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 links with Constraint Satisfaction Problems (CSP) are introduced, where I concentrate the main efforts, and I give some examples. The basic techniques used to solve these problems are introduced: constraint programing (Section 1.2) and meta–heuristic methods (Section 1.3). I also present some advanced techniques like hyper–heuristic methods in Section 1.4, 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

Combinatorial Optimization problems come from industrial and real world. We can find them in *Resource Allocations* [1], *Task Scheduling* [2], *Master-keying* [3], *Traveling Salesman* and *Knapsack* problems, among others, which are well-known examples of combinatorial optimization problems [4]. They are a particular case of *optimization* problems, and their main goal is to find an optimum value (minimal or maximal, depending on the problem) for a discrete function f, called *objective function*, involving a set variables $X = \{x_1, \dots, x_n\}$ defined over a set $D = \{D_1, \dots, D_n\}$ of discrete domains. These problems generally contain restrictions on the variables called *constraints*, defining the set of forbidden combinations of values for variables in X, tacking into account the problem characteristics.

A *configuration* $s \in D_1 \times D_2 \times \cdots \times D_n$ is a combination of values for the variables in X. The fact of assigning values $v_i \in D_i$ to all variables in $x_i \in X$ is called *evaluation.* When this evaluation is only performed to a given set of variables in X, we called *partial evaluation.* In combinatorial optimization, a *feasible* configuration is a configuration fulfilling all constraints. Finally, a *solution* s^* to the problem is a configuration such that $f(s^*)$ is optimal.

In many practical cases, the main goal is not to find the optimal solution, but finding one feasible configuration. This is the case of Constraint Satisfaction Problems. Formally, we present the definition of a CSP.

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

- $X = \{x_1, \dots, x_n\}$ *is finite a set of variables,*
- $D = \{D_1, \dots, D_n\}$ *is the set of associated domains. Each domain D_i specifies the set of possible values to the variable x_i.*
- $C = \{c_1, \dots, c_m\}$ *is a set of constraints. Each constraint is defined involving some variables from X, and specifies the possible combinations of values for these variables.*

In CSPs, a solution is a configuration satisfying all constraints $c_i \in C$.

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

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

Let $Var(c_i)$ be the set of involved variables $\{x_1, \dots, x_p\}$ in the constraint c_i, with $p \leq n$. Then, $c_i\,(s)$ denotes the evaluation using the values from the configuration s to the variables $Var(c_i)$. The set of all solutions of $\mathcal{P}$ is denoted by $Sol(\mathcal{P})$.

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.

Galinier et al. present in [5] a general approach for solving CSPs. In this work, authors present the concept of *penalty functions* that I 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:*

$$\omega_i : D\,(X) \times 2^{D(X)} \rightarrow \mathbb{R}^+ \textit{ where:}$$
$$\omega_i\,(s, c_i) = \begin{cases} 0 & \textit{if} \quad c_i(s) \textit{ is true} \\ k \in \mathbb{R}^+ \setminus 0 & \textit{otherwise} \end{cases}$$

where $D\,(X) = D_1 \times D_2 \times \cdots \times D_n$

This penalty function defines the cost of a configuration with respect to a given constraint, so if $\omega_i\,(s, c_i) = k$ we say that the configuration s has a local cost k with respect to the constraint c_i. 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:*

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

This global penalty function defines the cost of a configuration with respect to a given set of constraints, so if $\Omega\,(s, C) = k$ we say that the configuration s has a cost k with respect to C. We can now formulate a Constraint Satisfaction Problem as an *unrestricted optimization problem*:

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* $\mathcal{P}_{opt}\langle X, D, C, f\rangle$ *as follows:*

$$\min_{X} f\left(X, C\right)$$

Where: $f\left(X, C\right) \equiv \Omega\left(X, C\right)$ *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 $f\left(s, C\right) = 0$, which means that s satisfies all the constrains in the original CSP $\mathcal{P}$. This work focuses in solving the Constraint Satisfaction Problem using this formulation.

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. This field 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.

