axis or to the y–axis, depending on the characteristic of the sub-domain to be divided). The reference point to divide the sub-domain is called the *cut point* of the sub-domain.

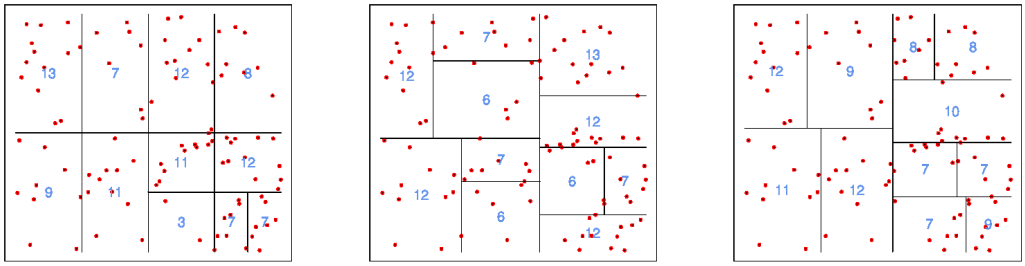
a) **Geometrical Split**:

- Split criterion: *Geometrical* $\rightarrow$ Dividing the region in tow parts with the same area.
- Cut point: The middle point of the x–axis (or y–axis, depending on the shape)
- Result: Two equal regions, but with different amount of clients (see Figure 3.1a). In the next step, the next sub-domain to divide will be, from those already divided, the most populated one (the sub-domain with more clients).



(a) Geometrical Split **(b)** Cardinal Split **(c)** Mean Split

Figure 3.1: Split domain of point normally distributed $\mathcal{N}(0, 0.35)$



(a) Geometrical Split **(b)** Cardinal Split **(c)** Mean Split

Figure 3.2: Split domain of point uniformly distributed

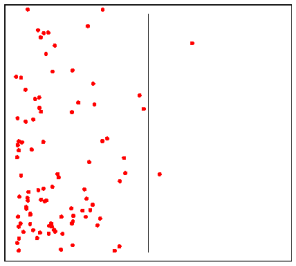


Figure 3.3: If we assume $C = 3$ then this set can't be divided

- Problem: Maybe we need to divide a sub-domain because it has a lot of client, but in the other hand it generate a sub-domain with a few clients, as you can see in the Figure 3.3.
- Benefit: Very fast split. If we attack scenarios with point uniformly distributed, the behavior is pretty much desirable, as we can see in the Figure 3.2a.

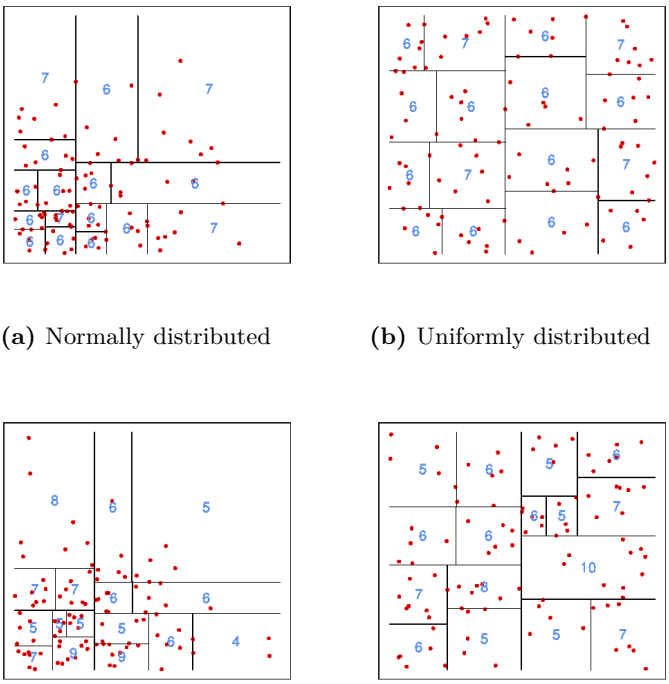
b) **Cardinal Split**

- Split criterion: The number of clients, i.e., a sub-domain is divided in such a way that in the resulting sub-sub-domains there will be the same number of clients.
- Cut point: The current sub-domain is ordered and the x–axis (or y–axis, depending on the shape) of the element (location of the client) right in the center is selected to be the *cut point*
- Result: Two regions with the same (± 1) amount of clients (see Figure 3.1a). In the next step, the next sub-domain to divide will be, from those already divided, the most populated one (the sub-domain with more clients).
- Problem: The subdivision process is a bit more costly: we need to group the clients on both sides of a *perfect pivot*[ii].
- Benefit: It guarantees the same cardinality in both new sub-sub-domains.

c) **Mean Split**

- Split criterion: This is a mid-point strategy between the two previous. The goal is to find a *cut point* to group the elements of the current sub-domain, but in a easy way (fast), for that reason we do not compute the exact middle point to produce two sub-sub-domains with the same cardinality, as in the previous approach, instead of that we work with his expected value: the arithmetic mean.
- Cut point: We compute the mean of the x–axis (or y–axis) of the elements (locations) of the current sub-domain, and it will be the *cut point*
- Result: Two regions with not exact information about their sizes or their cardinality.
- Problem: *Idem.*
- Benefit: The *cut point* can be obtained by a $O(n)$ number of operations. As we can see in the preliminary results[iii] (Figures 3.4c and 3.4d), the behavior of this technique is near to the *cardinal split* technique, at least for normally and uniformly distributed sets of point.

[ii] Element of a set with the same number of elements lower and greater than him
[iii] The experiments are coded in R[119].



(a) Normally distributed **(b)** Uniformly distributed

(c) Normally distributed **(d)** Uniformly distributed

Figure 3.4: Domain divided in $2^4 = 16$ sub-domains: (a) - (b) Cardinal; (c) - (d) Mean.

3.1.3 Conclusion

In this section we present a theoretical and not validated work where we applied the *search space subdivision* approach to solve *k-medoids problem* in parallel. We have proposed some different strategies and we have showed graphically some characteristics of them.

3.2 Tunning methods for local search algorithms

In this section we present our results applying ParamILS (version 2.3)[iv], a tool to find the optimum parameter settings for parametrized algorithms [108], to tune *Adaptive Search* solver[v]. It is an open source program (project) in *Ruby*, and the public source code include some examples and a detailed and complete *User Guide* with a compact explanation about how to use it with a specific solver [109].

[iv] Open source program (project) in *Ruby*, available at `http://cs.ubc.ca/labs/beta/Projects/ParamILS`. The `zip` file includes some examples that you can run and see how the tool works. It In addition, it brings a complete User Guide with a compact explanation about how to use it with a specific solver.

[v] An implementation from Daniel Díaz available at https://sourceforge.net/projects/adaptivesearch/

In order to facilitate the work we have decided to build a *wrapper* in C++ language, in order to tune more than one problem with the same code. The goal of doing this is using the tool to tune the solver, i.e., finding the best parameter configuration for a specific problem, but also the best parameter configuration to solve any kind of benchmark (a general parameter configuration).

Following we present in Table 3.1 the parameter list that we worked with:

Table 3.1: Adaptive Search parameters

Parameter	Type	Description
-P	PERCENT	probability to select a local min (instead of staying on a plateau)
-f	NUMBER	freeze variables of local min for NUMBER swaps
-F	NUMBER	freeze variables swapped for NUMBER swaps
-l	LIMIT	reset some variables when LIMIT variable are frozen
-p	PERCENT	reset PERCENT of variables

In this section, we explain in details the implementation and the experimentation process.

3.2.1 Using ParamILS

To use the tool ParamILS, we have installed Ruby 1.8.7 in our computer. We used a laptop Dell XPS 15(Intel Core i7-4702HQ 2.2 GHz, 16384 MB, Dual-channel DDR3L 1600 MHz) with Ubuntu 14.4. To run the tool, we needed to use the following command line:

```
>> ruby param_ils_2_3_run.rb -numRun 0 -scenariofile /.../<scenario_file> -validN 100
```

Where <scenario_file> is the name of the file where we have to put all the information that ParamILS needs to tune the solver (the *tuning scenario file*). We explain its content in the next section.

3.2.2 Tuning scenario files

The *tuning scenario file* is a text file with all needed information to tune the solver using ParamILS. It includes where to find the solver binary file, the parameters domains, etc. In our case, the *tuning scenario file* looks like the following:

```
algo = ./QtWrapper_wrapper
execdir = /.../src
deterministic = 0
```

```
run_obj = runtime
overall_obj = mean
cutoff_time = 50.0
cutoff_length = max
tunerTimeout = 3600
paramfile = instances/all_intervals-params.txt
outdir = instances/all_intervals-paramils-out
instance_file = instances/.../all_intervals-lower-instances.txt
test_instance_file = instances/.../all_intervals-upper-instances.txt
```

We explain in details each line in this file:

- **algo** → An algorithm executable or a call to a wrapper script around an algorithm that aims the input/output format of *ParamILS* (the wrapper).
- **execdir** → Directory to execute **algo** from: "cd <execdir>; <algo>"
- **run_obj** → A scalar quantifying how "good" a single algorithm execution is, such as its required runtime.
- **overall_obj** → While **run_obj** defines the objective function for a single algorithm run, **overall_obj** defines how those single objectives are combined to reach a single scalar value to compare two parameter configurations. Implemented examples include **mean**, **median**, **q90** (the 90% quantile), **adj_mean** (a version of the mean accounting for unsuccessful runs: total runtime divided by number of successful runs), **mean1000** (another version of the mean accounting for unsuccessful runs: (total runtime of successful runs + 1000 x runtime of unsuccessful runs) divided by number of runs – this effectively maximizes the number of successful runs, breaking ties by the runtime of successful runs; it is the criterion used in most of Frank experiments), and **geomean** (geometric mean, primarily used in combination with **run_obj** = **speedup**. The empirical statistic of the cost distribution (across multiple instances and seeds) to be minimized, such as the mean (of the single run objectives). [vi]
- **cutoff_time** → The time after which a single algorithm execution will be terminated unsuccessfully. This is an important parameter: if chosen too high, lots of time will be wasted with unsuccessful runs. If chosen too low the optimization is biased to perform well on easy instances only.
- **tunerTimeout** → The timeout of the tuner. Validation of the final best found parameter configuration starts after the timeout.

[vi] We use **mean** but maybe we can experiment with other values

- **`paramfile`** → Specifies the file with the parameters of the algorithms.
- **`outdir`** → Specifies the directory *ParamILS* should write its results to.
- **`instance_file`** → Specifies the file with a list of training instances.
- **`test_instance_file`** → Specifies the file with a list of test instances.

Another important file that we have to compose properly is the *algorithm parameter file*, just following the instruction from [109] *–[...] each line lists one parameter, in curly parentheses the possible values considered, and in square parentheses the default value [...].* Our *algorithm parameter file* looks like follows:

```
P {20, 25, 30, 35, 40, 45, 50, 55, 60} [50]
f {0, 1, 2, 3} [1]
F {0, 1, 2, 3} [0]
l {0, 1, 2, 3} [1]
p {1, 2, 3, 5, 10, 20} [5]
```

In the current *Adaptive Search* implementation, the solver binary file and the problem instance are the same thing. It means that we only have to use the following command to solve the *All–intervals* problem of size K, for example:

```
>> ./all-intervals K
```

So, to use ParamILS we modified a little the code: now our solver takes the size parameter from a text file. That way, the instance file is a text file only containing a number.

The solver we want to tune using ParamILS (*Adaptive Search* in this case), must aims specific input/output rules. For that reason instead of modifying the current code of *Adaptive Search* implementation, we preferred to build a wrapper.

3.2.3 Building the wrapper

The algorithm executable must follow the input/output criteria presented below:

Launch command:

```
>> <algo_exectuable> <instance_name> <instance-specific_information> ...
<cutoff_time> <cutoff_length> <seed> <params>
```

- <algo_exectuable> Solver

- <instance_name> In our case, a text file containing only the problem size
- <instance-specific_information> We don't use it
- <cutoff_time> Cut off time for each run of the solver (see above)
- <cutoff_length> We don't use it
- <seed> Random seed
- <params> Parameters and its values

Exmaple:

```
>> ./QtWrapper_320.txt "" 50.0 214483647 524453158 -p 5 -l 1 -f 1 -P 50 -F 0
```

Output:

```
>> <solved>, <runtime>, <runlength>, <best_sol>, <seed>
```

- <**solved**> SAT if the algorithm terminates successfully. TIMESOUT if the algorithm times out.
- <**runtime**> Runtime
- <**runlength**> -1 (as Frank Hutter recommended)
- <**best_sol**> -1 (as Frank Hutter recommended)
- <**cutoff_length**> We don't use it
- <**seed**> Used random seed

Exmaple:

```
>> SAT, 2.03435, -1, -1, 524453158
```

To build the wrapper we have followed a simple algorithm: launch two concurrent process. In the parent process we translate the input of the wrapper to the input of *Adaptive Search* solver. The solver is executed, and the runtime is measured. After that we post the output properly. In the child process a *sleep* command is executed for <runtime> seconds and after

that, if the parent process has not finished yet, it is killed, posting a time-out message. See Algorithm 2 for more details.

Algorithm 2: Costas Wrapper

input : Pth_π: problem instance path, k: cut off time, s: random seed, θ: parameters configuration
output: $PiLS_{out}$: Output in a ParamILS way

```
1  fork() /* Divide the execution in two processes                    */
2  if <in child process> then
3      t_0 ← clock_TIC()
4      strCall ← build_str(" ./AS_Wrapper %1 -s %2 %3", Pth_π, s, θ)
5      systemCall(strCall)
6      t_e ← clock_TOC()
7      killProcess(<parent process>)
8      t ← t_e- t_0
9      return paramilsOutput(SAT, t, s)
10 else
11     sleep(k)
12     killProcess(<child process>)
13     return paramilsOutput(TIMESOUT, k, s)
14 end
```

3.2.4 Using the wrapper

3.2.4.1 Factory call

The first thing we have to do is to implement the class ICallFactory. Here, the string-binary-name for the command call is statically obtained. We present, as example, the class All_IntervalCallFactory:

```
// all_interval_call_factory.h
class All_IntervalCallFactory:  public ICallFactory
{
    public:
       std::string BuildCall();
       std::string BuildDefaultCall();
};
```

```
// all_interval_call_factory.cpp
#define ALGO_EXECUTABLE "./all-interval"
```

```
#define DEFAULT_CALL "./all-interval _100.txt"

std::string All_IntervalCallFactory::BuildCall()
{
    return ALGO_EXECUTABLE;
}
std::string All_IntervalCallFactory::BuildDefaultCall()
{
    return DEFAULT_CALL;
}
```

All we have to do is to define our new macro **ALGO_EXECUTABLE** (**DEFAULT_CALL** is not being used)

3.2.4.2 Main method

Let's suppose now that we want to run an algorithm called *mySolver* that receives a file as parameter, called *my_instance_size.txt* (this is mandatory). We have to create (as we've explained before) the class My_SolverCallFactory and defining the macro as follows:

```
#define ALGO_EXECUTABLE "./mySolver"
```

Now, the main method would be exactly like this:

```
int main( int argc,  char* argv[])
{
   shared_ptr<ICallFactory> problem =
      make_shared<My_SolverCallFactory>();
   shared_ptr<TuningData> td =
      (make_shared<TuningData>(argc, argv, problem));

   shared_ptr<ADWrapper> w (make_shared<ADWrapper>());
   string output = w->tune(td);

   cout << output << endl;
    return 0;
}
```

3.2.5 Results

3.2.5.1 Tuning *All-intervals*

Study cases:

a) The *training instances set* is composed by instances of *All–Intervals* problems of order N with

$$N \in \{100, 110, 120, 130, 140, 150, 160, 170, 180\}$$

b) The *test instances set* is composed by instances of *All–Intervals* problems of order N with

$$N \in \{190, 200, 210, 220, 230, 240, 250, 260, 265\}$$

c) The time-out for each run is 50.0 seconds

d) The test quality is based on 100 runs

Experiment 1 Parameters domains:

- **P** `{41, 46, 51, 56, 60, 66, 71, 76, 80}`
- **F, f, l** `{0, 1, 2, 3}`
- **p** `{5, 10, 15, 20, 25, 30, 35}`

The results are presented in Table 3.2.

