Sequential Time Splitting and Bounds Communication for a Portfolio of Optimization Solvers

Roberto Amadini[1] and Peter J. Stuckey[2]

[1] Dept. of Computer Science and Engineering/Lab. Focus INRIA,
University of Bologna, Italy.
[2] National ICT Australia
Department of Computing and Information Systems
University of Melbourne 3010., Australia

Abstract. Scheduling a subset of solvers belonging to a given portfolio has proven to be a good strategy when solving Constraint Satisfaction Problems (CSPs). In this paper, we show that this approach can also be effective for Constraint Optimization Problems (COPs). Unlike CSPs, sequential execution of optimization solvers can communicate information in the form of bounds to improve the performance of the following solvers. We provide a hybrid and flexible portfolio approach that combines static and dynamic time splitting for solving a given COP. Empirical evaluations show the approach is promising and sometimes even able to outperform the best solver of the porfolio.

1 Introduction and Related Work

One of the main uses of Constraint Programming (CP) is to model and solve *Constraint Satisfaction Problems* (CSP) [19]. Solving CSPs is hard, and there are plenty of approaches that can be used to tackle them. One of the more recent trend in this research area—especially in the SAT field—is trying to solve a given problem by using a portfolio approach [12, 23].

An *algorithm portfolio* is a general methodology that exploits a number of different algorithms in order to get an overall better algorithm. A portfolio of CP solvers can therefore be seen as a particular solver, the *portfolio solver*, that exploits a collection of $m > 1$ different constituent solvers $s_1, \ldots, s_m$ in order to obtain a globally better CP solver. When a new unseen instance i arrives, the portfolio solver tries to predict which are the best constituent solvers $s_1, \ldots, s_k$ ($k \leq m$) for solving i and then runs such solver(s) on i. This solver selection process is clearly a fundamental part for the success of the approach and it is usually performed by exploiting Machine Learning techniques.

There has been a significant body of work in using portfolios to leverage and combine a number of different solvers in order to get an overall better solver [15,18]. A particular case of portfolio approach consists in *scheduling* (even in parallel) a subset of the constituent solvers within a certain time window (see

for instance [3, 7, 14, 16, 22, 24]. This would seem to be a winning strategy, in particular due to the fact that often the solving time of a satisfaction problem is either relatively short or very long. It hence naturally handles the heavy tailed nature of solving.

Surprisingly, most of the focus of algorithm portfolios has been on constraint satisfaction problems. In practice most of the combinatorial problems of interest are *Constraint Optimization Problems* (COPs), where we are interested in finding a solution which minimizes as much as possible a given objective function. As pointed out also in [25], there is a lack of use of meta-learning for algorithm selection: to the best of our knowledge, COP portfolios are mostly developed just for some specific optimization problems like Knapsack, Most Probable Explanation, Set Partitioning, Travel Salesman Problem [13, 15, 26] or for example to properly tune the parameters of a single COP solver [17, 27].

It is hence natural to ask how to create a scheduling algorithm portfolio for COPs. A crucial difference from CSPs arises because a COP solver may yield sub-optimal *partial solutions* before finding the best one (and possibly proving its optimality). This means that one solver can *transmit* useful bounds information to another if they are scheduled sequentially. This key feature is the basis of our work. Indeed, in a portfolio scenario, a partial solution found by a solver s_1 is a token of knowledge that s_1 can pass to another solver s_2 in order to prune its search space and therefore possibly improve its solving process. In this paper, we thus address the problem of boosting optimization by exploiting the sequential cooperation of different COP solvers. To do so, we will introduce the notion of solving *behaviour* for taking into account the anytime performance of the solvers.

A work related to this paper is [9], in which algorithm control techniques are used to share bounds information between the scheduled solvers without, however, explicitly rely on the solvers behaviours (as in our technical definition). In [20] the authors provide a generic approach to knowledge sharing, based on the communication of learned clauses and cuts information, which is suitable for sequential SAT solvers but is less likely to be useful when solvers are very disparate in nature. Finally, [4] reports an empirical evaluation of different portfolio approaches applied to COPs, without however taking into account the anytime performance of the solvers as well as the possible bounds communication between them.

Paper Structure. In Section 2 we give the technical definitions of solving behaviour and timesplit solver; then, in Section 3, we provide and evaluate `TimeSplit`: an algorithm aimed to determine the best time splitting according to already known behaviours. By exploiting the results of `TimeSplit` in Section 4 we introduce `TPS`, a generic and flexible *portfolio* approach that relies on two steps: when a new unseen problem arrives, a static solver schedule (computed off-line) is run first, while a dynamic schedule is executed then by possibly exploiting the best solution found in the first stage. In Section 5 we describe the methodology and the results achieved by `TPS`, using the SUNNY [3, 4] algorithm as a baseline for computing and evaluating different portfolio approaches. Finally, in Section 6 we conclude by providing some possible future directions.