Modeling a constrained problem, to be solved using constraint programing techniques, means properly choosing variables and their domains, and a right and efficient representation of the constraints set, aiming to declare as explicitly as possible, the solutions space. On the right election of the problem's model depends not only finding the solution, but also doing it in a fast and efficient way. 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 [6]. 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 [7]. The main goal is to choose one (or some) constraint(s) and enforcing certain consistency levels in the constraint. Achieving global consistency is desirable, because only using brute force algorithms one can arrive to a solution, but this is extremely hard to obtain. For that reason, some other consistency levels, easier to achieve, have been defined, 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 $c(v_1, \dots, v_k)$ 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 $\mathit{Bnd}(D_i) = \{\min(D_i), \max(D_i)\}$ 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 $v_j \in \mathit{Bnd}(D_j)$ with $1 \leq j \leq 2$:

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

such that $c(v_1, \dots, v_k)$ 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 notice that arc consistency is a stronger property, but heavier to enforce.

Apt and Monfroy have been proposed in [8] and [9], respectively, a formalization of constraint propagation through *chaotic iterations*, which is a technique that comes from numerical analysis to compute limits of iterations of finite sets of functions, and adapted for computer science needs for naturally explain constraint propagation [10, 11]. Another approach is presented by Monfroy in [12], a coordination-based chaotic iteration algorithm for constraint propagation, which is a scalable, flexible and generic framework for constraint propagation using coordination languages, not requiring special modeling of CSPs. Zoeteweij provides an implementation of this algorithm in DICE (Distributed Constraint Environment) [13] 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.

Another implementation of constraint propagation is proposed by Granvilliers et al. in [14], using 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 by Prud'homme et al. 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).

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 in testing 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 backtracking search.

It can be seen as performing a depth–first traversal of a search tree. This search tree is generated as the search progresses and represents alternative choices that may have to be examined in order to find a solution. Constraints are used to check whether a node may possibly lead to a solution of the CSP and to prune subtrees containing no solutions. A node in the search tree is a *dead-end* if it does not lead to a solution. Differences between searches 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, *OR-tools*, GECODE and `Choco`. 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* constraint, the *cumulative* constraint, 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.

1.3 Meta-heuristic methods

Meta-heuristic methods are algorithms generally applied to solve problems without deprived of satisfactory problem-specific algorithms to solve them. They are general purpose techniques widely used to solve complex optimization problems in industry and services, in areas ranging from finance to production management and engineering, with relatively few modifications: i) they are nature-inspired (based on some principles from physics or biology), ii) they involve random variables as an stochastic component, therefore approximate and usually non-deterministic, iii) and they have several parameters that need to be fitted [24].

A Meta-heuristic Method is formally defined as an iterative process which guides a subordinate heuristic by combining different concepts for *exploration* (also called *diversification*) i.e. guiding the search process through a much larger portion of the search space, and *exploitation* (also called *intensification*) i.e. guiding the search process into a limited, but promising, region of the search space [25].

In contrast with tree-search based methods, which are subject to combinatorial explosion (required time to find solutions of NP-hard problems increases exponentially w.r.t. the problem size), they do dot perform an ordered and complete search. For that reason they are not able to provide a proof that the optimal solution will be found in a finite (although often prohibitively large) amount of time. Meta-heuristics are therefore developed specifically to find a "*acceptably good*" solution "*acceptably*" fast. In the case of CSPs, finding a feasible solution is enough, for that reason, these methods have been proven to be effective solving these kind of problems.

Sometimes meta-heuristics use domain-specific knowledge in the form of heuristics controlled by an upper level strategy. Nowadays more advanced meta-heuristics use search experience to guide the search [26].

Meta-heuristics are divided into two groups:

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.* They usually start from a candidate configuration s (usually random) inside the search space, and then iteratively make local moves consisting of applying some local modifications to s to create a set of configuration called *neighborhood* $\mathcal{V}(s)$, and selecting a new configuration $s' \in \mathcal{V}(s)$, following some criteria, to be the new candidate solution for the next iteration. This process is repeated until a solution for the problem is found. These methods can be seen as an extension of *local search methods* [27]. Local search methods are the most widely used approaches to solve Combinatorial Optimization Problems because they often produces high–quality solutions in reasonable time [28].

