POSL: A Parallel-Oriented Solver Language

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
sous le sceau de l'Université Bretagne Loire

Alejandro Reyes Amaro

École doctorale : Sciences et technologies de l'information, et mathématiques
Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'Informatique de Nantes-Atlantique (LINA)

Directeur de thèse : M. Eric Monfroy, Professeur, Université de Nantes
Co-encadrant : Florian Richoux, Maître de Conferences, Université de Nantes

UNIVERSITÉ DE NANTES

Submitted: dd/mm/2016

Jury:

Prof. Arnaud LALLOUET
Huawei Technologies Ltd., France

Prof. Frédéric LARDEUX
Univeristé d'Angers, France

Prof. Christophe LECOUTRE
Université d'Artois, France

Prof. Salvador ABREU
Universidade de Évora, Portugal

POSL: A Parallel-Oriented Solver Language

Short abstract:

The multi-core technology and massive parallel architectures are nowadays more accessible for a broad public through hardware like the Xeon Phi or GPU cards. This architecture strategy has been commonly adopted by processor manufacturers to stick with Moore's law. However, this new architecture implies new ways of designing and implementing algorithms to exploit their full potential. This is in particular true for constraint-based solvers dealing with combinatorial optimization problems.

Furthermore, the developing time needed to code parallel solvers is often underestimated. In fact, conceiving efficient algorithms to solve certain problems takes a considerable amount of time. In this thesis we present POSL, a Parallel-Oriented Solver Language for building solvers based on meta-heuristic, in order to solve Constraint Satisfaction Problems (CSP) in parallel. The main goal of this thesis is to obtain a system with which solvers can be easily built, reducing therefore their development effort, by proposing a mechanism of code reusing between solvers. It provides a mechanism to implement solver-independent communication strategies. We also present a detailed analysis of the results obtained when solving some CSPs. The goal is not to outperform the state of the art in terms of efficiency, but showing that it is possible to rapidly prototyping with POSL in order to experiment different communication strategies.

Keywords: Constraint satisfaction, meta-heuristics, language.

Contents

Part I

POSL: Parallel Oriented Solver Language

1

A Parallel-Oriented Language for Modeling Meta-Heuristic-Based Solvers and communication strategies

In this chapter POSL *is introduced as the main contribution of this thesis, and a new way to solve CSPs (Section 1.1). Its characteristics and advantages are summarized, and a general methodology for building parallel solvers using* POSL *is described. Then a detailed description of each of the single steps is presented (Sections 1.2, 1.3, 1.4, 1.5, 1.6 and 1.7).*

Contents

1.1 Introduction

Meta-heuristic methods, despite showing very good results solving Constraint Satisfaction Problems, they are frequently not enough for solve them, when they are applied to problem instances with extremely large search spaces. Most of these methods are sensible to their large number of parameters. For that reason, a first direction of this thesis was tackling the one of the weakest points of meta-heuristic methods: theirs parameters. In Appendix **??** a performed study applying ParamILS to *Adaptive Search* in order to find a general parameter settings was presented. This experiment did not produce encouraging results. That is why it was decided to abandon the idea as the main direction of the thesis, but not as future work.

From the beginning of the current investigation, the target problems were big and complex instances of CSPs. For that reason, even if the current version of the framework does not provide auto tuning mechanisms, this thesis focuses on the implementation of a mechanism to easily build solvers based on local search meta-heuristic, providing an easy way of reusing algorithm's components commons to different methods.

With the development of parallelism, opening new ways to tackle constrained problems, the accessibility to this technology to a broad public has also increased. It is available through multi-core personal computers, Xeon Phi cards and GPU video cards. For that reason it was decided to focus the thesis completely on the parallel approach. In Appendix **??** it was presented a study in which the problem-subdivision approach was applied to the resolution of *K-Medoids Problem*. The main goal of this work was generalizing the proposed ideas to similar problems. It was only a theoretical study, performed in parallel with what would latter be the main scientific contribution of this thesis.

Many results from the literature indicate that the combination of meta-heuristic methods with parallelism provides very good results for large scale CSPs. This investigation focuses in the implementation of the multi-walk parallel approach. Most of the methods found in the state of the art of this field are based on applying clever techniques to accelerate the solution process of specific problems. The present work does not apply partitioning techniques neither for the search space nor for the target problem. This make the proposed framework applicable in a general and more easy way for a broad range of problems.

Another weak point of the development process that is frequently undervalued is the codding time, which is always long when codding parallel programs. This was the main motivation to start searching techniques for implementing parallel solution strategies with or without communication in a fast and easy way. The main goal was creating a tool providing 1- fast and simple mechanisms to connect solvers, ables to exchange information; 2- and a way to create numerous and different parallel strategies, where different communicating and not communicating solvers can be combined, exploiting to the maximum computation resources.

1.1.1 Precedents

During the development process, some inspired ideas were taken into account. HYPERION[2] [62] is a java framework for meta– and hyper–heuristics built providing generic templates for a variety of local search and evolutionary computation algorithms, allowing quick prototyping with the possibility of reusing source code. A similar idea was proposed by Fukunaga [7], introducing an evolutionary approach that uses a simple composition operator to automatically discover new local search heuristics for SAT and to visualize them as combinations of blocks. The goal of this thesis is to create a tool offering the same advantages, but providing also a mechanism to define communication protocols between solvers. It must also provide a way to create an abstract solver by combining simple functions that we call modules.

In [8] is presented a framework to facilitate the development of search procedures by using *combinators* to design features commonly found in search procedures as standard bricks and joining them. This approach can speed up the development and experimentation of search procedures when developing a specific solver based on local search. Martin et al. [9] propose an approach of using cooperating meta–heuristic based local search processes, using an asynchronous message passing protocol. The cooperation is based on the general strategies of pattern matching and reinforcement learning. The tool developed for this thesis, uses the combination of both ideas, where search process features can be combined and reused, and it is also possible to design communication strategies between solvers.

1.1.2 POSL

In this chapter is presented POSL, the main contribution of this thesis, as well as the different steps to build communicating parallel solvers with. It is proposed as a new way to implement *solution algorithms* to solve Constraint Satisfaction Problems, through local-search meta-heuristics using the multi-walk parallel approach. It is based on improving step by step an initial configuration, driven by a *cost function* provided by the user through the model. The implementation must follow the following stages.

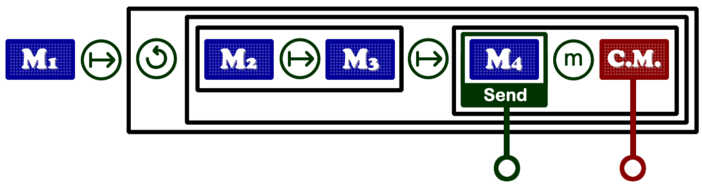
1. The conceived *solution algorithm* to solve the target problem is decomposed it into small modules of computation, which are implemented as separated *functions*. We name them computation modules (see Figure 1.1a, blue shapes). At this point it is crucial to find a good decomposition of its *solution algorithm*, because it will have a significant impact in its future re-usage.

