POSL: A Parallel-Oriented Solver Language

Thesis for the Degree of
Doctor of Computer Science

Alejandro Reyes Amaro

Doctoral School STIM

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

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

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

Submitted: dd/mm/2016

Assessment committee:

Prof. (1)

Institution (1)

Prof. (2)

Institution (2)

Prof. (3)

Institution (3)

ISBN ??

Contents

Part I

Presentation

Part II

Study and evaluation of POSL

1

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 [125] were we used POSL to solve the *Social Golfers Problem* and study some communication strategies. It was the first version of POSL, therefore it was able to solve only relatively easy instances. However, the efficacy of the communication was showed using this tool.

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.

1.1 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 in 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

1.1.1 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 [126]. 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.

1.1.2 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 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 modules V:

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

   V_{AS}: 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 1: Simple solvers for *SGP*

```
abstract solver as_simple                                   // ITR → number of iterations
computation : I, V, S, A
begin
    [↺ (ITR < K1) I ↦ [↺ (ITR % K2) [V ↦ S ↦ A] ] ]
end
solver S_std implements as_simple
    computation : I_BP, V_std, S_best, A_AI
solver S_as implements as_simple
    computation : I_BP, V_AS, S_best, A_AI
```

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

3. Selection 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 a *decision–pair*

   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 a *decision–pair*

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

4. Acceptance module A:

   A_{AI}: Given a decision–pair, returns the configuration marked as *"found"*

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_{AS}; and a combination of V_{std} and V_{AS} by applying the operator (ρ), already introduced in the previous chapter. Algorithms 1 and 2 present solvers for each case, respectively.

Algorithm 2: Solvers combining neighborhood functions using operator *RHO*

```
abstract solver as_rho                                      // ITR → number of iterations
computation : I, V1, V2, S, A
begin
    [↺ (ITR < K1) I ↦ [↺ (ITR % K2) [[V1 ρ V2] ↦ S ↦ A] ] ]
end
solver S_rho implements as_rho
    computation : I_BP, V_std, V_AS, S_best, A_AI
```

Solver	T	T(sd)	It.	It.(sd)
`S_as`	**1.06**	0.79	352	268
`S_rho`	41.53	26.00	147	72
`S_std`	87.90	41.96	146	58

Table 1.1: *Social Golfers*: Instance 10–10–3 in parallel

Results in Table 1.1 are not surprising. The neighborhood module V_{AS} is based on the *Adaptive Search* algorithm, which has shown very good results [5]. It selects the most culprit variable (i.e., a player), that is, the variable to most responsible for constraints violation. Then, it permutes this variable value with the value of each other variable, in all groups and all weeks. Each permutation gives a neighbor of the current configuration. V_{Std} uses no additional information, so it performs every possible swap between two players in different groups, every week. It means that this neighborhood is $g \times p$ times bigger than the previous one, with g the number of groups and p the number of players per group. 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 solvers with different combinations of these modules, using the $\binom{\rho}{}$ and the $\binom{\cup}{}$ operators. The $\binom{\rho}{}$ operator executes its first or second parameter depending on a given probability ρ, and the $\binom{\cup}{}$ operator returns the union of its parameters output. All these combinations spent more time searching the best configuration among the neighborhood, although with a lower number of iterations than V_{AS}. The V_{AS} neighborhood function being clearly faster, we have chosen it for our experiments, even if it shown a more spread standard deviation: 0.75 for AS versus 0.62 for Std, considering the ratio $\frac{T(sd)}{T}$.

***

With the selected neighborhood function, I have focused the experiment on choosing the best *selection* function. 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 Algorithm 3 have been created, using V_{AS} 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 not only spent more time searching a better configuration, but also is more sensitive to become trapped into local minima. The second computation module S_{first} selects the first configuration inside the neighborhood improving

Instance	Best improvement				First improvement			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
5–3–7	4.99	4.43	4,421	3,938	**0.23**	0.14	142	67
8–4–7	5.10	1.77	954	334	**0.28**	0.07	93	13
9–4–8	12.37	5.40	1,342	591	**0.60**	0.16	139	18

Table 1.2: *Social Golfers*: comparing selection functions in parallel

the current cost. Using this module, solvers favor exploration over intensification and of course spend clearly less time computing the neighborhood.

