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 ??

Contents

Part I

Presentation

1

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. In Section 1.1 the definition of a Combinatorial Optimization Problem and its link with Constraint Satisfaction Problems (CSP) are introduced, where I concentrate our main efforts, and I give some examples. The basic techniques used to solve these problems are introduced, like Constraint Programming (1.2), meta– and hyper–heuristic methods (Sections 1.3 and 1.4). I also present some advanced techniques like hybridization in Section 1.5, parallel computing in Section 1.6, and Solvers cooperation in Section 1.7. Finally, before ending the chapter with a brief summary, I present parameter setting techniques in Section 1.8.

Contents

1.1 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 whether 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* [1], *Task Scheduling* [2], *Master-keying* [3], *Traveling Salesman*, *Knapsack Problem*, among others, are well-known examples of Combinatorial Optimization Problems [4].

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:*

$$c_i\,(S) \;\; \textit{is true}\; \forall c_i \in C$$

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.

1.2 Constraint programming

CSPs find a lot of "real-world" applications in the industry. In practice, these problems are tackled through different techniques. One of the most popular is *constraint programming*, a combination of three main ingredients: i) a declarative model of the problem, ii) constraint reasoning techniques like *filtering* and *propagation*, and iii) search techniques.

For modeling CSPs, two tools can be highlighted. MiniZinc 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 [5]. XCSP is a readable, concise and structured XML-like language for coding CSPs. This format allows us to represent constraints defined either extensionally or in intensionally. 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 CSPs instances existing today.

Constraint reasoning techniques are filtering algorithms applied for each constraint to prune provably infeasible values from the domain of the involved variables. This process is called *constraint propagation*, and they 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 [6]. The main goal is to choose one (or some) constraint(s) and enforcing certain consistency levels in the constraint, like for example, *arc consistency and bound 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*. Local consistency restrictions on the filtering algorithms are necessary to ensure not loosing solution during the propagation process.

We said that a variable $x \in c$, if it is involved into the constraint c. Let the set $Var(c) = \{x_1 \dots x_k\}$ the set of variables involved into a constraint c be, denoted by $Var(c)$. Then, a constraint c is called *arc consistent* if for all $x_i \in Var(c)$ with $1 \leq i \leq k$, and for all $v_j \in D_j$ with $1 \leq j \leq \|D_j\|$:

$$\exists (v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_k) \in D_1 \times \dots \times D_{i-1} \times D_{i+1} \times \dots \times D_k$$

such that $c(v_1, \ldots, v_k)$ is fulfilled. In other words, c is arc consistent if for each value of each variable, there exist values for the other variables fulfilling c. In that case, we said that each value in the domain of x_i has a *support* in the domain of the other variables.

We denote by $Bnd(D_i) = \{\min(D_i), \max(D_i)\}$ the bounds of the domain D_i. Then, a constraint is *bound consistent* if for all $x_i \in Var(c)$ with $1 \leq i \leq k$, and for all $v_j \in Bnd(D_j)$ with $1 \leq j \leq 2$:

$$\exists (v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k) \in Bnd\,(D_1) \times \cdots \times Bnd\,(D_{i-1}) \times Bnd\,(D_{i+1}) \times \cdots \times Bnd\,(D_k)$$

such that $c(v_1, \ldots, v_k)$ is fulfilled. It means that each bound (min/max) in the domain of x_i has a support in the bounds (min/max) of the other variables.

As we can note arc consistency is a stronger property, but heavier to enforce.

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 [7, 8]. In [9, 10] a formalization of constraint propagation is proposed through *chaotic iterations*. In [11], 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) [12] 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 [13].

In [14] 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*

[15], supporting inheritance between elementary and compound reduction functions. The propagation mechanism uses the *Observer (Listener) Design Pattern* [16], 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* [17], that allows to parametrize parts of algorithms.

A propagation engine prototype with a *Domain Specific Language* (DSL) was implemented in [18]. 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. 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* [19] (controlling the behavior of variable–value pairs to trigger propagation) or *advisors* [20] (a method for supporting incremental propagation in propagator–centered setting). *Advisors* in [20] are used to modify propagator state and to decide whether a propagator must be propagated or "scheduled".

Most of the times, we can not solve CSPs only applying constraint propagation techniques. It is necessary to combine them with search algorithms. The complete search process consists test all possible configurations in an ordered way. Each time a partial evaluation is executed (evaluating just a set of variables), new constraints are posted, meaning that the propagation process can be relaunched. The simplest approach is using a depth first search, where the new constraints narrow domains causing propagation, and if no more evaluation can be done, a backtrack, and domains are restored. The difference between the kind of search lies in the selection criteria of the order of variables to be evaluated, and the order of the values to be assigned to the variables. A *static* search strategy is based on selecting the variable with me minimum index, to be evaluated first with the minimum value of its domain. Using this search, the tree *structure* of the search space does not change, but is good for testing propagators. A *dynamic* search strategy is based on selecting the variable with me minimum domain element, to be evaluated first with the minimum value of its domain. In this search strategy, propagators affect the variable selection order. A classical search strategy is based on on selecting the variable with me minimum domain size, to be evaluated first with any values of its domain. Based on the *first fail* principle which tells "*Focus first on the variable*

that is more likely to cause a fail". This strategy works pretty well in many cases because by branching early on variables with a few value, the search tree becomes smaller.

In the field of constraint programing we can find a lot of solvers, able to solve constrained problems using these techniques. As examples, we can cite CPLEX and GECODE. CPLEX is an analytical decision support toolkit for rapid development and deployment of optimization models using mathematical and constraint programming, to solve very large, real-world optimization problems. GECODE is an efficient open source environment for developing constraint-based system and applications, that provides a modular and extensible constraint solver [21], written in C++ (winner of all gold medals in the *MiniZinc Challenge* from 2008 to 2012). During the formation phase of this PhD, I had the opportunity to perform some pedagogical experiments using two other important and recognized solvers: *OR-tools* and `Choco`. The *OR-tools* is an open source, portable and documented software suite for combinatorial optimization. It contains an efficient constraint programming solver, used internally at Google, where speed and memory consumption are critical.

`Choco` is a free and open-source tool written in java, to describe hard combinatorial problems in the form of CSPs and solving them using Constraint Programing techniques. Mainly developed by people at École des Mines de Nantes (France), is a solver with a nice history, wining some awards, including seven medals in four entries in the *MiniZinc Challenge*. This solver uses multi-thread approach for the resolution, and provide a problem modeler able to manipulate a wide variety of variable types. This problem modeler accepts over 70 constraints, including all classical arithmetical constraints, the possibility of using boolean operations between constraints, table constraints, i.e., defining the sets of tuples that verify the intended relation for a set of variables and a large set of useful classical global constraints including the `alldifferent` constraint, the global cardinality, among others. `Choco` also contains a MINIZINC and *XCSP* instance parser. `Choco` can either deal with satisfaction or optimization problems. The search can be parameterized using a set of predefined variable and value selection heuristics, and also the variable and/or value selectors can be parametrized [22, 23].

Although constraint programing techniques have shown very good results solving constrained problems, the search space in practical instances becomes intractable for them. For that reason, these constrained 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 [24, 25].

1.3 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) [26], 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 [27].

They are often nature-inspired and are divided in two groups [28]:

1. *Single Solution Based:* more exploitation oriented, intensifying the search in some specific areas. (this work focuses its attention on this first group)
2. *Population Based:* more exploration oriented, identifying areas of the search space where there are (or where there could be) the best solutions.

1.3.1 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* [28]. 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) [29] 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 [27]. In [30] 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) [31], 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 [32] to solve instances of the *Social Golfers* problem.

The idea of *Guided Local Search* (GLS) [33] 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 [27]. In [34] 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 [35, 27]. In [36] is introduced a *generalized Variable Neighborhood Search* for Combinatorial Optimization Problems, and in [37] 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 [38]. 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 [27]. However, in [39] some extensions like alternative solution construction mechanisms and techniques to speed up the search are presented.

Galinier et al. present in [40] 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:*

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

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 [41, 42, 43]. In [44] Munera et al. based their solution method in *Adaptive Search* to solve the *Stable Marriage with Incomplete List and Ties* problem [45], a natural variant of the *Stable Marriage Problem* [46].