2. Deciding which information is interesting to *receive* from other solvers. This information is encapsulated into another kind of component called communication module, allowing data transmission between solvers (see Figure 1.1a, red shapes).

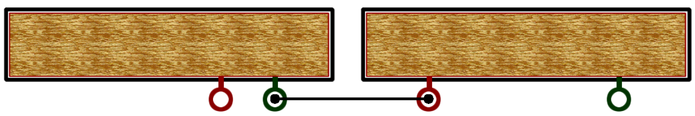
3. A third stage is to ensemble the modules through POSL's inner language to create independent solvers.

4. The parallel-oriented language based on operators provided by POSL (see Figure 1.1b, green shapes) allows the information exchange, and executing modules in parallel. In this stage the information that is interesting to be shared with other solvers is sent using operators. After that we can connect them using *communication operators*. This final entity is called a *solver set* (see Figure 1.1c).

(a) Creating POSL's modules



(b) Assembling modules using POSL's operators



(c) Connecting POSL solvers to create communication strategies

Figure 1.1: Solver construction process using POSL

In the following sections all these steps are explained in details, but first, I explain how to model the target benchmark using POSL.

1.2 Modeling the target benchmark

Target problems are modeled using the low-level framework provided by POSL (written in C++ programming language) They have to be coded respecting an object-oriented hierarchy designed for the optimum performance of the language. The most important functionalities that the proposed model have to provide are the following:

Cost function. This function must compute the *cost* of a given configuration. It must return an integer value taking into account the problem constraints. Given a configuration s, the *cost function*, as a mandatory rule, must return 0 if and only if s is a solution of the problem, i.e., s fulfills all the problem constraints. Otherwise, it must return an integer describing "how long" is the given configuration from a solution. An example of *cost function* is the one returning the number of violated constraints. However, the more expressive the cost function is, the better the performance of POSL is, leading to the solution.

Let us take the example of the *4-Queens Problem*. This problem is about placing 4 queens on a 4×4 chess board so that none of them can hit any other in one move. A configuration for this benchmark is a vector of 4 integer indicating the row where a queens is placed on each column. So, the configuration $s_a = (1, 3, 1, 2)$ corresponds to the example in Figure 1.2a.

Now, let us suppose two different *cost functions*:

1. $f_1(s) = c$ if and only if c is the maximum number of queens hitting another.
2. $f_2(s) = c$ if and only if c is the sum of the number of queens that each queen hits.

Tacking these two functions into account, it is easy to see that $f_1(s_a) = 3$ and $f_2(s_a) = 4$. If we take the example in Figure 1.2b, the corresponding configuration is $s_b = (0, 1, 0, 2)$ with $f_1(s_b) = 3$ and $f_2(s_b) = 6$. In this case, according to the *cost function* f_1 both configurations have the same opportunity of being selected, because they have the same cost. However, applying the *cost function* f_2, the best configuration is s_b in which a solution can be obtained just moving the queen *b3* to *a3*.

In that sense, f_2 is *more expressive* than f_1.

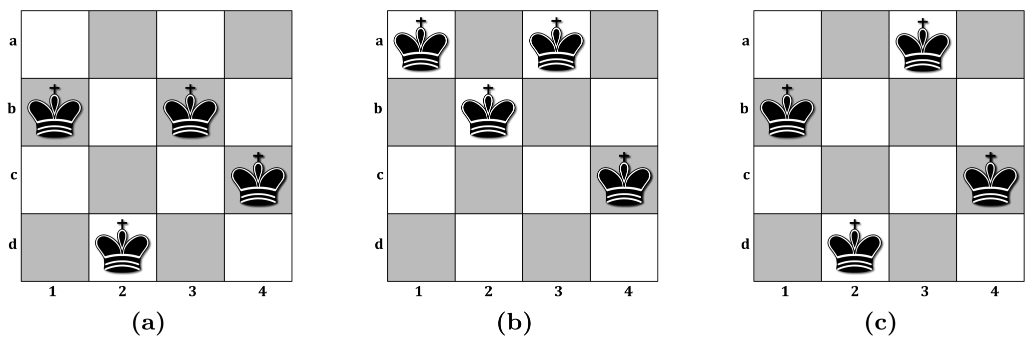


Figure 1.2: 4-Queens examples

Relative cost function. This function must compute the *cost* of a given configuration with respect to another, with the help of some stored information.

Coming back to the previews example, let us suppose that the current configuration is $s_a = (1,3,1,2)$ corresponding to the Figure 1.2a. Taking the *cost function* f_2, the cost of this configurations is $f_2(s_a) = 4$. If we want to compute the cost of $s_c = (1,3,0,2)$ (Figure 1.2c), knowing that the only change with respect to the current configuration is the queen in the column 3, we can use the following *relative cost function*:

$$\begin{aligned} rf(s_c) &= c - 2 \cdot q + a \\ &= 4 - 2 \cdot 2 + 0 \\ &= 0 \end{aligned}$$

where c is the current cost, q is the number of queens that the queen in column 3 hits (an information that can be stored), and a the number of queens that the queen in the column 3 hits in the new position (*a3*).

Showing result function. This function represents the way a benchmark shows a configuration, in order to provide more information about the structure.

For example, a configuration of the instance 3–3–2 of the *Social Golfers Problem* (see bellow for more details about this benchmark) can be written as follows:

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

This text is, nevertheless, very difficult to be read if the instance is larger. Therefore, it is recommended that the user implements this class in order to give more details and to make it easier to interpret the configuration. For example, for the same instance of the problem, a solution could be presented as follows:

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

Once we have modeled the target benchmark, it can be solved using POSL. In the following sections we describe how to use this parallel-oriented language to solve Constraint Satisfaction Problems.

1.3 First stage: creating POSL's modules

There exist two types of basic modules in POSL: *computation module* and *communication module*. A computation module is basically a function and a communication module is also a function, but in contrast, it can receive information from two different sources: through input parameters or from outside, i.e., by communicating with a module from another solver.

1.3.1 Computation module

A computation module is the most basic and abstract way to define a piece of computation. It is a function which receives an instance of a POSL data type as input, then executes an internal algorithm, and returns an instance of a POSL data type as output. The input and output types will characterize the computation module signature. It can be dynamically replaced by (or combined with) other computation modules, since they can be transmitted to other solvers working in parallel. They are joined through operators defined in Section 1.4.

Definition 1 ***(Computation Module)*** *A computation module Cm is a mapping defined by:*

$$Cm : \mathcal{I} \rightarrow \mathcal{O} \tag{1.1}$$

where I and O, for instance, can be independently a set of configurations, a set of sets of configurations, a set of values of some data type, etc.

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

$$Cm_{neighborhood} : I_1 \times I_2 \times \cdots \times I_n \rightarrow 2^{I_1 \times I_2 \times \cdots \times I_n}$$

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

Figure 1.3 shows an example of computation module: which receives a configuration S and then computes the set $\mathcal{V}$ of its neighbor configurations $\{S^1, S^2, \ldots, S^m\}$.