Simulated Annealing (SA) [29] is one of the first algorithms with an explicit strategy to escape from local minima. It is a method inspired by the annealing technique used by 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 under certain probability, in order to scape from local minima, which is decreased during the search process [26]. As an example of an implementation of this algorithm obtaining good results, it can be cited a work presented by Anagnostopoulos et al. in [30] which is an adaptation of 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 a very classic meta-heuristic for Combinatorial Optimization Problems. It explicitly maintains 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] by Iván Dotú and Pascal Van Hentenryck to solve instances of the *Social Golfers* problem, showing that local search is a very effective way to solve this problem. The used approach does not take symmetries into account, leading to an algorithm which is significant simpler than constraint programming solutions.

Guided Local Search (GLS) [33] consists of dynamically changing the objective function to change the search landscape, helping the search escape from local minima. 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 [26]. Mills et al. propose in [34] an implementation of a GLS, which is used to solve the satisfiability (SAT) problem, a special case of a CSP where variables take booleans values and constraints are disjunctions of literals (i.e. variables or theirs negations).

The *Variable Neighborhood Search* (VNS) is another meta-heuristic that systematically changes the neighborhood size during the search process. This neighborhood 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, 26]. Bouhmala et al. introduce in [36] a *generalized Variable Neighborhood Search* for Combinatorial Optimization Problems, where the order in which the neighborhood structures are selected during the search process offers a more effective mechanism for diversification and intensification.

Greedy Randomized Adaptive Search Procedures (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). The first one constructs an initial solution via an adaptive randomized greedy function. This function construct a solution performing partial evaluations using values of a restricted candidate list (RCL) formed by the best values, incorporating to the current partial solution values resulting in the smallest incremental costs (the greedy aspect of the algorithm). The value to be incorporated into the partial solution is randomly selected from those in the RCL (the random aspect of the algorithm). Then the candidate list is updated and the incremental costs are reevaluated (the adaptive aspect of the algorithm). The second phase applies a local search procedure to the constructed solution in to find an improvement [37]. 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 [26]. However, Resende et al. introduce in [38] some extensions like alternative solution construction mechanisms and techniques to speed up the search are presented.

Many other implementations of local search algorithms have been presented with good results. *Adaptive Search* is an algorithm based local search method, taking advantage of the structure of the problem in terms of constraints and variables. It uses also the concept of *penalty function*, 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 [39, 40, 41]. In [42] Munera et al. based their solution method in Adaptive Search to solve the *Stable Marriage with Incomplete List and Ties* problem [43], a natural variant of the *Stable Marriage Problem* [44], using a cooperative parallel approach. Michel and Van Hentenryck propose in [45] a constraint-based, object-oriented architecture to significantly reduce the development time of local search algorithms. This 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, illustrated using COMET, an optimization platform that provides a Java-like programming language to work with constraint and objective functions [46, 47], supporting the local search architecture. It also provides abstraction features to make a clean separation between the model an the search (promoting the reusing of the later) and novel control structures to implement nondeterminism.

1.3.2 Population Based Meta-heuristic

In the second group of meta-heuristic algorithms, we can find the methods based on populations. These methods do not work with a single configuration, but with a set of configurations named *population.* They were not part of the main investigation of this thesis, so I will nos get into details, but I thinks it is fair to mention some of the most important methods.

The most popular algorithms in this group are population-based methods. They are related to i) *Evolutionary Computation* (EC), inspired by the "Darwin's principle", where only the best adapted individuals will survive, where a population of individuals is modified through recombination and mutation operators, and ii) *Swarm Intelligence* (SI), where the idea is to produce computational intelligence by exploiting behaviors of social interaction [27].

