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 : M. Florian Richoux, Maître de Conferences, Université de Nantes

UNIVERSITÉ DE NANTES

Submitted: dd/mm/2016

Jury:

Président :	**M. Salvador ABREU**, *Professeur étranger*, Université d'Évora
Rapporteurs :	**M. Frédéric LARDEUX**, *Maître de conférences*, Université d'Angers
	M. Christophe LECOUTRE, *Professeur*, Université d'Artois
Examinateur :	**M. Arnaud LALLOUET**, *Chercheur industriel*, Huawei Technologies Ltd.

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, parallel, inter-process communication, 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 for solving Constraint Satisfaction Problems, are frequently not enough for solving instances with extremely large search spaces. Most of these methods are sensible to their large number of parameters. For that reason, a first direction for this thesis was to tackle 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 this 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 [1] is a java framework for building meta– and hyper–heuristics providing generic templates for a variety of local search and evolutionary computation algorithms, allowing quick prototyping with the possibility of reusing source code. This tool illustrate a bit one of the main goals of this thesis. Rapid and fast prototyping of algorithms through high-level languages is more and more imperative due to the increasing demand of algorithms to solve very complex algorithms coming from the development of the technology. Nevertheless, this solution does not take into account parallelism.

Alex S. Fukunaga propose in [2] 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. A similar idea is presented in [3] by Landtsheer et al., a framework to facilitate the development of local search methods by using *combinators* to design features commonly found in these methods as standard bricks and joining them. Authors define four types of bricks: 1- neighborhoods functions, 2- strategies to scape from local minima, 3- solution managers, allowing to store the best found solution during the search, and 4- stop criteria. This approach can speed up the development and experimentation of search procedures when developing a specific solver based on local search. 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 working in parallel. It must also provide a way to create an abstract solver by combining simple components or functions that called modules.

ParadisEO is a framework presented by Cahon et al. in [4] 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. Martin et al. propose in [5] 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.

We can cite a lot of works to prove that high-level framework for the development of efficient algorithms, as well as the parallel approach with or without communication have been widely used reporting good results. However, a solution combining a modular way to construct algorithms with tools to manage the parallelism is still missing. It is well know

that interprocess communication can help during the search process of constrained solvers, but it is also well know that this communication is very complicated to perform in practice, due to certain overheads. In that sense this thesis propose a framework to create parallel communication strategies, by providing tools to easily manipulate:

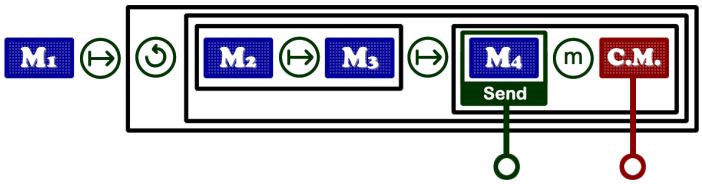
1. **where to perform the communication** → using operators to combine information reception modules with compound modules inside the algorithm,
2. **what to communicate** → configurations, neighborhoods, configuration costs, algorithms, etc., and
3. **how to perform the communication** → providing instructions to create and connect in different ways sets of solvers.

1.1.2 POSL

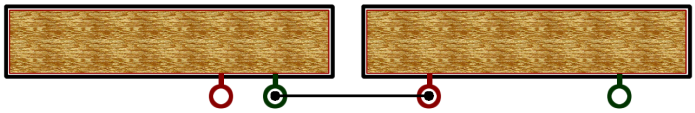
In this chapter is presented POSL, the main contribution of this thesis, as well as the different steps to build communicating parallel solvers with. It is proposed as a new way to implement *solution algorithms* to solve Constraint Satisfaction Problems, through local-search meta-heuristics using the 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.

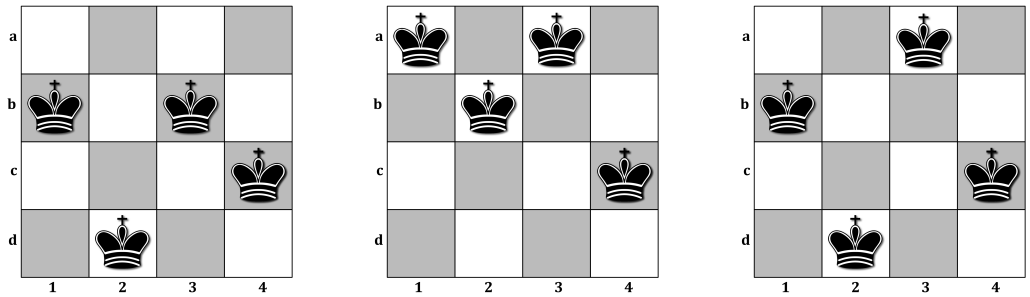


Figure 1.2: 4-Queens examples

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.

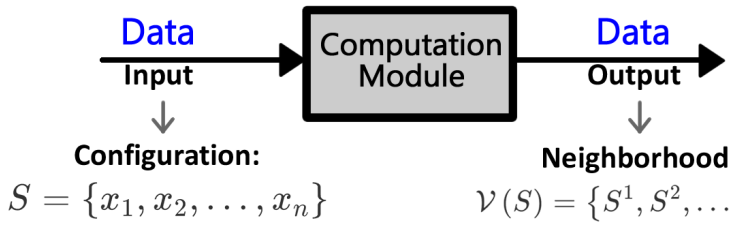


Figure 1.3: An example of a computation module computing a neighborhood

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 $\{S^1, S^2, \ldots, S^m\}$.

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 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).

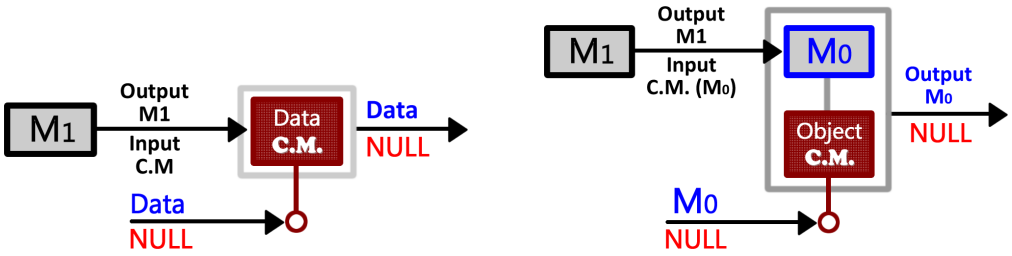
Definition 2 ***(Data Communication Module)*** *A* Data Communication Module *Ch is a module that produces a mapping defined as follows:*

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

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:*

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



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

Figure 1.4: Communication module

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.

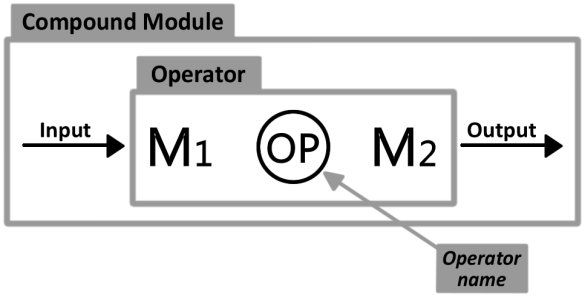


Figure 1.5: A compound module made of two modules M_1 and M_2

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. $\Sigma = \left\{\alpha, \beta, be, [\,], [\![\,]\!]_p, (\!|\,|\!)^d, |\!)^m, \textcircled{\scriptstyle\mapsto}, \textcircled{\scriptstyle ?}, \textcircled{\scriptstyle \circlearrowleft}, \textcircled{\scriptstyle \rho}, \textcircled{\scriptstyle \vee}, \textcircled{\scriptstyle \wedge}, \textcircled{\scriptstyle M}, \textcircled{\scriptstyle m}, \textcircled{\scriptstyle \downarrow}, \textcircled{\scriptstyle \cup}, \textcircled{\scriptstyle \cap}\right\}$ *is the set of terminals,*
3. $\mathbf{S} = \{CM\}$ *is the set of start variables,*
4. *and* $\mathbf{R} =$

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

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:
>
> | | |
> |---|---|
> | $Abstract\ computation\ module-1$ | I: Generating a configuration s |
> | $Abstract\ computation\ module-2$ | V: Defining the neighborhood $\mathcal{V}(s)$ |
> | $Abstract\ computation\ module-3$ | S: Selecting $s' \in \mathcal{V}(s)$ |
> | $Abstract\ computation\ module-4$ | 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 $(\mapsto)$ **Sequential Execution Operator.** *Let*

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

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

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

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:

$$\Big[\Big[\Big[I \textcircled{\mapsto} V\Big] \textcircled{\mapsto} S\Big] \textcircled{\mapsto} A\Big]$$

In the figure, each rectangle is a compound module.