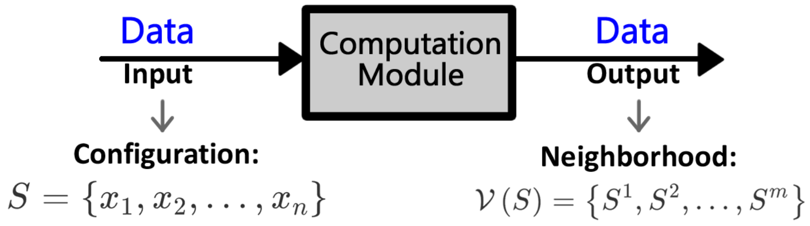


Figure 1.3: An example of a computation module computing a neighborhood

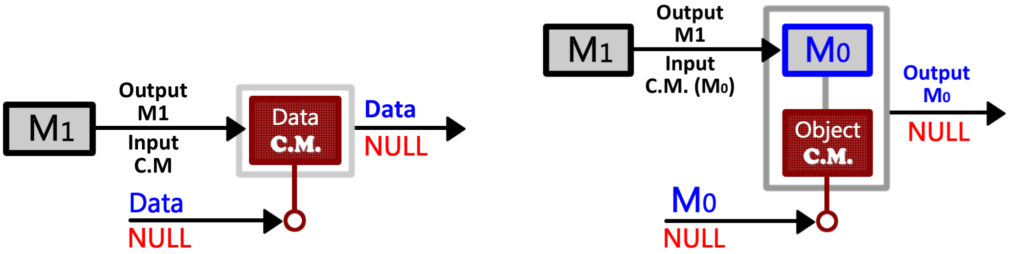
1.3.1.1 Creating new computation modules

Each computation module is written in C++. POSL provides a hierarchy of data types to work with and some abstract classes to inherit from, depending on the type of computation module the user wants to create. These abstract classes represent *abstract* computation modules and define a type of action to be executed. In the following we present the most important ones:

- **First Configuration Generation** → Represents computation modules returning a configuration s, usually used for generating the starting configuration on local search meta-heuristics.
- **Neighborhood Function** → Represent computation modules receiving a configuration s as input and returning its neighborhood $\mathcal{V}(s)$ as output. These output configurations are efficiently stored in term of space.
- **Selection Function** → Represents computation modules receiving a neighborhood as input and selecting a configuration s' from it as output. This function returns the pair (s, s') containing both current and selected configuration.
- **Decision Function** → Represents computation modules receiving a couple af configurations encapsulated into a pair (s, s'), and returning the configuration to be the current one for the next iteration.
- **Processing Configuration Function** → Represents computation modules receiving a configuration and returning another configuration as result of some arrangement, like for example, a reset.

1.3.2 Communication modules

A communication module is the component managing the information reception in the communication between solvers (I talk about information transmission in Section 1.4). They



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

Figure 1.4: Communication module

can interact with computation modules through operators (see Figure 1.4).

A communication module can receive two types of information from an external solver: data or computation modules. It is important to notice that by sending/receiving computation modules, I mean sending/receiving the required information to identify and being able to instantiate the computation module. For instance, an integer identifier.

In order to distinguish from the two types of communication modules, I will call *data communication module* the communication module responsible for the data reception (Figure 1.4a), and *object communication module* the one responsible for the reception and instantiation of computation modules (Figure 1.4b).

Definition 2 ***(Data Communication Module)*** *A* Data Communication Module *Ch is a module that produces a mapping defined as follows:*

$$Ch : I \times \{D \cup \{NULL\}\} \rightarrow D \cup \{NULL\} \tag{1.2}$$

No matter what the input I is, it returns the information D coming from an external solver.

Definition 3 ***(Object Communication Module)*** *If we denote by $\mathbb{M}$ the space of all the computation modules defined by Definition 1.1, then an* object communication module *Ch is a module that produces and executes a computation module coming from an external solver as follows:*

$$Ch : I \times \{\mathbb{M} \cup \{NULL\}\} \rightarrow O \cup \{NULL\} \tag{1.3}$$

It returns the output O of the execution of the computation module coming from an external solver, using I as the input.

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

Due to the fact that communication modules receive information coming from outside without having control on them, it is necessary to define the *NULL* information, in order to denote the absence of information. If a Data Communication Module receives information, it is returned automatically. If a Object Communication Module receives a computation module, it is instantiated and executed with the communication module's input and its result is returned. In both cases, if no available information exists (no communications performed), the communication module returns the *NULL* object.

1.4 Second stage: assembling POSL's modules

Modules mentioned above are grouped according to its signature. An *abstract module* is a module that represents all modules with the same signature. For example, the module showed in Figure 1.3 is a computation module based on an abstract module that receives a configuration and returns a neighborhood.

In this stage an *abstract solver* is coded using POSL. It takes abstract modules as *parameters* and combines them through operators. Through the abstract solver, we can also decide which information to send to other solvers.

The abstract solver is the solver's backbone. It joins the computation modules and the communication modules coherently. It is independent from the computation modules and communication modules used in the solver. It means that modules can be changed or modified during the execution, respecting the algorithm structure. Each time we combine some of them using POSL's operators, we are creating a *compound module*. Here we formally define the concept of *module* and *compound module*.

Definition 4 *Denoted by the letter $\mathcal{M}$, a* **module** *is:*

1. *a computation module; or*
2. *a communication module; or*
3. $[OP\ \mathcal{M}]$*, which is the composition of a module $\mathcal{M}$ to be executed sequentially, returning an output depending on the nature of the unary operator* OP*; or*
4. $[\mathcal{M}_1\ OP\ \mathcal{M}_2]$*, which is the composition of two modules $\mathcal{M}_1$ and $\mathcal{M}_2$ to be executed sequentially, returning an output depending on the nature of the binary operator* OP*; or*
5. $[\![\mathcal{M}_1\ OP\ \mathcal{M}_2]\!]_p$*, which is the composition of two modules $\mathcal{M}_1$ and $\mathcal{M}_2$ to be executed, returning an output depending on the nature of the binary operator* OP*. These two modules will be executed in parallel if and only if* OP *supports parallelism, or it throws an exception otherwise.*

I denote by $\mathbb{M}$ the space of the modules, and I call compound modules to the composition of modules described in 3., 4. and/or 5..

For a better understanding of Definition 4, Figure 1.5 shows graphically the structure of a compound module.