2 Solving Behaviour and Timesplit Solvers

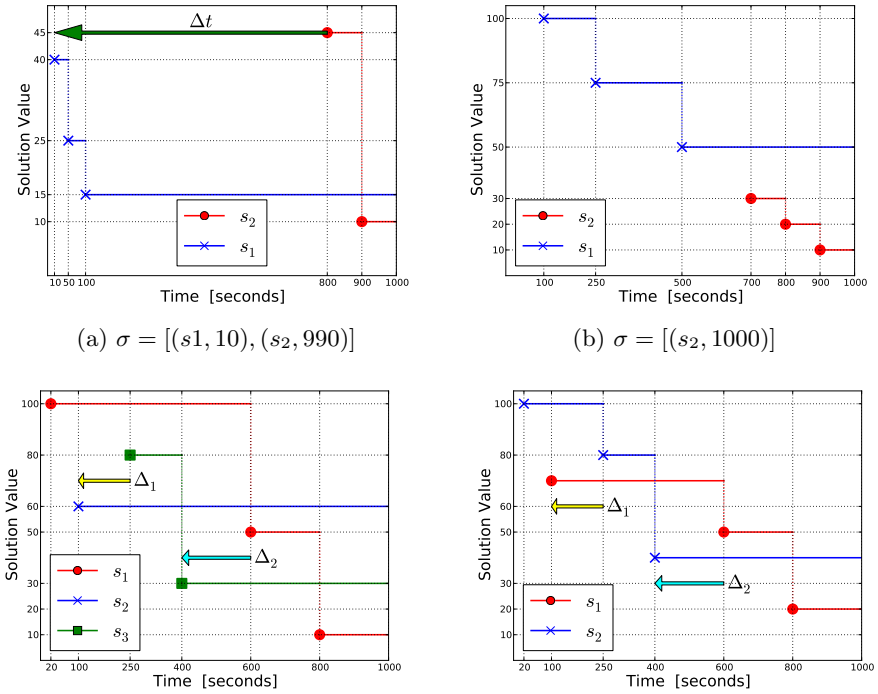
Let us fix a dataset of minimization[3] problems Δ, a universe of COP solvers Σ (which can include a particular portfolio $\Pi \subseteq \Sigma$) and a solving time window $[0, T]$. We wish to determine the best sequence of solvers in Σ to run on p and for how long to run each solver within the interval $[0, T]$ in order obtain the best result for instance p. Ideally we aim to *improve the best solver* of Σ for the instance p.

We define the *(solving) behaviour* of each solver $s \in \Sigma$ applied to a problem $p \in \Delta$ over time $[0, T]$ as a sequence of pairs $\mathcal{B}(s, p) = [(t_1, v_1), \ldots, (t_n, v_n)]$ where $t_i \in [0, T]$ is the time when s finds a solution, and v_i is the objective value of such a solution. Note that we can consider the pairs ordered so that $t_1 < \cdots < t_n$ while $v_1 > \cdots > v_n$ since we assume the solving process is *monotonic* (we can omit the non-monotonic entries if any). For example, consider the behaviours $\mathcal{B}(s_1, p) = [(10, 40), (50, 25), (100, 15)]$ and $\mathcal{B}(s_2, p) = [(800, 45), (900, 10)]$ illustrated in Figure 1a with a timeout of $T = 1000$ seconds. The best value $v^* = 10$ is found by s_2 after 900 seconds, but it takes 800 seconds to find its first solution ($v = 45$). Meanwhile, s_1 finds a better value ($v = 40$) after just 10 seconds and even better values in just 100 seconds. So, the question is: what happens if we *"inject"* the upper bound 40 from s_1 to s_2? Considering that starting from $v = 45$ the solver s_2 is able to find v^* in 100 seconds (from 800 to 900), hopefully starting from any better (or equal) value $v' \leq 45$ the time needed by s_2 to find v^* is no more than 100 seconds. Note that from a graphical point of view what we would like to do is therefore to *"shift"* the curve of s_2 towards the left from $t = 800$ to 10, by exploiting the fact that after 10 seconds s_1 can suggest to s_2 the upper bound $v = 40$. The cooperation between s_1 and s_2 would thereby reduce by $\Delta t = 790$ seconds the time needed to find v^*, and moreover would allow us to exploit the remaining Δt seconds for finding better solutions or even proving the optimality of v^*. However, note that the *virtual* behaviour may not occur: it may be that s_2 calculates important information in the first 800 seconds required to find the solution $v^* = 10$, and therefore the injection of $v = 40$ could be useless (if not harmful!).