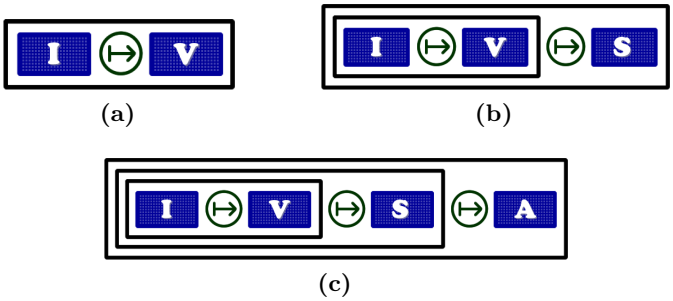


Figure 1.6: Using **sequential execution** operator

***

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

Definition 8 $\textcircled{?}$ **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 $\left|\mathcal{M}_1 \textcircled{?}_{<cond>} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{cond}$ *as result of the sequential execution of* $\mathcal{M}_1$ *if* $< cond >$ *is* **true** *or* $\mathcal{M}_2$*, otherwise:*

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

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):

$$\Big[\Big[\Big[I \textcircled{\mapsto} V\Big] \textcircled{\mapsto} \Big[S_1 \textcircled{?} S_2\Big]\Big] \textcircled{\mapsto} A\Big]$$

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 ⟲ **Cyclic Execution Operator** *Let* $\mathcal{M} : \mathcal{I} \rightarrow \mathcal{O}$ *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***:*

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

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):

$$\Big[I \mapsto \Big[\circlearrowleft \Big[\Big[V \mapsto S \Big] \mapsto A \Big] \Big] \Big]$$

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

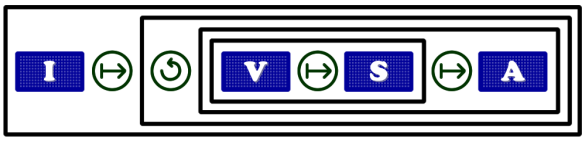


Figure 1.8: Using **cyclic execution** operator

***

Definition 10 ⟲ **Random Choice 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. 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* $(1-\rho)$*:*

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

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):

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

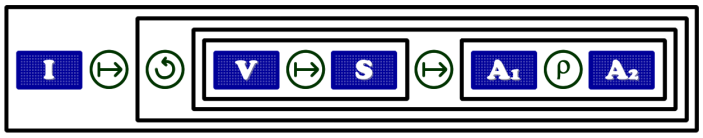


Figure 1.9: Using **random choice** operator

***

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:*

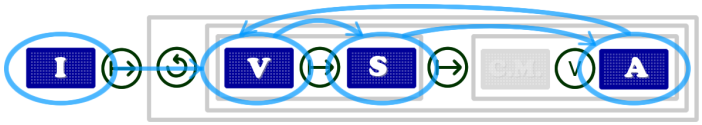
$$\mathcal{M}_{non} : \mathcal{I} \rightarrow \mathcal{O}_1 \cup \mathcal{O}_2$$

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:

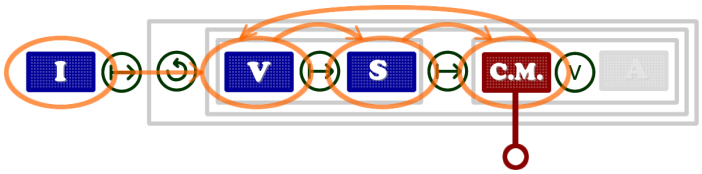
$\boxed{Communication\ module - 1:}$ $C.M.$: Receiving a configuration.

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:

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



(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 $\wedge$ *BOTH* **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 $\left|\mathcal{M}_1 \textcircled{\wedge} \mathcal{M}_2\right|$ *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 $\textcircled{m}$ **Minimum 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 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* $\left|\mathcal{M}_1 \textcircled{m} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{min}$ *that executes* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns* $\min\{o_1, o_2\}$*:*

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

***

Similarly we define the **Maximum** operator:

Definition 14 $\textcircled{M}$ **Maximum 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 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* $\left|\mathcal{M}_1 \textcircled{M} \mathcal{M}_2\right|$ *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):

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

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

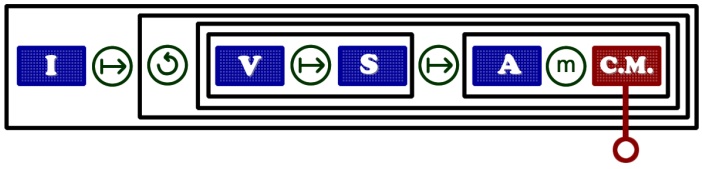


Figure 1.11: Using **minimum** operator

***

Definition 15 (↓) **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* $\left|\mathcal{M}_1 \textcircled{\downarrow} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_{race}$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns the output of the module ending first:*

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

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):

$$\left[I \textcircled{\mapsto} \left[\textcircled{\circlearrowleft} \left[\left[\left[V_1 \textcircled{\downarrow} V_2 \right]_p \textcircled{\mapsto} S \right] \textcircled{\mapsto} \left[A \textcircled{m} C.M. \right]_p \right] \right] \right]$$

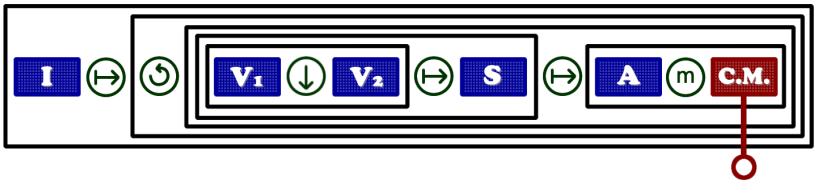


Figure 1.12: Using **race** operator

***

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

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

***

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

Definition 17 $\textcircled{\cap}$ **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* $\left|\mathcal{M}_1 \textcircled{\cap} \mathcal{M}_2\right|$ *defines the compound module* $\mathcal{M}_\cap$ *that executes both modules* $\mathcal{M}_1$ *and* $\mathcal{M}_2$*, and returns* $V_1 \cap V_2$*:*

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

***

Definition 18 $\textcircled{\setminus}$ **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* $V_1 \setminus V_2$*:*

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

***

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* $\mathcal{M} : \mathcal{I} \rightarrow \mathcal{O}$ *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:*

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

***

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

Definition 20 $(\!|.|\!)^m$ **Sending Module Operator** *Let* $\mathcal{M} : \mathcal{I} \rightarrow \mathcal{O}$ *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:*

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

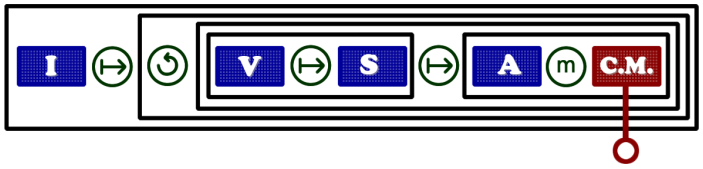
In the following example, I use one of the compound modules already presented in the previews examples, using a communication module to receive a configuration (see Figure 1.13a):

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

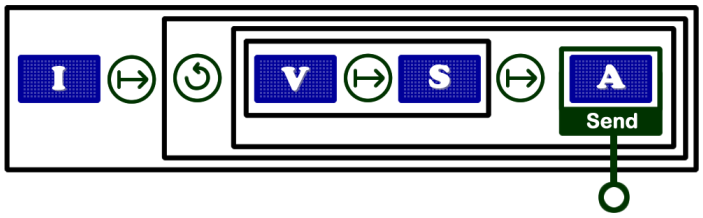
I also build another, as its complement: sending the accepted configuration to outside, using the **sending data operator** (see Figure1.13b):

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

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
- **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
$I \ (\mapsto) \ [(\circlearrowleft) \ (\text{ITR} < K_1) \ \left[V \ (\mapsto) \ S \ (\mapsto) \ \left[C.M. \ (m) \ (\!|A|\!)^d\right]\right] \ ]$
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

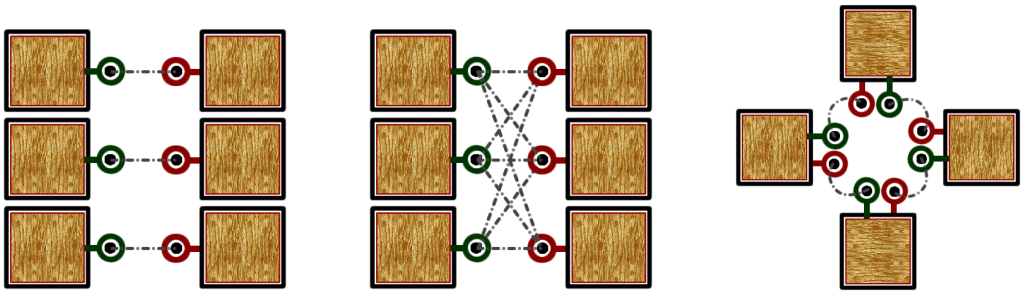
$$\mathcal{J} \boxed{\leftrightarrow} \mathcal{O}$$

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:

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

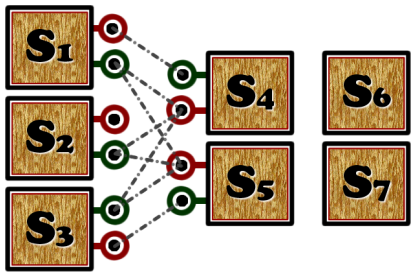


Figure 1.15: Graphic representation of communication operators

***

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 : $\mathcal{J}$ list of communication jacks,
$\mathcal{O}$ list of communication outlets

while *no available jacks or outlets remain* **do**
  $S_{jack} \leftarrow$ `GetNext(`$\mathcal{J}$`)`
  $R_{outlet} \leftarrow$ `GetNext(`$\mathcal{O}$`)`
  $S \leftarrow$ `GetSolverFromConnector(`S_{jack}`)`
  $R \leftarrow$ `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 **1 to N**:

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

If the operator **1 to N** is used with only with one solver in each list, the operation is equivalent to applying the operator **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

1 [S · *A* (3)] $\boxed{\rightsquigarrow}$ [Z · *C.M.* (3)] 2 ;

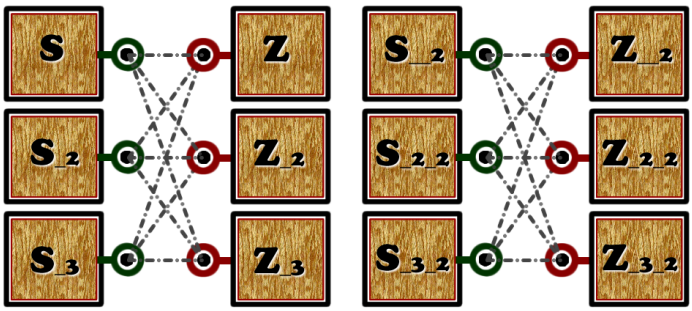


Figure 1.16: An example of connection strategy for 12 units of computation

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.