Experiment 2 Parameters domains:

- **P** `{41, 46, 51, 56, 60, 66, 71, 76, 80}`
- **F, f, l** `{0, 1, 2, 3}`
- **p** `{5, 10, 15, 20, 25, 30, 35}`

The results are presented in Table 3.3.

Experiment 3 **Parameters domains**:

- **P** `{10, 20, 30, 40, 50, 60, 70, 80, 90}`
- **F, f, l** `{0, 1, 2, 3, 4, 5, 6, 7, 8}`
- **p** `{10, 20, 30, 40, 50, 60, 70}`

The results are presented in Table 3.4.

Testing parameters In the next experiment, only the results obtained in the Tables 3.2 and 3.4 were took into account, since they were obtained by using longer times-out. As it can be observed in those tables, *Adaptive Search* seems to show a good behavior if the parameters **F**, **P** and **l** are in the following sets: **F** $\in$ `{ 0, 1}`, **P** $\in$ `{ 80, 90}` and **l** $\in$ `{ 0, 1}`.

In that sense, a specific configuration was extracted from the results above, and 60 runs of *Adaptive Search* were performed solving *All–Intervals* ($N = 600$) benchmark:

- 30 using the default parameter configuration (`-F 0 -P 66 -f 1 -l 1 -p 25`)
- 30 with an optimal parameter configuration extracted from the Tables 3.2, 3.4 (`-F 0 -P 80 -f 6 -l 1 -p 10`)

Table 3.5 shows the results by using the default parameter configuration, and Table 3.6 shows the results by using the parameter configuration found by *ParamILS*, and it is clear that the default configuration shows better results than *ParamILS*'s one.

3.2.5.2 Tuning *Costas Array*

Study cases:

a) The *training instances set* is composed by instances of *Costas Array* problems of order N with $9 \leq N \leq 15$

b) The *test instances set* is composed by instances of *Costas Array* problems of order N with $14 \leq N \leq 19$

c) The cutoff for each run was 60.0 seconds

d) The test quality is based on 100 runs

Tuning experiments **Parameters domains**:

- **P** `{10, 20, 30, 40, 50, 60, 70, 80, 90}`
- **F, f, l** `{0, 1, 2, 3, 4, 5, 6, 7, 8}`
- **p** `{5, 10, 20, 30, 40, 50, 60, 70}`

The results are presented in Table 3.7.

Testing parameters Table 3.7 shows how *Adaptive Search* seems to be not sensitive to parameters **F** and **p**, i.e. they don't change during the tuning process. On the other hand, the tuner seems to find some optimum values for the other parameters: $\mathbf{P} \in$ `{ 80, 90}`, $\mathbf{f} \in$ `{ 4, 5}` and $\mathbf{l} =$ `2`.

In that case also, an specific configuration was extracted from the results showed in Table 3.7, and 60 runs of *Adaptive Search* were performed solving *Costas Array* ($N = 20$) benchmark:

- 30 using the default parameter configuration (`-F 0 -P 50 -f 1 -l 0 -p 5`)
- 30 with an optimal parameter configuration extracted from the Table 3.7 (`-F 3 -P 90 -f 5 -l 2 -p 30`)

Table 3.8 shows the results by using the default parameter configuration, and Table 3.9 shows the results by using the parameter configuration found by *ParamILS*. One more time, "in the mean", the default configuration wins.

3.2.6 Tuning comparison

3.2.6.1 Experiment 1: Around Default parameters

Parameters domains:

- **P** `{43, 45, 47, 50, 53, 55, 57}`
- **F, f, l** `{0, 1, 2}`
- **p** `{5, 7, 10}`

The results are presented in Table 3.10.

3.2.6.2 Experiment 2: Around ParamILS parameters

Parameters domains:

- **P** `{75, 77, 80, 83, 85, 87, 90, 93, 95}`
- **f** `{4, 5, 6}`
- **F** `{2, 3, 4}`
- **l** `{1, 2, 3}`
- **p** `{20, 25, 30, 35, 40}`

The results are presented in Table 3.11.

3.2.7 Conclusion

The conclusion of this study is that the tunning process by hand in this case was more effective than using *ParamILS*. The results show that the found parameters are less effectives than the default parameters in both cases.

Table 3.2: Results with `tunerTimeout` = 20000 seconds

Initial configuration					Final best configuration					Training quality	Number of runs	Test quality
F	P	f	l	p	F	P	f	l	p			
0	66	1	1	25	0	80	2	1	35	0.79666	1780	8.274
2	56	2	2	20	1	80	1	1	10	0.795	1637	5.508
0	41	0	0	5	1	80	3	0	15	0.789	1547	5.8478
3	80	3	3	35	1	80	2	0	10	0.880686	1258	6.15398

Table 3.3: Results with `tunerTimeout` = 3600 seconds

Initial configuration					Final best configuration					Training quality	Number of runs	Test quality
F	P	f	l	p	F	P	f	l	p			
0	66	1	1	25	0	80	0	1	25	0.815	384	5.8191
0	66	1	1	25	1	80	1	1	35	0.737	452	6.267
0	66	1	1	25	1	56	0	1	35	1.03	371	9.056
0	66	1	1	25	0	76	0	1	20	0.814	385	4.915
0	66	1	1	25	0	80	3	1	20	0.76	469	5.417
2	56	2	2	20	0	41	0	1	10	0.919	239	18.364
2	56	2	2	20	0	56	1	1	20	0.819	407	5.409
2	56	2	2	20	1	80	1	1	35	0.772	457	5.43
2	56	2	2	20	1	80	0	1	10	0.858	504	5.566
2	56	2	2	20	0	80	1	1	10	0.7845	562	18.944
0	41	0	0	5	0	41	1	0	10	0.9749	367	5.97813
0	41	0	0	5	0	41	1	0	10	0.885	450	5.706
0	41	0	0	5	0	41	1	0	10	0.906	335	18.707
0	41	0	0	5	0	41	1	0	10	0.995	335	19.558
0	41	0	0	5	0	41	0	0	5	0.855	404	5.686
3	80	3	3	35	0	66	3	1	25	0.9118	230	26.585
3	80	3	3	35	0	80	1	1	10	0.732	310	7.875
3	80	3	3	35	0	80	0	1	20	0.816	303	7.2896
3	80	3	3	35	1	80	3	1	35	0.821	327	6.812
3	80	3	3	35	0	80	0	1	30	0.9203	443	5.401

Table 3.4: Results with `tunerTimeout` = 18000 seconds

Initial configuration					Final best configuration					Training quality	Number of runs	Test quality
F	P	f	l	p	F	P	f	l	p			
0	10	0	0	10	0	40	7	0	50	0.883188	936	6.3191
0	10	0	0	10	0	80	2	1	40	0.774659	1584	5.45674
0	10	0	0	10	0	40	2	0	10	0.96885	1104	6.82643
4	60	4	4	40	0	60	8	1	40	0.90358	1566	5.48127
4	50	4	4	40	0	80	5	1	20	0.78536	1662	11.5649
3	50	4	2	30	0	90	6	1	70	0.79440	1395	5.08108
0	90	0	0	10	1	90	6	1	10	0.859569	1379	5.4286
0	90	0	0	10	1	90	6	1	30	0.80738	1117	5.47126
8	90	8	8	60	0	80	5	1	10	0.834934	1384	5.5377
5	30	2	3	60	0	90	1	0	20	0.862013	1707	5.21837
3	20	2	4	60	0	80	6	1	10	0.805604	1630	5.4467
6	70	1	3	50	0	80	5	1	10	0.792600	1344	5.46558
6	40	1	3	30	1	80	7	0	20	0.822703	1977	5.41185

Table 3.5: Default configuration runtimes (secs)

37.210	411.300	112.510	171.000	73.770
327.880	214.910	124.910	482.740	530.440
212.660	99.370	287.400	533.540	**18.410**
197.290	1016.950	110.230	566.480	**1362.010**
94.860	819.700	434.460	620.600	95.920
80.580	333.370	121.590	489.700	248.370
mean: 341.005333				

Table 3.6: ParamILS configuration runtimes (secs)

154.460	264.530	169.840	26.990	108.790
550.210	104.900	31.100	**9.870**	1242.900
678.760	475.570	201.200	622.410	297.960
526.930	375.620	293.380	598.850	350.270
540.290	252.940	673.630	423.030	589.210
32.080	254.640	**2034.020**	571.100	207.090
mean: 422.085667				

Table 3.7: Results with `tunerTimeout` = 1800 seconds

Initial configuration					Final best configuration					Training quality	Number of runs	Test quality
F	P	f	l	p	F	P	f	l	p			
0	10	0	0	5	2	90	2	2	5	0.0493699	957	5.8461
0	10	0	0	5	0	90	5	2	5	0.0509388	1783	6.52742
0	10	0	0	5	0	90	5	2	5	0.049901	1759	5.21828
3	40	4	4	30	3	90	5	2	30	0.053974	856	6.3539
4	50	3	5	20	4	90	5	2	20	0.0500355	2000	5.4047
4	60	5	3	50	4	60	5	3	50	0.0520575	2000	6.09106
8	90	8	8	70	8	80	4	2	70	0.052685	550	3.85682
8	90	8	8	70	8	80	4	2	70	0.054104	536	4.17855
8	90	8	8	70	8	80	4	2	70	0.0497819	1284	3.90945
3	10	1	6	60	3	90	5	2	60	0.054934	2000	6.81675
5	70	6	1	10	5	90	4	2	10	0.0499895	2000	4.07365
1	30	5	7	5	1	90	4	2	5	0.0525747	1237	2.70091
7	80	2	0	70	7	90	5	2	70	0.0502264	212	5.2637

Table 3.8: Default configuration runtimes (secs)

452.980	91.420	31.510	**827.860**	96.670
635.030	295.830	272.360	151.040	170.660
183.550	161.340	91.240	426.470	62.020
138.090	236.030	**2.850**	187.240	21.510
165.370	90.440	195.580	15.390	229.720
170.840	174.210	30.520	6.570	115.880
mean: 191.007				

Table 3.9: ParamILS configuration runtimes (secs)

546.260	263.230	17.200	29.220	495.940
237.340	187.760	**7.810**	43.120	94.370
59.930	128.690	247.810	265.010	231.260
209.640	465.340	21.840	8.740	**1264.610**
57.700	122.890	450.610	229.580	174.540
414.080	402.250	91.150	677.190	58.640
mean: 250.125				

Table 3.10: Results with `tunerTimeout` = 18000 seconds

Initial configuration					**Final best configuration**					**Training quality**	**Number of runs**	**Test quality**
F	P	f	l	p	F	P	f	l	p			
2	43	0	0	7	0	45	1	0	5	0.0438025	952	3.13061
1	55	2	2	10	1	53	2	0	5	0.0434366	1120	6.8108005
1	55	2	2	10	1	53	2	0	5	0.0435660	2000	4.6961601

Table 3.11: Results with `tunerTimeout` = 18000 seconds

Initial configuration					**Final best configuration**					**Training quality**	**Number of runs**	**Test quality**
F	P	f	l	p	F	P	f	l	p			
2	85	6	1	35	2	85	6	1	35	0.0447855	2000	5.1182902
4	75	4	3	25	4	75	4	3	25	0.0458100	2000	3.4968102
3	95	5	2	40	3	95	5	2	40	0.0470930	2000	4.6591102

Part II

Parallel Oriented Solver Language

4

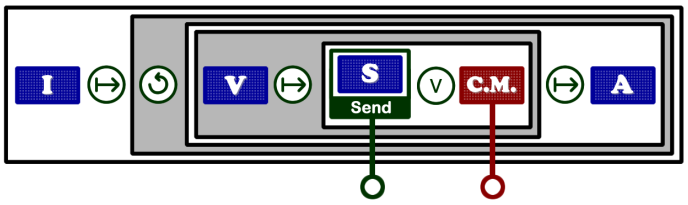
A Parallel-Oriented Language for Modeling Constraint-Based Solvers

In this chapter we introduce POSL *as our main contribution and a new way to solve CSPs. We resume its characteristics and advantages, and we get into details in the next sections. We describe a general outline to follow in order to build parallel solvers using* POSL*, and following each step is described in details.*

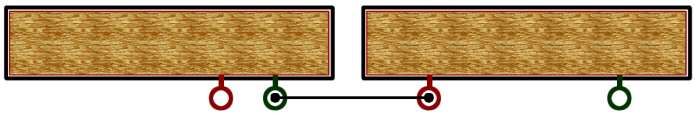
Contents

In this chapter we present the different steps to follow to build communicating parallel solvers with POSL. First of all, the algorithm that we have conceived to solve the target problem is modeled by decomposing it into small modules of computation. After that, they are implemented as separated *functions*. We name them *computation modules* (see Figure 4.1a, blue shapes). The coder's experience is crucial to find a good decomposition of its algorithm, because it will have an important impact in its future reuse and variability. The next step is to decide what information is interesting to receive from other solvers. This information is encapsulated into another kind of component called *communication module*, allowing data transmission among solvers (see Figure 4.1a, red shapes). In a third stage, is to glue the modules through POSL's inner language (the reader can see an example in Appendix [...]) to create independent solvers. The parallel-oriented language based on operators that POSL provides (see Figure 4.1b, green shapes) allows not only the information exchange, but also executing components in parallel. In this stage, the information that is interesting to share with other solvers, is sent using operators. After that, we can connect them using *communication operators*. We call this final entity a *solvers set* (see Figure 4.1c).

(a) Creating POSL's modules



(b) Assembling modules using POSL's operators



(c) Connecting POSL solvers to create *communication strategies*

Figure 4.1: Solver construction process using POSL

Once the solvers set is ready, the last step is to model the problem to solve. To do so, the user must follow the framework specification to implement the benchmark, respecting some requirements. The most important one is to implement a *cost function* computing the cost for a given configuration, i.e., an integer indicating how much the configuration violates the set of constraints. This integer equals zero if the configuration is a solution.

4.1 Modeling the target benchmark

Target problems are modelized in POSL using the C++ programing language. POSL provides the class **Benchmark** to inherit from. This class does not any method to override or implement, but in its constructor it receives three objects, instances from classes that the user must create, inheriting from **SolutionCostStrategy**, **RelativeCostStrategy** and **ShowStrategy** respectively. In these classes we write the most important functionalities of the benchmark model.

SolutionCostStrategy: In this class is implemented the strategy to compute the *cost* of a configuration. POSL is based on improving step by step an initial configuration, tacking into account a *cost function* provided by the user through the model (implementing the function *solutionCost(dots)*). The kind of problems that POSL solves are the *Constraint Satisfaction Problems*, so this *cost function* must be a function returning an integer taking into account the problem constraints. Given a configuration s, the *cost function*, as a mandatory rule, must return 0 if and only if s is a solution of the problem, i.e., s aims all the problem constraints. An example of *cost function* can be returning the number of violated constraints. However, the most ..... is the function cost, the better the behavior of POSL is leading to the solution.

The method to implement in this class is:

- `int solutionCost(std::vector<int> & c)` → It computes the cost of a given configuration (`c`).

RelativeCostStrategy: In this classe the user implements the strategy to compute the *cost* of a given configuration, with respect to another: the current one. If the user is able to compute the cost of a configuration, by knowing the performed changes with respect to the current configuration, the search process becomes more efficient, because this function is very often executed.

The methods to implement in this class are:

- `void initializeCostData(std::vector<int> & c)` → Initializes the information related to the cost (auxiliary data structures, the current configuration (`c`), the current cost, etc.)
- `void updateConfiguration(std::vector<int> & c)` → Updates the information related to the cost.