Algorithm 3: Solver for *SGP* to scape from local minima

abstract solver *as_eager* // ITR → number of iterations

computation : I, V, S_1, S_2, A

begin

$[\circlearrowleft (\text{ITR} < K_1)\ I \ (\mapsto)\ [\circlearrowleft (\text{ITR}\ \%\ K_2)\ [V\ (\mapsto)\ [S_1\ (?)_{\text{SCI}<K_3}\ S_2]\ (\mapsto)\ A]\ ]\ ]$

end

solver S_eager_best **implements** as_eager

computation : $I_{BP}, V_{std}, V_{AS}, S_{best}, S_{rand}, A_{AI}$

solver S_eager_fisrt **implements** as_eager

computation : $I_{BP}, V_{std}, V_{AS}, S_{first}, S_{rand}, A_{AI}$

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 1.3: *Social Golfers*: a single sequential solver using first improvement

Tables 1.2 and 1.3 present results of this experiment, showing that an local exploration-oriented strategy is better for the *SGP*. If we compare results of Tables 1.2 1.3 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_{AS}) 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 were the shared information is the current configuration. Algorithms 4 and 5 show

that the communication is performed while applying the acceptance criterion of the new configuration for the next iteration. Here, solvers receive a configuration from a sender solver, and match it with their current configuration. Then solvers select the configuration with the lowest global cost. 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).

Algorithm 4: Communicating abstract solver for *SGP* (sender)

```
abstract solver as_eager_sender                    // Itr → number of iterations
computation : I, V, S1, S2, A                      // Sci → number of iterations with the same cost
begin
    [↻ (Itr < K1)  I ↦ [↻ (Itr % K2) [V ↦ [S1 ?_{Sci<K3} S2] ↦ (|A|)^o] ] ]
end
solver S_eager_sender implements as_eager_sender
    computation : I_BP, V_AS, S_first, S_rand, A_AI
```

Algorithm 5: Communicating abstract solver for *SGP* (receiver)

```
abstract solver as_eager_receiver                  // Itr → number of iterations
computation : I, V, S1, S2, A                      // Sci → number of iterations with the same cost
communication : C.M.
begin
    [↻ (Itr < K1)
        I ↦ [↻ (Itr % K2)  V ↦ [S1 ?_{Sci<K3} S2] ↦ [A (m) C.M.] ]
    ]
end
solver S_eager_receiver implements as_eager_receiver
    computation : I_BP, V_AS, S_first, S_rand, A_AI
    communication : CM_last
```

In Algorithm 5, 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.

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.

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

Table 1.4: *Social Golfers*: test with 100% of communication

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

Table 1.5: *Social Golfers*: test with 50 % of communication

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.

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 $K_3 = 5$.

Correr los experimentos otra vez

1.1.3 Analysis of results

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

Each time a POSL meta-solver is launched, many independent search solvers are executed. We call "good" configuration a configuration with the lowest cost within the current con-

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

Table 1.6: *Social Golfers*: test with 25% of communication

figuration 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. I avoid this phenomenon through a simple (but effective) play: if a solver is not able to find a better configuration inside the neighborhood (executing S_{First}), it selects a random one at the next iteration (executing S_{Rand}). This strategy, using communication between solvers, produces some gain in terms of runtime (Table 1.2 with respect to Tables 1.4, 1.5 and 1.6. The percentage of the receiver solvers that were able to find the solution before the others did, was significant (see Appendix **??**). That shows that the communication played an important role during the search, despite inter–process communication's overheads (reception, information interpretation, making decisions, etc). 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. For this problem we have reduced the spread in the running times and iterations of the results for the two last instances (9–4–8 and 11–7–5) applying the communication strategy (0.71 without communication versus 0.44 with communication, for 9–4–8, and 0.31 without communication versus 0.20 with communication for 11–7–5).

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 1.1a).

   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 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. A solver 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 Figure1.1b). 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 to the receiver solver.