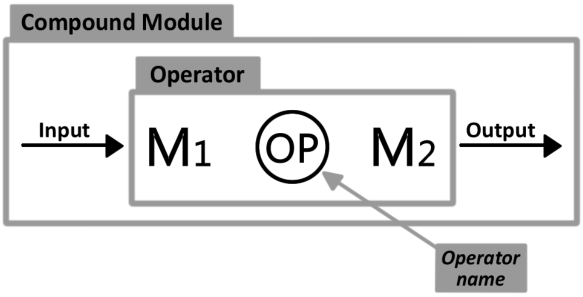


Figure 1.5: A compound module made of two modules M_1 and M_2

As mentioned before, the abstract solver is independent from the computation modules and communication modules used in the solver. It means that one abstract solver can be used to construct many different solvers, by implementing it using different modules. This is the reason why the abstract solver is defined only using abstract modules. Formally, we define an abstract solver as follows:

Definition 5 ***(Abstract Solver)*** *An* Abstract Solver *AS is a triple $(\mathbf{M}, \mathcal{L}^m, \mathcal{L}^c)$, where: $\mathbf{M}$ is a compound module (also called* root compound module*), $\mathcal{L}^m$ a list of abstract computation modules appearing in $\mathcal{M}$, and $\mathcal{L}^c$ a list of communication modules appearing in $\mathcal{M}$.*

Compound modules, and in particular the *root* compound module, can be defined also as a context-free grammar as follows:

Definition 6 *A* **compound module's grammar** *is the set $G_{POSL} = (\mathbf{V}, \Sigma, \mathbf{S}, \mathbf{R})$, where:*

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

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

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

4. and $\mathbf{R}$ =

$$
\begin{aligned}
CM &\longmapsto \alpha \mid \beta \mid (\!|CM|\!)^d \mid (\!|CM|\!)^m \mid [OP] \mid [\![OP]\!]_p \\
OP &\longmapsto CM \bigcirc\!\!\!\!\mapsto CM \mid CM \,?\, CM \mid CM \,\rho\, CM \mid CM \vee CM \mid CM \wedge CM \mid \\
&\qquad CM \,M\, CM \mid CM \,m\, CM \mid CM \downarrow CM \mid CM \cup CM \mid CM \cap CM \mid \\
&\qquad \circlearrowleft\ be\ CM
\end{aligned}
$$

is a set of rules

In the following I explain some of the concepts in Definition 6:

- The variables CM and OP correspond to a compound module and an *operator*, respectively.
- The terminals α and β represent a computation module and a communication module, respectively.
- The terminal *be* is a boolean expression.
- The terminals [ ], $[\![\ ]\!]_p$ are symbols for grouping and defining the way the involved compound modules are executed. Depending on the nature of the operator, this can be either sequentially or in parallel:
  1. [OP]: The involved operator will always executed sequentially.
  2. $[\![\text{OP}]\!]_p$: The involved operator will be executed in parallel if and only if OP supports parallelism. Otherwise, an exception is thrown.
- The terminals $(\!|.|\!)^d$, $(\!|.|\!)^m$ are operators to send information to other solvers (explained bellow).
- All other terminals are POSL operators that are detailed later.

In the following we define POSL operators. For grouping modules, like in Definition 4(4.) and 4(5.), we will use $|OP|$ as generic grouper. In order to help the reader to easily understand how to use operators, I use an example of a solver that I build step by step, while presenting the definitions.

> POSL creates solvers based on local search meta-heuristics algorithms. These algorithms have a common structure: 1. They start by initializing some data structures (e.g., a *tabu list* for *Tabu Search*, a *temperature* for *Simulated Annealing*, etc.). 2. An initial configuration s is generated. 3. A new configuration s' is selected from the neighborhood $\mathcal{V}(s)$. 4. If s' is a solution for the problem P, then the process stops, and s' is returned. If not, the data structures are updated, and s' is accepted or not for the next iteration, depending on a certain criterion.
>
> Abstract computation modules composing local search meta–heuristics are:

$\boxed{Abstract\ computation\ module - 1}$	I: Generating a configuration s
$\boxed{Abstract\ computation\ module - 2}$	V: Defining the neighborhood $\mathcal{V}(s)$
$\boxed{Abstract\ computation\ module - 3}$	S: Selecting $s' \in \mathcal{V}(s)$
$\boxed{Abstract\ computation\ module - 4}$	A: Evaluating an acceptance criterion for s'

The list of modules to be used in the examples have been presented. Now I present POSL operators.

***

Definition 7 $\bigcirc\!\!\!\!\mapsto$ **Sequential Execution Operator.** *Let*

1. $\mathcal{M}_1 : \mathcal{I}_1 \rightarrow \mathcal{O}_1$ *and*
2. $\mathcal{M}_2 : \mathcal{I}_2 \rightarrow \mathcal{O}_2$,

be modules, where $\mathcal{O}_1 \subseteq \mathcal{O}_2$*. Then the operation* $\left|\mathcal{M}_1 \bigcirc\!\!\!\!\mapsto \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{seq}$ *as the result of executing* $\mathcal{M}_1$ *followed by executing* $\mathcal{M}_2$*:*

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

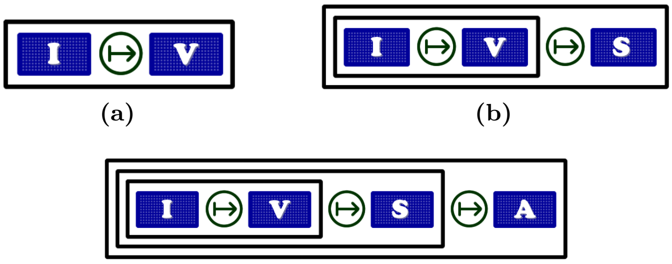
This is an example of an operator that does not support the execution of its involved compound modules in parallel, because the input of the second compound module is the output of the first one.

Coming back to the example, I can use defined abstract computation modules to create a compound module that performs only one iteration of a local search, using the **Sequential Execution** operator. I create a compound module to execute sequentially I and V (see Figure 1.6a), then I create another compound module to execute sequentially the compound module already created and S (see Figure 1.6b), and finally this compound module and the computation module A are executed sequentially (see Figure 1.6c). The compound module presented in Figure 1.6c can be coded as follows:

$$\left[\left[\left[I \bigcirc\!\!\!\!\mapsto V\right] \bigcirc\!\!\!\!\mapsto S\right] \bigcirc\!\!\!\!\mapsto A\right]$$

In the figure, each rectangle is a compound module.

***



(a) (b)

(c)

Figure 1.6: Using **sequential execution** operator

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

Definition 8 $?$ **Conditional Execution Operator** *Let*

1. $\mathcal{M}_1 : \mathcal{I} \to \mathcal{O}_1$ *and*

2. $\mathcal{M}_2 : \mathcal{I} \to \mathcal{O}_2$,

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

$$\mathcal{M}_{cond} : \mathcal{I} \to \mathcal{O}_1 \cup \mathcal{O}_2$$

This operator can be used in the example if I want to execute two different *selection* computation modules (S_1 and S_2) depending on certain criterion (see Figure 1.7):

$$\left[\left[\left[I \mapsto V\right] \mapsto \left[S_1 ? S_2\right]\right] \mapsto A\right]$$

In examples I remove the clause $< cond >$ for simplification.

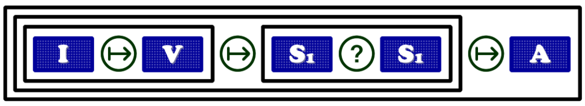


Figure 1.7: Using **conditional execution** operator

***

We can execute modules sequentially creating also cycles.