- `int relativeSolutionCost(std::vector<int> & c)` → Returns the relative cost of the configuration `c` with respect to the current configuration.
- `int currentCost()` → Property that returns the cost of the current configuration.
- `int costOnVariable(int variable_index)` → Returns a measure of how much some variable is contributing to the total cost of a configuration. AMPLIAR
- `int sickestVariable()` → Returns the variable contributing more to the cost.

SolutionCostStrategy: This class represents the way a benchmark shows a configuration, in order to be clearer and to give more information about the structure. For example, a configuration of the instance 3–3–2 of the *Social Golfers Problem* (see bellow for more details about this benchmark) can be written as follows:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 3, 4, 5, 6, 7, 8, 9, 1, 2]
```

But this text is very difficult to read if the instance is bigger. For that reason, the user should implement this class in order to give more details and make easier the configuration read. For example, for the same instance of the problem, a solution is presented as follows:

```
Golfers: players-3, groups-3, weeks-2
6        8        7
1        3        5
4        9        2
--
7        2        3
4        8        1
5        6        9
--
```

The method to implement in this class is:

- `std::string showSolution(std::shared_ptr<Solution> s)` → Returns a string to write in the standard output.

4.2 First Stage: Creating POSL's modules

There exist two types of basic modules in POSL: *computation modules* and *communication modules*. A *computation module* is a function receiving an input, then executes an internal algorithm, and returns an output. A *communication module* is also a function receiving and returning information, but in contrast, the *communication module* can receive information from two different ways: through parameter or from outside, i.e. by communicating with a module from another solver.

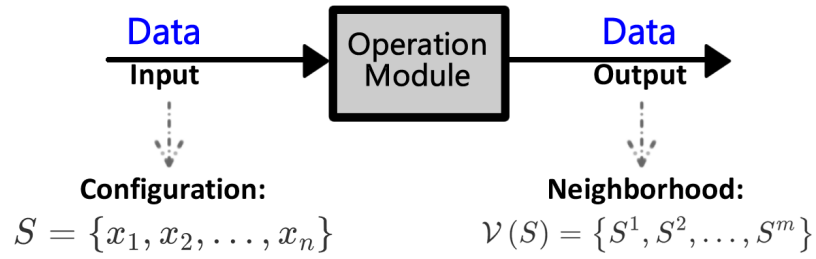


Figure 4.2: An example of a computation module computing a neighborhood

4.2.1 Computation Module

A *computation module* is the most basic and abstract way to define a piece of computation. It is a function receiving an input, then executing an internal algorithm, and returning an output. The input and output types will characterize the computation module signature. It can be dynamically replaced by or combined with other computation modules, since they can be shared among solvers working in parallel. They are joined through *abstract solvers*.

Definition 6 ***(Computation Module)*** *A computation module Cm is a mapping defined by:*

$$Cm : D \rightarrow I \tag{4.1}$$

D and I can be either a set of configurations, or set of sets of configurations, or a set of values of some data type, etc.

Consider a local search meta-heuristic solver. One of its *computation modules* can be the function returning the set of configurations composing the neighborhood of a given configuration:

$$Cm_{neighborhood} : D_1 \times D_2 \times \cdots \times D_n \rightarrow 2^{D_1 \times D_2 \times \cdots \times D_n}$$

where D_i represents the definition domains of each variable of the input configuration.

Figure 4.2 shows an example of computation module: it receives a configuration S and then computes the set V of its neighbor configurations $\{S^1, S^2, \ldots, S^m\}$.

4.2.2 Communication modules

A *communication module* is also a function receiving and returning its information, but in contrast, the *communication module* can receive information from two different ways: through input or from outside, i.e., by communicating with a module from another solver. A *communication module* is the component in charge of the information reception in the communication between solvers (we will talk about information transmission in the next section). They can interact with *computation modules* through operators (see Figure 4.3).

A *communication module* can receive two types of information, always coming from an external solver: data or *computation modules*. It is important to notice that when we are talking about sending/receiving *computation modules*, we mean sending/receiving only required information to identify it and being able to instantiate it.

In order to distinguish from the two types of *communication modules*, we will call Data Communication Module the *communication module* responsible for the data reception (Figure 4.3a), and Object Communication Module the one responsible for the reception and instantiation of *computation modules* (Figure 4.3b).

Definition 7 ***(Data Communication Module)*** *A* Data Communication Module *Ch is a component that produces a mapping defined as follows:*

$$Ch : U \rightarrow I \tag{4.2}$$

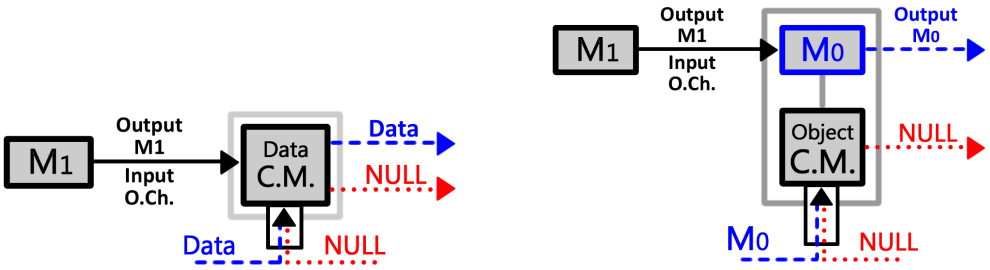
It returns the information I coming from an external solver, no matter what the input U is.

Definition 8 ***(Object Communication Module)*** *If we denote by $\mathbb{M}$ the space of all the computation modules defined by Definition 4.1, then an* Object Communication Module *Ch is a component that produces a computation module coming from an external solver as follows:*

$$Ch : \mathbb{M} \rightarrow \mathbb{M} \tag{4.3}$$

Users can implement new computation and connection modules but POSL already contains many useful modules for solving a broad range of problems.

Due to the fact that *communication modules* receive information coming from outside and have no control on them, it is necessary to define the *NULL* information, to denote the absence of any information. If a Data Communication Module receives a piece of information, it is returned automatically. If a Object Communication Module receives a *computation module*, the latter is instantiated and executed with the *communication module*'s input, and



(a) Data *communication module* **(b)** Object *communication module*

Figure 4.3: Communication module

its result is returned. In both cases, if no available information exists (no communications performed), the *communication module* returns the *NULL* object.

4.3 Second Stage: Assembling POSL's modules

Modules mentioned above are defined respecting the signature of some predefined abstract module. For example, the module showed in Figure 4.2 is a *computation module* based on an abstract module that receives a configuration and returns a neighborhood. In that sense, an example of a concrete *computation module* (or just *computation module* for simplification sake) can be a function receiving a configuration, and returns a neighborhood composed of N configurations only differing from one value to the input configuration.

In this stage an *abstract solver* is coded through POSL. It takes abstract modules as *parameters* and combines them with operators. Through the abstract solver, we can also decide what information to be sent to other solvers, by using some operators to send the result of a computation module (see below). Following we present a formal and more detailed presentation of POSL's operators specification.

The *abstract solver* is the solver's backbone. It joins the *computation modules* and the *communication modules* coherently. It is independent from the *computation modules* and *communication modules* used in the solver. It means they can be changed or modified during the execution, without altering the general algorithm, but still respecting the main structure. Each time we combine some of them using POSL's operators, we are creating a *compound module*. Following we define formally the concept of *module* and *compound module*.

Definition 9 *We call* **module** *to (and it is denoted by the letter $\mathcal{M}$):*

a) A computation module or

b) A communication module or

c) $[\mathcal{M}_1\ OP\ \mathcal{M}_2]$*: The composition of two modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *to be executed sequentially, one after the other, and returning an output depending on the nature of the operator* OP*; or*

d) $[\![\mathcal{M}_1\ OP\ \mathcal{M}_2]\!]$*: The composition of two modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *to be executed and returning an output depending on the nature of the operator* OP*. These two modules will be executed in parallel if and only if* OP *supports parallelism, (i.e. some modules will be executed sequentially although they were grouped this way); or sequentially otherwise.*

We denote the space of the modules by $\mathbb{M}$*. We call compound modules to the composition of modules described in c) and d).*

For better understand the Definition 9, Figure 4.4 shows graphically a *compound module*.

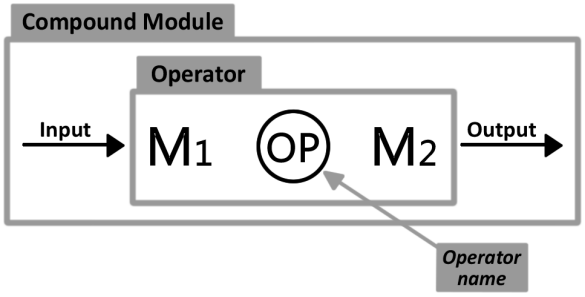


Figure 4.4: A *compound module*

We define an *abstract solver* as a free-context grammar as follows:

Definition 10 (POSL's Grammar) $G_{POSL} = (\mathbf{V}, \Sigma, \mathbf{S}, \mathbf{R})$*, where:*

a) $\mathbf{V} = \{CM, OP\}$ *is the set of variables,*

b) $\Sigma = \left\{\alpha, \beta, be, [,], [\![,]\!]_p, (,), \{, \}, (\!|, |\!)^m, |\!)^o, \textcircled{\scriptstyle\mapsto}, \textcircled{\scriptstyle ?}, \textcircled{\scriptstyle \circlearrowleft}, \textcircled{\scriptstyle \rho}, \textcircled{\scriptstyle \vee}, \textcircled{\scriptstyle \wedge}, \textcircled{\scriptstyle M}, \textcircled{\scriptstyle m}, \textcircled{\scriptstyle \downarrow}, \textcircled{\scriptstyle \cup}, \textcircled{\scriptstyle \cap}\right\}$ *is the set of terminals,*

c) $\mathbf{S} = \{CM\}$ *is the set of start variables,*

d) and the set of rules $\mathbf{R} =$

$$CM \longmapsto \alpha \mid \beta \mid (\!|CM|\!)^o \mid (\!|CM|\!)^m \mid [OP] \mid [\![OP]\!]_p$$
$$OP \longmapsto CM\textcircled{\scriptstyle\mapsto}CM \mid CM\textcircled{\scriptstyle ?}CM \mid CM\textcircled{\scriptstyle \rho}CM \mid CM\textcircled{\scriptstyle \vee}CM \mid CM\textcircled{\scriptstyle \wedge}CM$$
$$OP \longmapsto CM\textcircled{\scriptstyle M}CM \mid CM\textcircled{\scriptstyle m}CM \mid CM\textcircled{\scriptstyle \downarrow}CM \mid CM\textcircled{\scriptstyle \cup}CM \mid CM\textcircled{\scriptstyle \cap}CM$$
$$OP \longmapsto CM \circlearrowleft (be)\,\{CM\}$$

We would like to explain some of the concepts presented in Definition 10:

- The variable CM, as well as OP are two entities very important in the language, as can be seen in the grammar. We name them *compound module* and *operator* respectively.
- The terminals α and β represent a *computation module* and a *communication module* respectively,
- The terminal *be* is a boolean expression.
- The terminals $[\ ], [\![\ ]\!]_p$ are symbols for grouping and defining the way of how the involved *compound module*s are executed. Depending on the nature of the operator, they can be executed sequentially or in parallel:
  a) [OP]: The involved operator is executed sequentially.
  b) $[\![\text{OP}]\!]_p$: The involved operator is executed in parallel if and only if OP supports parallelism. Otherwise, an exception is threw.
- The terminals (and) are symbols for grouping the boolean expression in some operators.
- The terminals { and } are symbols for grouping *compound module*s in some operators.
- The terminals $(\!|.|\!)^m, (\!|.|\!)^o$, are operators to send information to other solvers (explained bellow).
- The rest of terminals are POSL operators.

Following we define POSL operators. In order to group modules, like in Definition $9(c)$ and $9(d)$, we will use $|.|$ as generic grouper.

Definition 11 (Operator Sequential Execution) *Let the modules*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be, where $\mathcal{I}_1 \subseteq \mathcal{D}_2$. *Then, the operation* $\left|\mathcal{M}_1 \textcircled{\mapsto} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{seq}$ *as result of the execution of* $\mathcal{M}_1$ *followed by* $\mathcal{M}_2$*:*

$$\mathcal{M}_{seq} : \mathcal{D}_1 \rightarrow \mathcal{I}_2$$

This operator is an example of an operator that does not support the execution of its involved *compound modules* in parallel, because the input of the second *compound module* is the output of the first one.

The following operator is useful to execute modules sequentially creating bifurcations, subject to some boolean condition:

Definition 12 (Operator Conditional Execution) *Let the modules*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Then, the operation* $\left|\mathcal{M}_1 \textcircled{?}_{<cond>} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{cond}$ *as result of the sequential execution of* $\mathcal{M}_1$ *if* $< cond >$ *is* **true** *or* $\mathcal{M}_2$ *otherwise:*

$$\mathcal{M}_{cond} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

We can execute modules sequentially creating also cycles.

Definition 13 (Operator Cyclic Execution) *Let the module* $\mathcal{M} : \mathcal{D} \rightarrow \mathcal{I}$ *be, where* $\mathcal{I} \subseteq \mathcal{D}$. *Then, the operation* $|\circlearrowleft_{<cond>} \mathcal{M}|$ *defines the compound module* $\mathcal{M}_{cyc}$ *as result of the sequential execution of* $\mathcal{M}$ *repeated while* $< cond >$ *remains* **true***:*

$$\mathcal{M}_{cyc} : \mathcal{D} \rightarrow \mathcal{I}$$

Definition 14 (Operator Random Choice) *Let the modules*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be, where $\mathcal{D}_1 \subset \mathcal{D}_2$, *and a probability* ρ. *Then, the operation* $\left|M_1 \textcircled{\rho} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{rho}$ *that executes and returns the output of* $\mathcal{M}_1$ *following the probability* ρ, *or executes and returns the output of* $\mathcal{M}_2$ *following* $(1 - \rho)$*:*

$$\mathcal{M}_{rho} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

The following operator is very useful if the user needs to use a *communication module* into an *abstract solver*. As we explained before, if a *communication module* does not receive any information from other solver, it returns *NULL*. This may cause the undesired termination of the solver if we do not pay attention correctly. Next, we introduce the operator **Operator Not** *NULL* **Execution** and we propose an example to illustrate how to use it in practice.

Definition 15 (Operator Not *NULL* **Execution)** *Let the modules*

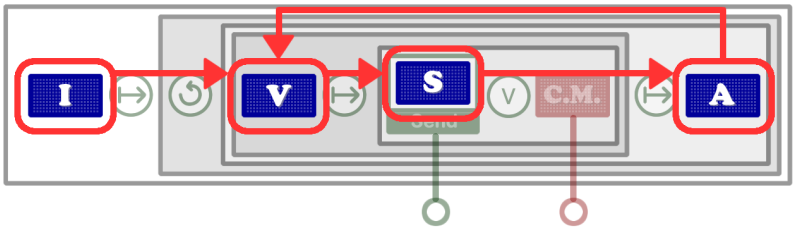
a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

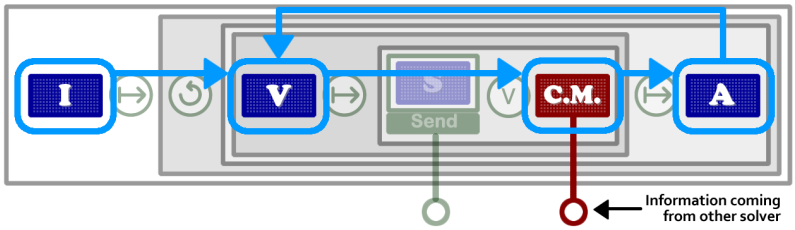
be, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Then, the operation* $\left|\mathcal{M}_1 (\vee) \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{non}$ *that executes* $\mathcal{M}_1$ *and returns its output if it is not NULL, or executes* $\mathcal{M}_2$ *and returns its output otherwise:*

$$\mathcal{M}_{non} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Figure 4.5 shows how to combine a connection module with the *computation module* `S` through the operator $(\vee)$. Here, the computation module `S` will be executed as long as the *communication module* remains *NULL*, i.e., there is no information coming from outside. This behavior is represented in Figure 4.5a by orange lines. If some data has been received through the connection module, it is executed instead of the module `S`, represented in Figure 4.5b by blue lines.



(a) The solver executes the computation module **S** if no information is received through the connection module



(b) The solver uses the information coming from an external solver

Figure 4.5: Two different behaviors in the same solver

This operator is *short-circuit*. It means that if the first argument (module) does not return *NULL*, the second will not be executed. POSL provides another operator the same functionality but not *short-circuit*:

Definition 16 (Operator *BOTH* Execution) *Let the modules*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Then, the operation* $\left|\mathcal{M}_1 (\wedge) \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{both}$ *that executes both* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, then returns the output of* $\mathcal{M}_1$ *if it is not NULL, or the output of* $\mathcal{M}_2$ *otherwise:*

$$\mathcal{M}_{both} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

In the following Definitions, the concepts of *cooperative parallelism* and *competitive parallelism* are implicitly included. We say that cooperative parallelism exists when two or more processes are running separately, they are independent, and the general result will be some combination of the results of all the involved processes (e.g. Definitions 17 and 18). On the other hand, competitive parallelism arise when the general result is the solution of the first process terminates (e.g. Definition 19).

Definition 17 (Operator Minimum) *Let the modules*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Let also* o_1 *and* o_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *respectively, and assume that there exist some order criteria between them. Then, the operation* $\left|\mathcal{M}_1 \textcircled{m} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{min}$ *that executes* $\mathcal{M}_1$ *and returns* $\min\{o_1, o_2\}$*:*

$$\mathcal{M}_{min} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Similarly we define the operator **Maximum**:

Definition 18 (Operator Maximum) *Let the modules*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Let also* o_1 *and* o_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *respectively, and assume that there exist some order criteria between them. Then, the operation* $\left|\mathcal{M}_1 \textcircled{M} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{max}$ *that executes* $\mathcal{M}_1$ *and returns* $\max\{o_1, o_2\}$*:*

$$\mathcal{M}_{max} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Definition 19 (Operator Race) *Let the modules*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$ *and* $\mathcal{I}_1 \subset \mathcal{I}_2$*. Then, the operation* $\left|\mathcal{M}_1 \textcircled{\downarrow} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{race}$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns the output of the module ending first:*

$$\mathcal{M}_{race} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

The operators presented in Definitions 17, 18 and 19 are very useful in terms of sharing not only information between solvers, but also *behaviors*. If one of the operands is a *communication module* it can receive an external solver's *computation module*, providing the opportunity to instantiate it in the current solver. The operator either instantiates that module if it is not null, and executes it; or it executes the other operand module otherwise.

Some others operators can be useful dealing with *sets*.

Definition 20 (Operator Union) *Let the modules*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$*,*

be, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Let also* V_1 *and* V_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *respectively. Then, the operation* $\left|\mathcal{M}_1 \textcircled{\cup} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{\cup}$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns* $V_1 \cup V_2$*:*

$$\mathcal{M}_{\cup} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Similarly we define the operators **Intersection** and **Subtraction**:

Definition 21 (Operator Intersection) *Let the modules*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$*,*

be, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$*. Let also* V_1 *and* V_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *respectively. Then, the operation* $\left|\mathcal{M}_1 \textcircled{\cap} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{\cap}$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns* $V_1 \cap V_2$*:*

$$\mathcal{M}_{\cap} : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Definition 22 (Operator Subtraction) *Let the modules*

a) $\mathcal{M}_1 : \mathcal{D}_1 \rightarrow \mathcal{I}_1$ *and*

b) $\mathcal{M}_2 : \mathcal{D}_2 \rightarrow \mathcal{I}_2$,

be, where $\mathcal{D}_1 \subseteq \mathcal{D}_2$. *Let also* V_1 *and* V_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *respectively. Then, the operation* $\left|\mathcal{M}_1 \ominus \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_-$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$, *and returns* $V_1 - V_2$*:*

$$\mathcal{M}_- : \mathcal{D}_1 \cap \mathcal{D}_2 \rightarrow \mathcal{I}_1 \cup \mathcal{I}_2$$

Now, we define the operators that allow us to send information to outside, i.e. other solvers. We can send two types of information: i) we can execute the *computation module* and send its output, or ii) we can send the *computation module* itself. . This utility is very useful in terms of sharing behaviors between solvers.

Definition 23 (Sending Data Operator) *Let the module* $\mathcal{M} : \mathcal{D} \rightarrow \mathcal{I}$ *be. Then, the operation* $|(\!|\mathcal{M}|\!)^o|$ *defines the compound module* $\mathcal{M}_{sendD}$ *that executes the module* $\mathcal{M}$ *then return and sends its output to outside:*

$$\mathcal{M}_{sendD} : \mathcal{D} \rightarrow \mathcal{I}$$

Similarly we define the operator **Send Module**:

Definition 24 (Sending Module Operator) *Let the module* $\mathcal{M} : \mathcal{D} \rightarrow \mathcal{I}$ *be. Then, the operation* $|(\!|\mathcal{M}|\!)^m|$ *defines the compound module* $\mathcal{M}_{sendM}$ *that executes the module* $\mathcal{M}$, *then returns its output and sends the module itself to outside:*

$$\mathcal{M}_{sendM} : \mathcal{D} \rightarrow \mathcal{I}$$

Once all desired abstract modules are linked together with operators, we obtain what we call an *abstract solver*. To implement a concrete solver from an *abstract solver*, one must instantiate each abstract module with a concrete one respecting the required signature. From the same *abstract solver*, one can implement many different concrete solvers simply by instantiating abstract modules with different concrete modules.

Including the topic of ROOT *compound module*

An *abstract solver* is defined as follow: after declaring an **abstract solver** and its name, the first line defines the list of signatures of *computation modules*, the second one the list of signatures of *communication modules*, then the algorithm of the solver is defined is the

solver's body, between **begin** and **end**. For instance, Algorithm 3 illustrate the abstract solver corresponding to Figure 4.1b.

Algorithm 3: POSL pseudo-code for the *abstract solver* presented in Figure 4.1b