Part II

Study and evaluation of POSL

2

Experiments design and results

In this Chapter I expose all details about the evaluation process of POSL*, i.e., all experiments I perform. For each benchmark, I explain used strategies in the evaluation process and the used environments were the runs were performed (Curiosiphi server). I describe all the experiments and I expose a complete analysis of the obtained result.*

Contents

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

The experiments[i] were performed on an Intel® Xeon™ E5-2680 v2, 10×4 cores, 2.80GHz. This server is called *Coriosiphi* and is located at *Laboratoire d'Informatique de Nantes Atlantique*, at the University of Nantes. Showed results are the means of 30 runs for each setup, presented in columns labeled **T**, corresponding to the run-time in seconds, and **It.** corresponding to the number of iterations; and their respective standard deviations (**T(sd)** and **It.(sd)**). In some tables, the column labeled **% success** indicates the percentage of solvers finding a solution before reaching a time–out (5 minutes).

The experiments in this Chapter are multi-walk runs. Parallel experiments use 40 cores for all problem instances. It is important to point out that POSL is not designed to obtain the best results in terms of performance, but to give the possibility of rapidly prototyping and studying different cooperative or non cooperative search strategies.

All benchmarks were coded using the POSL low-level framework in C++.

First results using POSL to solve constraint problems were published in [6] were we used POSL to solve the *Social Golfers Problem* and to study some communication strategies. It was the first version of POSL, therefore it was able to solve only relatively easy instances. However, results suggested that the communication can play an important role if we are able to find the proper communication strategy.

With the next and more optimized version of POSL, I decide to start to perform more detailed studies using the benchmark mentioned before and some others.

2.1 Solving the *Social Golfers Problem*

In this section I present the performed study using *Social Golfers Problem* (*SGP*) as a benchmark. The communication strategy analyzed here consists in applying a mechanism of cost descending acceleration, exchanging the current configuration between two solvers with different characteristics. Final obtained results show that this communication strategy works pretty well for this problem.

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

2.1.1 Problem definition

The *Social Golfers Problem* (*SGP*) consists in scheduling $g \times p$ golfers into g groups of p players every week for w weeks, such that two players play in the same group at most once. An instance of this problem can be represented by the triple $g - p - w$. This problem, and other closely related problems, arise in many practical applications such as encoding, encryption, and covering problems [7]. Its structure is very attractive, because it is very similar to other problems, like *Kirkman's Schoolgirl Problem* and the *Steiner Triple System*.

The cost function for this benchmark was implemented making an efficient use of the stored information about the cost of the previews configuration. Using integers to work with bit-flags, a table to store the information about the partners of each player in each week can be filled in $O(p^2 \cdot g \cdot w)$. So, if a configuration has $n = (p \cdot g \cdot w)$ elements, this table can be filled in $O(p \cdot n)$. This table is filled from scratch only one time in the search process (I explain in the next section why). Then, every cost of a new configuration, is calculated based on this information and the performed changes between the new configuration and the stored one. This relative cost is calculated in $O(c \cdot g)$, where c is the number of performed changed in the new configuration with respect to the stored one.

2.1.2 Experiment design and results

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

1. Generation abstract module I:

   I_{BP}: Returns a random configuration s, respecting the structure of the problem, *i.e.*, the configuration is a set of w permutations of the vector $[1..n]$, where $n = g \times p$.

2. Neighborhood abstract modules V:

   V_{std}: Given a configuration, returns the neighborhood $\mathcal{V}(s)$ swapping players among groups.

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

Algorithm 5: Simple solvers for *SGP*

```
1 abstract solver as_simple                                   // ITR → number of iterations
2 computation : I, V, S, A
3 begin
4     [(↺) (ITR < K1) I (↦)  [(↺) (ITR % K2) [V (↦) S (↦) A] ]  ]
5 end
6 solver SOLVER_Std implements as_simple
7     computation : I_BP, V_std, S_best, A_AI
8 solver SOLVER_AS implements as_simple
9     computation : I_BP, V_BAS, S_best, A_AI
```

$V_{BP}(p)$: Given a configuration, returns the neighborhood $\mathcal{V}(s)$ by swapping the most culprit player with other players in the same week, for all p randomly selected weeks.

3. Selection abstract modules S:

   S_{first}: Given a neighborhood, selects the first configuration $s' \in V(s)$ improving the current cost and returns it together with the current one into the pair (s', s).

   S_{best}: Given a neighborhood, selects the best configuration $s' \in V(s)$ improving the current cost and returns it together with the current one into the pair (s', s).

   S_{rand}: Given a neighborhood, selects randomly a configuration $s' \in V(s)$ and returns it together with the current one, into the pair (s', s).

4. Acceptance abstract module A:

   A_{AI}: Given a pair (s', s), returns always the configuration s'

These concrete modules are very useful and can be reused to solve tournament-like problems like *Sports Tournament Scheduling*, *Kirkman's Schoolgirl* and the *Steiner Triple System* problems.

In a first stage of the experiments I use the operator-based language provided by POSL to build and test many different non communicating strategies. The goal is to select the best concrete modules to run tests performing communication. A very first experiment was performed to select the best neighborhood function to solve the problem, comparing a basic solver using V_{std}; a new solver using V_{BAS}; and a combination of V_{std} and V_{BAS} by applying the operator (ρ), already introduced in the previous chapter. Algorithms 5 and 6 present solvers for each case, respectively.

Solver	T	T(sd)	It.	It.(sd)
SOLVER$_{AS}$	**1.06**	0.79	352	268
SOLVER$_{rho}$	41.53	26.00	147	72
SOLVER$_{Std}$	87.90	41.96	146	58

Table 2.1: *Social Golfers*: Instance 10–10–3 in parallel

Algorithm 6: Solvers combining neighborhood functions using operator *RHO*

```
1 abstract solver as_rho                          // ITR → number of iterations
2 computation : I, V1, V2, S, A
3 begin
4     [(↻) (ITR < K1) I (↦)  [(↻) (ITR % K2) [[V1 (ρ) V2] (↦) S (↦) A] ] ]
5 end
6 solver SOLVER_rho implements as_rho
7     computation : I_BP, V_std, V_BAS, S_best, A_AI
```

Results in Table 2.1 are not surprising. The neighborhood module V_{BAS} is based on the *Adaptive Search* algorithm, which has shown very good results [8]. It selects the most culprit variable (i.e., a player), that is, the variable the most responsible for constraints violation. Then, it permutes this variable value with the value of each other variable, in all groups and all weeks. Each permutation gives a neighbor of the current configuration. V_{Std} uses no additional information, so it performs every possible swap between two players in different groups, every week. It means that this neighborhood is $g \times p$ times bigger than the previous one, with g the number of groups and p the number of players per group. It allows for more organized search because the set of neighbors is pseudo-deterministic, i.e., the construction criteria is always the same but the order of the configuration is random. On the other hand, *Adaptive Search* neighborhood function takes random decisions more frequently, and the order of the configurations is random as well. We also tested a solver with combining these modules using the (ρ) operators. This operator executes its first or second parameter depending on a given probability ρ. This combination spent more time searching the best configuration among the neighborhood, although with a lower number of iterations than V_{BAS}. The V_{BAS} neighborhood function being clearly faster, we have chosen it for our experiments, even if it shown a more spread standard deviation: 0.75 for SOLVER$_{AS}$ versus 0.62 for SOLVER$_{Std}$, considering the ratio $\frac{T(sd)}{T}$.

***

Once the neighborhood computation module has been selected, I have focused the experiment on choosing the best *selection* computation module. Solvers mentioned above were too slow to solve instances of the problem with more than three weeks: they were very often trapped into local minima. For that reason, another solver implementing the abstract solver described in

Instance	Best improvement				First improvement			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.45	0.70	406	726	0.23	0.14	142	67
8–4–7	0.37	0.11	68	13	0.28	0.07	93	13
9–4–8	0.87	0.13	95	17	0.60	0.16	139	18

Table 2.2: *Social Golfers*: comparing selection functions in parallel

Algorithm 7 have been created, using V_{BAS} and combining S_{best} and S_{rand}: it tries a number of times to improve the cost, and if it is not possible, it picks a random neighbor for the next iteration. We also compared the S_{first} and S_{best} selection modules. The computation module S_{best} selects the best configuration inside the neighborhood. It did not only spend more time searching a better configuration, but also is was more sensitive to become trapped into local minima. The second computation module S_{first} selects the first configuration inside the neighborhood improving the current cost. Using this module, solvers favor exploration over intensification and of course spend clearly less time searching into the neighborhood.

Algorithm 7: Solver for *SGP* to scape from local minima

```
1 abstract solver as_eager                          // ITR → number of iterations
2 computation : I, V, S1, S2, A
3 begin
4     [↻ (ITR < K1) I ↦ [↻ (ITR % K2) [V ↦ [S1 ?_{SCI<K3} S2] ↦ A] ] ]
5 end
6 solver SOLVER_best implements as_eager
7     computation : I_BP, V_std, V_BAS, S_best, S_rand, A_AI
8 solver SOLVER_first implements as_eager
9     computation : I_BP, V_std, V_BAS, S_first, S_rand, A_AI
```

Instance	T	T(sd)	It.	It.(sd)
5–3–7	1.25	1.05	2,907	2,414
8–4–7	0.60	0.33	338	171
9–4–8	1.04	0.72	346	193

Table 2.3: *Social Golfers*: a single sequential solver using first improvement

Tables 2.2 and 2.3 present results of this experiment, showing that a local exploration-oriented strategy is better for the *SGP*. If we compare results of Tables 2.2 2.3 with respect to the standard deviation, we can some gains in robustness with parallelism. The spread in the running times and iterations for the instance 5–3–7 is 24% lower (0.84 sequentially versus 0.60 in parallel), for 8–4–7 is 30% lower (0.55 sequentially versus 0.25 in parallel) and for 9–4–8 (the hardest one) is 43% lower (0.69 sequentially versus 0.26 in parallel), using the same ratio $\frac{T(sd)}{T}$).