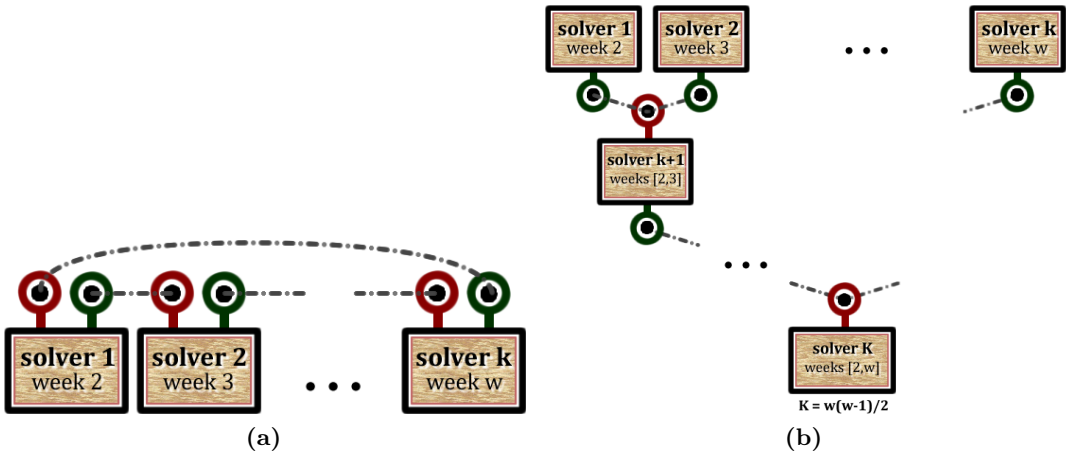

Figure 1.1: Unsuccessful communication strategies to solve *SGP*

1.2 Solving the *N-Queens Problem*

In this section I present the performed study using *N-Queens Problem* (*NQP*) as a benchmark.

1.2.1 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 [127].

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 [128]. 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].

1.2.2 Experiment design Nr. 1

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

1. Neighborhood module:

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

Fors this problem I used a simple abstract solver showing good results with no communication, based on the idea introduced in the section 1.1, using the computation module S_{rand} to scape from local minima. The abstract solver is presented in Algorithm 6.

Algorithm 6: Abstract solver for *NQP*

abstract solver *as_eager* // ITR → number of iterations
computation : I, V, S_1, S_2, A // SCI → number of iterations with the same cost
begin
$I \ (\mapsto) \ [\circlearrowleft (\text{ITR} < K_1) \ V \ (\mapsto) \ [S_1 \ (?)_{\text{SCI}<K_2} \ S_2] \ (\mapsto) \ A \ ]$
end

Using solvers implementing this abstract solver we create communicating solvers to compare their performance with the non communicating strategies. The shared information is the current configuration. Algorithms 7 and 8 show that the communication is performed while selecting a new configuration for the next iteration. We design different communication strategies. Either I execute a full connected solvers set, or a tuned combination of connected and unconnected solvers. Between connected solvers, I have applied two different connections

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

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

Algorithm 7: Abstract solver for *NQP* (sender)

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

Algorithm 8: Abstract solver for *NQP* (receiver)

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

1.2.3 Results analysis of experiment Nr. 1

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

Instance	Sequential (1 core)				No Comm. (40 cores)			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
2000	6.20	0.12	947	21	6.15	0.20	952	20
3000	14.19	0.21	1,415	22	14.06	0.33	1,413	25
4000	25.63	0.36	1,900	28	25.46	0.51	1,898	34
5000	41.37	0.44	2,367	26	40.57	0.91	2,377	32
6000	60.42	0.52	2,837	31	60.10	0.70	2,849	43

Table 1.7: Results for *NQP* (no communication)

POSL, as it can be seen in Tables 1.8 and 1.9, works very well without communication, for instances relatively big. This confirms once again the success of the computation module V_{AS} based on *Adaptive Search* algorithm to solve these kind of problems. That is the reason why the parallel approach does not outperforms significantly the sequential one, as we can

Instance	25% Comm.				50% Comm.				All Comm.			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
2000	6.05	0.25	934	36	6.01	0.19	920	41	5.92	0.17	885	49
3000	13.89	0.28	1,387	48	13.91	0.30	1,368	51	13.67	0.39	1,346	40
4000	25.26	0.63	1,868	43	25.14	0.50	1,855	50	25.11	0.39	1,834	58
5000	40.38	0.93	2,338	71	40.33	0.66	2,312	69	39.62	1.07	2,287	44
6000	59.28	1.34	2,794	78	58.97	1.19	2,775	67	58.97	1.38	2,729	78

Table 1.8: Results for *NQP* (40 cores, communication 1 to 1)