Definition 9 $\circlearrowleft$ **Cyclic Execution Operator** *Let $\mathcal{M} : \mathcal{I} \to \mathcal{O}$ be a module, where $\mathcal{O} \subseteq \mathcal{I}$. Then, the operation $\left| \circlearrowleft_{<cond>} \mathcal{M} \right|$ defines the compound module $\mathcal{M}_{cyc}$ repeating sequentially the execution of $\mathcal{M}$ while $< cond >$ remains* **true***:*

$$\mathcal{M}_{cyc} : \mathcal{I} \to \mathcal{O}$$

Using this operator I can model a local search algorithm, by executing the *abstract* computation module I and then the other computation modules (V, S and A) cyclically, until finding a solution (i.e, a configuration with cost equals to zero) (see Figure 1.8):

$$\left[I \mapsto \left[\circlearrowleft \left[\left[V \mapsto S\right] \mapsto A\right]\right]\right]$$

In the examples, I remove the clause $< cond >$ for simplification.

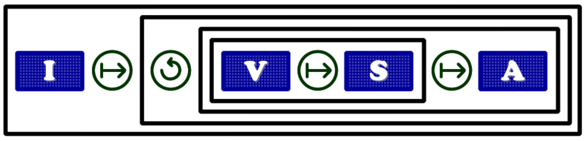


Figure 1.8: Using **cyclic execution** operator

***

Definition 10 ρ **Random Choice Operator** *Let*

1. $\mathcal{M}_1 : \mathcal{I} \to \mathcal{O}_1$ *and*
2. $\mathcal{M}_2 : \mathcal{I} \to \mathcal{O}_2$*,*

be modules. and a real value $\rho \in (0,1)$. Then the operation $\left| M_1 \rho \mathcal{M}_2 \right|$ defines the compound module $\mathcal{M}_{rho}$ executing $\mathcal{M}_1$ with probability ρ, or executing $\mathcal{M}_2$ with probability $(1-\rho)$:

$$\mathcal{M}_{rho} : \mathcal{I} \to \mathcal{O}_1 \cup \mathcal{O}_2$$

In the example I can create a compound module to execute two *abstract* computation modules A_1 and A_2 following certain probability ρ using the **random choice** operator as follows (see Figure 1.9):

$$\left[I \mapsto \left[\circlearrowleft \left[\left[V \mapsto S\right] \mapsto \left[A_1 \rho A_2\right]\right]\right]\right]$$

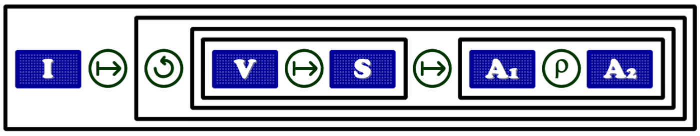


Figure 1.9: Using **random choice** operator

***

The following operator is very useful if the user needs to use a communication module inside an abstract solver. As explained before, if a communication module does not receive any information from another solver, it returns *NULL*. This may cause the undesired termination of the solver if this case is not correctly handled. Next, I introduce the **Not** *NULL* **Execution Operator** and illustrate how to use it in practice with an example.

Definition 11 $\vee$ **Not** *NULL* **Execution Operator** *Let*

1. $\mathcal{M}_1 : \mathcal{I} \rightarrow \mathcal{O}_1$ *and*
2. $\mathcal{M}_2 : \mathcal{I} \rightarrow \mathcal{O}_2$,

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

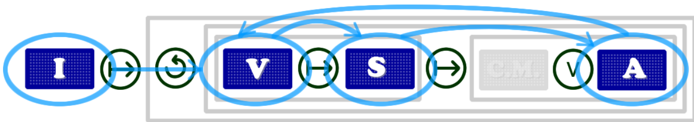
$$\mathcal{M}_{non} : \mathcal{I} \rightarrow \mathcal{O}_1 \cup \mathcal{O}_2$$

Let us make consider a slightly more complex example: When applying the acceptance criterion, suppose that we want to receive a configuration from other solver to combine the computation module A with a communication module:

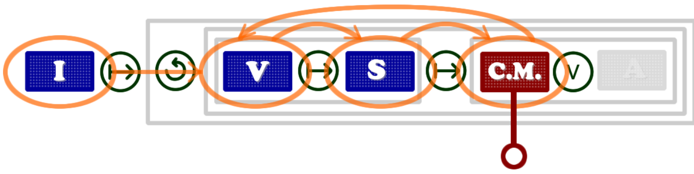
$\boxed{Communication\ module - 1 :}$ $C.M.$: Receiving a configuration.

Figure 1.10 shows how to combine a communication module with the computation module A through the operator $\vee$. Here, the computation module A will be executed as long as the communication module remains *NULL*, i.e., there is no information coming from outside. This behavior is represented in Figure 1.10a by the blue lines. If some data has been received through the communication module, the later is executed instead of the module A, represented in Figure 1.10b by orange lines. The code can be written as follows:

$$\left[I \mapsto \left[\circlearrowleft \left[\left[V \mapsto S\right] \mapsto \left[C.M. \vee A\right]\right]\right]\right]$$



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



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

Figure 1.10: Two different behaviors within the same solver

∗∗∗

The operator that I have just defined is a *short-circuit* operator. It means that if the first argument (module) does not return *NULL*, the second will not be executed. POSL provides another operator with the same functionality but not *short-circuit*. This operator is necessary if the user wants a side effect by always executing the second module also.

Definition 12 $\bigcirc\!\!\!\!\wedge$ *BOTH* **Execution Operator** *Let*

1. $\mathcal{M}_1 : \mathcal{I} \rightarrow \mathcal{O}_1$ *and*

2. $\mathcal{M}_2 : \mathcal{I} \rightarrow \mathcal{O}_2$*,*

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

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

∗∗∗

In the following I introduce the concepts of *cooperative parallelism* and *competitive parallelism*. We say that cooperative parallelism exists when two or more processes are running separately, and the general result will be some combination of the results of at least some involved processes (e.g. Definitions 13 and 14). On the other hand, competitive parallelism arise when the general result comes from an unique process, usialy the one finishing first (e.g. Definition 15).

Definition 13 (m) **Minimum Operator** *Let*

1. $\mathcal{M}_1 : \mathcal{I} \to \mathcal{O}_1$ *and*
2. $\mathcal{M}_2 : \mathcal{I} \to \mathcal{O}_2$,

be modules. Let also o_1 *and* o_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, respectively. Assume that there exists a total order in* $O_1 \cup O_2$ *where the object* NULL *is the greatest value. Then the operation* $\left|\mathcal{M}_1 (m) \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{min}$ *that executes* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns* $\min\{o_1, o_2\}$*:*

$$\mathcal{M}_{min} : \mathcal{I} \to \mathcal{O}_1 \cup \mathcal{O}_2$$

***

Similarly we define the **Maximum** operator:

Definition 14 (M) **Maximum Operator** *Let*

1. $\mathcal{M}_1 : \mathcal{I} \to \mathcal{O}_1$ *and*
2. $\mathcal{M}_2 : \mathcal{I} \to \mathcal{O}_2$,

be modules. Let also o_1 *and* o_2 *be the outputs of* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, respectively. Assume that there exists a total order in* $O_1 \cup O_2$ *where the object* NULL *is the smallest value. Then the operation* $\left|\mathcal{M}_1 (M) \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{max}$ *that executes* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns* $\max\{o_1, o_2\}$*:*

$$\mathcal{M}_{max} : \mathcal{I} \to \mathcal{O}_1 \cup \mathcal{O}_2$$

The **minimum operator** can be applied in the previews example to obtain an interesting behavior: When applying the acceptance criteria, suppose that we want to receive a configuration from another solver, to compare it with ours and select the one with the lowest cost. We can do that by applying the (m) operator to combine the computation module A with a communication module $C.M.$ (see Figure1.11):

$$\left[I \,(\mapsto)\, \left[(\circlearrowleft) \left[\left[V \,(\mapsto)\, S \right] \,(\mapsto)\, \left[\!\left[A \,(m)\, C.M. \right]\!\right]_p \right] \right] \right]$$

Notice that in this example, I can use the grouper $[\![.]\!]_p$ since the **minimum operator** supports parallelism.