***

The conclusion of the last experiment was that the best solver to solve *SGP* using POSL is the one using a neighborhood computation module based on *Adaptive Search* algorithm (V_{BAS}) and a selection computation module selecting the first configuration improving the cost. Using this solver as a base, the next step was to design a simple communication strategy where the shared information is the current configuration. Algorithms 8 and 9 show that the communication is performed while applying the acceptance criterion of the new configuration for the next iteration. Here, receiver solvers receive a configuration from a sender solver, and match it with their current configuration. Then, the configuration with the lowest global cost is selected. This operation is coded using the *minimum* operator (m) in Algorithm 9. This way, the receiver solver continues the search from a more promising place into de search space. Different communication strategies were designed, either executing a full connected solvers set, or a tuned combination of connected and unconnected solvers. Between connected solvers, two different connections operations were applied: connecting each sender solver with one receiver solver (one to one), or connecting each sender solver with all receiver solvers (one to N). The code for the different communication strategies are presented in Algorithms 10 to 15.

Algorithm 8: Communicating abstract solver for *SGP* (sender)

```
1  abstract solver as_eager_sender                    // ITR → number of iterations
2  computation : I, V, S1, S2, A                      // SCI → number of iterations with the same cost
3  begin
4      [(↺) (ITR < K1)  I (↦) [(↺) (ITR % K2) [V (↦) [S1 (?)SCI<K3 S2] (↦) (|A|)^o] ] ]
5  end
6  solver SOLVER_sender implements as_eager_sender
7      computation : I_BP, V_BAS, S_first, S_rand, A_AI
```

Algorithm 9: Communicating abstract solver for *SGP* (receiver)

```
1  abstract solver as_eager_receiver                  // ITR → number of iterations
2  computation : I, V, S1, S2, A                      // SCI → number of iterations with the same cost
3  communication : C.M.
4  begin
5      [(↺) (ITR < K1)
6          I (↦) [(↺) (ITR % K2)  V (↦) [S1 (?)SCI<K3 S2] (↦) [A (m) C.M.] ]
7      ]
8  end
9  solver SOLVER_receiver implements as_eager_receiver
10     computation : I_BP, V_BAS, S_first, S_rand, A_AI
11     communication : CM_last
```

Algorithm 10: Communication strategy one to one 100%

```
1  [SOLVER_sender · A] [→] [SOLVER_receiver · C.M.] 20;
```

Algorithm 11: Communication strategy one to N 100%

1 $[\text{SOLVER}_{sender} \cdot A(20)] \boxed{\rightsquigarrow} [\text{SOLVER}_{receiver} \cdot C.M.(20)]$;

Algorithm 12: Communication strategy one to one 50%

1 $[\text{SOLVER}_{sender} \cdot A] \boxed{\rightarrow} [\text{SOLVER}_{receiver} \cdot C.M.]\, 10$;
2 $[\text{SOLVER}_{first}]\, 20$;

In Algorithm 9, the abstract communication module $C.M.$ was instantiated with the concrete communication module CM_{last}, which takes into account the last received configuration at the time of its execution.

Each time a POSL meta-solver is launched, many independent search solvers are executed. We call "good" configuration a configuration with the lowest cost within the current configuration neighborhood and with a cost strictly lesser than the current one. Once a good configuration is found in a sender solver, it is transmitted to the receiver one. At this moment, if the information is accepted, there are some solvers searching in the same subset of the search space, and the search process becomes more exploitation–oriented. This can be problematic if this process makes solvers converging too often towards local minima. In that case, we waste more than one solver trapped into a local minima: we waste all solvers that have been attracted to this part of the search space because of communications. This phenomenon is avoided through a simple (but effective) play: if a solver is not able to find a better configuration inside the neighborhood (executing S_{first}), it selects a random one at the next iteration (executing S_{rand}).

In all Algorithms in this section, three parameter can be found: 1. K_1: the maximum number of *restarts*, 2. K_2: the maximum number of iterations in each *restart*, and K_3: the maximum number of iterations with the same current cost. 3.

After the selection of the proper modules to study different communication strategies, I proceeded to tune these parameter. Only a few runs were necessaries to conclude that the mechanism of using the computation module S_{rand} to scape from local minima was enough. For that reason, since the solver never perform restarts, the parameter K_1 was irrelevant. So the reader can assume $K_1 = 1$ for every experiment.

With the certainty that solvers do not performs restarts during the search process, I select the same value for $K_2 = 5000$ in order to be able to use the same abstract solver for all instances.

Algorithm 13: Communication strategy one to N 50%

1 $[\text{SOLVER}_{sender} \cdot A(10)] \boxed{\rightsquigarrow} [\text{SOLVER}_{receiver} \cdot C.M.(10)]$;
2 $[\text{SOLVER}_{first}]\, 20$;

Algorithm 14: Communication strategy one to one 25%

1 $[\text{SOLVER}_{sender} \cdot A] \boxed{\rightarrow} [\text{SOLVER}_{receiver} \cdot C.M.]\, 5;$
2 $[\text{SOLVER}_{first}]\, 30;$

Algorithm 15: Communication strategy one to N 25%

1 $[\text{SOLVER}_{sender} \cdot A(5)] \boxed{\rightsquigarrow} [\text{SOLVER}_{receiver} \cdot C.M.(5)]\,;$
2 $[\text{SOLVER}_{first}]\, 30;$

Finally, in the tuning process of K_3, I notice only slightly differences between using the values 5, 10, and 15. So I decided to use $K_3 = 5$.

This communication strategy produces some gain in terms of runtime (Table 2.2 with respect to Tables 2.4, 2.5 and 2.6). Having many solvers searching in different places of the search space, the probability that one of them reaches a promising place is higher. Then, when a solver finds a good configuration, it can be communicated, and receiving the help of one or more solvers in order to find the solution. Using this strategy, the spread in the running times and iterations was reduced for the instance 9–4–8 (0.22 using communication one to one and 50% of communication solvers), but not for instances 5–3–7 and 8–4–7 (0.70 using communication one to N and 50% of communicating solvers, and 0.28 using communication one to one and 50% of communicating solvers, respectively).

Other two strategies were analyzed in the resolution of this problem, with no success, both based on the sub-division of the work by weeks, i.e., solvers trying to improve a configuration only working with one or some weeks. To this end two strategies were designed:

A **Circular strategy:** K solvers try to improve a configuration during a during a number of iteration, only working on one week. When no improvement is obtained, the current configuration is communicated to the next solver (circularly), which tries to do the same working on the next week (see Figure 2.1a).

This strategy does not show better results than previews strategies. The reason is because, although the communication in POSL is asynchronous, most of the times solvers were trapped waiting for a configuration coming from its neighbor solver.

B **Dichotomy strategy:** Solvers are divided by levels. Solvers in level 1, only work

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.20	0.20	165	110	0.20	0.17	144	108
8–4–7	0.27	0.09	88	28	0.24	0.05	95	12
9–4–8	0.52	0.14	117	25	0.55	0.14	126	20

Table 2.4: *Social Golfers*: 100% of communicating solvers

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.18	0.13	125	88	0.17	0.12	139	81
8–4–7	0.21	0.06	89	18	0.22	0.06	90	20
9–4–8	0.49	0.11	119	24	0.51	0.15	124	21

Table 2.5: *Social Golfers*: 50% of communicating solvers

Instance	Communication 1 to 1				Communication 1 to N			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.22	0.20	181	130	0.23	0.16	143	80
8–4–7	0.24	0.08	95	22	0.29	0.09	93	12
9–4–8	0.55	0.14	134	21	0.55	0.11	130	20

Table 2.6: *Social Golfers*: 25% of communicating solvers

on one week, solvers on level 2, only work on 2 consecutive weeks, and so on, until the solver that works on all (except the first one) weeks. Solvers in level 1 improve a configuration during some number of iteration, then this configuration is sent to the corresponding solver. Solvers in level 2 do the same, but working on weeks k to $k+1$. It means that it receives configurations from the solver working on week k and from the solver working on week $k+1$, and sends its configuration to the corresponding solver working on weeks k to $k+3$; and so on. The solver in the last level works on all (except the first one) weeks and receive configuration from the solver working on weeks 2 to $w/2$ and from the solver working on weeks $w/2+1$ to w (see Figure2.1b). We tested this strategy with all possible levels.

The goal of this strategy was testing if focused searches rapidly communicated can help at the beginning of the search. However, The failure of this strategy is in the fact that most of the time the sent information arrives to late from the bottom to the top.

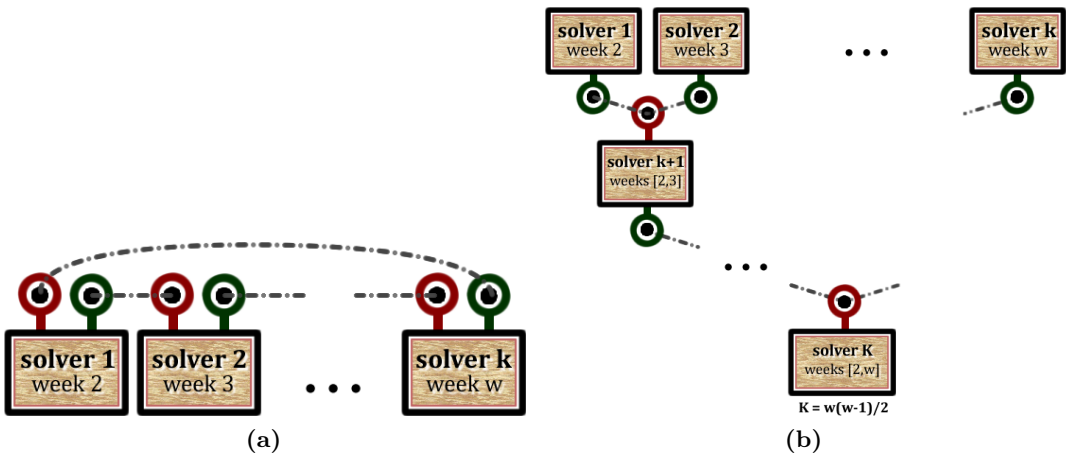


Figure 2.1: Unsuccessful communication strategies to solve *SGP*

***

One last experiment using this benchmark was implementing a communication strategy which applies a mechanism of cost descending acceleration, exchanging the current configuration between two solvers with different characteristics. Results show that this communication strategy works pretty well for this problem.