Michel and Van Hentenryck [47] 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) [48, 49], that supports the local search architecture with a number of novel concepts, abstractions, and control structures.

1.3.2 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 [50].

Probably the most popular in this group are the *Genetic Algorithms* [51], 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 [52], that simulate the behavior of an ant swarm to find the shortest path from the food source to the nest.

1.4 Hyper-heuristic Methods

Hyper-heuristics are automated methodologies for selecting or generating heuristics to solve hard computational problems [53]. 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 [54].

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 [55]. In [56] 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 [57] 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:

1. Promoting interoperability via component interfaces,
2. Allowing rapid prototyping of meta– and hyper– heuristics, with the potential to use the same source code in either case,
3. Providing generic templates for a variety of local search and evolutionary computation algorithms,
4. Making easier the construction of novel meta- and hyper- heuristics by hybridization (via interface interoperability) or extension (subtype polymorphism),
5. *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 for the acceptance criteria [58].

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 [59] 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.

1.5 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 [60] 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 becomes more complex.

In [61] 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 [62, 63] a general hybridization framework, 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 [64]; and *ParadisEO*, a framework to design parallel and distributed hybrid meta-heuristics showing very good results [65], 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*) [66].

In [67] 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 [68] is proposed a tool (`xcsp2mzn`) for converting problem instances from the format [69] 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 [5]. 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 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.

1.6 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 [70]. That is why this field is in constant development and it is the topic where I 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 [71]. 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 [72, 73]. We can find in [74] a survey of the

different parallel programming models and available tools, emphasizing on their suitability for high-performance computing.

Falcou propose in [75] 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 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 compilation time.

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:

1. *Problem subdivision*,
2. *Scalability* and
3. *Inter-process communication*.

In a preliminary review of literature on parallel constraint solving [76], 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. It presents an analysis of some limiting factors on performance such as *Amdahl*'s law, and then reviews recent literature on parallel constraint programming, grouping the paper in 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 [77] 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 [78] 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 [79] to evaluate *Zobrist Hashing*, an efficient hash function designed for table games like chess and Go, to mitigate communication overheads.

In [80] 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 [81]); and second, *k-Means* algorithm, one of the most popular clustering algorithms [82].

In [83] 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 1.7.

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 [41].

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 doing 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 [84]. *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 [41]. For this algorithm, an implementation of a cooperative multi-walks strategy has been published in [85]. 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 [86] is presented a combination of this method with an *Extremal Optimization* procedure: a nature-inspired general-purpose meta-heuristic [87].

A lot of studies have been published around this topic. A parallel solver for numerical CSPs is presented in [88] showing good results scaling on a number of cores. In [89], 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 [90] 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 [85] 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 [91] 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 [92] 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 [93] 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 [94], 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 [95], 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 [77, 96]; 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 [97].

A component-based constraint solver in parallel is proposed in [98]. 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.

1.7 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 [99] a parallelization of the an algorithm A* (Hash Distributed A*) for *optimal sequential planning* [100], 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 [78], 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) [101].

In [102] 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 [90], 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. [102] 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 [91], which allows to tackle problems, through the cooperation of arbitrary domain–specific constraint solvers. *Meta–S* [91] 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 [57] 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 [83] 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 [13].

1.8 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 [103]. 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:

1. *Off-line tunning*: Also known just as parameter tuning, were parameters are computed before the run.
2. *On-line tunning*: Also known as parameter control, were parameters are adjusted during the run, and

1.8.1 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 [104] is presented a study of methods to tune these algorithms.

In [105] 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 [106] 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 [107]. 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 [108].

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 [109]. In [110] 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 [111] .

1.8.2 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 [112].

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 [113] 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 [114], 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) [115].

Drozdik et al. present in [116] 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 [117] 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.

1.9 Summary and discussion

In this chapter I have presented an overview of the different techniques to solve Constraint Satisfaction Problems. Special attention was given to the *local-search meta-heuristics*, as well as *parallel computing*, which are directly related to this investigation.

In contrast with tree-based methods (complete methods), *Meta-heuristic methods* have shown good results solving large and complex CSPs, where the search space is huge. They are algorithms applying different techniques to guide the search as direct as possible through the solution. The main contribution of this thesis is presented in Chapter **??**, where is proposed a framework to build local-search meta-heuristics combining small functions (computation modules) through an operator-based language. *Hybridization* is also an important point in this investigation due to their good results in solving CSPs. With the proposed framework, many different solvers can be created using solvers templates (abstract solvers), that can be instantiated with different computation modules.

The era of multi/many-core computers, and the development of parallel algorithms have opened new ways to solve constraint problems. In this field, the solver cooperation has become a very popular technique. In general, the main goal of parallelism is to improve some limitations imposed by the use of unique solver. The present investigation attempts to show the importance and the success of this technique, by proposing a deep study of some parallel communication strategies in Chapter **??**.

2

Bibliography

[1] 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.

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

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

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

[5] 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.

[6] 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.

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

[8] 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.

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

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

[22] 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.

[23] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. Choco Documentation. Technical report, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

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

[25] 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.

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

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

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

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

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

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

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

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

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

[56] 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.

[57] 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.

[58] 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.

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

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

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

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

[63] 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.

[64] 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.

[65] 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.

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

[67] 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.

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

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

[70] 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.

[71] 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.

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

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

[74] 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.

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

[76] 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.

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

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

[79] 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)*.

[80] 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.

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

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

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

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

[85] 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.

[86] 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.

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

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

[89] 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.

[90] 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.

[91] 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.

[92] 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.

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

[94] 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.

[95] 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.

[96] 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.

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

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

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

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

[101] 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.

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

[103] 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.

[104] 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.

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

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

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

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

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

[110] 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.

[111] 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.

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

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

[114] 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.

[115] 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.

[116] 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.

[117] 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.

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

[119] Renaud De Landtsheer, Yoann Guyot, Gustavo Ospina, and Christophe Ponsard. Combining Neighborhoods into Local Search Strategies. In *11th MetaHeuristics International Conference*, Agadir, 2015. Springer.

[120] Simon Martin, Djamila Ouelhadj, Patrick Beullens, Ender Ozcan, Angel A Juan, and Edmund K Burke. A Multi-Agent Based Cooperative Approach To Scheduling and Routing. *European Journal of Operational Research*, 2016.

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

[122] 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.

Part II

Appendix

A

French summary

This appendix contains a detailed summary of the thesis in French language.

Contents

A.1 Introduction

L'optimisation combinatoire a plusieurs applications dans différents domaines tels que l'apprentissage de la machine, l'intelligence artificielle, et le génie du logiciel. Dans certains cas, le but principal est seulement de trouver une solution, comme pour les Problèmes de Satisfaction de Contraintes (CSP). Une solution sera une affectation de variables répondant aux contraintes fixées, en d'autres termes: une solution faisable.

Plus formellement, un CSP (dénoté par $\mathcal{P}$) est défini par le trio $\langle X, D, C\rangle$ où $X = \{x_1, x_2, \ldots, x_n\}$ est un ensemble fini de variables; $D = \{D_1, D_2, \ldots, D_n\}$, est l'ensemble des domaines associés à chaque variable dans X; et $C = \{c_1, c_2, \ldots, c_m\}$, est un ensemble de contraintes. Chaque contrainte est définie en impliquant un ensemble de variables, et spécifie les combinaisons possibles de valeurs de ces variables. Une configuration $s \in D_1 \times D_2 \times \cdots \times D_n$, est une combinaison de valeurs des variables dans X. Nous disons que s est une solution de $\mathcal{P}$ si et seulement si s satisfait toutes les contraintes $c_i \in C$.

Les *CSP*s sont connus pour être des problèmes extrêmement difficiles. Parfois les méthodes complètes ne sont pas capables de passer à l'échelle de problèmes de taille industriel. C'est la raison pour laquelle les techniques méta–heuristiques sont de plus en plus utilisées pour la résolution de ces derniers. Par contre, dans la plupart des cas industriels, l'espace de recherche est assez important et devient donc intraitable, même pour les méthodes méta-heuristiques. Cependant, les récents progrès dans l'architecture de l'ordinateur nous conduisent vers les ordinateurs *multi/many–cœur*, en proposant une nouvelle façon de trouver des solutions à ces problèmes d'une manière plus réaliste, ce qui réduit le temps de recherche.