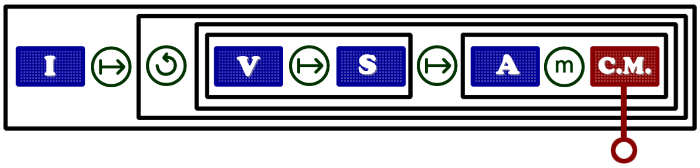


Figure 1.11: Using **minimum** operator

***

Definition 15 $\downarrow$ **Race Operator** *Let*

1. $\mathcal{M}_1 : \mathcal{I} \to \mathcal{O}_1$ *and*
2. $\mathcal{M}_2 : \mathcal{I} \to \mathcal{O}_2$,

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

$$\mathcal{M}_{race} : \mathcal{I} \to \mathcal{O}_1 \cup \mathcal{O}_2$$

Sometimes nighborhood functions are slow depending on the configuration. In that case two neighborhood computation modules can be executed and take into account the output of the module ending first (see Figure1.12):

$$\left[I \mapsto \left[\circlearrowleft \left[\left[\left[\!\left[V_1 \downarrow V_2\right]\!\right]_p \mapsto S\right] \mapsto \left[\!\left[A \; m \; C.M.\right]\!\right]_p\right]\right]\right]$$

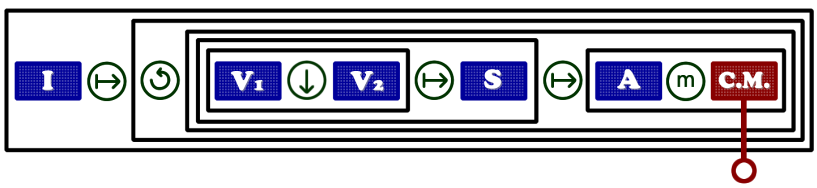


Figure 1.12: Using **race** operator

***

Some POSL's data types are related to sets, like neighborhoods. For that reasson, it is useful to define operators to handle that kind of data. Algthogh at this moment POSL is designed only to create solvers based on local-search meta-heuristic, is was conceived to be able to create population-based solvers as a future direction. In that sens, these operators are useful also.

Definition 16 $\bigcirc\!\!\!\!\cup$ **Union Operator** *Let*

1. $\mathcal{M}_1 : \mathcal{I} \rightarrow \mathcal{O}_1$ *and*

2. $\mathcal{M}_2 : \mathcal{I} \rightarrow \mathcal{O}_2$,

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

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

***

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

Definition 17 $\bigcirc\!\!\!\!\cap$ **Intersection Operator** *Let*

1. $\mathcal{M}_1 : \mathcal{I} \rightarrow \mathcal{O}_1$ *and*

2. $\mathcal{M}_2 : \mathcal{I} \rightarrow \mathcal{O}_2$,

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

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

***

Definition 18 $\bigcirc\!\!\!\!\setminus$ **Subtraction Operator** *Let*

1. $\mathcal{M}_1 : \mathcal{I} \rightarrow \mathcal{O}_1$ *and*

2. $\mathcal{M}_2 : \mathcal{I} \rightarrow \mathcal{O}_2$,

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

$$\mathcal{M}_{\setminus} : \mathcal{I} \to \mathcal{O}_1$$

***

Now, I define the operators which allows to send information to other solvers. Two types of information can be sent: i) the output of the computation module as result of its execution, or ii) the computation module itself. This feature is very useful in terms of sharing behaviors between solvers.

Definition 19 $(\!|.|\!)^d$ **Sending Data Operator** *Let* $\mathcal{M} : \mathcal{I} \to \mathcal{O}$ *be a module. Then the operation* $\left|(\!|\mathcal{M}|\!)^d\right|$ *defines the compound module* $\mathcal{M}_{sendD}$ *that executes the module* $\mathcal{M}$ *and sends its output to a communication module:*

$$\mathcal{M}_{sendD} : \mathcal{I} \to \mathcal{O}$$

***

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

Definition 20 $(\!|.|\!)^m$ **Sending Module Operator** *Let* $\mathcal{M} : \mathcal{I} \to \mathcal{O}$ *be a module. Then the operation* $|(\!|\mathcal{M}|\!)^m|$ *defines the compound module* $\mathcal{M}_{sendM}$ *that executes the module* $\mathcal{M}$*, then returns its output and sends the module itself to a communication module:*

$$\mathcal{M}_{sendM} : \mathcal{I} \to \mathcal{O}$$

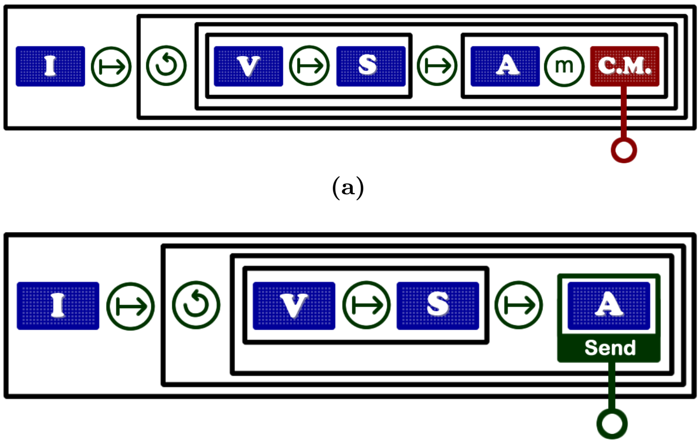
In the following example, I use one of the compound modules already presented in the previews examples, using a communication module to receive a configuration (see Figure 1.13a):

$$\left[I \textcircled{\mapsto} \left[\textcircled{\circlearrowleft} \left[\left[V \textcircled{\mapsto} S\right] \textcircled{\mapsto} [\![A \textcircled{m} C.M.]\!]_p\right]\right]\right]$$

I also build another, as its complement: sending the accepted configuration to outside, using the **sending data operator** (see Figure1.13b):

$$\left[I \textcircled{\mapsto} \left[\textcircled{\circlearrowleft} \left[\left[V \textcircled{\mapsto} S\right] \textcircled{\mapsto} (\!|A|\!)^d\right]\right]\right]$$

In the Section 1.6 I explain how to connect solvers to each other.