```
1 abstract solver as_01
2 computation : I, V, S, A
3 connection: C.M.
4     begin
5         I (↦)
6         [↻ (Itr % K_2) begin
7             V (↦) [C.M. (∨) (|S|)^o] (↦) A
8         end]
9     end
```

4.4 Third Stage: Creating POSL solvers

With *computation* and *communication modules* composing an *abstract solver*, we can create solvers by instantiating *modules*. This is simply done by specifying that a given **solver** will **implements** a given *abstract solver*, followed by the list of *computation* then *communication modules* matching signatures required by the *abstract solver*. Algorithm 4 gives an example of implementation of Algorithm 3 by instantiating modules shown in Figure 4.5.

Algorithm 4: An instantiation of the *abstract solver* presented in Algorithm 3

```
1 solver solver_01 implements as_01
2 computation : I_rand, V_std, S_best, A_alw
3 connection: CM_last
```

Algorithm 4 is just an example of a solver instantiation, using some *computation modules* provided by POSL provides, that we use and explain in details later in this document:

- I_{rand} creates a random configuration.
- V_{std} creates a neighborhood of a given configuration, changing one element at once.
- S_{best} selects the configuration of a neighborhood with the lowest cost.
- A_{alw} always accepts the incoming configuration.
- CM_{last} returns the last arrived configuration, if at the time of its execution, there is more than one waiting.

4.5 Forth Stage: Connecting the solvers

We call *solver set* the pool of (concrete) solvers we plan to use in parallel to solve a problem. Once we have our solvers set, the last stage is to connect the solvers each others. Up to here, solvers are disconnected, but they have everything to establish the communication. POSL provides to the user a platform to easily define cooperative strategies that solvers must follow.

Following, we present two important concepts before we can formalize the *communication operators*.

Definition 25 (Communication Jack) *Let a solver $\mathcal{S}$ be. Then, the operation $\mathcal{S} \cdot \mathcal{M}$ opens an outgoing connection from the solver $\mathcal{S}$ sending to the outside either a) the output of $\mathcal{M}$ if it is affected by a sending data operator presented in Definition 23, or b) $\mathcal{M}$ itself, if it is affected by a sending module operator presented in Definition 24.*

Definition 26 (Communication Outlet) *Let a solver $\mathcal{S}$ be. Then, the operation $\mathcal{S} \cdot \mathcal{CM}$ opens an ingoing connection to the solver $\mathcal{S}$ receiving from the outside either a) the output of some computation module if $\mathcal{CM}$ is a data communication module, or b) a computation module if $\mathcal{CM}$ is an object communication module.*

The communication is established by following the next rules guideline:

a) Each time a solver sends any kind of information by using a *sending* operator, it creates a *communication jack*.

b) Each time a solver defines a *communication module*, it creates a *communication outlet*.

c) Solvers can be connected each others by linking *communication jacks* to *communication outlets*.

Following, we define the *connection operators* that POSL provides.

Definition 27 (Connection Operator One-to-One) *Let*

a) the list $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ of communication jacks, and

b) the list $\mathcal{O} = [\mathcal{Z}_0 \cdot \mathcal{CM}_0, \mathcal{Z}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{Z}_{N-1} \cdot \mathcal{CM}_{N-1}]$ of communication outlets

be. Then, the operation

$$\mathcal{J} \ \textcircled{\rightarrow} \ \mathcal{O}$$

Connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ *with the corresponding communication outlet* $\mathcal{Z}_i \cdot \mathcal{CM}_i \in \mathcal{O}$, $\forall 0 \leq i \leq N-1$ *(see Figure 4.6a).*

Definition 28 (Connection Operator One-to-N) *Let*

a) the list $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *of communication jacks, and*

b) the list $\mathcal{O} = [\mathcal{Z}_0 \cdot \mathcal{CM}_0, \mathcal{Z}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{Z}_{M-1} \cdot \mathcal{CM}_{M-1}]$ *of communication outlets*

be. Then, the operation

$$\mathcal{J} \ \textcircled{\rightsquigarrow} \ \mathcal{O}$$

Connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ *with every communication outlet* $\mathcal{Z}_j \cdot \mathcal{CM}_j \in \mathcal{O}$, $\forall 0 \leq i \leq N-1$ *and* $0 \leq j \leq M-1$ *(see Figure 4.6b).*

Definition 29 (Connection Operator Ring) *Let*

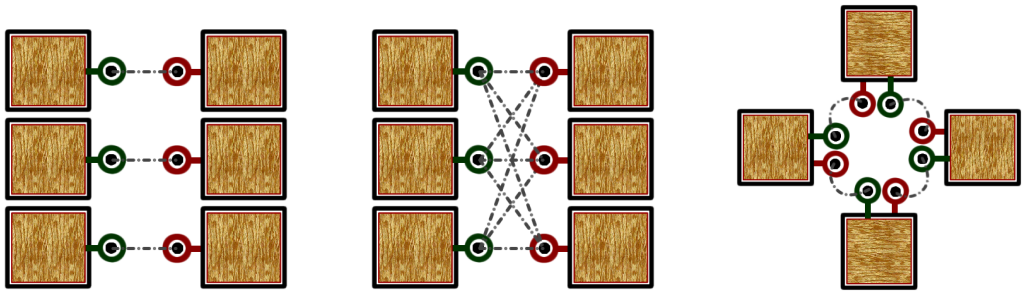
a) the list $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *of communication jacks, and*

b) the list $\mathcal{O} = [\mathcal{S}_0 \cdot \mathcal{CM}_0, \mathcal{S}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{CM}_{N-1}]$ *of communication outlets*

be. Then, the operation

$$\mathcal{J} \ \textcircled{\leftrightarrow} \ \mathcal{O}$$

Connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ *with the corresponding communication outlet* $\mathcal{Z}_{(i+1)\%N} \cdot \mathcal{CM}_{(i+1)\%N} \in \mathcal{O}$, $\forall 0 \leq i \leq N-1$ *(see Figure 4.6c).*



(a) Communication 1 to 1 **(b)** Communication 1 to N **(c)** Cyclic communication

Figure 4.6: Graphic representation of communication operators

POSL allows also declare not communicating solvers to be executed in parallel, declaring only the list of solvers names:

$$[\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_{N-1}]$$

When we apply a connection operator (op) between a *communication jacks* list $\mathcal{J}$ and a *communication outlets* list $\mathcal{O}$, internally we are assigning an *abstract computation unit* (typically a thread) to each solver that we declare in each list. This assignation receives the name of *Solver Scheduling*. Before running the *solver set* this *abstract unit of computation* is just an integer $\tau \in [0..N]$ as unique identifier for each solver. When the *solver set* is launched, the solver with the identifier τ runs into the computation unit τ. This identifier assignation remains independent of the real availability of resources of computation. It just take into account the user declaration. It means that, if the user declares 30 solvers (15 senders and 15 receivers) and the *solver set* is launched using 20 cores, only the firsts 20 solvers will be executed, and in consequence, there will be 10 solvers sending information to nowhere. Users should take this into account at the time of declaring the *solver set*.

The connection process depends on the applied conection operator. In each case the goal is to assign to the sending operator ($(\!|.|\!)^o$ or $(\!|.|\!)^m$) into the *abstract solver*, the identifier of the solver (or solvers, depending on the connection operator) where the information will be sent. Algorithm 5 presents the connection process.

Algorithm 5: Scheduling and connection main algorithm