Grâce à ces développements, les algorithmes parallèles ont ouvert de nouvelles façons de résoudre les problèmes de contraintes: Adaptive Search [41] est un algorithme efficace, montrant de très bonnes performances et passant à l'échelle de plusieurs centaines ou même milliers de cœurs, en utilisant la recherche locale *multi-walk* en parallèle. Munera et al. [85] ont présenté une autre implémentation d'Adaptive Search en utilisant la coopération entre des stratégies de recherche. *Meta–S* est une implémentation d'un cadre théorique présenté dans [91], qui permet d'attaquer les problèmes par la coopération de solveurs de contraintes de domaine spécifique. Ces travaux ont montré l'efficacité du schéma parallèle multi-walk.

De plus, le temps de développement nécessaire pour coder des solveurs en parallèle est souvent sous-estimé, et dessiner des algorithmes efficaces pour résoudre certains problèmes consomment trop de temps. Dans cette thèse nous présentons POSL, un langage orienté parallèle pour construire des solveurs de contraintes basés sur des méta-heuristiques, qui résolvent des *CSP*s. Il fournit un mécanisme pour dessiner des stratégies de communication.

L'autre but de cette thèse est de présenter une analyse détaillée des résultats obtenus en résolvant plusieurs instances des problèmes CSP. Sachant que créer des solveurs utilisant différentes stratégies de solution peut être compliqué et pénible, POSL donne la possibilité de faire des prototypes de solveurs communicants facilement.

A.2 Des travaux reliés

Beaucoup de chercheurs se concentrent sur la programmation par contraintes, particulièrement dans le développement de solution à haut-niveau qui facilitent la construction de stratégies de recherche. Cela permet de citer plusieurs contributions.

Hyperion [57] est un système codé en Java pour méta et hyper-heuristiques basé sur le principe d'interopérabilité, fournissant des patrons génériques pour une variété d'algorithmes de recherche locale et évolutionnaire, et permettant des prototypages rapides avec la possibilité de réutiliser le code source. POSL offre ces avantages, mais il fournit également un mécanisme permettant de définir des protocoles de communication entre solveurs. Il fournit aussi, à travers d'un simple langage basé sur des opérateurs, un moyen de construire des abstract solvers, en combinant des modules déjà définis (computation modules et communication modules). Une idée similaire a été proposée dans [118] sans communication, qui introduit une approche évolutive en utilisant une simple composition d'opérateurs pour découvrir automatiquement les nouvelles heuristiques de recherche locale pour SAT et les visualiser comme des combinaisons d'un ensemble de blocs.

Récemment, [61] a montré l'efficacité de combiner différentes techniques pour résoudre un problème donné (hybridation). Pour cette raison, lorsque les composant du solveurs peuvent être combinés, POSL est dessiné pour exécuter en parallèle des ensembles de solveurs différents, avec ou sans communication. Une autre idée intéressante est proposée dans Templar. Il s'agit d'un système qui génère des algorithmes en changeant des composants prédéfinis, et en utilisant des méthodes hyper-heuristiques [64]. Dans la dernière phase du processus de codage avec POSL, les solveurs peuvent être connectés les uns aux autres, en fonction de la structure de leurs communication modules, et de cette façon, ils peuvent partager non seulement des informations, mais aussi leur comportement, en partageant leurs computation modules. Cette approche donne aux solveurs la capacité d'évoluer au cours de l'exécution.

Renaud De Landtsheer et al. présentent dans [119] un cadre facilitant le développement des systèmes de recherches en utilisant des *combinators* pour dessiner les caractéristiques trouvées très souvent dans les procédures de recherches comme des briques, et les assembler. Dans [120] est proposée une approche qui utilise des systèmes coopératifs de recherche locale

basée sur des méta-heuristiques. Celle-ci se sert de protocoles de transfert de messages. POSL combine ces deux idées pour assembler des composants de recherche locale à travers des opérateurs fournis (ou en créant des nouveaux), mais il fournit aussi un mécanisme basé sur opérateurs pour connecter et combiner des solveurs, en créant des stratégies de communication.

Dans cette thèse, nous présentons quelques nouveaux opérateurs de communication afin de concevoir des stratégies de communication. Avant de clore cet article par une brève conclusion et de travaux futurs, nous présentons quelques résultats obtenus en utilisant POSL pour résoudre certaines instances des problèmes *Social Golfers*, *Costas Array*, *N-Queens* et *Golomb Ruler*.

A.3 Solveurs parallèles POSL

POSL permet de construire des solveurs suivant différentes étapes :

1. L'algorithme du solveur considéré est exprimé via une décomposition en modules de calcul. Ces modules sont implémentés à la manière de *fonctions* séparées. Nous appelons *computation module* ces morceaux de calcul (figure A.1a, blocs bleus). En suite, il faut décider quelles sont les types d'informations que l'on souhaite recevoir des autres solveurs. Ces informations sont encapsulées dans des composants appelés *communication module*, permettant de transmettre des données entre solveurs (figure A.1a, bloc rouge)

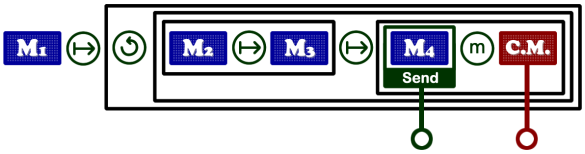
2. Une *stratégie générique* est codée à travers POSL, en utilisant les opérateurs fournis par le langage appliqués sur des modules *abstraite* qui représentent les *signatures* des composants donnés lors l'étape 1., pour créer *abstract solvers*. Cette stratégie définie non seulement les informations échangées, mais détermine également l'exécution parallèle de composants. Lors de cette étape, les informations à partager sont transmises via les opérateurs ad-hoc. On peut voir cette étape comme la définition de la colonne vertébrale des solveurs (figure A.1b).

3. Les solveurs sont créés en instanciant l'abstract solver, par computation modules et communication module.

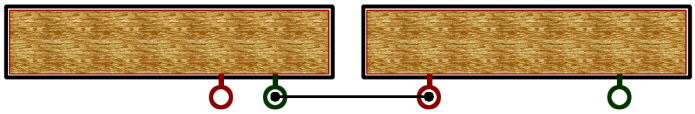
4. Les solveurs sont assemblés en utilisant les opérateurs de communication fournis par le langage, pour creér des strategies de communication. Cet entité final s'appelle *solver set* (figure A.1c).

Les sous-sections suivantes expliquent en détail chacune des étapes ci-dessus.

(a) Definition des modules et les communication modules



(b) Definition de l'abstract solver



(c) Definition de la strategie de communication

Figure A.1: Construire des solveurs parallèles avec POSL

A.3.1 Computation module

Un computation module est la plus basique et abstraite manière de définir un composant de calcul. Il reçoit une entrée, exécute un algorithme interne et retourne une sortie. Dans ce papier, nous utilisons ce concept afin de décrire et définir les composants de base d'un solveur, qui seront assemblés par l'abstract solver.

Un computation module représente un morceau de l'algorithme du solveur qui est susceptible de changer au cours de l'exécution. Il peut être dynamiquement remplacé ou combiné avec d'autres computation modules, puisque les computation modules sont également des informations échangeables entre les solveurs. De cette manière, le solveur peut changer/adapter son comportement à chaud, en combinant ses computation modules avec ceux des autres solveurs. Ils sont représentés par des blocs bleus dans la figure A.1.

Definition 6 ***(Computation Module)*** *Un computation module $\mathcal{O}m$ est une application définie par :*

$$\mathcal{C}m : \mathcal{I} \rightarrow \mathcal{O} \tag{A.1}$$

Dans (6), la nature de $\mathcal{D}$ et $\mathcal{I}$ dépend du type de computation module. Ils peuvent être soit une configuration, ou un ensemble de configurations, ou un ensemble de valeurs de différents types de données, etc.

Soit une méta-heuristique de recherche locale, basée sur un algorithme bien connu, comme par exemple *Tabu Search*. Prenons l'exemple d'un computation module retournant le voisinage d'une configuration donnée, pour une certaine métrique de voisinage. Cet computation module peut être défini par la fonction suivante:

$$\mathcal{C}m : D_1 \times D_2 \times \cdots \times D_n \rightarrow 2^{D_1 \times D_2 \times \cdots \times D_n} \tag{A.2}$$

où D_i représente la définition des domaines de chacune des variables de la configuration d'entrée.