Instance	25% Comm.				50% Comm.				All Comm.			
	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)	T	T(sd)	It.	It.(sd)
2000	6.07	0.15	925	41	5.98	0.19	915	41	6.01	0.19	887	57
3000	13.97	0.34	1,402	49	13.96	0.31	1,386	52	13.79	0.32	1,365	65
4000	25.30	0.57	1,867	52	25.29	0.42	1,851	66	25.17	0.47	1,838	65
5000	40.45	0.80	2,338	80	40.37	0.56	2,312	56	39.88	0.71	2,291	51
6000	59.77	1.50	2,824	49	59.53	0.98	2,773	69	59.16	1.37	2,773	57

Table 1.9: Results for *NQP* (40 cores, communication 1 to N)

see in Table 1.7. However, 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.

We can see the improvement with respect to the percentage of communicating solvers in Figure 1.2. The bigger the instance is, the more significant the observed improvement is. This phenomenon suggests that a deeper study and an efficient implementation can make the communication playing a more significant role in the solution process. For that reason, I decided to design another experiment to try to improve the results using communicating strategies using POSL.

1.2.4 Experiment design Nr. 2

The strategy of work sub-division proposed in the previews section with *Social Golfers Problem* seemed interesting to me and I did not want to give up on it. So I tried to apply it to *N-Queens Problem*.

In some experimental runs, I launched some *partial* solvers (i.e., solvers only performing permutations between variables into certain range), together with a *full* solver (i.e., a solver working with the entire configuration). I used the instance *4000-queens* as test and I built the following solvers:

1. Solver S_1 only permuting the first 1000 variables
2. Solver S_2 only permuting the first 2000 variables

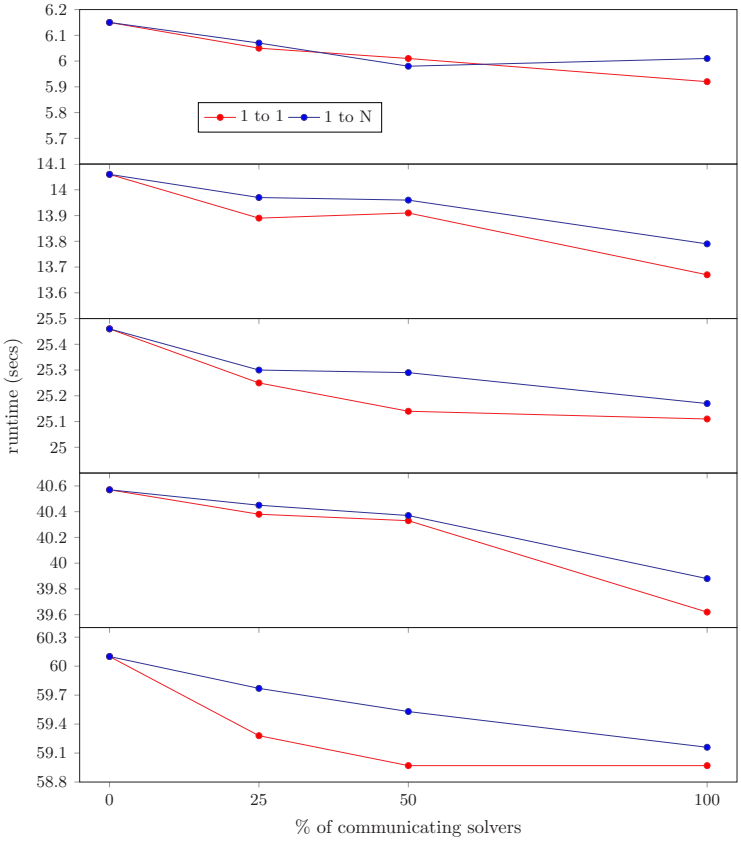

Figure 1.2: Runtime means of instances 2000-, 3000-, 4000-, 5000- and 6000-queens

3. Solver S_3 only permuting the first 3000 variables

4. Solver S_4 a *full* solver.

Obviously, the first three solvers were not able to find a solution to the problem, but at the beginning of runs, it was possible to observe that these solvers were able to obtain configurations with costs considerably lower with respect to the *full* solver S_4. For that reason I put in practice the idea of connecting *partial* solvers together with *full* solvers. This way, the search process can be accelerated at the beginning.

Before designing the solution strategy (abstract solver) many experiments were launched to select: 1. The number of sub-divisions of the configuration, i.e., how many *partial* solvers works in different sections of the configuration. They are connected to the *full* solver. 2. The size of the section where the *partial* solvers work in.