Algorithms based on evolutionary computation have a general structure. Every iteration of the algorithm corresponds to a *generation*, where a population of candidate solutions (called individuals) to a given problem, is capable of reproducing and is subject to genetic variations and environmental pressure that causes natural selection. New solutions are created by applying recombination, by combining two or more selected individuals (parents) to produce one or more new individuals (the offspring). Mutation can be applied allowing the appearance of new traits in the offspring to promote diversity. The fitness (how good the solutions are) of the resulting solutions is evaluated and a suitable selection strategy is then applied to determine which solutions will be maintained into the next generation. As a termination condition, a predefined number of generations (or function evaluations) of simulated evolutionary process is usually used. 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 [48].

Probably the most popular evolutionary algorithms are *Genetic Algorithms* (GA) [49], where operators are based on the simulation of the genetic variation process to achieve individuals (solutions in this case) more adapted. GAs are usually differently implemented according to the problem: representation of solution (chromosomes), selection strategy, type of crossover (the recombination operator) and mutation operators, etc. The most common representation of the chromosomes is a fixed-length binary string, because simple bit manipulation operations allow the easy implementation of crossover and mutation operations.

Swarm intelligence based methods are inspired by the collective behavior in society of groups different form of live. SI systems are typically made up of a population of simple element, capable of performing certain operations, interacting locally with one another and with their environment. These elements have very limited individual capability, but in cooperation with others can perform many complex tasks necessary for their survival. Ant colony optimization, Particle Swarm Optimization and Bee Colony Optimization are examples to this approach.

Ant Colony optimization algorithms are inspired by the behavior of real ants. Ants searching for food, initially explore the area surrounding the nest by performing a randomized walk. Along the path between food source and nest, ants deposit a pheromone trail on the ground in order to mark some promising path that should guide other ants to the food source. After some time, the shortest path between the nest and the food source has a higher concentration of pheromone, so it attracts more ants [50].

Particle Swarm optimization uses the metaphor of the flocking behavior of birds to solve optimization problems. Each element of the swarm is a candidate solution to the problem, stochastically generated in the search space, and they are connected to some others elements called *neighbors*. It is represented by a velocity, a location in the search space and has a memory which helps it to remember its previous best position. This values describe the *influence* of each element over its neighbors [51].

Bee Colony optimization consists of three groups of bees: employed bees, onlookers and scout bees. A food source is a possible solution to the problem. Employed bees are currently exploiting a food source. They exploit the food source, carry the information about food source back to the hive and share it with onlooker bees. Onlookers bees wait in the hive for the information to be shared with the employed bees to update their knowledge about discovered food sources. Scouts bees are always searching for new food sources near the hive. Employed bees share information about the nectar amount of a food source by dancing in the designated dance area inside the hive. This information represents the quality of the solution. The nature of dance is proportional to the nectar content of food source. Onlooker bees watch the dance and choose a food source according to the probability proportional to the quality of that food source. In that sense, good food sources attract more onlooker bees. Whenever a food source is fully exploited, all employed bees associated with it abandon the food source, and become scouts [52].

1.4 Hyper-heuristic Methods

Hyper-heuristics are automated methodologies for selecting or generating meta-heuristics algorithms 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. 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].

This approach have been followed for solving constrained problems. HYPERION^2 [56] is a Java 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. It promotes interoperability via component interfaces, allowing rapid prototyping of meta- and hyper-heuristics, with the potential of using the same source code in either case. It also provides generic templates for a variety of local search and evolutionary computation algorithms, making easier the construction of novel meta- and hyper-heuristics by hybridization (via interface interoperability) or extension (subtype polymorphism). HYPERION^2 is faithful to "*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, but using a new object-oriented architecture. It 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 the other one for the acceptance criteria [57].

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 [58] Dioşan et al. use linear genetic programming and multi-expression genetic programming to optimize the EA solving unimodal mathematical functions and another EA to adjust 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. An different but interesting point of view is presented in [59], where Samulowitz et al. present *Snappy*, a *Simple Neighborhood-based Algorithm Portfolio* written in *Python*. It is a very resent framework that aims to provide a tool able to 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. It incorporates available knowledge coming from portfolio's runs, by considering the following ways incrementally: 1- Every time a test instance is considered, it is added to the current set of training instances. 2- After selecting an algorithm for a given test instance, the actual runtime information for the selected algorithm on this instance is added to the data set. It means that the difference between neighborhoods of different algorithms represent how often algorithms will be selected. Other interesting idea is proposed by Swan et al. in TEMPLAR, a framework to generate algorithms changing predefined components using hyper-heuristics methods [69].