A.3.2 Communication module

Les communication modules sont les composants des solveurs en charge de la réception des informations communiquées entre solveurs. Ils peuvent interagir avec les computation modules, en fonction de l'abstract solver. Les communication modules jouent le rôle de prise, permettant aux solveurs de se brancher et de recevoir des informations. Il sont représentés en rouge dans la figure A.1a.

Un communication module peut recevoir deux types d'informations, provenant toujours d'un solveur tiers : des données et des computation modules. En ce qui concerne les computation modules, leur communication peut se faire via la transmission d'identifiants permettant à chaque solveur de les instancier.

Pour faire la distinction entre les deux différents types de communication modules, nous appelons *data communication module* les communication modules responsables de la réception de données et *object communication module* ceux s'occupant de la réception et de l'instanciation de computation modules.

Definition 7 ***(Data communication module)*** *Un data communication module $\mathcal{C}h$ est un composant produisant une application définie comme suit :*

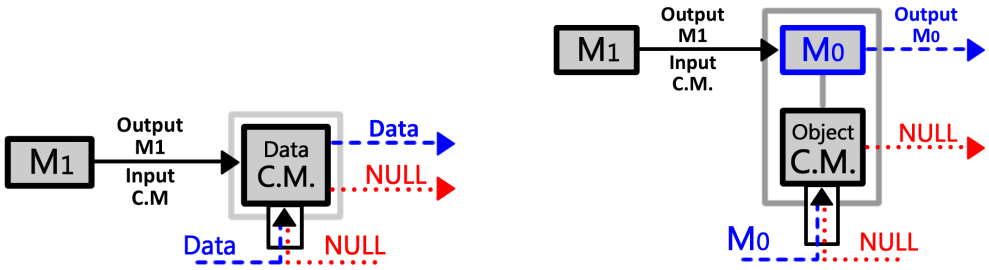
$$\mathcal{C}h : I \times \{D \cup \{NULL\}\} \rightarrow D \cup \{NULL\} \tag{A.3}$$

et retournant l'information $\mathcal{I}$ provenant d'un solveur tiers,quelque soit l'entrée $\mathcal{U}$.

Definition 8 ***(Object communication module)*** *Si nous notons* **M** *l'espace de tous les computation modules de la définition 6, alors un object communication module $\mathcal{C}h$ est un composant produisant un computation module venant d'un solveur tiers défini ainsi :*

$$\mathcal{C}h : I \times \{\mathbf{M} \cup \{NULL\}\} \rightarrow O \cup \{NULL\} \tag{A.4}$$

Puisque les communication modules reçoivent des informations provenant d'autres solveurs sans pour autant avoir de contrôle sur celles-ci, il est nécessaire de définir l'information



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

Figure A.2: Mécanisme interne du communication module

NULL, signifiant l'absence d'information. La figure A.2 montre le mécanisme interne d'un communication module. Si un data communication module reçoit une information, celle-ci est automatiquement retournée (figure A.2a, lignes bleues). Si un object communication module reçoit un computation module, ce dernier est instancié et exécuté avec l'entrée de l'communication module, et le résultat est retourné (figure A.2b, lignes bleues). Dans les deux cas, si aucune information n'est reçue, l'communication module retourne l'objet *NULL* (figure A.2, lignes rouges).

A.3.3 Abstract solver

L'abstract solver est le cœur du solveur. Il joint les computation modules et les communication modules de manière cohérente, tout en leur restant indépendant. Ceci signifie qu'elle peut changer ou être modifiée durant l'exécution, sans altérer l'algorithme général et en respectant la structure du solveur. À travers l'abstract solver, on peut décider également des informations à envoyer aux autres solveurs. Chaque fois que nous combinons certains composants en utilisant des opérateurs POSL, nous créons un *module*.

Definition 9 *Noté par la lettre* $\mathcal{M}$*, un* **module** *est:*

1. *un computation module; ou*
2. *un communication module; ou*
3. $[OP\ \mathcal{M}]$*, la composition d'un module* $\mathcal{M}$ *exécuté sequentielement, en retournant une sortie, en dépendant de la nature de l'opérateur unaire* OP*; ou*
4. $[\mathcal{M}_1\ OP\ \mathcal{M}_2]$*, la composition de deux modules* $\mathcal{M}_1$ *et* $\mathcal{M}_2$ *exécuté séquentiellement, en retournant une sortie, en dépendant de la nature de l'opérateur binaire* OP*.*

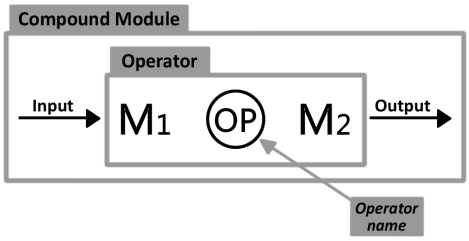


Figure A.3: Un compound module

5. $[\mathcal{M}_1\ OP\ \mathcal{M}_2]$, *la composition de deux modules* $\mathcal{M}_1$ *et* $\mathcal{M}_2$ *exécuté, en retournant une sortie, en dépendant de la nature de l'opérateur binaire* OP. *Ces deux opérateurs vont être exécutes en parallèle si et seulement si* OP *support le parallélisme, ou il lance une exception en cas contraire.*

Nous notons par **M** *l'espace des modules, et nous appelons compound modules à la composition de modules présentés en 3. 4., et/ou 5..*

Pour illustrer la définition 9, la figure A.3 montre graphiquement le concept de compound module.

Dans le cas particulier où un des compound modules impliqués est un communication module, chaque opérateur gère l'information *NULL* à sa manière.

Afin de grouper des modules, nous utiliserons la notation |.| comme un groupe générique qui pourra être indifféremment interprété comme [.] ou comme $[\![.]\!]_p$.

Ensuite, les opérateurs fournis par POSL sont présentés.

Sequential Execution Operator: L'opération $\left|\mathcal{M}_1 \ (\mapsto)\ \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{seq}$ comme le résultat de l'exécution de $\mathcal{M}_1$ suivi de $\mathcal{M}_2$. C'est un exemple d'opérateur ne supportant pas une exécution parallèle de ses compound modulesimpliqués, puisque l'entrée du second compound module est la sortie du premier.

Conditional Execution Operator: L'opération $\left|\mathcal{M}_1 (?)_{<cond>} \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{cond}$ le résultat de l'exécution en séquentiel de $\mathcal{M}_1$ si $< cond >$ es **vrai** or $\mathcal{M}_2$, autrement.

Cyclic Execution Operator: L'opération $|\circlearrowleft_{<cond>} \mathcal{M}|$ définit le compound module $\mathcal{M}_{cyc}$ en répétant séquentiellement l'exécution de $\mathcal{M}$ tant que $< cond >$ est **vrai**.

Random Choice Operator: L'operation $\left|M_1 (\rho) \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{rho}$ qui exécute $\mathcal{M}_1$ en suivant une probabilité ρ, ou en exécutant $\mathcal{M}_2$ en suivant une probabilité $(1-\rho)$.

Not *NULL*: L'operation $\left|\mathcal{M}_1 (\vee) \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{non}$ qui exécute $\mathcal{M}_1$ et retourne une sortie si elle n'est pas *NULL*, ou exécute $\mathcal{M}_2$ et retourne une sortie autrement.

Minimum Operator: Soient o_1 et o_2 les sorties de $\mathcal{M}_1$ et $\mathcal{M}_2$, respectivement. Nous assumons qu'il existe un ordre total dans $I_1 \cup I_2$ où l'objet *NULL* est la plus grand valeur. Alors, l'opération $\left|\mathcal{M}_1 (\mathrm{m}) \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{min}$ qui exécute $\mathcal{M}_1$ et $\mathcal{M}_2$, et retourne $\min\{o_1, o_2\}$.

Maximum Operator: Soient o_1 et o_2 les sorties de $\mathcal{M}_1$ et $\mathcal{M}_2$, respectivement. Nous assumons qu'il existe un ordre total dans $I_1 \cup I_2$ où l'objet *NULL* est la plus petite valeur. Alors, l'opération $\left|\mathcal{M}_1 (\mathrm{M}) \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{max}$ qui exécute $\mathcal{M}_1$ et $\mathcal{M}_2$, et retourne $\max\{o_1, o_2\}$.