After many runs of these experiments, it was decided to work with two *partial* sender solvers (implementing the abstract solver in Algorithm 9). In this algorithm a and b are parameters of the module $V_{[a,b]}$ used in the *partial* solvers. They represent the variables defining the range of the configuration where the *partial* solver works. They were chosen such that $b - a = \frac{n}{4}$

These solvers send their configurations to the *full* solver that implements the abstract solver in Algorithm 10.

Algorithm 9: Abstract solver for *NQP* (partial solver sender)

abstract solver *as_partial_sender* // ITR → number of iterations
computation : $I, V_{[a,b]}, S, A$ // SCI → number of iterations with the same cost
begin
$[\circlearrowleft (\text{ITR} < K_1)$
$I \ (\mapsto) \ [\circlearrowleft (\text{ITR} \ \% \ K_2 || \ \text{SCI} < K_3) \ \left[V_{[a,b]} \ (\mapsto) \ S \ (\mapsto) \ \left[(\!|A|\!)^o \ (?)_{\text{ITR}\%K_4} \ A\right]\right] \ ]$
$]$
end

Algorithm 10: Abstract solver for *NQP* (full solver receiver)

abstract solver *as_full_receiver* // ITR → number of iterations
computation : I, V, S, A // SCI → number of iterations with the same cost
communication : $C.M.$
begin
$I \ (\mapsto)$
$[\circlearrowleft (\text{ITR} < K_1)$
$\left[V \ (\mapsto) \ S \ (\mapsto) \ \left[A \ (?)_{\text{ITR}\%K_2} \ \left[A \ (m) \ C.M.\right]\right]\right]$
$]$
end

1.2.5 Results analysis of experiment Nr. 2

Results in Table 1.10 show that this strategy is effective to solve the *N-Queens Problem* improving the runtimes already obtained in the previews experiment. In the resolution of this problem, the improvement rate of the current configuration cost is very slow (yet stable). The *partial* solvers work only on a section of the configuration, and for that reason, they are able to obtain configuration with costs considerably lower than the obtained by the *full* solver more quickly. This characteristic is taken into account: *partial* solvers send their obtained configurations to the *full* solvers. By doing this, the improvement rate of the current configuration can be accelerated at the beginning of the search.

Instance	T	T(sd)	It.	It.(sd)
2000	**5.11**	0.83	**841**	37
3000	**11.55**	1.96	**1,275**	67
4000	**21.27**	3.76	**1,656**	108
5000	**34.77**	4.99	**2,082**	108
6000	**51.72**	5.73	**2,501**	176

Table 1.10: Results for *NQP* (40 cores, communication partial-full solvers)

1.3 Solving the *Costas Array Problem*

In this section I present the performed study using *Costas Array Problem* (*CAP*) as a benchmark.

1.3.1 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 [129].

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

1.3.2 Experiment design

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 rights parameter 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* module (R).

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

The basic solver I use to solve this problem is presented in Algorithm 11, and I take it as a base to build all the different communication strategies. Basically, it is a classical local search iteration, where instead of performing restarts, it performs resets. After a deep analysis of this implementation and results of some runs, I decided to use $K_1 = 24000$ (maximum number of iterations) big enough to solve the chosen instance $n = 17$; and $K_2 = 3$ (the number of iteration before performing the next *reset*).

Algorithm 11: Reset-based abstract solver for *CAP*

```
abstract solver as_hard                                  // Itr → number of iterations
computation : I, R, V, S, A
begin
    I (↦)
    [↺ (Itr < K1)
        R (↦) [↺ (Itr % K2) [V (↦) S (↦) A] ]
    ]
end
```

The abstract solver for the sender solver is presented in Algorithm 12. Like for the *Social Golfers Problem*, we design different communication strategies combining different percentages of communicating solvers and our two communication operators (*1 to 1* and *1 to N*). However for this problem, we study the behavior of the communication performed at two different moments: while applying the acceptance criteria (Algorithm 13), and while performing a

reset (Algorithms 13, 14 and 15).