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. We can find hybridization in combining algorithms in the same branch of investigation. This is the case of *ParadisEO*, a framework to design parallel and distributed hybrid meta-heuristics showing very good results, including a broad range of reusable features to easily design evolutionary algorithms and local search methods [70]. But we can find hybridization also in combining very different techniques, like the work of El-Ghazali Talbi presented in [66], which is 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.

However, maybe one of the most common in this field, is the combination of meta-heuristic methods and constraint programming techniques. Constraint programming algorithms are based on backtracking mechanisms. These algorithms, also called *complete method* usually explore the search space systematically, and thus guarantee to find a solution if one exists. Meta-heuristic methods may find a solution to a problem, but they can fail even if the problem is satisfiable, because of its local nature. They perform a probabilistic exploration of the search space, so they are not able to guarantee finding a solution. For that reason they are also know as *incomplete methods*. However, they are more efficient (wit respect to response time) than complete methods. The challenge is trying to get the best of both of these methods: exploration of a neighborhood from meta-heuristics, and the power of propagation from constraint programming [60, 61, 62].

Hooker J.N. presents in [65] 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. Following this direction, Monfroy et al. present in [67, 68] 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.

A popular way of hybridization is the *portfolio approach*, which is a methodology exploiting the significant variety in performances of different algorithms and combining them in order to create a globally better solver. In [73], Amadini et al. propose `xcsp2mzn`, a tool for converting problem instances from the XCSP format, to MiniZinc. and `mzn2feat`, a tool to extract static and dynamic features from the MiniZinc representation, with the help of the Gecode interpreter, allowing a better and more accurate selection of the solvers to use according to the instances to solve. Based also on the portfolio approach, Amadini et al. propose in [72] a *time splitting* technique 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: 1. runs solver Σ_1 on P for a period of time t_1, 2. then, for $i = 1, \ldots, n-1$, runs solver 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 during t_i units of time. *Autonomous search* is a technique based on supervised or controlled learning. This system are another portfolio point of view presented by Hamadi et al. in [71], which improves its performance while it solves problems, either modifying its internal components to take advantage of the opportunities in the search space, or choosing adequately the solver to use.

An interesting hybridization point of view, is the integration of operations research into constraint programming. Fontaine et al. use in [63] a generalization of the optimization paradigm *Lagrangian relaxation*, to relax the hard constraints into the objective function, and applying them into constraint-programming and local search models. It combine the concepts of constraint violation (typically used in constraint programming and local search) and constraint satisfiability (typically used in mathematical programming). Hooker J.N. presents in [64] a detailed description of how operation research models like mixed integer linear programming (MILP) models (which can themselves be relaxed), Lagrangian relaxations, and dynamic programming models can be applied to constraint programming.

1.6 Parallel computing

Despite advances previously presented, hard instances of many problems are still complicated to solve through these techniques. Thanks to *parallel computing*, we have been capable of going one step further in solving CSPs. Parallel computing is a way to solve problems using several computation resources at the same time. It is a powerful alternative to solve problems which would require too much time by using sequential algorithms [75].

Since the late 2000's 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 constrained problems [76]. 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 [77, 78].