Race Operator: L'opération $\left|\mathcal{M}_1 (\downarrow) \mathcal{M}_2\right|$ définit le compound module $\mathcal{M}_{race}$ qui exécute les deux modules $\mathcal{M}_1$ et $\mathcal{M}_2$, et retourne la sortie du module qui termine en premier

Les opérateurs (ρ), $(\vee)$ et (m) sont très utiles en terme de partage d'informations entre solveurs, mais également en terme de partage de comportements. Si un des opérandes est un communication module alors l'opérateur peut recevoir le computation module d'un autre solveur, donnant la possibilité d'instancier ce module dans le solveur le réceptionnant. L'opérateur va soit instancier le module s'il n'est pas *NULL* et l'exécuter, soit exécuter le module donné par le second opérande.

Maintenant, nous présentons les opérateurs nous permettant d'envoyer de l'information vers d'autres solveurs. Deux types d'envois sont possibles : i) on exécute un module et on envoie sa sortie, ii) ou on envoie le module lui-même.

Sending Data Operator: L'opération $\left|(\!|\mathcal{M}|\!)^d\right|$ définit le compound module $\mathcal{M}_{sendD}$ qui exécute le module $\mathcal{M}$ puis envoie la sortie vers un communication module.

Sending Module Operator: L'opération $|(\!|\mathcal{M}|\!)^m|$ définit le compound module $\mathcal{M}_{sendM}$ qui exécute le module $\mathcal{M}$, puis envoie le module lui même vers un communication module.

Avec les opérateurs présentés jusqu'ici, nous sommes en mesure de concevoir les abstract solvers (ou algorithme) de résolution d'un problème de contraintes. Une fois un tel abstract solver définie, on peut changer les composants (computation modules et communication modules) auxquels elle fait appel, permettant ainsi d'implémenter différents solveurs à partir du même abstract solver mais composés de différents modules, du moment que ces derniers respectent la signature attendue, à savoir le types des entrées et sorties.

Un abstract solver est déclaré comme suit: après déclarer les noms de l'**abstract solver** (*name*), la première ligne définit la liste des computation modules abstraites ($\mathcal{L}^m$), la seconde ligne, la liste des communication modules abstraites (**M**), puis l'algorithme du solver est définit comment le corps su solver (the root compound module **M**), entre **begin** et **end**.

Un abstract solver peut être déclaré par l'expression régulière suivante:

abstract solver *name* **computation**: $\mathcal{L}^m$ (**communication**: $\mathcal{L}^c$)? **begin** **M** **end**

Par exemple, l'algorithme 1 montre l'abstract solver correspondant a la figure A.1b.

Algorithm 1: Pseudo-code POSL pour l'abstract solver de la figure A.1b

abstract solver *as_01*
computation : I, V, S, A
connection: $C.M.$
begin
$I \textcircled{\mapsto}$ [$\circlearrowleft$ (Itr $< K_1$) $\left[V \textcircled{\mapsto} S \textcircled{\mapsto} \left[C.M. \textcircled{m} (\!|A|\!)^d\right]\right]$]
end

A.3.4 Créer les solveurs

Maintenant on peut créer les solveurs en instanciant les modules. Il est possible de faire ceci en spécifiant que un **solver** donné doit implémenter (en utilisant le mot clé **implements**) un abstract solver donné, suivi par la liste de computation puis communication modules. Ces modules doivent correspondre avec les signatures exigé par l'abstract solver.

Algorithm 2: Une instanciation de l'abstract solver présenté dans l'algorithme 1

solver solver_01 **implements** as_01
computation : $I_{rand}, V_{1ch}, S_{best}, A_{AI}$
connection: CM_{last}

A.3.5 Connecter les solveurs : créer le solver set

La dernière étape est connecter les solveurs entre eux. POSL fournit des outils pour créer des stratégies de communication très facilement. L'ensemble des solveurs connectés qui seront exécutés en parallèle pour résoudre un CSP s'appelle *solver set*.

Les communications sont établies en respectant les règles suivantes :

1. À chaque fois qu'un solveur envoie une information via les opérateurs $(\!|.|\!)^d$ ou $(\!|.|\!)^m$, il créé une *prise mâle de communication*
2. À chaque fois qu'un solveur contient un communication module, il créé une *prise femelle de communication*
3. Les solveurs peuvent être connectés entre eux en reliant *prises mâles* et *femelles*.

Avec l'opérateur $(\cdot)$, nous pouvons avoir accès aux computation modules envoyant une information et aux noms des communication modulesd'un solveur. Par exemple : $Solver_1 \cdot \mathcal{M}_1$ fournit un accès à le computation module $\mathcal{M}_1$ du $Solver_1$ si et seulement s'il est utilisé par l'opérateur $(\!|.|\!)^d$ (ou $(\!|.|\!)^m$), et $Solver_2 \cdot \mathcal{C}h_2$ fournit un accès au communication module $\mathcal{C}h_2$ de $Solver_2$.

Maintenant, nous définissons les opérateurs de communication que POSL fournit.

Definition 10 Connection One-to-One Operator *Soient*

1. $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *une liste de prises mâles, et*
2. $\mathcal{O} = [\mathcal{Z}_0 \cdot \mathcal{CM}_0, \mathcal{Z}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{Z}_{N-1} \cdot \mathcal{CM}_{N-1}]$ *une liste de prises femelles*

Alors, l'opération

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

connecte chaque prise mâles $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ avec la correspondante prise femelle $\mathcal{Z}_i \cdot \mathcal{CM}_i \in \mathcal{O}$, $\forall\ 0 \leq i \leq N-1$ (voir figure A.4a).

Definition 11 Connection One-to-N Operator *Soient*

1. $\mathcal{J} = [\mathcal{S}_0 \cdot \mathcal{M}_0, \mathcal{S}_1 \cdot \mathcal{M}_1, \ldots, \mathcal{S}_{N-1} \cdot \mathcal{M}_{N-1}]$ *une liste de prises mâles, et*
2. $\mathcal{O} = [\mathcal{Z}_0 \cdot \mathcal{CM}_0, \mathcal{Z}_1 \cdot \mathcal{CM}_1, \ldots, \mathcal{Z}_{M-1} \cdot \mathcal{CM}_{M-1}]$ *une liste de prises femelles*

Alors, l'opération

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

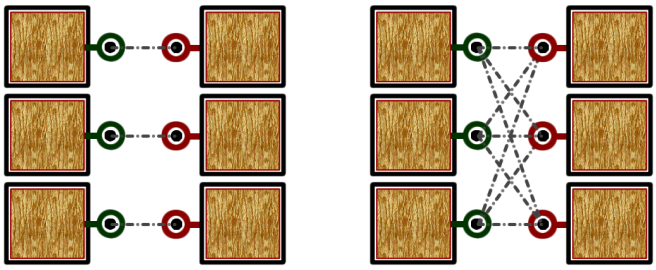
connecte chaque prise mâles $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ avec chaque prise femelle $\mathcal{Z}_j \cdot \mathcal{CM}_j \in \mathcal{O}$, $\forall\ 0 \leq i \leq N-1$ et $0 \leq j \leq M-1$ (see Figure A.4b).

POSL permet aussi de déclarer des solveurs non communicatifs pour les exécuter en parallèle, en déclarant seulement la liste des noms:

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

A.4 Les résultats

Le but principal de cette section est de sélectionner quelques instances de problèmes de référence, pour analyser et illustrer la versatilité de POSL pour étudier des stratégies de solution basées sur la recherche locale méta-heuristique avec communication. Grâce à POSL



(a) Communication 1 à 1 **(b)** Communication 1 à N

Figure A.4: Représentation graphique des opérateurs de communication

nous pouvons analyser des résultats et formuler des conclusions sur le comportement de la stratégie de recherche, mais aussi sur la structure de l'espace de recherche du problème. Dans cette section, nous expliquons la structure des solveurs de POSL que nous avons générés pour les expériences, et les résultats.