Algorithm 12: Reset-based abstract solver for *CAP* (sender)

abstract solver *as_hard_sender* // ITR → number of iterations
computation : I, R, V, S, A
begin
  $I \ (\mapsto)$
  $[\circlearrowleft (\text{ITR} < K_1)$
    $R \ (\mapsto) \ [\circlearrowleft (\text{ITR} \ \% \ K_2) \ \left[V \ (\mapsto) \ S \ (\mapsto) \ (\!|A|\!)^o\right] \ ]$
  $]$
end

Algorithm 13: Reset-based abstract solver for *CAP* (receiver, variant A)

abstract solver *as_hard_receiver_a* // ITR → number of iterations
computation : I, R, V, S, A
communication : $C.M.$
begin
  $I \ (\mapsto)$
  $[\circlearrowleft (\text{ITR} < K_1)$
    $R \ (\mapsto) \ [\circlearrowleft (\text{ITR} \ \% \ K_2) \ \left[V \ (\mapsto) \ S \ (\mapsto) \ \left[A \ (m) \ C.M.\right]\right] \ ]$
  $]$
end

Algorithm 14: Reset-based abstract solver for *CAP* (receiver, variant B)

abstract solver *as_hard_receiver_b* // ITR → number of iterations
computation : I, R, V, S, A // SCI → number of iterations with the same cost
communication : $C.M.$
begin
  $I \ (\mapsto)$
  $[\circlearrowleft (\text{ITR} < K_1)$
    $\left[R \ (?)_{\text{SCI}\%K_3} \ \left[R \ (m) \ C.M.\right]\right] \ (\mapsto) \ [\circlearrowleft (\text{ITR} \ \% \ K_2) \ \left[V \ (\mapsto) \ S \ (\mapsto) \ A\right] \ ]$
  $]$
end

Algorithm 15: Reset-based abstract solver for *CAP* (receiver, variant C)

abstract solver *as_hard_receiver_c* // ITR → number of iterations
computation : I, R, V, S, A
communication : $C.M.$
begin
  $I \ (\mapsto)$
  $[\circlearrowleft (\text{ITR} < K_1)$
    $\left[R \ (m) \ C.M.\right] \ (\mapsto) \ [\circlearrowleft (\text{ITR} \ \% \ K_2) \ \left[V \ (\mapsto) \ S \ (\mapsto) \ A\right] \ ]$
  $]$
end

1.3.3 Analysis of results

I present in Table 1.11 results of launching *solver sets* to solve each instance of *Costas Array Problem* sequentially. Runtimes and iteration means showed in this table are bigger than those presented in Table 1.12, confirming once again the success of the parallel approach.

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

Table 1.11: *Costas Array* 17: no communication

I chose directly the neighborhood module (V_{AS}), the selection module (S_{First}) and the acceptance module A, to create solvers. I ran experiments to study parallel communicating strategies taken into account the structure of the communication, and the communication operator used, but in this problem, I perform the communication at two different times: at the time of applying the acceptance criteria, and at the time of performing the *reset*.

Table 1.12 shows that the abstract solver A (receiving the configuration at the time of applying the acceptance criteria) is more effective. The reason is that the others, interfere with the proper performance of the *reset*, that 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 sabe 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

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

Table 1.12: *Costas Array* 17: with communication

sender solvers apply different *reset* in the same configuration, and results showed the efficacy of this communication strategy.

Table 1.12 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.

1.4 Solving the *Golomb Ruler Problem*

In this section I present the performed study using *Golomb Ruler Problem* (*GRP*) as a benchmark.

1.4.1 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 [130]. When I apply POSL to solve an instance of this problem sequentially, I can notice that it performs many *restarts* before finding a solution. For that reason I have chosen this problem 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). I also store all distances where a variable is participating. This information is usefull to compute the more culprit variable (the variable that interfiers less in the representd measures), in case of the user wants to apply algorithms like *Adaptive Search*. This cost is calculated in $O\left(o^2 + l\right)$.

1.4.2 Experiment design

I use *Golomb Ruler Problem* instances to study a different communication strategy. This time I communicate the current configuration, to avoid its neighborhood, i.e., a *tabu* configuration. I have reused some modules used in the resolution of *Social Golfers* and *Costas Array*

problems to design the solvers: the *Selection* and *Acceptance* modules. The new modules are:

1. Generation module:

   I: Generates a random configuration s, respecting the structure of the problem, i.e., the configuration is an ordered vector of integers. This module takes into account a set of *tabu* configurations arrived from the same solver, and also via solver-communication through a communication module $C.M.$ that receives a set of configurations. This module constructs the new configuration far enough from the *tabu* configurations.