```
input  : 𝒥 list of communication jacks,
         𝒪 list of communication outlets
1 while no available jacks or outlets do
2     S_jack ← GetNext(𝒥)
3     R_oulet ← GetNext(𝒪)
4     S ← GetSolverFromConnector(S_jack)
5     R ← GetSolverFromConnector(R_oulet)
6     Schedule(S)
7     R_id ← Schedule(R)
8     Connect(root(S),S_jack, R_id)
9 end
```

In Algorithm 5:

- `GetNext(...)` Returns the next available solver-jack (or solver-outlet) in the list, depending on the connection operator, e.g., for the connection operator One-to-N, each *communication jack* in $\mathcal{J}$ must be connected with each *communication outlet* in $\mathcal{O}$.
- `GetSolverFromConnector(...)` Returns the solver name given a connector declaration.
- `Schedule(...)` Schedule a solver and returns its identifier.
- `Root(...)` Returns the *root compound module* of a solver.
- `Connect(...)` Searches the *computation module* S_{jack} recursively inside the *root compound module* of S and places the identifier R_{id} into its list of destination solvers.

4.5.1 Solver name space expansion

Includeing an introduction here. POSL provides to ways of space name expansion:

Solver name expansion - Using an integer K to denote how many times the solver name S will appear in the declaration. $[\dots S_i \cdot \mathcal{M}(K), \dots]$ expands as $\left[\dots S_i \cdot \mathcal{M}, S_i^2 \cdot \mathcal{M}, \dots S_i^K \cdot \mathcal{M} \dots\right]$
and all new solvers $S_i^j, j \in [2..K]$ are created using the same solver declaration of solver S_i.

Connection declaration expansion - Using an integer K to denote how many times the connection will be repeated in the declaration. Let a) the list $[\mathcal{S}_1 \cdot \mathcal{M}_1, \dots, \mathcal{S}_N \cdot \mathcal{M}_N]$ of *communication jacks*, b) the list $[\mathcal{R}_1 \cdot \mathcal{CM}_1, \dots, \mathcal{R}_M \cdot \mathcal{CM}_M]$ of *communication outlets*, and c) an connection operator $\textcircled{\text{op}}$ be. Then,

$$[\mathcal{S}_1 \cdot \mathcal{M}_1, \dots, \mathcal{S}_N \cdot \mathcal{M}_N] \ \textcircled{op} \ [\mathcal{R}_1 \cdot \mathcal{CM}_1, \dots, \mathcal{R}_M \cdot \mathcal{CM}_M] \, K$$

expands as

$$\begin{gathered}
[\mathcal{S}_1 \cdot \mathcal{M}_1, \dots, \mathcal{S}_N \cdot \mathcal{M}_N] \ \textcircled{op} \ [\mathcal{R}_1 \cdot \mathcal{CM}_1, \dots, \mathcal{R}_N \cdot \mathcal{CM}_N] \\
\left[\mathcal{S}_1^2 \cdot \mathcal{M}_1, \dots, \mathcal{S}_N^2 \cdot \mathcal{M}_N\right] \ \textcircled{op} \ \left[\mathcal{R}_1^2 \cdot \mathcal{CM}_1, \dots, \mathcal{R}_N^2 \cdot \mathcal{CM}_N\right] \\
\dots \\
\left[\mathcal{S}_1^K \cdot \mathcal{M}_1, \dots, \mathcal{S}_N^K \cdot \mathcal{M}_N\right] \ \textcircled{op} \ \left[\mathcal{R}_1^K \cdot \mathcal{CM}_1, \dots, \mathcal{R}_N^K \cdot \mathcal{CM}_N\right]
\end{gathered}$$

and all new solvers $S_i^k, i \in [1..N]$ and $R_j^k, j \in [1..M]$, with $k \in [2..K]$ are created using the same solver declaration of solvers S_i and R_j respectively.

4.6 Step-by-step POSL code example

In this section we explain how to create a solver using POSL through an example. POSL creates solvers based on local search meta-heuristics algorithms. These algorithms have a common structure: 1. they start by initializing some data structures (e.g. a *tabu list* for *Tabu Search* [34], a *temperature* for *Simulated Annealing* [32], etc.). 2. An initial configuration s is generated. 3. A new configuration s' is selected from the neighborhood $\mathcal{V}(s)$. 4. If s' is a solution for the problem P, then the process stops, and s' is returned. If not, the data

structures are updated, and s' is accepted or not for the next iteration, depending on some criterion.An example of such data structure can be the penalizing features of local optima defined by Boussaïd et al [31] in their algorithm *Guided Local Search*.

Abstract computation modules composing *local search meta–heuristics* are the following:

$\boxed{\textit{Abstract Computation module} - 1}$ I: Generating a configuration s

$\boxed{\textit{Abstract Computation module} - 2}$ V: Defining the neighborhood $\mathcal{V}(s)$

$\boxed{\textit{Abstract Computation module} - 3}$ S: Selecting $s' \in \mathcal{V}(s)$

$\boxed{\textit{Abstract Computation module} - 4}$ A: Evaluating an acceptance criteria for s'

To be more specific in our example, we describe come concrete *computation modules* that we can use:

$\boxed{\textit{Computation module} - 1}$ I_{rand}: Generates a random configuration s

$\boxed{\textit{Computation module} - 2}$ V_{1ch}: Defines the neighborhood $\mathcal{V}(s)$ changing only one element

$\boxed{\textit{Computation module} - 3}$ S_{best}: Selects the best configuration $s' \in \mathcal{V}(s)$ improving the current cost.

$\boxed{\textit{Computation module} - 4}$ $A_{a.i.}$: Evaluates an acceptance criteria for s'. We have chosen the classical module selecting the configuration with the lowest global cost, *i.e.*, the one which is likely the closest to a solution.

We can combine modules to create more complex ones. For example, in the example we use in this section, we want to apply some selection criteria, but if there is not improvement in the current configuration, we select a configuration randomly. To do so, we use the operator (?) to combine the *computation module* 3 with another:

$\boxed{\textit{Computation module} - 3.1}$ S_{rand}: Selects a random configuration $s' \in \mathcal{V}(s)$.

Let's make this solver a little bit more complex: in the time of applying the acceptance criteria, suppose that we want to receive a configuration from other solver, to compare it with ours and take it if its cost is lower. We can do that applying the operator (m) to combine the *computation module* 4 with a *communication module*:

$\boxed{\textit{Communication module} - 1:}$ $C.M.$: Receiving a configuration.

Figure 4.7 shows a graphic representation of the *abstract solver* that we create for our example. In this Figure all the modules are abstract, so we can instantiate them afterwards to create concrete solvers. Note that we also want to send the output of the *computation module A*, in order to share the current configuration with other solvers.

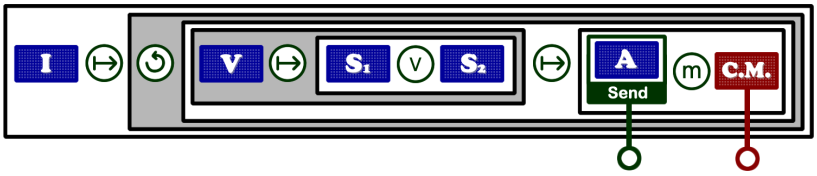


Figure 4.7: Graphic representation of an *abstract solver*

Algorithm 6 shows the POSL pseudo-code for the *abstract solver* described above, using predefined operators, and Algorithm 7 shows the concrete solver definition using the concrete *computation* and *communication modules* already presented. In the later, we have used the concrete *communication module* CM_{last}, that receives all the configuration sent to it, and selects the last arrived.

Algorithm 6: POSL pseudo-code for *abstract solver* presented in Figure 4.7

```
1 abstract solver as_example
2 computation : I, V, S1, S2, A
3 connection: C.M.
4     begin
5         I (↦)
6         [(↻) (Itr % K2) begin
7             [V (↦) [S1 (∨) S2]] (↦) [(|A|)^o (m) C.M.]
8         end]
9     end
```

Algorithm 7: An instantiation of the *abstract solver* in Algorithm 6

```
1 solver solver_example implements as_example
2 computation : I_rand, V_1ch, S_best, A_a.i.
3 connection: CM_last
```

Part III

Study and evaluation of POSL

5

Experiments design

In this chapter we expose all details about the process of evaluation of POSL*, i.e. all the experiments we perform. For each benchmark, we explain also the strategy (or strategies) used in the solving (evaluation) process.*

Contents

In this chapter we illustrate and analyze the versatility of POSL studying different ways to solve constraint problems based on local search meta-heuristics. We have chosen the *Social Golfers Problem*, the *N-Queens Problem*, the *Costas Array Problem* and the *Golomb Ruler Problem* as benchmarks since they are two challenging yet differently structured problems. In this chapter we explain the structure of POSL's solvers that we generated for experiments.

5.1 Solving the *Social Golfers Problem*

The *Social Golfers Problem* (*SGP*) consists in scheduling $n = g \times p$ golfers into g groups of p players every week for w weeks, such that two players play in the same group at most once. An instance of this problem can be represented by the triple $g - p - w$. This problem, and other closely related problems, arise in many practical applications such as encoding, encryption, and covering problems [121]. Its structure is very attractive, because it is very similar to other problems, like *Kirkman's Schoolgirl Problem* and the *Steiner Triple System*, so we can build efficient modules to solve a broad range of problems.

Here, we give the abstract solver designed for this problem as well as concrete computation modules composing the different solvers we have tested:

a) Generation module:

I: Generates a random configuration s, respecting the structure of the problem, *i.e.*, the configuration is a set of w permutations of the vector $[1..n]$.

b) Neighborhood modules:

V_{Std}: Defines the neighborhood $\mathcal{V}(s)$ swapping players among groups.

V_{AS}: Defines the neighborhood $\mathcal{V}(s)$ swapping the most culprit player with other players from the same week. It is based on the *Adaptive Search* algorithm.

c) Selection modules:

S_{First}: Selects the first configuration $s' \in \mathcal{V}(s)$ improving the current cost.

S_{Best}: Selects the best configuration $s' \in \mathcal{V}(s)$ improving the current cost.

S_{Rand}: Selects a random configuration $s' \in \mathcal{V}(s)$.

d) Acceptance module:

A: Evaluates an acceptance criteria for s'. We have chosen the classical module selecting the configuration with the lowest global cost, *i.e.*, the one which is likely the closest to a solution.

A very first experiment was performed to select the best neighborhood function to solve the problem, comparing a basic solver V_{Std}; a new solver using V_{AS}; and a combination of V_{Std} and V_{AS} by applying the operators (ρ), already introduced in the previous section, and $(\cup)$, that computes and returns the union of two neighborhoods. Algorithms 8, 9 and 10 present the *abstract solver* for each case, respectively.

Algorithm 8: Standard *abstract solver* for *SGP*

```
1 abstract solver as_union                          // Itr → number of iterations
2 computation : I, V, S, A
3 begin
4     [↺ (Itr < K1) begin
5         I (↦) [↺ (Itr % K2) begin V (↦) S (↦) A end]
6     end]
7 end
```

Algorithm 9: *Abstract solver* combining neighborhood functions using operator *RHO*

```
1 abstract solver as_union                          // Itr → number of iterations
2 computation : I, V1, V2, S, A
3 begin
4     [↺ (Itr < K1) begin
5         I (↦) [↺ (Itr % K2) begin [V1 (ρ) V2] (↦) S (↦) A end]
6     end]
7 end
```

Algorithm 10: *Abstract solver* combining neighborhood functions using operator *Union*

```
1 abstract solver as_union                          // Itr → number of iterations
2 computation : I, V1, V2, S, A
3 begin
4     [↺ (Itr < K1) begin
5         I (↦) [↺ (Itr % K2) begin [V1 (∪) V2] (↦) S (↦) A end]
6     end]
7 end
```

Solvers mentioned above were too slow to solve instances of the problem with more than 3 weeks, so we created another solver implementing the *abstract solver* described in Algorithm 11 using V_{AS} and combining S_{First} and S_{Rand}: it tries a number of times to improve the cost, and if it is not possible, it picks a random neighbor for the next iteration. We also compared

the S_{First} and S_{Best} selection modules.

Algorithm 11: *Abstract solver* for *SGP* to scape from local minima

```
1 abstract solver as_eager                                   // Itr → number of iterations
2 computation : I, V, S1, S2, A
3 begin
4     [↺ (Itr < K1) begin
5         I ↦ [↺ (Itr % K2) begin V ↦ [S1 ?_{Sci<K3} S2] ↦ A end]
6     end]
7 end
```

After that, we have chosen the best solver to be communicating solvers to compare their performance with the non communicating strategies. The shared information is the current configuration. Algorithms 12 and 13 show that the communication is performed while selecting a new configuration for the next iteration. Here, solvers receive a configuration from an outer solver, and match it with their current configuration. Then solvers select the configuration with the lowest global cost. We design different communication strategies. Either we execute a full connected solvers set, or a tuned combination of connected and unconnected solvers. Between connected solvers, we applied two different connections operations: connecting each sender solver with one receiver solver (*1 to 1*), or connecting each sender solver with all receiver solvers (*1 to N*).

Algorithm 12: Communicating *abstract solver* for *SGP* (sender)

```
1 abstract solver as_eager_sender                            // Itr → number of iterations
2 computation : I, V, S1, S2, A      // Sci → number of iterations with the same cost
3 begin
4     [↺ (Itr < K1) begin
5         I ↦ [↺ (Itr % K2) begin V ↦ [(|S1|)^o ?_{Sci<K3} S2] ↦ A end]
6     end]
7 end
```

Algorithm 13: Communicating *abstract solver* for *SGP* (receiver)

```
1 abstract solver as_eager_receiver                          // Itr → number of iterations
2 computation : I, V, S1, S2, A      // Sci → number of iterations with the same cost
3 communication : C.M.
4 begin
5     [↺ (Itr < K1) begin
6         I ↦
7         [↺ (Itr % K2) begin
8             V ↦ [[S1 m C.M.] ?_{Sci<K3} S2] ↦ A
9         end]
10    end]
11 end
```

Obviously, the communication frequency have to be controlled, because it can slow down the

search process.

5.2 Solving the *N-Queens Problem*

The *N-Queens Problem* (*NQP*) asks how to place N queens on a chess board so that none of them can hit any other in one move. This problem was introduced in 1848 by the chess player Max Bezzelas the *8-queen problem*, and years latter it was generalized as *N-queen problem* by Franz Nauck. Since then many mathematicians, including Gauss, have worked on this problem. It finds a lot of applications, e.g., parallel memory storage schemes, traffic control, deadlock prevention, neural networks, constraint satisfaction problems, among others [123]. Some studies suggest that the number of solution grows exponentially with the number of queens (N), but local search methods have been shown very good results for this problem [124]. For that reason we tested some communication strategies using POSL, to solve a problem relatively easy to solve using non communication strategies.

To handle this problem, we reused some modules used for the *Social Golfers Problem*: the *Selection* and *Acceptance* modules. The new module is:

a) Neighborhood module:

V_{AS}: Defines the neighborhood $V(s)$ swapping the variable which contributes the most to the cost with other.

Fors this problem we used a simple *abstract solver* showing good results with no communication, based on the idea introduced in the section 5.1, using the *computation module* S_{rand} to scape from local minima. The *abstract solver* is presented in Algorithm 14.

Algorithm 14: *Abstract solver* for *NQP*

```
1 abstract solver as_eager                          // Itr → number of iterations
2 computation : I, V, S1, S2, A      // Sci → number of iterations with the same cost
3 begin
4     I ↦ [↻ (Itr < K1) begin V ↦ [S1 ?_{Sci<K2} S2] ↦ A end]
5 end
```

Using solvers implementing this *abstract solver* we create communicating solvers to compare their performance with the non communicating strategies. The shared information is the current configuration. Algorithms 15 and 16 show that the communication is performed while selecting a new configuration for the next iteration. We design different communication strategies. Either we execute a full connected solvers set, or a tuned combination of connected and unconnected solvers. Between connected solvers, we applied two different connections

operations: connecting each sender solver with one receiver solver (*1 to 1*), or connecting each sender solver with all receiver solvers (*1 to N*).

Algorithm 15: *Abstract solver* for *NQP* (sender)

```
1 abstract solver as_eager_sender                    // Itr → number of iterations
2 computation : I, V, S1, S2, A      // Sci → number of iterations with the same cost
3 begin
4     I (↦) [↻ (Itr < K1) begin V (↦) [(|S1|)^o (?)_{Sci<K2} S2] (↦) A end]
5 end
```

Algorithm 16: *Abstract solver* for *NQP* (receiver)

```
1 abstract solver as_eager_receiver                  // Itr → number of iterations
2 computation : I, V, S1, S2, A      // Sci → number of iterations with the same cost
3 communication : C.M.
4 begin
5     I (↦)
6     [↻ (Itr < K1) begin
7         V (↦) [[S1 (?)_{Itr%K2} [S1 (m) C.M.]] (?)_{Sci<K3} S2] (↦) A
8     end]
9 end
```

5.3 Solving the *Costas Array Problem*

The *Costas Array Problem* (*CAP*) consists in finding a *costas array*, which is an $n \times n$ grid containing n marks such that there is exactly one mark per row and per column and the $n(n-1)/2$ vectors joining each couple of marks are all different. This is a very complex problem that finds useful application in some fields like sonar and radar engineering. It also presents an interesting characteristic: although the search space grows factorially, from order 17 the number of solutions drastically decreases [122].

To handle this problem, we reused some modules used for the *Social Golfers Problem* and *N-Queens Problem*: the *Neighborhood computation module* used for *N-Queens*, and the *Selection* and *Acceptance computation modules* used for both. The new modules are:

a) Generation module:

I: Generates a random configuration s, as a permutation of the vector $[1..n]$.

b) Neighborhood module:

V_{AS}: Defines the neighborhood $V(s)$ swapping the variable which contributes the most to the cost with other.

We also add a *Reset* module (R), a mechanism to escape from local minima[i]. The basic solver we use to solve this problem is presented in Algorithm 17, and we take it as a base to build all the different communication strategies. Basically, it is a classical local search iteration, where instead of performing restarts, it performs resets.

Algorithm 17: Reset-based *abstract solver* for *CAP*