Nous avons choisi l'une des méthodes de solutions les plus classique pour des problèmes combinatoires: l'algorithme méta-heuristique de recherche locale. Ces algorithmes ont une structure commune: ils commencent par l'initialisation des structures de données. Ensuite, une configuration initiale s est générée. Après cela, une nouvelle configuration s' est sélectionnée dans le voisinage $V(s)$. Si s' est une solution pour le problème P, alors le processus s'arrête, et s' est renvoyée. Dans le cas contraire, les structures de données sont mises à jour, et s' est acceptée, ou non, pour l'itération suivante, en fonction de certains critères (par exemple, en pénalisant les caractéristiques des optimums locaux).

Les expériences ont été effectuées sur un processeur Intel ® Xeon ™ E5-2680 v2, 10 × 4 cœurs, 2.80GHz. Les résultats montrés dans cette section sont les moyennes de 30 runs pour chaque configuration. Dans les tableaux de résultats, les colonnes marquées **T** correspondent au temps de l'exécution en secondes et les colonnes marquées **It.** correspondent au nombre d'itérations. Toutes les expériences de cette section sont basées sur différentes stratégies en parallèle, avec 40 cœurs.

A.4.1 Social Golfers Problem

Le problème de *Social Golfers* (*SGP*) consiste à planifier $n = g \times p$ golfeurs en g groupes de p joueurs chaque semaine pendant w semaines, de telle manière que deux joueurs jouent dans le même groupe au plus une fois. Une instance de ce problème peut être représentée par le triplet $g - p - w$. Ce problème, et d'autres problèmes étroitement liés, trouvent de nombreuses applications pratiques telles que le codage, le cryptage et les problèmes couvrants. Sa structure nous a semblé intéressante car elle est similaire à d'autres problèmes, comme

Kirkman's Schoolgirl et *Steiner Triple System*, donc nous pouvons construire des modules efficaces pour résoudre un grand éventail de problèmes.

Nous avons utilisé uns stratégie de communication cyclique pour résoudre ce problème, en échangeant la configuration courante entre deux solveurs avec des caractéristiques différentes. Les résultats montrent que cette stratégie marche très bien pour ce problème.

Algorithm 3: Solveur pour *SGP*

```
abstract solver as_eager                          // ITR → nombre d'itérations
computation : I, V, S1, S2, A                     // SCI → nombre d'itérations avec le même coût
begin
    I (↦) [ (↻) (ITR < K1)  V (↦) [S1 (?)_{SCI%K2} S2] (↦) A ]
end
solver SOLVER_eager implements as_eager
    computation : I_BP, V_BAS, S_first, S_rand, A_AI
```

L'algorithme 3 montre l'abstract solver utilisé pour résoudre me manière séquentielle le *SGP*. L'utilisation de deux modules de sélection (S_1 et S_2) est un simple chamanisme pour éviter les minimums locaux: il tente d'améliorer le coût un certain nombre de fois, en exécutant le computation module S_1. S'il n'y arrive pas, il exécute le computation module S_2. L'abstract solver a été instancié par les computation modules suivantes:

1. S_{BP} génère une configuration aléatoire s, en respectant la structure du problème, c'est-à-dire que la configuration est un ensemble de w permutations du vecteur $[1..n]$.

2. V_{BAS} définit le voisinage $V(s)$ permutant le joueur qui a contribué le plus au coût, avec d'autres joueurs dans la même semaine.

3. S_{rirst} sélectionne la première configuration $s' \in V(s)$ qui améliore le coût actuel, et retourne (s, s')

4. S_{rand} sélectionne une configuration aléatoire $s' \in V(s)$, et retourne (s, s')

5. A_{AI} retourne toujours la configuration sélectionnée (s').

Pour *SGP*, nous avons utilisé une stratégie de communication, où un solveur "compagnon", incapable de trouver une solution au final, mais capable de trouver des configurations avec an coût considérablement plus petit que ceci trouvé par le solveur *standard* dans le même instant de temps, au début de la recherche. L'idée c'est d'échanger leurs configurations cycliquement, jusqu'à trouver une solution. Les algorithmes 4 et 5 montrent les solveurs utilisés pour cette stratégie, ou $V_{BP}(p)$ est le computation module de voisinage pour le solveur "compagnon", qui cherche des configurations seulement changeant des joueurs parmi p semaines. Le communication module instancié CM_{last}, prend en compte la dernière configuration reçue quand il est au moment de l'exécution.

Algorithm 4: Solveur standard pour *SGP*

abstract solver *as_standard*
computation : I, V, S_1, S_2, A
communication : $C.M.$
begin
$I \ (\mapsto) \ [(\circlearrowleft) \ (\text{ITR} < K_1) \ \ V \ (\mapsto) \ [S_1 \ (?)_{\text{SCI}\%K_2} \ S_2] \ (\mapsto) \ [C.M. \ (m) \ (\!|A|\!)^d]$]
end
solver SOLVER$_{standard}$ **implements** as_standard
computation : $I_{BP}, V_{BAS}, S_{first}, S_{rand}, A_{AI}$
communication : CM_{last}

Algorithm 5: Solveur compagnon pour *SGP*

abstract solver *as_compagnon*
computation : I, V, S_1, S_2, A
communication : $C.M.$
begin
$I \ (\mapsto) \ [(\circlearrowleft) \ (\text{ITR} < K_1) \ \ V \ (\mapsto) \ [S_1 \ (?)_{\text{SCI}\%K_2} \ S_2] \ (\mapsto) \ [C.M. \ (\vee) \ (\!|A|\!)^d]$]
end
solver SOLVER$_{compagnon}$ **implements** as_compagnon
computation : $I_{BP}, V_{BP}(p), S_{first}, S_{rand}, A_{AI}$
communication : CM_{last}

Nous avons dessiné aussi des différentes stratégies de communication, en combinant des solveurs connectés et non-connectés, et en appliquant des différents opérateurs de communication : one to one et one to N.

Instance	Séquentielle		Parallèle		Coopérative	
	T	It.	T	It.	T	It.
5–3–7	1.25	2,903	0.23	144	**0.10**	98
8–4–7	0.60	338	0.28	93	**0.14**	54
9–4–8	1.04	346	0.59	139	**0.36**	146

Table A.1: Résultats pour *SGP*

Comme nous nous attendions, le tableau A.1 confirme le succès de l'approche parallèle sur le séquentielle. Plus intéressante, les expériences confirment que la stratégie de communication proposée pour cet benchmark est la correcte: en comparant par rapport aux runs en parallèle sans communication, il améliore les runtimes par un facteur de 1.98 (facteur moyen parmi les trois instances). Les résultats coopératifs de ce tableau on été obtenus en utilisant l'opérateur de communication one to one avec 100% de solveurs communicatifs (algorithme 6).

Algorithm 6: Stratégie 100% de communication *compagnon*

$[\text{SOLVER}_{compagnon} \cdot A] \ (\rightarrow) \ [\text{SOLVER}_{standard} \cdot C.M.] \ 20;$
$[\text{SOLVER}_{standard} \cdot A] \ (\rightarrow) \ [\text{SOLVER}_{compagnon} \cdot C.M.] \ 20;$

A.4.2 Costas Array Problem

Le problème *Costas Array* (*CAP*) consiste à trouver une matrice *Costas*, qui est une grille de $n \times n$ contenant n marques avec exactement une marque par ligne et par colonne et les $n(n-1)/2$ vecteurs reliant chaque couple de marques de cette grille doivent tous être différents. Ceci est un problème très complexe trouvant une application utile dans certains domaines comme le sonar et l'ingénierie de radar, et présente de nombreux problèmes mathématiques ouverts. Ce problème a aussi une caractéristique intéressante: même si son espace de recherche grandit factoriellement, à partir de l'ordre 17 le nombre de solutions diminue drastiquement.

Pour ce problème nous avons testé une stratégie de communication simple, où l'information a communiquer est la configuration courante. Pour construire les solveurs, nous avons réutilisé les computation modules de sélection (S_{first}) et de d'acceptation (A_{AI}) et le communication module utilisés dans la résolution de *SGP*. Les autres computation modules sont les suivantes :

1. I_{perm}: génère une configuration aléatoire s, comme une permutation du vecteur $[1..n]$.
2. V_{AS}: définit le voisinage $V(s)$ permutant la variable qui a contribué le plus au coût, avec d'autres.