For this strategy, new solvers were built reusing same modules used for the communication strategies exposed before, and another different neighborhood computation module: $V_{BP}(p)$, which given a configuration, returns the neighborhood $\mathcal{V}(s)$ by swapping the culprit player chosen for all p randomly selected weeks with other players in the same week. This new solver was called *companion solver*, and it descends quicker the cost of its current solution at the beginning because its neighborhood generates less values, but the convergence is slower and yet not sure. It was combined with the solver similar to the one used for the communication strategies exposed before. It was called *standard solver*, and converges in a stable way to the solution. So, the companion solver uses the same neighborhood function that the standard solver, but parametrized in such a way that it builds neighbors only swapping players among two weeks.

The idea of the communication strategy is to communicate a configuration from the companion solver to the standard solver, to be able to continue the search from a more promising place into de search space. After some iterations, is the standard solver who sends its configuration to the companion solver. The companion solver takes this received configuration and starts its search from it and finds quickly a much better configuration to send to the standard solver again. To force the companion solver to take the received configurations over its own, we use the *not null* operator together with the communication module $C.M.$ (Algorithm 17). This process is repeated until a solution is found.

Figure 2.2 shows standard solver's run versus companion solver's run. In this chart we can see that, at the beginning of the run, found configurations by the companion solver have costs significantly lower than those found by the standard solver. At the 60-th millisecond the standard solver current configuration has cost 123, and the companion solver's one, 76. So for example, the communication at this time, can accelerate the process significantly.

We also design different communication strategies, combining connected and unconnected solvers in different percentages, and applying two different communication operators: one to one and one to N.

This strategy produces some gain in terms of runtime as we can see in Tables 2.7 and 2.8 with respect to Table 2.2. It produces also more robust results in terms of runtime. The spread of results in iterations show higher variances, because there are included also

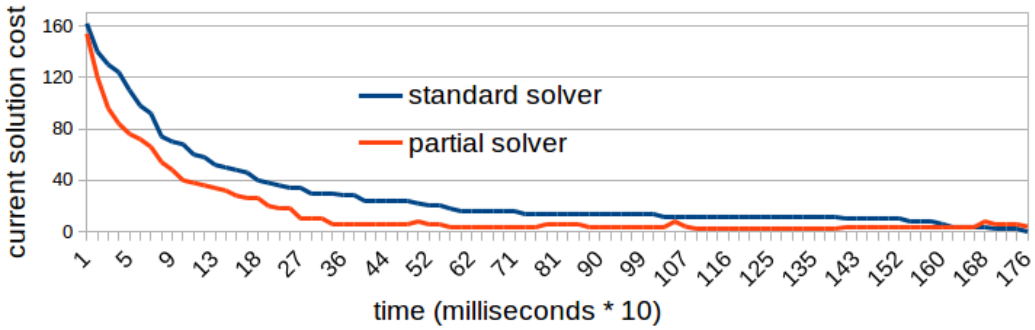


Figure 2.2: Companion solver vs. standard solver (solving *Social Golfers Problem*)

Algorithm 16: Standard solver for *SGP*

```
1 abstract solver as_standard
2 computation : I, V, S_1, S_2, A
3 communication : C.M.
4 begin
5     I ↦ [ ↺ (Itr < K_1)  V ↦ [ S_1 ?_{Sci%K_1} S_1 ] ↦ [ C.M. m (|A|)^d ] ]
6 end
7 solver Solver_standard implements as_standard
8     computation : I_BP, V_BAS, S_first, S_rand, A_AI
9     communication : CM_last
```

Algorithm 17: Companion solver for *SGP*

```
1 abstract solver as_companion
2 computation : I, V, S_1, S_2, A
3 communication : C.M.
4 begin
5     I ↦ [ ↺ (Itr < K_1)  V ↦ [ S_1 ?_{Sci%K_1} S_2 ] ↦ [ C.M. ∨ (|A|)^d ] ]
6 end
7 solver Solver_companion implements as_companion
8     computation : I_BP, V_BP(2), S_first, S_rand, A_AI
9     communication : CM_last
```

Instance	Comm. one to N				(Comm. one to N)/2				(Comm. one to N)/4			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.14	0.08	102	53	0.14	0.07	97	73	0.12	0.08	175	162
8–4–7	0.30	0.13	101	24	0.22	0.06	92	29	0.22	0.06	88	45
9–4–8	0.55	0.15	125	20	0.53	0.14	107	20	0.40	0.14	101	70

Table 2.7: Companion communication strategy with communication one to N

Instance	Comm. one to one (100%)				Comm. one to one (50%)				Comm. one to one (25%)			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	0.10	0.05	98	75	**0.08**	0.04	139	122	0.11	0.05	190	142
8–4–7	**0.14**	0.05	100	64	0.22	0.06	119	74	0.21	0.5	101	64
9–4–8	0.37	0.14	86	65	**0.36**	0.12	144	92	0.45	0.11	150	96

Table 2.8: companion communication strategy with communication one to N

results of companion solvers, which performs many times more iterations that the standard solvers. The percentage of the receiver solvers that were able to find the solution before the others did, was significant (see Appendix **??**, Figures **??**, **??** and **??**). That shows that the communication played an important role during the search, despite inter–process communication's overheads (reception, information interpretation, making decisions, etc).

The code for the communication strategy of 100% of communicating solvers is presented in Algorithm 18 and for 50% of communicating solvers in Algorithm 19.

Algorithm 18: Companion communication strategy 100% communication

1 $[\text{SOLVER}_{companion} \cdot A] \boxed{\rightarrow} [\text{SOLVER}_{standard} \cdot C.M.]\, 20;$
2 $[\text{SOLVER}_{standard} \cdot A] \boxed{\rightarrow} [\text{SOLVER}_{companion} \cdot C.M.]\, 20;$

Algorithm 19: Companion communication strategy 50% communication

1 $[\text{SOLVER}_{companion} \cdot A] \boxed{\rightarrow} [\text{SOLVER}_{standard} \cdot C.M.]\, 10;$
2 $[\text{SOLVER}_{standard} \cdot A] \boxed{\rightarrow} [\text{SOLVER}_{companion} \cdot C.M.]\, 10;$
3 $[\text{SOLVER}_{first}]\, 20;$

2.2 Solving the *N-Queens Problem*

In this section I present the performed study using *N-Queens Problem* (*NQP*) as a benchmark. The communication strategy analyzed here consists in exchanging cyclically the configuration between solvers using different neighborhood functions, in order to accelerate the process of generating very promising configurations. Final obtained results show that this communication strategy works pretty well for some instances of this problem.

2.2.1 Problem definition

The *N-Queens Problem* (*NQP*) asks how to place N queens on a chess board so that none of them can hit any other in one move. This problem was introduced in 1848 by the chess player Max Bezzelas as the *8-queen problem*, and years latter it was generalized as *N-queen problem* by Franz Nauck. Since then many mathematicians, including Gauss, have worked on this problem. It finds a lot of applications, e.g., parallel memory storage schemes, traffic control, deadlock prevention, neural networks, constraint satisfaction problems, among others [9]. Some studies suggest that the number of solution grows exponentially with the number of queens (N), but local search methods have been shown very good results for this problem [10]. For that reason we tested some communication strategies using POSL, to solve a problem relatively easy to solve using non communication strategies.

The cost function for this benchmark was implemented in C++ based on the current implementation of *Adaptive Search*[ii].

2.2.2 Experiments and results

To handle this problem, some modules used for the *Social Golfers Problem* have been reused: the selection computation modules S_{first} and S_{best}, and the acceptance computation module A_{AI}. It was used a simple abstract solver presented in Algorithm 20:

Algorithm 20: Abstract solver for *NQP*

```
1 abstract solver as_simple                          // ITR → number of iterations
2 computation : I, V, S, A                  // SCI → number of iterations with the same cost
3 begin
4     I (↦) [(↺) (ITR < K1) V (↦) S (↦) A ]
5 end
6 solver SOLVER_as implements as_simple
7     computation : I_perm, V_AS, S_first, A_AI
8 solver SOLVER_selective implements as_simple
9     computation : I_perm, V_PAS(p), S_first, A_AI
```

Solvers used for the experiments without communications, are presented in Algorithm 20, where the abstract solver is instantiated in the solver SOLVER_{as} with the neighborhood computation module V_{AS}, which given a configuration, returns a neighborhood $V(s)$ swapping the variable which contributes the most to the cost, with all others. This solver was able to find solutions but taking too much time (a minute for 6000-queens, for example). For that

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

Instance	Sequential				Parallel			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
250	0.29	0.072	8,898	2,158	0.19	0.003	4,139	913
500	0.35	0.087	4,203	998	0.24	0.036	2,675	366
1000	0.35	0.126	2,766	445	0.30	0.037	2,102	222
3000	1.50	0.138	2,191	77	1.33	0.055	2,168	71
6000	4.71	0.183	3,339	51	4.57	0.123	3,323	43

Table 2.9: Results for *NQP* (sequential and parallel without communication)

reason it was implemented a neighborhood computation module $S_{PAS}(p)$ which performs the same algorithm of S_{AS}, but instead of generating neighbors swapping the most costly variable with all others, it is swapped only with a percentage of rest of variables. Solver $\text{SOLVER}_{selective}$ instantiates the abstract solver with this computation module, showing much better results than SOLVER_{as}.