(a)

(b)

Figure 1.13: Sender and receiver behaviors

Sending modules through this operator is performed by sending an identifier, in the case of computation modules, or the corresponding POSL code in the case of compound modules. The receptor data communication module create the module, executes it and returns its output. There exists other way to do this task more efficiently, for example, compiling modules in a pre–processing stage and store them in memory, to be executed afterwards by sending only the reference. However, POSL does it in this way, because it was thought to be able to apply learning techniques to the received modules in the future, to adapt them to the experience of the solver during the search.

***

Once all desired abstract modules are linked together with operators, we obtain the root compound module, i.e., the algorithmic part of an abstract solver. To implement a concrete solver from an abstract solver, one must instantiate each abstract module with a concrete one respecting the required signature. From the same abstract solver, one can implement many different concrete solvers simply by instantiating abstract modules with different concrete modules.

An abstract solver is declared as follows: after declaring the **abstract solver**'s name, the first line defines the list of abstract computation modules, the second one the list of abstract communication modules, then the algorithm of the solver is defined as the solver's body (the root compound module **M**), between **begin** and **end**.

An abstract solver can be declared through the simple regular expression:

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

where:

- *name* is the identifier of the abstract solver,
- $\mathcal{L}^m$ is the list of abstract computation modules,
- $\mathcal{L}^c$ is the list of abstract communication modules, and
- $\mathbf{M}$ is the root compound module.

For instance, Algorithm 1 illustrates the abstract solver corresponding to Figure 1.1b.

Algorithm 1: POSL pseudo-code for the abstract solver presented in Figure 1.1b

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

1.5 Third stage: creating POSL solvers

With computation and communication modules composing an abstract solver, one can create solvers by instantiating modules. This is simply done by specifying that a given **solver** must **implements** a given abstract solver, followed by the list of computation then communication modules. These modules must match signatures required by the abstract solver.

In the following example, I describe some concrete computation modules that can be used to implement the abstract solver declared in Algorithm 1:

I_{rand}	Generates and returns a random configuration s
V_{1ch}	Returns the neighborhood $\mathcal{V}(s)$ changing only one element on the input configuration s
S_{best}	Selects the configuration $s' \in \mathcal{V}(s)$ with the lowest global cost, *i.e.*, the one which is likely the closest to a solution, and then returns the pair (s, s').
A_{AI}	Receives a pair of configurations (s, s'), and always returns s'.

I use also the following concrete communication module:

CM_{last}	Returns the last configuration arrived, if at the moment of its execution, there is more than one configuration waiting to be received.

Algorithm 2: An instantiation of the abstract solver presented in Algorithm 1

```
solver solver_01 implements as_01
computation : I_rand, V_1ch, S_best, A_AI
connection: CM_last
```

1.6 Forth stage: connecting solvers

Once a set of solvers is created, the last stage is to connect them to each other. Up to this point, solvers are disconnected, but they are ready to establish the communication. POSL provides tools to the user to easily define cooperative strategies based on communication jacks and outlets. The pool of (concrete) connected solvers to be executed in parallel to solve a problem is called a *solver set*.

Definition 21 Communication Jack *Let $\mathcal{S}$ be a solver and a module $\mathcal{M}$. Then the operation $\mathcal{S} \cdot \mathcal{M}$ opens an outgoing connection from the solver $\mathcal{S}$, sending either a) the output of $\mathcal{M}$, if a sending data operator is applied to $\mathcal{M}$, as presented in Definition 19, or b) $\mathcal{M}$ itself, if a sending module operator is applied to $\mathcal{M}$, as presented in Definition 20.*

Definition 22 Communication Outlet *Let $\mathcal{S}$ be a solver and a communication module $\mathcal{CM}$. Then, the operation $\mathcal{S} \cdot \mathcal{CM}$ opens an ingoing connection to the solver $\mathcal{S}$, receiving either a) the output of some computation module, if $\mathcal{CM}$ is a data communication module, or b) a computation module, if $\mathcal{CM}$ is an object communication module.*

***

The communication is established by following the following rules guideline:

1. Each time a solver sends any kind of information by using a *sending* operator, it creates a *communication jack*.
2. Each time a solver defines a communication module, it creates a *communication outlet*.
3. Solvers can be connected to each other by linking communication jacks to communication outlets.

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

Definition 23 $\boxed{\rightarrow}$ **Connection One-to-One Operator** *Let*

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

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

Then the operation

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

connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ *with the corresponding communication outlet* $\mathcal{Z}_i \cdot \mathcal{CM}_i \in \mathcal{O}$, $\forall\ 0 \leq i \leq N-1$ *(see Figure 1.14a).*

∗∗∗

Definition 24 $\boxed{\rightsquigarrow}$ **Connection One-to-N Operator** *Let*

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

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

Then the operation

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

connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ *with every communication outlet* $\mathcal{Z}_j \cdot \mathcal{CM}_j \in \mathcal{O}$, $\forall\ 0 \leq i \leq N-1$ *and* $0 \leq j \leq M-1$ *(see Figure 1.14b).*

∗∗∗

Definition 25 $\boxed{\leftrightarrow}$ **Connection Ring Operator** *Let*

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

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

Then the operation

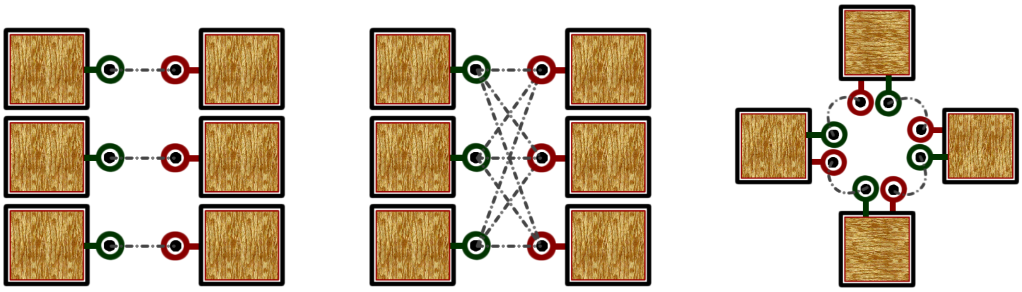
$$\mathcal{J} \boxed{\leftrightarrow} \mathcal{O}$$

connects each communication jack $\mathcal{S}_i \cdot \mathcal{M}_i \in \mathcal{J}$ *with the corresponding communication outlet* $\mathcal{Z}_{(i+1)\%N} \cdot \mathcal{CM}_{(i+1)\%N} \in \mathcal{O}$, $\forall 0 \leq i \leq N-1$ *(see Figure 1.14c).*

∗∗∗

POSL also allows to declare non-communicating solvers to be executed in parallel, declaring only the list of solver names:

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



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

Figure 1.14: Graphic representation of communication operators

These operators can be combined beteewn themself to contruct any kind of communication strategy. Figure 1.15 shows a simple example combining solvers doubly connected and non connected solvers. Assuming that all solvers $S_i, i \in [1..5]$ have a module $\mathcal{M}$ sent by a send operator, and a communication module $\mathcal{CM}$, the corresponding code is the following:

$$[\mathcal{S}_1 \cdot \mathcal{M}, \mathcal{S}_2 \cdot \mathcal{M}, \mathcal{S}_3 \cdot \mathcal{M}] \ \boxed{\rightsquigarrow} \ [\mathcal{S}_4 \cdot \mathcal{CM}, \mathcal{S}_5 \cdot \mathcal{CM}]$$
$$[\mathcal{S}_4 \cdot \mathcal{M}, \mathcal{S}_5 \cdot \mathcal{M}] \ \boxed{\rightarrow} \ [\mathcal{S}_1 \cdot \mathcal{CM}, \mathcal{S}_3 \cdot \mathcal{CM}]$$
$$[\mathcal{S}_6]$$
$$[\mathcal{S}_7]$$

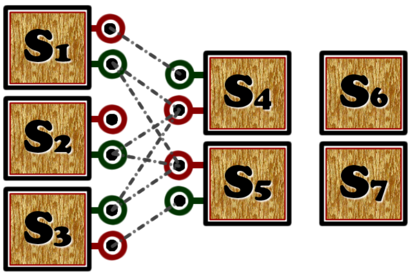


Figure 1.15: Graphic representation of communication operators

* * *

The connection process depends on the applied connection operator. In each case the goal is to assign, to the sending operator ($(\!|.|\!)^d$ or $(\!|.|\!)^m$) inside the abstract solver, the identifier of the solver (or solvers, depending on the connection operator) where the information will be sent. Algorithm 3 presents the connection process.

Algorithm 3: Connection main algorithm

```
input  : 𝒥 list of communication jacks,
         𝒪 list of communication outlets
while no available jacks or outlets remain do
    S_jack ← GetNext(𝒥)
    R_outlet ← GetNext(𝒪)
    S ← GetSolverFromConnector(S_jack)
    R ← GetSolverFromConnector(R_outlet)
    Connect(root(S),S_jack, R)
end
```

In Algorithm 3:

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

Let us suppose that we have declared two solvers S and Z, both implementing the abstract solver in Algorithm 1, so they can be either sender or receiver. The following code connects them using the operator $\mathbf{1\ to\ N}$:

$$[S \cdot A] \ \boxed{\rightsquigarrow} \ [Z \cdot C.M.]$$

If the operator $\mathbf{1\ to\ N}$ is used with only with one solver in each list, the operation is equivalent to applying the operator $\mathbf{1\ to\ 1}$. However, to obtain a communication strategy like the one showed in Figure 1.14b, six solvers (three senders and three receivers) have to be declared to be able to apply the following operation:

$$[S_1 \cdot A, S_2 \cdot A, S_3 \cdot A] \ \boxed{\rightsquigarrow} \ [Z_1 \cdot C.M., Z_2 \cdot C.M., Z_3 \cdot C.M.]$$

POSL provides a mechanism to make this easier, through two *syntactic sugars* explained below.

***

One of the goals of POSL is to provide a way to declare sets of solvers to be executed in parallel easily. For that reason, POSL provides two syntactic sugars in order to create sets of solvers using already declared ones:

1. Using an integer to denote how many times a solver name will appear in the declaration.
2. Using an integer to denote how many times the connection will be repeated in the declaration.

The following example explains clearly these syntactic sugars:

> Suppose that I have created solvers S and Z mentioned in the previews example. As a communication strategy, I want to connect them through the operator **1 to N**, using S as sender and Z as receiver. Then, we need to declare how many solvers I want to connect. Algorithm 4 shows the desired communication strategy. Notice in this example that the connection operation is affected also by the number 2 at the end of the line. In that sense, and supposing that 12 units of computation are available, a solver set working on parallel following the topology described in Figure 1.16 can be obtained.

Algorithm 4: A communication strategy

$[\ \mathsf{S} \cdot A\ (3)\ ]\ \boxed{\rightsquigarrow}\ [\ \mathsf{Z} \cdot C.M.\ (3)\ ]\ 2$;

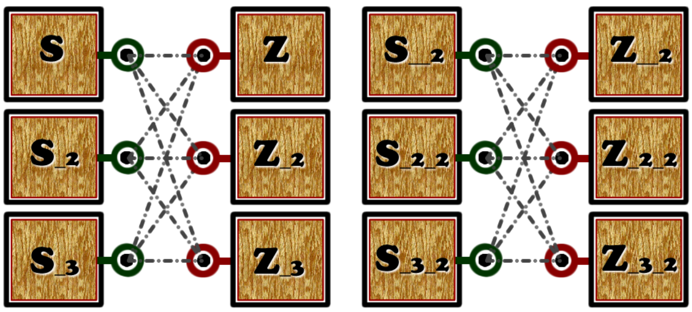


Figure 1.16: An example of connection strategy for 12 units of computation

1.7 Summarize

In this chapter POSL have been formally presented, as a Parallel–Oriented Solver Language to build meta-heuristic-based solver to solve Constraint Satisfaction Problems. This language provides a set of computation modules useful to solve a wide range of constrained problems. It is also possible to create new ones, through the low-level framework in C++ programming language. POSL also provides a set of communication modules, essential features to share information between solvers.

One of the POSL's advantages is the possibility of creating, using an operator-based language, abstract solvers remaining independent from concrete computation and communication modules. It is then possible to create many different solvers builded upon the same abstract solver by only instantiating different modules. It is also possible to create different communication strategies upon the same solver set by using communication operators that POSL provides.

In the next chapter, a detailed study of various communicating and non-communicating strategies is presented using some Constraint Satisfaction Problems as benchmarks.

2

Bibliography

[1] Vangelis Th Paschos, editor. *Applications of combinatorial optimization.* John Wiley & Sons, 2013.

[2] Francisco Barahona, Martin Groetschel, Michael Juenger, and Gerhard Reinelt. An Application of Combinatorial Optimization To Statistical Physics and Circuit Layout Design. *Operations Research*, 36(3):493 – 513, 1988.

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

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

[5] 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.

[6] 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.

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

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

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

[10] 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.

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

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

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

[31] 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.

[32] 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.

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

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

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

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

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

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

[53] 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.

[54] 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.

[55] 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.

[56] 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.

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

[58] 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.

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

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

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

[62] 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.

[63] 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.

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

[65] 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.

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

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

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

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

[70] 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.

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

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

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

[74] 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.

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

[76] 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.

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

[78] 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.

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

[80] 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.

[81] 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.

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

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

[84] 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.

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

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

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

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

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

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

[91] 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.

[92] 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.

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

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

[95] 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.

[96] 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).*

[97] 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.

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

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

[100] 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.

[101] 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.

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

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

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

[105] 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.

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

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

[108] 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.

[109] 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.

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

[111] 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.

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

[113] 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.

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

[115] 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.

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

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

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

[119] 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.

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

[121] 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.

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

[123] 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.

[124] 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.

[125] 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.

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

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

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

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