Pour résoudre *CAP* nous avons eu besoin d'utiliser un computation module de *reset* (T_{AS}) comme machinisme d'exploration. L'algorithme 7 montre le solveur utilisé pour résoudre ce problème séquentiellement. Les résultats des runs en séquentiel et en parallèle sans communication sont montrés dans le tableau A.2. Ils montrent le succès de l'approche en parallèle est montré encore une fois. Afin d'améliorer ces résultats, nous avons appliqué uns stratégies simple de communication : communiquer la configuration courante au moment d'exécuter le critère d'acceptation. Les algorithmes 8 et 9 montrent les solveurs envoyeur et récepteur.

Algorithm 7: Solveur pour *CAP*

```
abstract solver as_hard
computation : I, T, V, S, A
begin
    I (↦) [↻ (Itr < K1)  T (↦) [↻ (Itr % K2) [V (↦) S (↦) A] ] ]
end
solver Solver1 implements as_hard
    computation : I_perm, T_AS, V_AS, S_first, A_AI
```

Un des buts principaux de cette étude a été d'explorer des différentes stratégies de communication. Nous avons ensuite mis en place et testé différentes variantes de la stratégie exposée

Algorithm 8: Solveur envoyeur pour *CAP*

abstract solver *as_hard_sen*
computation *:* I, T, V, S, A
begin
$I \ (\mapsto) \ [\circlearrowleft (\text{Itr} < K_1) \ T \ (\mapsto) \ [\circlearrowleft (\text{Itr} \ \% \ K_2) \ \left[V \ (\mapsto) \ S \ (\mapsto) \ (\!|A|\!)^d\right] \ ] \ ]$
end
solver Solver$_{sender}$ **implements** as_hard_sen
computation : $I_{perm}, T_{AS}, V_{AS}, S_{first}, A_{AI}$

Algorithm 9: Solveur récepteur pour *CAP*

abstract solver *as_hard_rec*
computation *:* I, T, V, S, A
communication *:* $C.M.$
begin
$I \ (\mapsto) \ [\circlearrowleft (\text{Itr} < K_1) \ T \ (\mapsto) \ [\circlearrowleft (\text{Itr} \ \% \ K_2) \ \left[V \ (\mapsto) \ S \ (\mapsto) \ \left[A \ (m) \ C.M.\right]\right] \ ] \ ]$
end
solver Solver$_{receiver}$ **implements** as_hard_rec
computation : $I_{perm}, T_{AS}, V_{AS}, S_{first}, A_{AI}$
communication: CM_{last}

STRATÉGIE	T	It.	% success
Séquentielle	132.73	2,332,088	40.00
Parallèle	25.51	231,262	100.00
Coopérative	**10.83**	**79,551**	100.00

Table A.2: Résultats pour *CAP* 19

ci-dessus en combinant deux opérateurs de communication (one to one et one to N) et des pourcentages différents de solveurs communicantes.

Comme prévu, la meilleure stratégie était basée sur 100% de communication avec l'opérateur one to N (algorithme 10), parce que cette stratégie permet de communiquer un lieu prometteur à l'intérieur de l'espace de recherche à un maximum de solveurs, en reforçant l'intensification.

Algorithm 10: Stratégie de communication one to N 100% pour *CAP*

$[\text{Solver}_{sender} \cdot A(20)] \ (\rightsquigarrow) \ [\text{Solver}_{receiverA} \cdot C.M.(20)]\,;$

A.4.3 N-Queens Problem

Le problème *N-Queens* (*NQP*) demande de placer N rennes sur un échiquier, tel que aucune d'elles peuvent attaquer une autre avec un seul mouvement. C'est un problème introduit en 1848 par le joueur d'échecs Max Bezzelas comme le problème *8-Queens*, et un an après il a

été généralisé comme le problème *N-Queens* par Franz Nauck. Depuis sa création, plusieurs mathématiciens, Gauss inclut, ont travaillé en ce problème. Il a beaucoup d'applications, par exemple, en le stockage en parallèle de la mémoire, le contrôle du trafique, la prévention des deadlocks, les réseaux neurales, etc. [121]. Quelques études suggèrent que le numéro de solutions augmente exponentiellement avec numéro de rennes (N), mais les méthodes de recherche local ont montré des bons résultats avec ce problème [122]. C'est pour cela que nous avons testé quelques stratégies de communication en utilisant POSL, pour résoudre en problème relativement facile a résoudre, mais un utilisant la communication.

Pour construire les solveurs, nous avons réutilisé presque tous les computation modules utilisés pour résoudre le problème *Costas Array* (S_{first}, S_{first}, A_{AI}, et aussi le communication module CM_{last}. Les solveurs utilisés pour les expériences sans communication sont présentés dans l'algorithme 11, ou l'abstract solver est instancié dans le solveurs $\text{SOLVER}_{selective}$ avec le computation module de voisinage $V_{PAS}(p)$, qui reçoie une configuration, et retourne un voisinage $V(s)$ en permutant la variable qui a contribué le plus au coût, avec un pourcentage p des autres.

Algorithm 11: Solveur simple pour *NQP*

abstract solver *as_simple*
computation : I, V, S, A
begin
$I \ (\mapsto) \ [\circlearrowleft (\text{ITR} < K_1) \ V \ (\mapsto) \ S \ (\mapsto) \ A \ ]$
end
solver $\text{SOLVER}_{selective}$ **implements** as_simple
computation : $I_{perm}, V_{PAS}(p), S_{first}, A_{AI}$

Le tableau A.3 présente les résultats des runs en séquentielle et en parallèle. Ces résultats montrent que l'amélioration de l'approche en parallèle par rapport à l'approche séquentielle en utilisant POSL diminue au fur et à mesure que l'ordre du problème augmente.

Pour appliquer l'approche coopératif a la résolution de ce problème, nous avons implémenté une stratégie de communication similaire à cela appliquée avec *SGP*, mais dans ce cas, avec des solveurs qui utilisent le même module de voisinage $V_{PAS}(p)$ mais avec une valeur différente de p et avec un module de sélection différent aussi. Dans cette stratégie de communication la configuration courante est échangée cycliquement entre les solveurs différents. Un solveur *compagnon* utilise le computation module $V_{PAS}(p)$ avec une valeur plus petite de p et le computation moduleS_{best}, donc capable de trouver des configurations prometteuses plus rapidement, mais avec une convergence lente. L'autre solveur est très similaire au solveur utilisé pour les expériences sans communication, mais dans cette stratégie de communication, les solveurs sont à la fois des envoyeurs et des récepteurs (voir l'algorithme 12).

Grâce a cette expérience nous avons été capables de trouver une stratégie de communication (algorithme 13) pour améliorer les temps d'exécution, mais seulement pour des instances

Instance	Sequential		Parallel	
	T	It.	T	It.
250	0.29	8,898	0.19	4,139
500	0.35	4,203	0.24	2,675
1000	0.35	2,766	0.30	2,102
3000	1.50	2,191	1.33	2,168
6000	4.71	3,339	4.57	3,323

Table A.3: Résultats pour *NQP* (séquentielle et en parallèle sans communication)

Algorithm 12: Des solveurs cycliques pour *NQP*

abstract solver *as_cyc*
computation *:* I, V, S_1, S_2, A
communication *:* $C.M.$
begin
$I \ (\mapsto) \ [\circlearrowleft (\text{ITR} < K_1) \ V \ (\mapsto) \ S \ (\mapsto) \ \left[A \ (?)_{\text{ITR}\%K_2} \ \left[(\!|A|\!)^d \ (m) \ C.M.\right]\right]]$
end
solver $\text{SOLVER}_{standard}$ **implements** as_cyc
computation : $I_{perm}, V_{PAS}(2.5), S_{first}, A_{AI}$
communication: CM_{last}
solver $\text{SOLVER}_{compagnon}$ **implements** as_cyc
computation : $I_{perm}, V_{PAS}(1), S_{best}, A_{AI}$
communication: CM_{last}

Instance	Communication 1-1		Communication 1-n		I.R.
	T	It.	T	It.	
250	**0.09**	1,169	0.10	1,224	2.00
500	**0.14**	864	0.15	977	1.65
1000	0.22	889	**0.21**	807	1.39
3000	1.25	1,602	**1.02**	1,613	1.17
6000	4.83	2,938	**4.24**	2,537	1.01

Table A.4: Résultats de la communication cyclique avec *NQP*

petites du problème. Ce résultat confirme l'hypothèse de que quand l'ordre du problème monte, le gain en utilisant la communication pendant la recherche de la solution diminue. Le tableau A.4 montre comment l'*improvement ratio* (colonne **I.R.**) diminue avec l'ordre N.