```
1 abstract solver as_hard                          // ITR → number of iterations
2 computation : I, R, V, S, A
3 begin
4     I (↦)
5     [↻ (ITR < K1) begin
6         R (↦) [↻ (ITR % K2) begin V (↦) S (↦) A end]
7     end]
8 end
```

The *abstract solver* for the sender solver is presented in Algorithm 18. Like for the Social Golfers Problem, we design different communication strategies combining different percentages of communicating solvers and our two communication operators (*1 to 1* and *1 to N*). However for this problem, we study the behavior of the communication performed at two different moments: while applying the acceptance criteria (Algorithm 19), and while performing a

[i]It is based on the code from Daniel Díaz available at https://sourceforge.net/projects/adaptivesearch/

reset (Algorithms 19, 20 and 21).

Algorithm 18: Reset-based *abstract solver* for *CAP* (sender)

```
1 abstract solver as_hard_sender                    // Itr → number of iterations
2 computation : I, R, V, S, A
3 begin
4     I (↦)
5     [↺ (Itr < K_1) begin
6         R (↦) [↺ (Itr % K_2) begin V (↦) S (↦) (|A|)^o end]
7     end]
8 end
```

Algorithm 19: Reset-based *abstract solver* for *CAP* (receiver, variant A)

```
1 abstract solver as_hard_receiver_a                // Itr → number of iterations
2 computation : I, R, V, S, A
3 communication : C.M.
4 begin
5     I (↦)
6     [↺ (Itr < K_1) begin
7         R (↦) [↺ (Itr % K_2) begin V (↦) S (↦) [A (m) C.M.] end]
8     end]
9 end
```

Algorithm 20: Reset-based *abstract solver* for *CAP* (receiver, variant B)

```
1 abstract solver as_hard_receiver_b                // Itr → number of iterations
2 computation : I, R, V, S, A      // Sci → number of iterations with the same cost
3 communication : C.M.
4 begin
5     I (↦)
6     [↺ (Itr < K_1) begin
7         [R (?)_{Sci<K_3} [R (m) C.M.]] (↦) [↺ (Itr % K_2) begin V (↦) S (↦) A end]
8     end]
9 end
```

Algorithm 21: Reset-based *abstract solver* for *CAP* (receiver, variant C)

```
1 abstract solver as_hard_receiver_c                // Itr → number of iterations
2 computation : I, R, V, S, A
3 communication : C.M.
4 begin
5     I (↦)
6     [↺ (Itr < K_1) begin
7         [R (m) C.M.] (↦) [↺ (Itr % K_2) begin V (↦) S (↦) A end]
8     end]
9 end
```

5.4 Solving the *Golomb Ruler Problem*

The *Golomb Ruler Problem* (*GRP*) problem consists in finding an ordered vector of n distinct non-negative integers, called *marks*, $m_1 < \cdots < m_n$, such that all differences $m_i - m_j$ $(i > j)$ are all different. An instance of this problem is the pair (o, l) where o is the order of the problem, (i.e., the number of *marks*) and l is the length of the ruler (i.e., the last *mark*). We assume that the first *mark* is always 0. This problem has been applied to radio astronomy, x-ray crystallography, circuit layout and geographical mapping [125]. When we apply POSL to solve an instance of this problem sequentially, we can notice that it performs many *restarts* before finding a solution. For that reason we chose this problem to study a new communication strategy.

We use *Golomb Ruler Problem* instances to study a different communication strategy. This time we communicate the current configuration, to avoid its neighborhood, i.e., a *tabu* configuration.

We reused some modules used in the resolution of *Social Golfers* and *Costas Array* problems to design the solvers: the *Selection* and *Acceptance* modules. The new modules are:

a) Generation module:

I: Generates a random configuration s, respecting the structure of the problem, i.e., the configuration is an ordered vector of integers. This module takes into account a set of *tabu* configurations arrived via solver-communication (and also from the same solver) to construct the new configuration far enough from them.

b) Neighborhood module:

V: Defines the neighborhood $\mathcal{V}(s)$ by changing one value while keeping the order, i.e., replacing the value s_i by all possible values $s'_i \in D_i$ that satisfy $s_{i-1} < s'_i < s_{i+1}$.

We also add a module to insert a configuration into a *tabu* list inside the solver. In Algorithm 22 we present the *abstract solver* used to send information (sender *abstract solver*). When the module T is executed, the solver is unable to find a better configuration around the current one, so it is assumed to be a local minimum, and it is sent to the receiver solver. Algorithm 23 presents an *abstract solver* used to receive information (receiver *abstract solver*). Based on the connection operator used in the communication strategy, this solver might receives one or many configurations. These configurations are the input of the generation

module (I), and this module inserts all received configurations into a *tabu* list, and then generates a new first configuration, far from all configurations in the *tabu* list.

Algorithm 22: *Abstract solver* for *GRP* (sender)

```
1 abstract solver as_golomb_sender                    // Itr → number of iterations
2 computation : I, V, S, A, T
3 begin
4     [↺ (Itr < K1) begin
5         I ↦ [↺ (Itr % K2) begin V ↦ S ↦ A end] ↦ (|T|)^o
6     end]
7 end
```

Algorithm 23: *Abstract solver* for *GRP* (receiver)