Table 2.9 presents results of sequential and parallel runs, using $\text{SOLVER}_{selective}$ with a tunned value of $p = 2.5$. Results show that the improvement of the parallel scheme using POSL is not big. While the number of solutions of this problem is only known for the very small value of $N = 23$, studies suggest that the number of solutions grows exponentially with N. It implies that as the problem grows in order, it becomes easier to solve through local search methods. The behavior of POSL solving this problem matches with this hypothesis: the search process solving larger instances is more stable and the convergence is direct. In a well spread search space with a lot of solutions, the parallelism using 40 cores do not provide a lot of improvement. In that sense, as a future work, experiments using POSL to solve *NQP* in parallel with more cores are planned.

***

In order to test the cooperative approach with this problem, a first and simple experiment was performed. Using the previous defined abstract solver, communicating solvers were built, to create a simple communication strategy in which the shared information is the current configuration, and it is communicated in one sense (from sender solvers to receivers). Algorithms 21 and 22 show that the communication is performed while applying the acceptance criterion. We design different communication strategies:

- a set of sender solvers sending information to receiver solvers, using operator one to one (see see Algorithm 23) and operator one to N (see Algorithm 24 with $K = 1$)
- some sets of sender solvers sending information to receiver solvers, using operator one to N (see see Algorithm 24), with $K \in \{2, 4\}$

Instance	Communication 1-1			
	T	T(sd)	It.	It.(sd)
250	0.18	0.040	3,433	697
500	0.25	0.047	2,216	427
1000	0.26	0.056	1,735	424
3000	1.21	0.088	1,873	227
6000	4.38	0.111	3,178	121

Table 2.10: Simple communication strategy one to one for *NQP*

Algorithm 21: Sender solver for *NQP* (simple communication strategy)

```
1 abstract solver as_sender                          // Itr → number of iterations
2 computation : I, V, S, A                  // Sci → number of iterations with the same cost
3 begin
4     I ↦ [↺ (Itr < K1) V ↦ S ↦ (|A|)^d ]
5 end
6 solver Solver_sender implements as_sender
7     computation : I_perm, V_PAS(p), S_first, A_AI
```

Algorithm 22: Receiver solver for *NQP* (simple communication strategy)

```
1 abstract solver as_receiver                        // Itr → number of iterations
2 computation : I, V, S, A                  // Sci → number of iterations with the same cost
3 communication : C.M.
4 begin
5     I ↦ [↺ (Itr < K1) V ↦ S ↦ [A ?_{Itr%K2} [A m C.M.]] ]
6 end
7 solver Solver_receiver implements as_receiver
8     computation : I_perm, V_PAS(p), S_first, A_AI
9     communication: CM_last
```

Algorithm 23: Simple communication strategy one to one for *NQP*

```
1 [Solver_sender · A] → [Solver_receiver · C.M.] 20;
```

Algorithm 24: Simple communication strategy one to N for *NQP*

```
1 [Solver_sender · A(20/K)] ⇝ [Solver_receiver · C.M.(20/K)] K;
```

Tables 2.10 and 2.11 show how the communication improve the non communicating results in terms of runtime and iterations, but this improvement is not significant. In contrast to *SGP*, POSL does not get trapped so often into local minima during the resolution of *NQP*. For that reason, the shared information, once received and accepted by the receivers solvers, does not improves largely the current cost.

***

Instance	Communication 1-n				Communication (1-n)×2				Communication (1-n)×4			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
250	0.16	0.032	2,621	894	0.15	0.036	2,459	892	0.15	0.036	2,494	547
500	0.20	0.040	1,592	428	0.19	0.053	1,521	539	0.18	0.057	1,719	593
1000	0.26	0.055	1,329	286	0.25	0.046	1,435	369	0.23	0. 056	1,400	426
3000	1.26	0.078	1,657	212	1.22	0.101	1,598	249	1.20	0.078	1,704	252
6000	4.40	0.118	2,771	197	4.35	0.127	2,840	148	4.33	0.120	2,975	188

Table 2.11: Simple communication strategy one to N for *NQP*

In the following experiment, with the goal of improving results, another communication strategy was implemented, very similar to the one applied to *SGP*, but in this case, with solvers using the same neighborhood function $V_{PAS}(p)$ but with different values of p and different selection functions. In this communication strategy a cyclic exchange of the current configuration is performed between to different solvers. One solver *companion* using the neighborhood computation module $V_{PAS}(p)$ with a smaller value of p and using the selection computation module S_{best}, meaning that it is able to find promising configuration faster, but its convergence is slower. The other solver is very similar to the one used for non communicating experiments, but in this communication strategy solvers are both senders and receivers (see Algorithm 25). Before designing communication strategies (Algorithms 26, and 27), many experiments were launched to select: 1. The percentage of variables that the companion solver swaps with the culprit one, when executing the neighborhood computation module (p). This value was decided to be 1. 2. The number of companion solvers to connect with the standard one, for the communication strategy using operator one to N. This vale was decide to be 2, as we can see in Algorithm 27.

Algorithm 25: Solvers for cyclic communication strategy to solve *NQP*

```
1  abstract solver as_cyc                                  // ITR → number of iterations
2  computation : I, V, S1, S2, A          // SCI → number of iterations with the same cost
3  communication : C.M.
4  begin
5      I (↦) [(↻) (ITR < K1) V (↦) S (↦) [A (?)_{ITR%K2} [(|A|)^d (m) C.M.]] ]
6  end
7  solver SOLVER_standard implements as_cyc
8      computation : I_perm, V_PAS(2.5), S_first, A_AI
9      communication: CM_last
10 solver SOLVER_companion implements as_cyc
11     computation : I_perm, V_PAS(1), S_best, A_AI
12     communication: CM_last
```

Algorithm 26: Cyclic communication strategy one to one for *NQP*

```
1  [SOLVER_companion · A] [→] [SOLVER_standard · C.M.] 20;
2  [SOLVER_standard · A] [→] [SOLVER_companion · C.M.] 20;
```

With this experiment, it was possible to find a communication strategy which improves

Algorithm 27: Cyclic communication strategy one to N for *NQP*

1 $[\text{SOLVER}_{companion} \cdot A(2)] \boxed{\rightsquigarrow} [\text{SOLVER}_{standard} \cdot C.M.]$ 13;
2 $[\text{SOLVER}_{standard} \cdot A] \boxed{\rightsquigarrow} [\text{SOLVER}_{companion} \cdot C.M.(2)]$ 13;

Instance	Communication 1-1				Communication 1-n				I.R.
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)	
250	**0.09**	0.021	1,169	254	0.10	0.021	1,224	254	2.00
500	**0.14**	0.027	864	121	0.15	0.030	977	220	1.65
1000	0.22	0.041	889	247	**0.21**	0.056	807	196	1.39
3000	1.25	0.090	1,602	90	**1.02**	0.145	1,613	206	1.17
6000	4.83	0.121	2,938	746	**4.24**	0.746	2,537	779	1.01

Table 2.12: Cyclic communication strategy for *NQP*

runtimes significantly, but only for small instances of the problem, where the number of solutions, with respect to the order N, is lower. This result confirms experimentally the hypothesis already introduced, which propose that as the size of the problem grows, (and with it, the number ef solutions inside the search space with respect to N) lower is the gain using communication during the search process. Table 2.12 shows how the *improvement ratio* (column **I.R.**) decreases with the instance order N. This ratio was computed using the following equation:

$$\frac{\text{runtime without communication}}{\frac{(\text{runtime using communication 1-1}+\text{runtime using communication 1-n})}{2}}$$

2.3 Solving the *Costas Array Problem*

In this section I present the performed study using *Costas Array Problem* (*CAP*) as a benchmark. This time, a simple communication strategy, in which the information to communicate between solvers is the current configuration was tested, showing good results.

2.3.1 Problem definition

The *Costas Array Problem* (*CAP*) consists in finding a *costas array*, which is an $n \times n$ grid containing n marks such that there is exactly one mark per row and per column and the $n(n-1)/2$ vectors joining each couple of marks are all different. This is a very complex problem that finds useful application in some fields like sonar and radar engineering. It also

presents an interesting characteristic: although the search space grows factorially, from order 17 the number of solutions drastically decreases [11].

The cost function for this benchmark was implemented in C++ based on the current implementation of *Adaptive Search*[iii].

2.3.2 Experiment design and results

To handle this problem, I have reused all modules used for solving the *N-Queens Problem*. First attempts to solve this problems were using the same strategies (abstract solvers) used to solve the *Social Golfers Problem* and *N-Queens Problem*, without success: POSL was not able to solve instances larger than $n = 8$ in a reasonable amount of time (seconds). After many unsuccessful attempts to find the right parameters of *maximum number of restarts*, *maximum number of iterations*, and *maximum number of iterations with the same cost*, I decided to implement the mechanism used by Daniel Díaz in the current implementation of *Adaptive Search* to escape from local minima: I have added a *Reset* computation module R_{AS} based on the abstract computation module R. The rest of computation modules were the same used for solving the *N-Queens Problem*.

The basic solver used to solve this problem is presented in Algorithm 28, and it was taken as a base to build all the different communication strategies. Basically, it is a classical local search iteration, where instead of performing restarts, it performs resets. After a deep analysis of this implementation and results of some runs, I decided to use $K_1 = 2,000,000$ (maximum number of iterations) big enough to solve the chosen instance $n = 19$; and $K_2 = 3$ (the number of iteration before performing the next *reset*).

Algorithm 28: Reset-based abstract solver for *CAP*

```
1 abstract solver as_hard                          // ITR → number of iterations
2 computation : I, R, V, S, A
3 begin
4     I (↦) [(↺) (ITR < K1)  R (↦) [(↺) (ITR % K2) [V (↦) S (↦) A] ] ]
5 end
6 solver SOLVER_single implements as_hard
7     computation : I_perm, R_AS, V_AS, S_first, A_AI
```

Table 2.13 shows results of launching solver sets to solve each instance of *Costas Array Problem* 19 sequentially and in parallel without communication. Runtimes and iteration means showed in this confirm once again the success of the parallel approach.