A.4.4 Golomb Ruler Problem

Le *Golomb Ruler Problem* (*GRP*) consiste à trouver un vecteur ordonné de n entiers non négatifs différents, appelés *marques*, $m_1 < \cdots < m_n$, tel que toutes les différences $m_i - m_j$, $(i > j)$ sont toutes différentes. Une instance de ce problème est défini par le paire (o, l) où o est l'ordre du problème, (le nombre de *marques*) et l est la longueur de la règle (la dernière *marque*). Nous supposons que la première *marque* est toujours 0. Lorsque nous appliquons POSL pour résoudre une instance de problème séquentiellement, nous pouvons remarquer qu'il effectue de nombreux *restars* avant de trouver une solution. Pour cette raison, nous avons choisi ce problème pour étudier une stratégie de communication intéressante: communiquer la configuration actuelle afin d'éviter son voisinage, c'est à dire, une configuration *tabu*.

Nous réutilisons les modules de sélection et d'acceptation des études antérieures (S_{first} etA_{AI}) pour concevoir les abstract solvers. Les nouvelles modules sont:

1. I_{sort}: renvoie une configuration aléatoire s en tant que vecteur d'entiers trié. La configuration est générée *loin* de l'ensemble des configurations *tabu* arrivés via communication entre solveurs.

2. V_{sort}: donné une configuration, retourne le voisinage en changeant une valeur tout en gardant l'ordre, à savoir, le remplacement de la valeur s_i par toutes les valeurs possibles $s'_i \in D_i$ en satisfaisant $s_{i-1} < s'_i < s_{i+1}$.

Nous avons également ajouté un module de reset T: il reçoit et renvoie une configuration. Le computation module utilisé pour l'instancier (T_{tabu}) insère la configuration reçue dans une liste *tabu* à l'intérieur du solveur et retourne la configuration d'entrée telle quelle. L'algorithme 14 présente le solveur utilisé pour envoyer des informations (solveur envoyeur).

Algorithm 14: Solveur envoyeur pour *GRP*

abstract solver *as_golomb_sender*
computation : I, V, S, A, T
begin
$[\circlearrowleft (\text{Itr} < K_1)\ \ I \ (\mapsto)\ \ [\circlearrowleft (\text{Itr } \mathbf{\%}\ K_2)\ \left[V \ (\mapsto)\ S \ (\mapsto)\ A\right]\ ]\ \ (\mapsto)\ (\!|T|\!)^d\ \ ]$
end
solver Solver_{sender} **implements** as_golomb_sender
computation : $I_{sort}, V_{sort}, S_{first}, A_{AI}, , T_{tabu}$

Le module T_{tabu} est exécuté lorsque le solveur est incapable de trouver une meilleure configuration autour de l'actuelle: elle est supposée être un minimum local, et elle est envoyée au solveur récepteur. L'algorithme 15 présente solveur utilisé pour recevoir l'information. Le communication module CM_{set} reçoit plusieurs configurations qui sont reçus par le computation module I_{sort} comme entrées.

Algorithm 15: Solveur récepteur pour *GRP*

abstract solver *as_golomb_receiver*
computation *:* I, V, S, A, T
connection *:* $C.M.$
begin
$[\circlearrowleft (\text{Itr} < K_1) \ \left[C.M. \circledcirc{\mapsto} I\right] \circledcirc{\mapsto} [\circlearrowleft (\text{Itr} \ \mathbf{\%} \ K_2) \ \left[V \circledcirc{\mapsto} S \circledcirc{\mapsto} A\right]] \circledcirc{\mapsto} T \ ]$
end
solver $\text{Solver}_{receiver}$ **implements** as_golomb_receiver
computation : $I_{sort}, V_{sort}, S_{first}, A_{AI}, , T_{tabu}$
communication: CM_{set}

Le bénéfice de l'approche en parallèle avec POSL est aussi prouvé pour le *GRP* (voir le tableau A.5). Dans ce tableau, la colonne **R** représente le nombre de redémarrages exécutées. Cette expérience a été réalisée en utilisant des solveurs similaires à ceux présentés précédemment, mais sans communication modules.

Instance	Séquentiel				Parallèle		
	T	It.	R	% success	T	It.	R
8–34	0.66	10,745	53	100.00	0.43	349	1
10–55	67.89	446,913	297	88.00	4.92	20,504	13
11–72	117.49	382,617	127	30.00	85.02	155,251	51

Table A.5: Résultats non coopératifs pour *GRP*

Pour *GRP*, la stratégie de communication que nous adoptions a été différente. L'idée de cette stratégie est de profiter des nombreux redémarrages indiqués dans le tableau A.5. Chaque fois qu'un solveur redémarre, la configuration actuelle est communiquée pour alerter les solveurs et éviter son voisinage. De cette façon, chaque fois qu'un solveur redémarre, il génère une nouvelle configuration assez loin de ces "zones pénalisées".

Sur la base de l'opérateur de connexion utilisé dans la stratégie de communication, ce solveur peut recevoir une ou plusieurs configurations. Ces configurations sont l'entrée du module de génération (I_{sort}). Ce module insère toutes les configurations reçues dans une liste *tabu*, puis il génère une nouvelle première configuration loin de toutes les configurations dans la liste *tabu*.

Instance	Communication one to one			Communication one to N		
	T	It.	R	T	It.	R
8–34	0.44	309	1	**0.43**	283	1
10–55	3.90	15,437	10	**3.16**	12,605	8
11–72	85.43	156,211	52	**60.35**	110,311	36

Table A.6: Résultats avec communication pour *GRP*.

Comme nous pouvons voir dans le tableau A.6 l'amélioration en temps d'exécution avec communication est plus visible quand on utilise l'opérateur de communication one to N (al-

gorithme 16), parce-que à chaque nouvelle itération, le solveur récepteur a plus d'information afin de générer une nouvelle configuration loin des "zones pénalisées".

Algorithm 16: Stratégie de communication one to N pour *GRP*

$[\text{SOLVER}_{sender} \cdot R(20)] \ (\rightsquigarrow) \ [\text{SOLVER}_{receiver} \cdot C.M.(20)]$;

A.5 Conclusions

Dans cette thèse, nous avons présenté POSL, un système pour construire des solveurs parallèles coopératifs. Il propose une manière modulable pour créer des solveurs capables d'échanger n'importe quel type d'informations, comme par exemple leur comportement même, en partageant leurs computation modules. Avec POSL, de nombreux solveurs différents pourront être créés et lancés en parallèle, en utilisant une unique stratégie générique mais en instanciant différents computation modules et communication modules pour chaque solveur.

Il est possible d'implémenter différentes stratégies de communication, puisque POSL fournit une couche pour définir les canaux de communication connectant les solveurs entre eux.

Nous avons présenté aussi des résultats en utilisant POSL pour résoudre des instances des problèmes classiques CSP. Il a été possible d'implémenter différentes stratégies communicatives et non communicatives, grâce au langage basé sur des opérateurs fournis, pour combiner différents computation modules. POSL donne la possibilité de relier dynamiquement des solveurs, étant capable de définir des stratégies différentes en terme de pourcentage de solveurs communicatifs. Les résultats montrent la capacité de POSL à résoudre ces problèmes, en montrant en même temps que la communication peut jouer un rôle décisif dans le processus de recherche.

POSL a déjà une importante bibliothèque de computation modules et de communication modules prête à utiliser, sur la base d'une étude approfondie sur les algorithmes méta-heuristiques classiques pour la résolution de problèmes combinatoires. Dans un avenir proche, nous prévoyons de la faire grandir, afin d'augmenter les capacités de POSL.

En même temps, nous prévoyons d'enrichir le langage en proposant de nouveaux opérateurs. Il est nécessaire, par exemple, d'améliorer le langage de *définition du solveur*, pour permettre la construction plus rapide et plus facile des ensembles de nombreux nouveaux solveurs. En plus, nous aimerions élargir le langage des opérateurs de communication, afin de créer des stratégies de communication polyvalentes et plus complexes, utiles pour étudier le comportement des solveurs.