```
1 abstract solver as_golomb_receiver                  // Itr → number of iterations
2 computation : I, V, S, A, T
3 connection : C.M.
4 begin
5     [↺ (Itr < K1) begin
6         [C.M. ↦ I] ↦ [↺ (Itr % K2) begin V ↦ S ↦ A end] ↦ (|T|)^o
7     end]
8 end
```

6

Analysis of results

In this chapter we explain the used environments were we run the experiments (description of my desktop machine, Curiosiphi server, and eventually Grid5000). We describe all the experiments and we expose a complete analysis of the obtained result.

Contents

The experiments[i] were performed on an Intel® Xeon™ E5-2680 v2, 10×4 cores, 2.80GHz. Results showed in this section are the means of 30 runs for each setup, presented in columns labeled **T**, corresponding to the run-time in seconds, and **It.** corresponding to the number of iterations; and their respective standard deviations (**T(sd)** and **It.(sd)**). In some tables, the column labeled **% success** indicates the percentage of solvers finding a solution before reaching a time–out (5 minutes).

The experiments in this section are multi-walk runs using the same solver main structure (except differents w.r.t. communication operations). Parallel experiments use 40 cores for all problem instances. It is important to point out that POSL is not designed to obtain the best results in terms of performance, but to give the possibility of rapidly prototyping and studying different cooperative or non cooperative search strategies.

6.1 *Social Golfers Problem*

We present in Table 6.1 results of launching *solvers sets* to solve each instance of the problem sequentially. Not surprisingly, the means of sequential runtimes and iterations (Table 6.1) are bigger than those means of parallel runs, with or without communication (all other tables).

Instance	T	T(sd)	It.	It.(sd)	% success
5–3–7	8.31	7.64	17,347	15,673	100.00
8–4–7	16.92	15.15	7,829	7,019	100.00
9–4–8	79.60	64.07	20,779	16,537	94.28
11–7–5	3.37	2.16	664	380	100.00

Table 6.1: *Social Golfers*: a single sequential solver

In a first stage of the experiments we use the operator-based language provided by POSL to build and test many different non communicating strategies. The goal is to select the best concrete modules to run tests performing communication. In particular, we have tested two kind of computation modules: the one computing the neighborhood of a given configuration and the one choosing the current configuration for the next solver iteration.

We focused on choosing the right neighborhood function. In the case of the *Social Golfers Problem*, this experiment was launched using a basic abstract solver showed in Algorithm 8. Solvers implemented from this abstract solver was not able to solve instances beyond three weeks: they were very often trapped into local minima. This is the reason why we perform this first experiment with the instance 10–10–3 whereas next experiments scale above 3

[i]POSL source code is available on GitHub:https://github.com/alejandro-reyesamaro/POSL

Abstract solvers	T	T(sd)	It.	It.(sd)
Adaptive Search (AS)	**1.06**	0.79	352	268
Std (ρ) AS	41.53	26.00	147	72
Std $(\cup)$ AS	59.65	55.01	198	110
Standard (Std)	87.90	41.96	146	58

Table 6.2: *Social Golfers*: Instance 10–10–3 in parallel

Instance	O.M. Best Improvement				O.M. First Improvement			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	4.99	4.43	4,421	3,938	**1.32**	0.68	1322	676
8–4–7	5.10	1.77	954	334	**1.82**	0.84	445	191
9–4–8	12.37	5.40	1,342	591	**6.43**	4.60	873	591
11–7–5	5.19	1.67	351	114	**2.22**	0.69	273	58

Table 6.3: *Social Golfers*: comparing selection functions

weeks. This was not a problem though, since the goal of this first experiment was only to find the right concrete neighborhood module.

Results in Table 6.2 are not surprising. The neighborhood neighborhood module V_{AS} is based on the *Adaptive Search* algorithm, which has shown very good results [1]. It selects the most culprit variable (i.e. a player), that is, the variable to most responsible for constraints violation. Then, it permutes this variable value with the value of each other variable, in all groups and all weeks. Each permutation gives a neighbor of the current configuration. V_{Std} uses no additional information, so it performs every possible swap between two players in different groups, every week. It means that this neighborhood is $g \times p$ times bigger than the previous one, with g the number of groups and p the number of players per group. We also tested abstract solvers with different combinations of these modules, using the (ρ) and the $(\cup)$ operators. The (ρ) operator executes its first or second parameter depending on a given probability ρ, and the $(\cup)$ operator returns the union of its parameters output. All these combinations spent more time searching the best configuration among the neighborhood, although with a lower number of iterations than V_{AS}. The V_{AS} neighborhood function being clearly faster, we have chosen it for our experiments, even if it shown a more spread standard deviation: 0.75 for AS versus 0.62 for Std, considering the ratio $\frac{T(sd)}{T}$.

With the selected neighborhood function, we focused on choosing the best *selection* function. We compared two different concrete modules used within the abstract solver in Algorithm 11, which combines selection modules (S_{First} or S_{Best}) with S_{Rand}, to avoid being trapped into local minima: it tries to improve the cost in a limited number of iterations. If it is not possible, it selects a random neighbor for the next iteration. The first module was S_{Best} that selects the best configuration inside the neighborhood. It not only spent more time searching a better configuration, but also is more sensitive to become trapped into local minima. The second module was S_{First} which selects the first configuration inside the neighborhood

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	1.19	0.64	1,156	608	1.11	0.49	1,067	484
8–4–7	**1.30**	0.72	**317**	161	1.46	0.57	347	128
9–4–8	4.38	2.72	**597**	347	5.51	3.06	736	389
11–7–5	1.76	0.41	214	44	**1.62**	0.34	**202**	30

Table 6.4: *Social Golfers*: test with 100% of communication

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	1.04	0.45	1,019	456	1.04	0.53	1,031	530
8–4–7	1.40	0.57	337	122	1.43	0.76	353	167
9–4–8	4.64	2.17	637	279	5.75	3.06	776	389
11–7–5	1.81	0.40	220	33	1.82	0.39	222	39

Table 6.5: *Social Golfers*: test with 50 % of communication

improving the current cost. Using this module, solvers favor exploration over intensification and of course spend clearly less time computing the neighborhood. Table 6.3 presents results of this experiment, showing that an exploration-oriented strategy is better for the *Social Golfers* problem. If we compare results of Tables 6.1 and 6.3 with respect to the standard deviation, we can some gains in robustness with parallelism. The spread in the running times and iterations for the instance 9–4–8 (the hardest one) is 10% lower (0.80 sequentially versus 0.71 in parallel), and for the others, it is around 40% lower (0.91, 0.89 and 0.64 sequentially versus 0.51, 0.45 and 0.31 in parallel, for 5–3–7, 8–4–7 and 11–7–5 respectively, with the same ratio $\frac{T(sd)}{T}$).

Then we ran experiments to study POSL's behavior solving target problems in communicating scenarios. Some compositions of solvers set were taken into account: i. the structure of the communication (with/without communication or a mix), and ii. the used communication operator.

Each time a POSL meta-solver is launched, many independent search solvers are executed. We call "good" configuration a configuration with the lowest cost within the current configuration neighborhood and with a cost strictly lesser than the current one. Once a good configuration is found in a sender solver, it is transmitted to the receiver one. At this

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	**0.90**	0.51	**881**	492	1.19	0.67	1,170	655
8–4–7	1.39	0.43	341	94	1.46	0.43	352	96
9–4–8	**4.33**	1.92	599	248	4.53	2.01	625	251
11–7–5	1.99	0.54	242	51	1.63	0.35	224	28

Table 6.6: *Social Golfers*: test with 25% of communication

moment, if the information is accepted, there are some solvers searching in the same subset of the search space, and the search process becomes more exploitation–oriented. This can be problematic if this process makes solvers converging too often towards local minima. In that case, we waste more than one solver trapped into a local minima: we waste all solvers that have been attracted to this part of the search space because of communications. We avoid this phenomenon through a simple (but effective) play: if a solver is not able to find a better configuration inside the neighborhood (executing S_{First}), it selects a random one at the next iteration (executing S_{Rand}). This strategy, using communication between solvers, produces some gain in terms of runtime (Table 6.3 w.r.t. Tables 6.4, 6.5 and 6.6. The percentage of the receiver solvers that were able to find the solution before the others did, was significant See Anexes. That shows that the communication played an important role during the search, despite inter–process communication's overheads (reception, information interpretation, making decisions, etc). For this problem we have reduced the spread in the running times and iterations of the results for the two last instances (9–4–8 and 11–7–5) applying the communication strategy (0.71 without communication versus 0.44 with communication, for 9–4–8, and 0.31 without communication versus 0.20 with communication for 11–7–5).

6.2 *N-Queens Problem*

We use directly the neighborhood module V_{AS} based on the *Adaptive Search* algorithm, and the selection module S_{First} which selects the first configuration inside the neighborhood improving the current cost, to create the solvers, and studying communicating and non communicating strategies.

Instance	Sequential (1 core)		No Comm. (40 cores)	
	T	It.	T	It.
2000			6.15	952
3000			14.06	1,413
4000			25.46	1,898
5000			40.57	2,377
6000			60.10	2,849

Table 6.7: Results for *NQP* (no communication)

POSL, as we can see in Tables 6.8 and 6.9, works very well without communication, for instances relatively big. This confirms once again the success of the *computation module* V_{AS} based on *Adaptive Search* algorithm to solve these kind of problems. The parallel approach outperforms significantly the sequential one, as we can see in Table 6.7. Runtimes and iteration means showed in this Table are bigger than those presented in Tables 6.8 and 6.9. However, the communication improve the non communicating results in terms of runtime

Instance	25% Comm.		50% Comm.		All Comm.	
	T	It.	T	It.	T	It.
2000	6.05	934	6.01	920	**5.92**	**885**
3000	13.89	1,387	13.91	1,368	**13.67**	**1,346**
4000	25.26	1,868	25.14	1,855	**25.11**	**1,834**
5000	40.38	2,338	40.33	2,312	**39.62**	**2,287**
6000	59.28	2,794	58.97	2,775	**58.97**	**2,729**

Table 6.8: Results for *NQP* (40 cores, communication 1 to 1)

Instance	25% Comm.		50% Comm.		All Comm.	
	T	It.	T	It.	T	It.
2000	6.07	925	**5.98**	915	6.01	**887**
3000	13.97	1,402	13.96	1,386	**13.79**	**1,365**
4000	25.30	1,867	25.29	1,851	**25.17**	**1,838**
5000	40.45	2,338	40.37	2,312	**39.88**	**2,291**
6000	59.77	2,824	59.53	2,773	**59.16**	**2,773**

Table 6.9: Results for *NQP* (40 cores, communication 1 to N)

and iterations, this improvement is not significant. In contrast to *SGP*, POSL does not get trapped so often into local minima during the resolution of *NQP*. For that reason, the shared information, once received and accepted by the receivers solvers, does not improves largely the current cost.

We can see the improvement with respect to the percentage of communicating solvers in Figure 6.1. The bigger the instance is, the more significant the observed improvement is. This phenomenon suggests that a deeper study and an efficient implementation can make the communication playing a more significant role in the solution process.

6.3 *Costas Array Problem*

We present in Table 6.10 results of launching *solver sets* to solve each instance of *Costas Array Problem* sequentially. Runtimes and iteration means showed in this Table are bigger than those presented in Table 6.11, confirming once again the success of the parallel approach.

STRATEGY	T	T(ds)	It.	It.(sd)	% success
Sequential (1 core)	2.12	0.87	44,453	18,113	42.00
Parallel (40 cores)	0.73	0.46	9,556	6,439	100.00

Table 6.10: *Costas Array* 17: no communication

We chose directly the neighborhood module (V_{AS}), the selection module (S_{First}) and the acceptance module A, to create the solvers. We ran experiments to study parallel communicat-

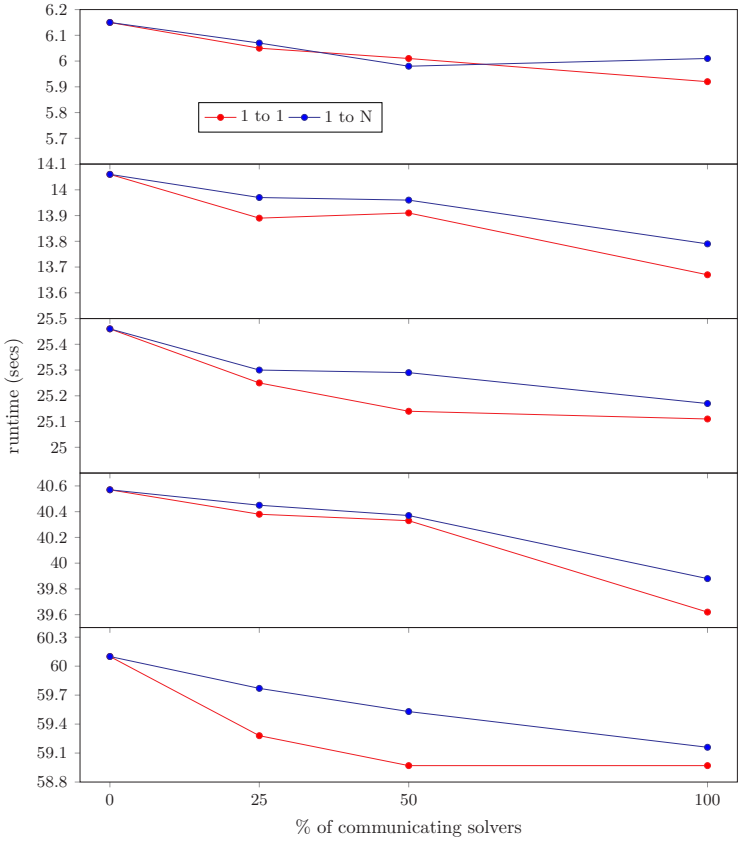


Figure 6.1: Runtime means of instances 2000-, 3000-, 4000-, 5000- and 6000-queens

ing strategies taken into account the structure of the communication, and the communication operator used, but in this problem, we perform the communication at two different times: at the time of applying the acceptance criteria, and at the time of performing the *reset*.

Table 6.11 shows that the *abstract solver A* (receiving the configuration at the time of applying the acceptance criteria) is more effective. The reason is that the others, interfere with the proper performance of the *reset*. Table 6.11 shows also high values of standard deviation. This is not surprising, due to the highly random nature of the neighborhood function and the selecting criterion, as well as the execution of many resets during the search process.

	100% COMM				50% COMM			
STRATEGY	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
Str A: 1 to 1	**0.41**	0.30	4,973	3,763	0.55	0.43	8,179	7,479
Str A: 1 to N	0.43	0.31	5,697	4,557	0.57	0.46	8,420	7,564
Str B: 1 to 1	0.48	0.41	6,546	5,562	0.51	0.49	8,004	7,998
Str B: 1 to N	0.45	0.46	5,701	6,295	0.48	0.51	7,245	8,379
Str C: 1 to 1	0.48	0.43	6,954	6,706	0.58	0.43	8,329	6,593
Str C: 1 to N	0.49	0.38	6,457	5,875	0.58	0.50	8,077	8,319

Table 6.11: *Costas Array* 17: with communication

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)	% success
8–34	0.79	0.66	13,306	11,154	66	55.74	100.00
8–34 (t)	0.66	0.63	10,745	10,259	53	51.35	100.00
10–55	66.44	49.56	451,419	336,858	301	224.56	80.00
10–55 (t)	67.89	50.02	446,913	328,912	297	219.30	88.00
11–72	160.34	96.11	431,623	272,910	143	90.91	26.67
11–72 (t)	117.49	85.62	382,617	275,747	127	91.85	30.00

Table 6.12: *Golomb Ruler*: a single sequential solver

6.4 *Golomb Ruler Problem*

The benefit of the parallel approach is also proved for the *Golomb Ruler Problem* (see Table 6.12 w.r.t. 6.13, 6.14, 6.15 and 6.16).

For *Golomb Ruler Problem*, the communication strategy that we adopt was different. Solvers do not communicate the current configuration to have more solvers searching in its neighborhood, but a configuration that we assume is a local minimum to be avoided. We consider that the current configuration is a local minimum since the solver (after a given number of iteration) is not able to find a better configuration in its neighborhood.

The first experiment compares the runs of non communitaing solvers not using a *tabu* list with non communicating solvers using a *tabu* list. The results in Tables 6.13 and 6.14 demonstrate that using a *tabu* list can help the search process. Without communication, the improvement is not substantial (8% for 8–34, 7% for 10–55 and 5% for 11–72) because only one configuration is inserted in the *tabu* list after each restart. When we use *one to one* communication, after the restart k, the receiving solver has twice the number of configurations in the *tabu* list (one *tabu* configuration from itself and the received one after each restart). Table 6.15 shows that this strategy is not sufficient for some instances, but when we use *one to N* communication, the number of *tabu* configurations after the restart k, in the receiving solver is considerably higher, e.g., after the restart k a receiving solver has $k(N+1)$ configurations in his *tabu* list. Hence, these solvers can generate configurations far enough from many potentially local minima. This phenomenon is more visible when the problem order increases. Table 6.16 shows that the improvement for the higher case (11-72) is about 32% w.r.t non communicating solvers not using a *tabu* list (Table 6.13), and about 29% w.r.t non communicating solvers using a *tabu* list (Table 6.14).

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)
8–34	0.47	34.82	436	330.10	2	1.63
10–55	5.31	38.63	22,577	16,488	15	11.00
11–72	89.76	55.85	164,763	102,931	54	34.32

Table 6.13: *Golomb Ruler*: parallel, without tabu list.

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)
8–34	0.43	0.37	349	334	1	1.64
10–55	4.92	4.68	20,504	19,742	13	13.07
11–72	85.02	67.22	155,251	121,928	51	40.64

Table 6.14: *Golomb Ruler*: parallel, with tabu list.

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)
8–34	0.44	0.31	309	233	1	1.23
10–55	3.90	3.22	15,437	12,788	10	8.52
11–72	85.43	52.60	156,211	97,329	52	32.43

Table 6.15: *Golomb Ruler*: parallel, communication 1 to 1.

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)
8–34	0.43	0.29	283	225	1	1.03
10–55	3.16	2.82	12,605	11,405	8	7.61
11–72	60.35	43.95	110,311	81,295	36	27.06

Table 6.16: *Golomb Ruler*: parallel, communication 1 to n.

Part IV

Conclusions and future works

7

Conclusion

We resume our work, emphasizing on our contribution and obtained results, and we expose the conclusions of the work. We also discus future branches to follow that can be derived from our work. Finally we give our conclusions.

In this paper we have analyzed some results using POSL, a Parallel–Oriented Solver Language to solve constraint-based problems [126, 127], to solve instances of the Social Golfers and Costas Array problems. It was possible to implement different communicating and non communicating strategies using the operator-based language provided. POSL gives the possibility to define different solver connections, to tune the percentage of communicating solvers, the moment when this communication takes place inside each iteration and the type of information to communicate. Results show POSL ability to solve these problems, showing at the same time that communication can play a decisive role in the search process, and provide more stable results. We show also that for some problems, the communication of the current configuration, as a mechanism of search intensification, has a positive effect in the search process. Although, a deep study to the nature of information to communicate would been very beneficial to make efficient parallel solvers. We also show that the communication of *tabu* configurations can accelerate the convergence to a solution.

POSL already has an important library of ready–to–use computation and connection modules, based on a deep study of classical meta-heuristics algorithms for solving combinatorial problems. In the near future we plan to make it grow, in order to increase possibilities of POSL by proposing new operators. It is necessary, for example, to improve the solver definition language, allowing to build sets of many new solvers faster and easier. Furthermore, we are aiming to expand the communication definition language, in order to create versatile and more complex and dynamic communication strategies, to allow a communication strategy to change during runtime.

8

Bibliography

[1] Daniel Diaz, Florian Richoux, Philippe Codognet, Yves Caniou, and Salvador Abreu. Constraint-Based Local Search for the Costas Array Problem. In *Learning and Intelligent Optimization*, pages 378–383. Springer, 2012.

[2] Danny Munera, Daniel Diaz, Salvador Abreu, and Philippe Codognet. A Parametric Framework for Cooperative Parallel Local Search. In *Evolutionary Computation in Combinatorial Optimisation*, volume 8600 of *LNCS*, pages 13–24. Springer, 2014.

[3] Stephan Frank, Petra Hofstedt, and Pierre R. Mai. Meta-S: A Strategy-Oriented Meta-Solver Framework. In *Florida AI Research Society (FLAIRS) Conference*, pages 177–181, 2003.

[4] Alex S Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary computation*, 16(1):31–61, 2008.

[5] Mahuna Akplogan, Jérome Dury, Simon de Givry, Gauthier Quesnel, Alexandre Joannon, Arnauld Reynaud, Jacques Eric Bergez, and Frédérick Garcia. A Weighted CSP approach for solving spatio-temporal planning problem in farming systems. In *11th Workshop on Preferences and Soft Constraints Soft 2011.*, Perugia, Italy, 2011.