In the literature on parallel constraint solving [81], two main limiting factors on performance are addressed: 1- inter-process communication overheads (explained in details in Section 1.7), and 2- the *Amdahl*'s law. The Amdahl's law of parallel computing states that the *speed-up* of a parallel algorithm is limited by the fraction of the program that must be executed sequentially. It means that adding more processors may not make the program run faster. It assumes that some percentage of the program or code cannot be parallelized ($T_{sequential}$), and states that the ratio called speed-up of $T_{sequential}$ over $1 - T_{Sequerntial} = T_{parallel}$ is bounded by $1 \setminus T_{sequential}$ when the number of processors $P \to \infty$:

$$Sepeed - Up = \frac{T_{sequential}}{T_{parallel}} \leq \frac{1}{T_{sequential}} \tag{1.1}$$

Another issue, usually underestimated, is the codification. Writing efficient code for parallel machines is less trivial, as it usually involves dealing with low-level APIs such as OpenMP and 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 sequential approaches. In that sense, Falcou proposes in [80] a programming model: *parallel algorithmic skeletons* (along with a C++ implementation called QUAFF) to make parallel application development easier. 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. It relies on C++ template meta-programming to reduce the overhead traditionally associated with object-oriented implementations of such libraries allowing some code generation at compilation time. Cahon et al. also propose *ParadisEO*, a framework to design parallel and distributed hybrid meta-heuristics showing very good results, including a broad range of reusable features to easily design evolutionary algorithms and local search methods [70].

The development of techniques and algorithms to solve problems in parallel focuses principally on three fundamentals aspects:

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

They all pursue the same objective: achieving good levels of *scalability*. Scalability is the ability of a system to handle the increasing growth of workload. *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 of cores [39]. For this algorithm, an implementation of a cooperative multi-walks strategy has been published by Munera et al. in [90]. 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, authors propose a parallel solution strategy able to solve hard instances of *Stable Marriage with Incomplete List and Ties Problem* quickly. This technique has been combined in [91] with an *Extremal Optimization* procedure: a nature-inspired general-purpose meta-heuristic [92].

The issue of subdividing 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. For that reason it is imperative to apply some complementary 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 [82, 89]. In [88] Arbab and Monfory propose a mechanism to create sub-CSPs (whose union contains all the solutions of the original CSP) by splitting the domain of the variables. The coordination is achieved though communication between processes. The contribution of this work is explained in details in Section 1.7.

In [100] Yasuhara et al. propose 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 [82, 101]; 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 [102]. 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 there 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 oe ore some objective function values, implies the deterioration of at least one of the other objective function. A collection of pareto optimal points defines a pareto front.

Related to parallelizing 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*, consisting of 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 [39].

Kishimoto et al. present in [83] a 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 *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 it if they become unoccupied. Authors use MPI, the 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) [106].

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) [106]

The same approach is used by Jinnai and Fukunaga in [84] to evaluate *Zobrist Hashing*, an efficient hash function designed for table games like Chess and Go, to mitigate communication overheads.

In [85] Arbelaez et al. present 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 method, this work aims to improve the scalability of the sequential algorithm, which is measured in terms of the quality of the solution within a given timeout. 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 [86]); and second, *k-Means* algorithm, one of the most popular clustering algorithms [87].

An interesting work is presented by Truchet et al. in [94], which is an estimation of the speed-up (a performance prediction of a parallel algorithm) through statistical analysis of its sequential algorithm is presented. Using this approach it is possible to have a rough idea of the resources needed to solve a given problem in parallel. In this work, authors study the parallel performances of *Las Vegas* algorithms (randomized algorithms whose runtime might vary from one execution to another, even with the same input) under independent multi-walk scheme, and predict the performances of the parallel execution from the runtime distribution of their sequential runs. These predictions are compared to actual speedups obtained for a parallel implementation of the same algorithm and show that the prediction can be quite accurate.

The other important aspect in parallel computing is the inter-process communication, also called *solver cooperation* and it is treated in the next section.

1.7 Solvers cooperation

The interaction between solvers exchanging information is called *solver cooperation.* 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.

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 the work presented by Hamadi et al. in [95], where 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. This approach allows to perform the clause-sharing, controlling the exchange between any pair of processes.

This is a very changeling field, that is way we can find a lot of interesting ideas in the literature to improve parallel solutions through solver cooperation techniques.

Kishimoto et al. present in [104] a parallelization of an algorithm A* (Hash Distributed A*) for *optimal sequential planning* [105], 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 [107] Pajot and Monfroy present a paradigm that enables the user to properly separate strategies combining solver applications, from the way the search space is explored in solver cooperations. The cooperation must be supervised by the user, through *cooperation strategy language*, which defines the solver interactions during the search process.