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

STRATEGY	T	T(ds)	It.	It.(sd)	% success
Sequential (1 core)	132.73	80.32	2,332,088	1,424,757	40.00
Parallel (40 cores)	25.51	15.75	231,262	143,789	100.00

Table 2.13: *Costas Array* 19: no communication

***

In order to improve results, a simple communication strategy was applied: communicating the current configuration to other solvers. To do so, we insert a *sending output* operator to the abstract solver in Algorithm 28. This results in the sender solver presented in Algorithm 29.

Algorithm 29: Sender solver for *CAP*

```
1 abstract solver as_hard_sender
2 computation : I, R, V, S, A
3 begin
4     I (↦) [(↻) (Itr < K1)  T (↦) [(↻) (Itr % K2) [V (↦) S (↦) (|A|)^d] ] ]
5 end
6 solver Solver_sender implements as_hard_sender
7     computation : I_perm, R_AS, V_AS, S_first, A_AI
```

Studying some runs of POSL solving *CAP*, it was observed that the cost of the current configuration of the first solver finding a solution, describes an oscillatory descent due to the repeated resets, but not so pronounced. For that reason, it was decided to apply a simple communication strategy that shares the current configuration while applying the acceptance criterion: its goal is to accelerate the cost descent. To do so, a communication module using a *minimum* operator (*m*) together with the abstract computation module A was inserted, as shown in Algorithm 30.

One of the main purpose of this study is to explore different communication strategies. We have then implemented and tested different variations of the strategy exposed above by combining two communication operators (one to one and one to N) and different percentages of communicating solvers. For this problem, it was study also the behavior of the communication performed at two different moments: while applying the acceptance criteria (Algorithm 30),

	100% COMM				50% COMM			
STRATEGY	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
Str A: 1 to 1	11.60	9.17	84,159	68,958	16.78	13.43	148,222	121,688
Str A: 1 to N	**10.83**	8.72	79,551	63,785	13.03	13.46	106,826	120,894
Str B: 1 to 1	14.84	13.54	119,635	112,085	14.51	13.88	125,982	123,261
Str B: 1 to N	22.99	23.82	199,930	207,851	16.62	15.16	138,840	116,858

Table 2.14: *Costas Array* 19: with communication

and while performing a *reset* (Algorithm 31).

Algorithm 30: Receiver solver for *CAP* (variant A)

```
1 abstract solver as_hard_receiver_a                    // ITR → number of iterations
2 computation : I, T, V, S, A
3 communication : C.M.
4 begin
5     I ↦ [↺ (ITR < K1)  T ↦ [↺ (ITR % K2) [V ↦ S ↦ [A (m) C.M.]] ] ]
6 end
7 solver SOLVER_receiverA implements as_hard_receiver_a
8     computation : I_perm, R_AS, V_AS, S_first, A_AI
9     communication: CM_last
```

Algorithm 31: Receiver solver for *CAP* (variant B)

```
1 abstract solver as_hard_receiver_b                    // ITR → number of iterations
2 computation : I, R, V, S, A
3 communication : C.M.
4 begin
5     I ↦ [↺ (ITR < K1) [R (m) C.M.] ↦ [↺ (ITR % K2) [V ↦ S ↦ A] ] ]
6 end
7 solver SOLVER_receiverB implements as_hard_receiver_b
8     computation : I_perm, R_AS, V_AS, S_first, A_AI
9     connection: CM_last
```

The instantiation for receiver solvers instantiates the abstract communication module $C.M.$ with the concrete communication module CM_{last}, which takes into account the last received configuration at the time of its execution.

Table 2.14 shows that solver sets executing the strategy A (receiving the configuration at the time of applying the acceptance criteria) is more effective. The reason is that the others, interfere with the proper performance of the *reset*, which is a very important step in the algorithm. This step can be performed on three different ways:

1. Trying to shift left/right all sub-vectors starting or ending by the variable which contributes the most to the cost, and selecting the configuration with the lowest cost.

2. Trying to add a constant (circularly) to each element in the configuration.

3. Trying to shift left from the beginning to some culprit variable (i.e., a variable contributing to the cost).

Then, one of these 3 generated configuration has the same probability of being selected, to be the result of the *reset* step. In that sense, some different *resets* can be performed for the same configuration. Here is when the communication play its important role: receiver and sender solvers apply different *reset* in the same configuration, providing an exploratory factor. Results showed the efficacy of this communication strategy.

Analyzing the whole information obtained during the experiments, we can observe that the percentage of communicating solvers finding the solution thanks to the received information was high. That shows that the communicated information was very helpful during the search process. With the simplicity of the operator-based language provided by POSL, we were able to find a simple communication strategy to obtain better results than applying sequential and parallel independent multi-walk approaches. As expected, the best strategy was based on 100% of communication and a one to N communication, because this strategy allows to communicate a promising place inside the search space to a maximum of solvers, helping the decisive intensification process. Algorithm 32 shows the code of this communication strategy. Using the one to N operator $\boxed{\rightsquigarrow}$ each sender solver sends information to every receiver solver.

Algorithm 32: Communication strategy one to N 100% for *CAP*

1 $[\text{SOLVER}_{sender} \cdot A(20)] \boxed{\rightsquigarrow} [\text{SOLVER}_{receiverA} \cdot C.M.(20)]$;

Table 2.14 shows also high values of standard deviation. This is not surprising, due to the highly random nature of the neighborhood function and the selecting criterion, as well as the execution of many resets during the search process.

2.4 Solving the *Golomb Ruler Problem*

In this section, the performed study using *Golomb Ruler Problem* (*GRP*) as a benchmark is presented. Using this benchmark, a different communication strategy was tested: we communicate the current configuration in order to avoid its neighborhood, i.e., a *tabu* configuration.

2.4.1 Problem definition

The *Golomb Ruler Problem* (*GRP*) problem consists in finding an ordered vector of n distinct non-negative integers, called *marks*, $m_1 < \cdots < m_n$, such that all differences $m_i - m_j$ $(i > j)$ are all different. An instance of this problem is the pair (o, l) where o is the order of the problem, (i.e., the number of *marks*) and l is the length of the ruler (i.e., the last *mark*). We assume that the first *mark* is always 0. This problem has been applied to radio astronomy, x-ray crystallography, circuit layout and geographical mapping [12]. When POSL is applied to solve an instance of this problem sequentially, It can be noticed that it performs many *restarts* before finding a solution. For that reason this problem was chosen to study a new communication strategy.

The cost function is implemented based on the storage of a counter for each measure present in the rule (configuration). All distances where a variable is participating are also stored. This information is useful to compute the more culprit variable (the variable that interferes less in the represented measures), in case of the user wants to apply algorithms like *Adaptive Search*. This cost is calculated in $O\left(o^2 + l\right)$.

2.4.2 Experiment design and results

Golomb Ruler Problem instances were used to study a different communication strategy. This time the current configuration is communicated, to avoid its neighborhood, i.e., a *tabu* configuration. Some modules used in the resolution of *Social Golfers* and *Costas Array* problems have been reused to design the solvers: the *Selection* and *Acceptance* modules. The new modules are:

1. I_{sort}: returns a random configuration s as an ordered vector of integers. The configuration is generated *far* from the set of *tabu* configurations arrived via solver-communication (in communicating strategies) (based on the *generation* abstract module I).

2. V_{sort}: given a configuration, returns the neighborhood $V\left(s\right)$ by changing one value while keeping the order, i.e., replacing the value s_i by all possible values $s'_i \in D_i$ satisfying $s_{i-1} < s'_i < s_{i+1}$ (based on the *neighborhood* abstract module V).

It was also added an abstract module R for reset: it receives and returns a configuration. The concrete reset module used for this problem (R_{tabu}) inserts the input configuration into a *tabu* list inside the solver and returns the input configuration as-is. As Algorithm 34 shows, this module is executed just before performing a restart, so the solver was unable to find a

better configuration around the current one. Therefore, the current configuration is assumed to be a local minimum, and it is inserted into a tabu list.

Algorithm 33 and 34 present both solvers, using a tabu list and without using it. They were used to obtain results presented in Tables 2.15 and 2.16 to show that the approach explained above provides some gain in terms of runtime.

Algorithm 33: Solver without using tabu list, for *GRP*

```
1 abstract solver as_golomb_notabu                    // Itr → number of iterations
2 computation : I, V, S, A
3 begin
4     [↺ (Itr < K1)  I ↦  [↺ (Itr % K2) [V ↦ S ↦ A] ]  ]
5 end
6 solver Solver_notabu implements as_notabu
7     computation : I_sort, V_sort, S_first, A_AI
```

Algorithm 34: Solver using tabu list, for *GRP*

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

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)	% success
8–34	0.79	0.66	13,306	11,154	66	55.74	100.00
10–55	66.44	49.56	451,419	336,858	301	224.56	80.00
11–72	160.34	96.11	431,623	272,910	143	90.91	26.67

Table 2.15: A single sequential solver without using tabu list for *GRP*

Instance	T	T(sd)	It.	It.(sd)	R	R(sd)	% success
8–34	0.66	0.63	10,745	10,259	53	51.35	100.00
10–55	67.89	50.02	446,913	328,912	297	219.30	88.00
11–72	117.49	85.62	382,617	275,747	127	91.85	30.00

Table 2.16: A single sequential solver using tabu list for *GRP*

The benefit of the parallel approach is also proved for the *Golomb Ruler Problem*, as we can see in Table 2.17. However, without communication, the improvement is not substantial (8% for 8–34, 7% for 10–55 and 5% for 11–72). The reason is because only one configuration is inserted in the tabu list after each restart.

***

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

Table 2.17: Parallel solvers using tabu list for *GRP*

The main goal of choosing this benchmark was to study a different communication strategy, since for solving this problem, POSL needs to perform some restarts. In this communication strategy, solvers do not communicate the current configuration to have more solvers searching in its neighborhood, but a configuration that we assume is a local minimum to be avoided. We consider that the current configuration is a local minimum since the solver (after a given number of iteration) is not able to find a better configuration in its neighborhood, so it will communicate this configuration just before performing the restart.