2. Neighborhood module:

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

I also add a module to insert a configuration into a *tabu* list inside the solver. In Algorithm 16 the abstract solver used to send information (sender abstract solver) is presented. When the module T is executed, the solver is unable to find a better configuration around the current one, so it is assumed to be a local minimum, and it is sent to the receiver solver. Algorithm 17 presents an abstract solver used to receive information (receiver abstract solver). Based on the connection operator used in the communication strategy, this solver might receives one or many configurations. These configurations are the input of the generation module (I), and this module inserts all received configurations into a *tabu* list, and then generates a new first configuration, far from all configurations in the *tabu* list.

Algorithm 16: Abstract solver for *GRP* (sender)

```
abstract solver as_golomb_sender                    // Itr → number of iterations
computation : I, V, S, A, T
begin
    [↻ (Itr < K1)
        I ↦ [↻ (Itr % K2) [V ↦ S ↦ A] ] ↦ (|T|)^o
    ]
end
```

Algorithm 17: Abstract solver for *GRP* (receiver)

```
abstract solver as_golomb_receiver                  // Itr → number of iterations
computation : I, V, S, A, T
connection : C.M.
begin
    [↻ (Itr < K1)
        [C.M. ↦ I] ↦ [↻ (Itr % K2) [V ↦ S ↦ A] ] ↦ (|T|)^o
    ]
end
```

In this communcation 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 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) or from outside (other solvers) depending on the strategy (non-communicating or communicating).

This module applies 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 communication strategy study. 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, 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, the process returns to the Step 1.

1.4.3 Analysis of results

The benefit of the parallel approach is also proved for the *Golomb Ruler Problem* (see Table 1.13 with respect to 1.14, 1.15, 1.16 and 1.17). But 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.

The first experiment compares the runs of non communitaing solvers not using a *tabu* list with non communicating solvers using a *tabu* list. The results in Tables 1.14 and 1.15

demonstrate that using a *tabu* list can help the search process. Without communication, the improvement is not substantial (8% for 8–34, 7% for 10–55 and 5% for 11–72). The reason is because only one configuration is inserted in the *tabu* list after each restart. When we use *1 to 1* communication, after the restart k, the receiving solver has twice the number of configurations in the *tabu* list (one *tabu* configuration from itself and the received one after each restart). Table 1.16 shows that this strategy is not sufficient for some instances, but when we use *1 to N* communication, the number of *tabu* configurations after the restart k, in the receiving solver is considerably higher, e.g., after the restart k a receiving solver has $k(N+1)$ configurations in his *tabu* list (one *tabu* configuration from itself and N received from the other solvers, each restart). Hence, these solvers can generate configurations far enough from many potentially local minima. This phenomenon is more visible when the problem order increases. Table 1.17 shows that the improvement for the higher case (11-72) is about 32% with respect to non communicating solvers without using a *tabu* list (Table 1.14), and about 29% with respect to non communicating solvers using a *tabu* list (Table 1.15).

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

Table 1.13: *Golomb Ruler*: a single sequential solver

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

Table 1.14: *Golomb Ruler*: parallel, without tabu list.

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

Table 1.15: *Golomb Ruler*: parallel, with tabu list.

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

Table 1.16: *Golomb Ruler*: parallel, communication 1 to 1.

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

Table 1.17: *Golomb Ruler*: parallel, communication 1 to n.

1.5 Summarizing

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

In the solution process of *Social Golfers Problem*, it was showed the success of an exploitation-oriented communication strategy, in which the current configuration is communicated to ask other solvers for help, concentrating the effort in a more promising area. I was able also to study some other unsuccessful strategies, to show that strategies based on the sub-division of the effort by weeks, is not a good idea.

It was showed that simple communication strategies as they applied to solve *Social Golfers Problem* does not improve enough the results without communication for the *N-Queens Problem*. However, a deep study of the POSL's behavior during the search process allows to design a communication strategy able to improve the results obtained using non-communicating strategies.

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 existent algorithms. However, thanks to some studies of different communication strategies, based on the configuration of the current communication at different times (places) in the algorithm, it was possible to find a communication strategy to improve the performance.

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 tu 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.

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 largely help to the search process, working with complex constrained problems.

2

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[126] 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.

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

[128] 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.

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

[130] 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.

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