META–S is an implementation of a theoretical framework proposed in [96] by Franc et al., which allows to tackle constrained problems, through 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.

Arbab and Monfory propose in [88] 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 information to the *worker* processes. 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 [108].

A component-based constraint solver in parallel is proposed in [103] by Zoeteweij and Arbab. 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 cannot 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 idle, and there are no subproblems available, another solver produces new sub-problems when its time-out elapses.

Munera et al. present in [90] 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. All teams are distributed and thus have limited inter–node communication. This tool provides diversification through communication between teams, extending the search to different regions of the search space. Intensification is ensured through communication between explorers, and it is achieved swarming to the most promising neighborhood found by explorers. 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.

A similar approach is presented by Guo et al. in [99], exploring 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. Results lead to an original intensification strategy which outperforms the best parallel SAT solver *ManySAT*, and solves some open SAT instances.

Hamadi et al. propose in [97] 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. The frequency of exchanges (conflict-clauses) influences considerably 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 backtrack search (the problem of finding the right time to to restart the search after some fails), Cire et al. have developed in [98] a simple technique for parallelizing restarted search deterministically. They demonstrate experimentally that they can achieve near–linear speed–ups in practice, when the number of processors is constant and the number of restarts grows to infinity. The proposed technique is 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.

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 [109]. 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 [110] is presented a study of methods to tune these algorithms.

In [111] 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 [112] 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 [113]. 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 [114].

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 [115]. In [116] 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 [117] .

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 [118].

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 [119] 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 [120], 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) [121].

Drozdik et al. present in [122] 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 [123] 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] 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.

[6] 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.

[7] 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.

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

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

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

[11] 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.

[12] 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.

[13] 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.

[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] Johann Dréo, Patrick Siarry, Alain Pétrowski, and Eric Taillard. Introduction. In *Metaheuristics for Hard Optimization*. Springer, 2006.

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

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

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

[28] Stefan Voss, Silvano Martello, Ibrahim H. Osman, and Catherine Roucairol, editors. *Meta-heuristics: Advances and trends in local search paradigms for optimization*. Springer Science+Business Media, LLC, 2012.

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

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

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

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

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

[52] Weifeng Gao, Sanyang Liu, and Lingling Huang. A global best artificial bee colony algorithm for global optimization. *Journal of Computational and Applied Mathematics*, 236(11):2741–2753, 2012.

[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] 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.

[57] 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.

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

[59] 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.

[60] Narendra Jussien and Olivier Lhomme. Local Search with Constant Propagation and Conflict-Based Heuristics. *Artificial Intelligence*, 139(1):21–45, 2002.

[61] Gilles Pesant and Michel Gendreau. A View of Local Search in Constraint Programming. In *Second International Conference on Principles an Practice of Constraint Programming*, number 1118, pages 353–366. Springer, 1996.

[62] Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. *Computer*, 1520(Springer):417–431, 1998.

[63] Daniel Fontaine, Laurent Michel, and Pascal Van Hentenryck. Constraint-Based Lagrangian Relaxation. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, pages 324–339. Springer, 2014.

[64] John N. Hooker. Operations Research Methods in Constraint Programming. In *Handbook of Constraint Programming*, chapter 15. 2006.

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

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

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

[68] 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.

[69] 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.

[70] 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.

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

[72] 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.

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

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

[75] 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.

[76] 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.

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

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

[79] 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.

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

[81] 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.

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

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

[84] 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).*

[85] 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.

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

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

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

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

[90] 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.

[91] 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.

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

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

[94] 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.

[95] 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.

[96] 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.

[97] 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.

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

[99] 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.

[100] 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.

[101] 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.

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

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

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

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

[106] 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.

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

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

[109] 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.

[110] 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.

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

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

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

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

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

[116] 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.

[117] 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.

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

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

[120] 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.

[121] 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.

[122] 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.

[123] 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.