Algorithm 35 presents the sender solver and Algorithm 36 presents the receiver solver. Based on the connection operator used in the communication strategy, this solver might receives one or many configurations. These configurations are the input of the generation module (I), and this module inserts all received configurations into a *tabu* list, and then generates a new first configuration, far from all configurations in the *tabu* list.

Algorithm 35: Sender solver for *GRP*

```
1 abstract solver as_golomb_sender                    // ITR → number of iterations
2 computation : I, V, S, A, R
3 begin
4     [↺ (ITR < K1)  I ↦ [↺ (ITR % K2) [V ↦ S ↦ A] ] ↦ (|R|)^o ]
5 end
6 solver SOLVER_sender implements as_golomb_sender
7     computation : I_sort, V_sort, S_first, A_AI, R_tabu
```

Algorithm 36: Receiver solver for *GRP*

```
1 abstract solver as_golomb_receiver                  // ITR → number of iterations
2 computation : I, V, S, A, R
3 connection : C.M.
4 begin
5     [↺ (ITR < K1)  [C.M. ↦ I] ↦ [↺ (ITR % K2) [V ↦ S ↦ A] ] ↦ R ]
6 end
7 solver SOLVER_receiver implements as_golomb_receiver
8     computation : I_sort, V_sort, S_first, A_AI, R_tabu
9     communication: CM_last
```

In this communication strategy there are some parameters to be tuned. The first ones are: 1. K_1, the number of restarts, and 2. K_2, the number of iterations by restart. Both are instance dependent, so, after many experimental runs, I choose them as follows:

- *Golomb Ruler* 8–34: $K_1 = 300$ and $K_2 = 200$

- *Golomb Ruler* 10–55: $K_1 = 1000$ and $K_2 = 1500$
- *Golomb Ruler* 11-72: $K_1 = 1000$ and $K_2 = 3000$

The idea of this strategy (abstract solver) follows the following steps:

Step 1

The computation module I_{sort} generates an initial configuration tacking into account a set of configurations into a tabu list. The configuration arriving to this tabu list come from the same solver (Step 3) and/or from outside (other solvers) depending on the strategy (non-communicating or communicating), and on the type of the solver (sender or receiver).

This module executes some other modules provided by POSL to solve the *Sub-Sum Problem* in order to generates *valid* configurations for *Golomb Ruler Problem*. A valid configuration s for *Golomb Ruler Problem* is a configuration that fulfills the following constraints:

- $s = (a_1, \ldots, a_o)$ where $a_i < a_j, \forall i < j$, and
- all $d_i = a_{i+1} - a_i$ are all different, for all $d_i, i \in [1...o-1]$

The *Sub-sum Problem* is defined as follows: Given a set E of integers, with $|E| = N$, finding a sub set e of n elements that sums exactly z. In that sense, a solution $S_{sub-sum} = \{s_1, \ldots, s_{o-1}\}$ of the *Sub-sum problem* with $E = \left\{1, \ldots, l - \frac{(o-2)(o-1)}{2}\right\}$, $n = o - 1$ and $z = l$, can be traduced to a *valid* configuration C_{grp} for *Golomb Ruler Problem* as follows:

$$C_{grp} = \{c_1, c_1 + s_1, \ldots, c_{o-1} + s_{o-1}\}$$

where $c_1 = 0$.

In the selection module applied inside the module I, the selection step of the search process selects a configuration from the neighborhood *far* from the *tabu* configurations, with respect to certain vectorial norm and an epsilon. In other words, a configuration C is selected if and only if:

1. the cost of the configuration C is lower than the current cost, and
2. $||C - C_t||_p > \varepsilon$, for all *tabu* configuration C_t

where p and ε are parameters.

I experimented with 3 different values for p. Each value defines a different type of norm of a vector $x = \{x_1, \ldots, x_n\}$:

- $p = 1$: $||x||_1 = \sum_{i=0}^{n} |x_i|$
- $p = 2$: $||x||_2 = \sqrt{\sum_{i=0}^{n} |x_i|^2}$
- $p = \infty$: $||x||_\infty = \max(x)$

After many experimental runs with these values I choose $p = \infty$ and $\varepsilon = 4$ for the study of the communication strategy. I also made experiments trying to find the right size for the $tabu$ list and the conclusion was that the right sizes were 15 for non-communicating strategies and 40 for communicating strategies, taking into account that in the latter, I work with 20 receivers solvers.

Step 2

After generating the first configuration, the next step is to apply a local search to improve it. In this step I use the neighborhood computation module V_{sort}, that creates neighborhood $\mathcal{V}(s)$ by changing one value while keeping the order in the configuration, and the other modules (selection and acceptance). The local search is executed a number K_2 of times, or until a solution is obtained.

Step 3

If no improvement is reached, the current configuration is classified as a *potential local minimum* and inserted into the $tabu$ list, then send it (on the case of sender solvers). Then, the process returns to the Step 1.

The POSL code of the communication strategy using the one to N operator is presented in Algorithm 37.

Algorithm 37: Communication strategy one to N for *GRP*

```
1 [SOLVER_sender · R(20)] ⇝ [SOLVER_receiver · C.M.(20)] ;
```

When we use communication one to one, after k restarts the receiver solver has $2k$ configurations inside its tabu list: its own tabu configurations and the received ones. Table 2.18 shows that this strategy is not sufficient for some instances, but when we use communication one to N, the number of tabu configurations after k restarts in the receiver solver is considerably higher: $k(N+1)$: its own tabu configurations and the ones received from N sender solvers the receiver solver is connected with. Hence, these solvers can generate configurations far enough from many potentially local minima. This phenomenon is more visible when the problem order o increases. Table 2.19 shows that the improvement for the higher case (11-72) is about 29% w.r.t non communicating solvers (Table 2.17).

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

Table 2.18: *Golomb Ruler*: parallel, communication one to one

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

Table 2.19: *Golomb Ruler*: parallel, communication one to N

2.5 Summarizing

In this chapter various Constraint Satisfaction Problems as benchmarks have been chosen to 1. evaluate the POSL behavior solving these kind of problems, and 2. to study different solution strategies, specially communication strategies. To this end, benchmarks with different characteristics have been selected, to help me having a wide view of the POSL behavior.

In the solution process of *Social Golfers Problem*, it was studied an exploitation-oriented communication strategy, in which the current configuration is communicated to ask other solvers for help to concentrate the effort in a more promising area. Results show that this communication strategy can provide some gain in terms of runtime. It was also presented results showing the success of a cost descending acceleration communication strategy, exchanging the current configuration between two solvers with different characteristics. Some other unsuccessful communication strategies were studied, showing that the sub-division of the effort by weeks, do not work well. Table 2.20 summarizes the obtained results solving *SGP*.

Instance	Sequential		Parallel		Cooperation	
	T	It.	T	It.	T	It.
5–3–7	1.25	2,907	0.23	142	**0.08**	139
8–4–7	0.60	338	0.28	93	**0.14**	100
9–4–8	1.04	346	0.60	139	**0.36**	144

Table 2.20: Summarizing results for *SGP*

It was showed that simple communication strategies as they were applied to solve *Social Golfers Problem* does not improve enough the results without communication for the *N-Queens Problem*. In this problem, the number of solution with respect to the order N increase

exponentially, then higher order instances are "easer" to solve using local search. For that reason, the communication can not provide a lot on gain. However, a deep study of the POSL's behavior during the search process allowed to design a communication strategy able to improve the results obtained using non-communicating strategies for small instances. Table 2.21 summarizes the obtained results solving *NQP*.

Instance	Sequential		Parallel		Cooperation	
	T	It.	T	It.	T	It.
250	0.29	8,898	0.19	4,139	**0.09**	1,169
500	0.35	4,203	0.24	2,675	**0.14**	864
1000	0.35	2,766	0.30	2,102	**0.21**	807
3000	1.50	2,191	1.33	2,168	**1.02**	1,613
6000	4.71	3,339	4.57	3,323	**4.24**	2,537

Table 2.21: Summarizing results for *NQP*

The *Costas Array Problem* is a very complicated constrained problem, and very sensitive to the methods to solve it. For that reason I used some ideas from already existing algorithms. However, thanks to some studies of different communication strategies, based on the communication of the current configuration at different times (places) in the algorithm, it was possible to find a communication strategy to improve the performance. Table 2.22 summarizes the obtained results solving *CAP*.

STRATEGY	T	It.	% success
Sequential	132.73	2,332,088	40.00
Parallel	25.51	231,262	100.00
Cooperative Strategy	**10.83**	79,551	100.00

Table 2.22: Summarizing results for *CAP* 19

During the solution process of the *Golomb Ruler Problem*, POSL needs to perform many restarts. For that reason, this problem was chosen to study a different (and innovative up to my knowledge) communication strategy, in which the communicated information is a potential local minimum to be avoided. This new communication strategy showed to be effective to solve these kind of problems. Table 2.23 summarizes the obtained results solving *GRP*.

Instance	Sequential				Parallel			Cooperation		
	T	It.	R	% success	T	It.	R	T	It.	R
8–34	0.66	10,745	53	100.00	0.43	349	1	**0.43**	283	1
10–55	67.89	446,913	297	88.00	4.92	20,504	13	**3.16**	12,605	8
11–72	117.49	382,617	127	30.00	85.02	155,251	51	**60.35**	110,311	36

Table 2.23: Summarizing results for *GRP*

In all cases, thanks to the operator-based language provided by POSL it was possible to test many different strategies (communicating and non-communicating) fast and easily. Whereas

creating solvers implementing different solution strategies can be complex and tedious, POSL gives the possibility to make communicating and non-communicating solver prototypes and to evaluate them with few efforts. In this chapter it was possible to show that a good selection and management of inter-solvers communication can help to the search process, working with complex constrained problems.

3

Bibliography

[1] 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.

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

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

[10] 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.

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

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