[6] Louise K. Sibbesen. *Mathematical models and heuristic solutions for container positioning problems in port terminals.* Doctor of philosophy, Technical University of Danemark, 2008.

[7] Wolfgang Espelage and Egon Wanke. The combinatorial complexity of masterkeying. *Mathematical Methods of Operations Research*, 52(2):325–348, 2000.

[8] Barbara M Smith. Modelling for Constraint Programming. *Lecture Notes for the First International Summer School on Constraint Programming*, 2005.

[9] Ignasi Abío and Peter J Stuckey. Encoding Linear Constraints into SAT. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 75–91. Springer, 2014.

[10] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. *AIIDE*, pages 216–217, 2008.

[11] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–49, 2012.

[12] Christian Bessiere. Constraint Propagation. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, chapter 3, pages 29–84. Elsevier, 1st edition, 2006.

[13] Daniel Chazan and Willard Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199–222, 1969.

[14] Patrick Cousot and Radhia Cousot. Automatic synthesis of optimal invariant assertions: mathematical foundations. In *ACM Symposium on Artificial Intelligence amd Programming Languages*, volume 12, pages 1–12, Rochester, NY, 1977.

[15] Krzysztof R. Apt. From Chaotic Iteration to Constraint Propagation. In *24th International Colloquium on Automata, Languages and Programming (ICALP'97)*, pages 36–55, 1997.

[16] Éric Monfroy and Jean-Hugues Réty. Chaotic Iteration for Distributed Constraint Propagation. In *ACM symposium on Applied computing SAC '99*, pages 19–24, 1999.

[17] Éric Monfroy. A coordination-based chaotic iteration algorithm for constraint propagation. In *Proceedings of The 15th ACM Symposium on Applied Computing, SAC 2000*, pages 262–269. ACM Press, 2000.

[18] Peter Zoeteweij. Coordination-based distributed constraint solving in DICE. In *Proceedings of the 18th ACM Symposium on Applied Computing (SAC 2003)*, pages 360–366, New York, 2003. ACM Press.

[19] Farhad Arbab. Coordination of Massively Concurrent Activities. Technical report, Amsterdam, 1995.

[20] Laurent Granvilliers and Éric Monfroy. Implementing Constraint Propagation by Composition of Reductions. In *Logic Programming*, pages 300–314. Springer Berlin Heidelberg, 2001.

[21] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. The Iterator and Composite Patterns. Well-Managed Collections. In *Head First Design Patterns*, chapter 9, pages 315–384. O'Relliy, 1st edition, 2004.

[22] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. The Observer Pattern. Keeping your Objects in the know. In *Head First Design Patterns*, chapter 2, pages 37–78. O'Relliy, 1st edition, 2004.

[23] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. Introduction to Design Patterns. In *Head First Design Patterns*, chapter 1, pages 1–36. O'Relliy, 1st edition, 2004.

[24] Charles Prud'homme, Xavier Lorca, Rémi Douence, and Narendra Jussien. Propagation engine prototyping with a domain specific language. *Constraints*, 19(1):57–76, sep 2013.

[25] Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched Literals for Constraint Propagation in Minion. *Lecture Notes in Computer Science*, 4204:182–197, 2006.

[26] Mikael Z. Lagerkvist and Christian Schulte. Advisors for Incremental Propagation. *Lecture Notes in Computer Science*, 4741:409–422, 2007.

[27] Narendra Jussien, Hadrien Prud'homme, Charles Cambazard, Guillaume Rochart, and François Laburthe. Choco: an Open Source Java Constraint Programming Library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08),*, Paris, France, 2008.

[28] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. MiniZinc: Towards A Standard CP Modelling Language. In *Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.

[29] Ibrahim H Osman and Gilbert Laporte. Metaheuristics : A bibliography. *Annals of Operations research*, 63(5):511–623, 1996.

[30] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

[31] Ilhem Boussaïd, Julien Lepagnot, and Patrick Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82–117, jul 2013.

[32] Alexander G. Nikolaev and Sheldon H. Jacobson. Simulated Annealing. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 1, pages 1–39. Springer, 2nd edition, 2010.

[33] Aris Anagnostopoulos, Laurent Michel, Pascal Van Hentenryck, and Yannis Vergados. A simulated annealing approach to the travelling tournament problem. *Journal of Scheduling*, 2(9):177—-193, 2006.

[34] Michel Gendreau and Jean-Yves Potvin. Tabu Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 2, pages 41–59. Springer, 2nd edition, 2010.

[35] Iván Dotú and Pascal Van Hentenryck. Scheduling Social Tournaments Locally. *AI Commun*, 20(3):151—-162, 2007.

[36] Christos Voudouris, Edward P.K. Tsang, and Abdullah Alsheddy. Guided Local Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 11, pages 321–361. Springer, 2 edition, 2010.

[37] Patrick Mills and Edward Tsang. Guided local search for solving SAT and weighted MAX-SAT problems. *Journal of Automated Reasoning*, 24(1):205–223, 2000.

[38] Pierre Hansen, Nenad Mladenovie, Jack Brimberg, and Jose A. Moreno Perez. Variable neighborhood Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 3, pages 61–86. Springer, 2010.

[39] Noureddine Bouhmala, Karina Hjelmervik, and Kjell Ivar Overgaard. A generalized variable neighborhood search for combinatorial optimization problems. In *The 3rd International Conference on Variable Neighborhood Search (VNS'14)*, volume 47, pages 45–52. Elsevier, 2015.

[40] Edmund K. Burke, Jingpeng Li, and Rong Qu. A hybrid model of integer programming and variable neighbourhood search for highly-constrained nurse rostering problems. *European Journal of Operational Research*, 203(2):484–493, 2010.

[41] Thomas A. Feo and Mauricio G.C. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optimization*, (6):109–134, 1995.

[42] Mauricio G.C Resende. Greedy randomized adaptive search procedures. In *Encyclopedia of optimization*, pages 1460–1469. Springer, 2009.

[43] Philippe Galinier and Jin-Kao Hao. A General Approach for Constraint Solving by Local Search. *Journal of Mathematical Modelling and Algorithms*, 3(1):73–88, 2004.

[44] Philippe Codognet and Daniel Diaz. Yet Another Local Search Method for Constraint Solving. In *Stochastic Algorithms: Foundations and Applications*, pages 73–90. Springer Verlag, 2001.

[45] Yves Caniou, Philippe Codognet, Florian Richoux, Daniel Diaz, and Salvador Abreu. Large-Scale Parallelism for Constraint-Based Local Search: The Costas Array Case Study. *Constraints*, 20(1):30–56, 2014.

[46] Danny Munera, Daniel Diaz, Salvador Abreu, Francesca Rossi, and Philippe Codognet. Solving Hard Stable Matching Problems via Local Search and Cooperative Parallelization. In *29th AAAI Conference on Artificial Intelligence*, Austin, TX, 2015.

[47] Kazuo Iwama, David Manlove, Shuichi Miyazaki, and Yasufumi Morita. Stable marriage with incomplete lists and ties. In *ICALP*, volume 99, pages 443–452. Springer, 1999.

[48] David Gale and Lloyd S. Shapley. College Admissions and the Stability of Marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[49] Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. *ACM SIGPLAN Notices*, 37(11):83–100, 2002.

[50] Dynamic Decision Technologies Inc. *Dynadec. Comet Tutorial.* 2010.

[51] Laurent Michel and Pascal Van Hentenryck. The comet programming language and system. In *Principles and Practice of Constraint Programming*, pages 881–881. Springer Berlin Heidelberg, 2005.

[52] Jorge Maturana, Álvaro Fialho, Frédéric Saubion, Marc Schoenauer, Frédéric Lardeux, and Michèle Sebag. Adaptive Operator Selection and Management in Evolutionary Algorithms. In *Autonomous Search*, pages 161–189. Springer Berlin Heidelberg, 2012.

[53] Colin R. Reeves. Genetic Algorithms. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 146, chapter 5, pages 109–139. Springer, 2010.

[54] Marco Dorigo and Thomas Stützle. Ant colony optimization: overview and recent advances. In *Handbook of Metaheuristics*, volume 146, chapter 8, pages 227–263. Springer, 2nd edition, 2010.

[55] Konstantin Chakhlevitch and Peter Cowling. Hyperheuristics : Recent Developments. In *Adaptive and multilevel metaheuristics*, pages 3–29. Springer, 2008.

[56] Patricia Ryser-Welch and Julian F. Miller. A Review of Hyper-Heuristic Frameworks. In *Proceedings of the Evo20 Workshop, AISB*, 2014.

[57] Kevin Leyton-Brown, Eugene Nudelman, and Galen Andrew. A portfolio approach to algorithm selection. In *IJCAI*, pages 1542–1543, 2003.

[58] Horst Samulowitz, Chandra Reddy, Ashish Sabharwal, and Meinolf Sellmann. Snappy: A simple algorithm portfolio. In *Theory and Applications of Satisfiability Testing - SAT 2013*, volume 7962 LNCS, pages 422–428. Springer, 2013.

[59] Alexander E.I. Brownlee, Jerry Swan, Ender Özcan, and Andrew J. Parkes. Hyperion 2. A toolkit for {meta- , hyper-} heuristic research. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1133–1140, Vancouver, BC, 2014. ACM.

[60] Enrique Urra, Daniel Cabrera-Paniagua, and Claudio Cubillos. Towards an Object-Oriented Pattern Proposal for Heuristic Structures of Diverse Abstraction Levels. *XXI Jornadas Chilenas de Computación 2013*, 2013.

[61] Laura Dioşan and Mihai Oltean. Evolutionary design of Evolutionary Algorithms. *Genetic Programming and Evolvable Machines*, 10(3):263–306, 2009.

[62] John N. Hooker. Toward Unification of Exact and Heuristic Optimization Methods. *International Transactions in Operational Research*, 22(1):19–48, 2015.

[63] El-Ghazali Talbi. Combining metaheuristics with mathematical programming, constraint programming and machine learning. *4or*, 11(2):101–150, 2013.

[64] Éric Monfroy, Frédéric Saubion, and Tony Lambert. Hybrid CSP Solving. In *Frontiers of Combining Systems*, pages 138–167. Springer Berlin Heidelberg, 2005.

[65] Éric Monfroy, Frédéric Saubion, and Tony Lambert. On Hybridization of Local Search and Constraint Propagation. In *Logic Programming*, pages 299–313. Springer Berlin Heidelberg, 2004.

[66] Jerry Swan and Nathan Burles. Templar - a framework for template-method hyper-heuristics. In *Genetic Programming*, volume 9025 of *LNCS*, pages 205–216. Springer International Publishing, 2015.

[67] Sébastien Cahon, Nordine Melab, and El-Ghazali Talbi. ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics*, 10(3):357–380, 2004.

[68] Youssef Hamadi, Éric Monfroy, and Frédéric Saubion. An Introduction to Autonomous Search. In *Autonomous Search*, pages 1–11. Springer Berlin Heidelberg, 2012.

[69] Roberto Amadini and Peter J Stuckey. Sequential Time Splitting and Bounds Communication for a Portfolio of Optimization Solvers. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, volume 1, pages 108–124. Springer, 2014.

[70] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Features for Building CSP Portfolio Solvers. *arXiv:1308.0227*, 2013.

[71] Christophe Lecoutre. XML Representation of Constraint Networks. Format XCSP 2.1. *Constraint Networks: Techniques and Algorithms*, pages 541–545, 2009.

[72] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. *Modeling and Programming with Gecode*. 2013.

[73] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Introduction to Parallel Computing. In *Introduction to Parallel Computing*, chapter 1, pages 1–9. Addison Wesley, 2nd edition, 2003.

[74] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, 2007. ACM.

[75] Mark D. Hill and Michael R. Marty. Amdahl's Law in the multicore era. *IEEE Computer*, (7):33–38, 2008.

[76] Peter Sanders. Engineering Parallel Algorithms: The Multicore Transformation. *Ubiquity*, 2014(July):1–11, 2014.

[77] Javier Diaz, Camelia Muñoz-Caro, and Alfonso Niño. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

[78] Joel Falcou. Parallel programming with skeletons. *Computing in Science and Engineering*, 11(3):58–63, 2009.

[79] Ian P Gent, Chris Jefferson, Ian Miguel, Neil C A Moore, Peter Nightingale, Patrick Prosser, and Chris Unsworth. A Preliminary Review of Literature on Parallel Constraint Solving. In *Proceedings PMCS 2011 Workshop on Parallel Methods for Constraint Solving*, 2011.

[80] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Embarrassingly Parallel Search. In *Principles and Practice of Constraint Programming*, pages 596–610. Springer, 2013.

[81] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Evaluation of a simple, scalable, parallel best-first search strategy. *Artificial Intelligence*, 195:222–248, 2013.

[82] Yuu Jinnai and Alex Fukunaga. Abstract Zobrist Hashing : An Efficient Work Distribution Method for Parallel Best-First Search. *30th AAAI Conference on Artificial Intelligence (AAAI-16)*.

[83] Alejandro Arbelaez and Luis Quesada. Parallelising the k-Medoids Clustering Problem Using Space-Partitioning. In *Sixth Annual Symposium on Combinatorial Search*, pages 20–28, 2013.

[84] Hue-Ling Chen and Ye-In Chang. Neighbor-finding based on space-filling curves. *Information Systems*, 30(3):205–226, may 2005.

[85] Pavel Berkhin. Survey Of Clustering Data Mining Techniques. Technical report, Accrue Software, Inc., 2002.

[86] Farhad Arbab and Éric Monfroy. Distributed Splitting of Constraint Satisfaction Problems. In *Coordination Languages and Models*, pages 115–132. Springer, 2000.

[87] Mark D. Hill. What is Scalability? *ACM SIGARCH Computer Architecture News*, 18:18–21, 1990.

[88] Danny Munera, Daniel Diaz, and Salvador Abreu. Solving the Quadratic Assignment Problem with Cooperative Parallel Extremal Optimization. In *Evolutionary Computation in Combinatorial Optimization*, pages 251–266. Springer, 2016.

[89] Stefan Boettcher and Allon Percus. Nature's way of optimizing. *Artificial Intelligence*, 119(1):275–286, 2000.

[90] Daisuke Ishii, Kazuki Yoshizoe, and Toyotaro Suzumura. Scalable Parallel Numerical CSP Solver. In *Principles and Practice of Constraint Programming*, pages 398–406, 2014.

[91] Charlotte Truchet, Alejandro Arbelaez, Florian Richoux, and Philippe Codognet. Estimating Parallel Runtimes for Randomized Algorithms in Constraint Solving. *Journal of Heuristics*, pages 1–36, 2015.

[92] Youssef Hamadi, Said Jaddour, and Lakhdar Sais. Control-Based Clause Sharing in Parallel SAT Solving. In *Autonomous Search*, pages 245–267. Springer Berlin Heidelberg, 2012.

[93] Youssef Hamadi, Cedric Piette, Said Jabbour, and Lakhdar Sais. Deterministic Parallel DPLL system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:127–132, 2011.

[94] Andre A. Cire, Sendar Kadioglu, and Meinolf Sellmann. Parallel Restarted Search. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 842–848, 2011.

[95] Long Guo, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Diversification and Intensification in Parallel SAT Solving. *Principles and Practice of Constraint Programming*, pages 252–265, 2010.

[96] M Yasuhara, T Miyamoto, K Mori, S Kitamura, and Y Izui. Multi-Objective Embarrassingly Parallel Search. In *IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pages 853–857, Singapore, 2015. IEEE.

[97] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. Improvement of the Embarrassingly Parallel Search for Data Centers. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 622–635, Lyon, 2014. Springer.

[98] Prakash R. Kotecha, Mani Bhushan, and Ravindra D. Gudi. Efficient optimization strategies with constraint programming. *AIChE Journal*, 56(2):387–404, 2010.

[99] Peter Zoeteweij and Farhad Arbab. A Component-Based Parallel Constraint Solver. In *Coordination Models and Languages*, pages 307–322. Springer, 2004.

[100] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Scalable, Parallel Best-First Search for Optimal Sequential Planning. *In ICAPS-09*, pages 201–208, 2009.

[101] Claudia Schmegner and Michael I. Baron. Principles of optimal sequential planning. *Sequential Analysis*, 23(1):11–32, 2004.

[102] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Programming Using the Message-Passing Paradigm. In *Introduction to Parallel Computing*, chapter 6, pages 233–278. Addison Wesley, second edition, 2003.

[103] Brice Pajot and Éric Monfroy. Separating Search and Strategy in Solver Cooperations. In *Perspectives of System Informatics*, pages 401–414. Springer Berlin Heidelberg, 2003.

[104] Mauro Birattari, Mark Zlochin, and Marrco Dorigo. Towards a Theory of Practice in Metaheuristics Design. A machine learning perspective. *RAIRO-Theoretical Informatics and Applications*, 40(2):353–369, 2006.

[105] Agoston E Eiben and Selmar K Smit. Evolutionary algorithm parameters and methods to tune them. In *Autonomous Search*, pages 15–36. Springer Berlin Heidelberg, 2011.

[106] Maria-Cristina Riff and Elizabeth Montero. A new algorithm for reducing metaheuristic design effort. *IEEE Congress on Evolutionary Computation*, pages 3283–3290, jun 2013.

[107] Holger H. Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous Search*, pages 37–71. Springer Berlin Heidelberg, 2012.

[108] Frank Hutter, Holger H Hoos, and Kevin Leyton-brown. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

[109] Frank Hutter. Updated Quick start guide for ParamILS, version 2.3. Technical report, Department of Computer Science University of British Columbia, Vancouver, Canada, 2008.

[110] Volker Nannen and Agoston E. Eiben. Relevance Estimation and Value Calibration of Evolutionary Algorithm Parameters. *IJCAI*, 7, 2007.

[111] S. K. Smit and A. E. Eiben. Beating the 'world champion' evolutionary algorithm via REVAC tuning. *IEEE Congress on Evolutionary Computation*, pages 1–8, jul 2010.

[112] E. Yeguas, M.V. Luzón, R. Pavón, R. Laza, G. Arroyo, and F. Díaz. Automatic parameter tuning for Evolutionary Algorithms using a Bayesian Case-Based Reasoning system. *Applied Soft Computing*, 18:185–195, may 2014.

[113] Agoston E. Eiben, Robert Hinterding, and Zbigniew Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2):124–141, 1999.

[114] Junhong Liu and Jouni Lampinen. A Fuzzy Adaptive Differential Evolution Algorithm. *Soft Computing*, 9(6):448–462, 2005.

[115] A Kai Qin, Vicky Ling Huang, and Ponnuthurai N Suganthan. Differential evolution algorithm with strategy adaptation for global numerical optimization. *IEEE Transactions on Evolutionary Computation*, 13(2):398–417, 2009.

[116] Vicky Ling Huang, Shuguang Z Zhao, Rammohan Mallipeddi, and Ponnuthurai N Suganthan. Multi-objective optimization using self-adaptive differential evolution algorithm. *IEEE Congress on Evolutionary Computation*, pages 190–194, 2009.

[117] Martin Drozdik, Hernan Aguirre, Youhei Akimoto, and Kiyoshi Tanaka. Comparison of Parameter Control Mechanisms in Multi-objective Differential Evolution. In *Learning and Intelligent Optimization*, pages 89–103. Springer, 2015.

[118] Jeff Clune, Sherri Goings, Erik D. Goodman, and William Punch. Investigations in Meta-GAs: Panaceas or Pipe Dreams? In *GECOO'05: Proceedings of the 2005 Workshop on Genetic an Evolutionary Computation*, pages 235–241, 2005.

[119] Emmanuel Paradis. R for Beginners. Technical report, Institut des Sciences de l'Evolution, Université Montpellier II, 2005.

[120] Scott Rickard. Open Problems in Costas Arrays. In *IMA International Conference on Mathematics in Signal Processing at The Royal Agricultural College*, Cirencester, UK., 2006.

[121] Frédéric Lardeux, Éric Monfroy, Broderick Crawford, and Ricardo Soto. Set Constraint Model and Automated Encoding into SAT: Application to the Social Golfer Problem. *Annals of Operations Research*, 235(1):423–452, 2014.

[122] Konstantinos Drakakis. A review of Costas arrays. *Journal of Applied Mathematics*, 2006:32 pages, 2006.

[123] Jordan Bell and Brett Stevens. A survey of known results and research areas for n-queens. *Discrete Mathematics*, 309(1):1–31, 2009.

[124] Rok Sosic and Jun Gu. Effcient Local Search with Conflict Minimization: A Case Study of the N-Queens Problem. *IEEE Transactions on Knowledge and Data Engineering*, 6:661–668, 1994.

[125] Stephen W. Soliday, Abdollah. Homaifar, and Gary L. Lebby. Genetic algorithm approach to the search for Golomb Rulers. In *International Conference on Genetic Algorithms*, volume 1, pages 528–535, Pittsburg, 1995.

[126] Alejandro Reyes-amaro, Éric Monfroy, and Florian Richoux. POSL: A Parallel-Oriented metaheuristic-based Solver Language. In *Recent developments of metaheuristics*, to appear. Springer.

[127] Alejandro Reyes-Amaro, Éric Monfroy, and Florian Richoux. A Parallel-Oriented Language for Modeling Constraint-Based Solvers. In *Proceedings of the 11th edition of the Metaheuristics International Conference (MIC 2015)*. Springer, 2015.