Given a problem $p \in \Delta$ and a schedule $\sigma = [(s_1, t_1), \ldots, (s_k, t_k)]$ of k solvers we define the corresponding *timesplit solver* as a particular solver such that: *i)* first, runs s_1 on p for t_1 seconds; *ii)* then, for $i = 1, \ldots, k-1$, runs s_{i+1} on p for t_{i+1} seconds possibly exploiting the best solution found by the previous solver s_i in t_i seconds. We will use the notation $\underline{\sigma}$ to indicate the *base* of timesplit solver σ where we omit the last solver in the schedule, i.e. $\underline{\sigma} = [(s_1, t_1), \ldots, (s_{k-1}, t_{k-1})]$. Intuitively, if s_k is the last solver of the schedule, $\underline{\sigma}$ is the timesplit solver that ideally contributes to improve s_k.

As an example, in Figure 1a the ideal timesplit solver would be defined by $\sigma = [(s1, 10), (s2, 990)]$, but note that there are cases in which the timesplit solver is actually a single solver, since the best solver is not virtually improvable

[3] We can convert a maximization problem to a minimization problem by simply negating the objective function.



(a) $\sigma = [(s1, 10), (s_2, 990)]$

(b) $\sigma = [(s_2, 1000)]$

(c) $\sigma = [(s_2, 100), (s_3, 150), (s_1, 750)]$

(d) $\sigma = [(s_1, 100), (s_2, 150), (s_3, 750)]$

Fig. 1: Examples of solving behaviours and corresponding time splitting σ.

by any other: this happens when every solution found by the best solver is also the best solution found so far (e.g., see Figure 1b). Moreover, there may be also cases in which splitting the time window in more than two slots (even alternating the same solvers) may ideally lead to better performances. Indeed, the "overall" best solver at the time edge T might no longer be the best one at a previous time $t < T$. For example, in Figure 1c the best solver at time $t \geq 800$ is s_1, at time $400 \leq t < 800$ is s_3 while for $t \leq 400$ is s_2; in Figure 1d the best solver is s_1 if $t < 400$ or $t \geq 800$, while for $400 \leq t < 800$ is s_2.

3 Splitting Selection and Evaluation

Once we have informally hypothesized the potential benefits of timesplit solvers, some questions naturally arise. First, which metric(s) is reasonable to formally define the "best solver"? Furthermore, how do we split the time window between solvers for determining the (virtually) best timesplit solver? Finally, to what extent do timesplit solvers act like the virtual timesplit solvers? In order to answer these questions, we fixed some proper metrics, defined a splitting algorithm and empirically evaluated the assumptions previously introduced.

3.1 Evaluation Metrics

In order to evaluate the performances of different COP solvers (and thus formally define the notion of best solver) we examine a number of metrics for grading a solver s on a problem p over a time limit T.

Analogous to the usual metric for CSP solvers, let $\texttt{proven}(s,p) = 1$ if solver s finds and proves the optimal solution (including proving unsatisfiability or unboundedness) for problem p in T seconds, and 0 otherwise. A slightly better metric measures *optimization time*, i.e. the time to find an optimal solution. Let $\texttt{otime}(s,p) = t$ if s finds and proves the optimal solution of p in time t, and $\texttt{otime}(s,p) = T$ if $\texttt{proven}(s,p) = 0$. Unfortunately, both these metrics are rather poor at discriminating: for many optimization problems no solver may be able to prove optimality.

The `score` function introduced in [4] gives to each solver a score in $[0.25, 0.75]$ proportional to the distance between the best solution it finds and the best known solution. An additional reward ($\texttt{score} = 1$) is given if $\texttt{proven}(s,p) = 1$ while a punishment ($\texttt{score} = 0$) is given if no solution is found without proving unsatisfiability. Let $\texttt{val}(s,p,t) = \min\left(\{+\infty\} \cup \{v \mid (t',v) \in \mathcal{B}(s,p),\ t' \leq t\}\right)$ be the (possible) best objective value found by solver s for instance p at time t. Let $V_p = \{\texttt{val}(s,p,T) \in \mathbb{Z} \mid s \in \Sigma\}$ be the set of all the objective values found by any solver s at the time limit T. The score of solver s on a problem p (at the time limit T) is a value $\texttt{score}(s,p) \in \{0,1\} \cup [0.25, 0.75]$ such that:

$$\texttt{score}(s,p) = \begin{cases} 0 & \text{if } \texttt{val}(s,p,T) = +\infty;\ \textit{else} \\ 1 & \text{if } \texttt{proven}(s,p) = 1;\ \textit{else} \\ 0.75 & \text{if } \texttt{val}(s,p,T) = \min V_p = \max V_p;\ \textit{else} \\ 0.75 - 0.5 \cdot \dfrac{\texttt{val}(s,p,T) - \min V_p}{\max V_p - \min V_p} \end{cases}$$

Under this measure, a better solver has a higher score. While more discriminatory than the previous measures, the `score` measure is still only considers the result at the time limit T, without considering how it was reached (i.e. the behaviour).

In this work we introduce a new metric able to estimate the *anytime* solver performance. Let $W_p = \{\texttt{val}(s,p,t) \in \mathbb{Z} \mid s \in \Sigma,\ t \in [0,T]\}$ be the set of all the solutions found by any solver at any time, so $\min W_p$ is the best solution found for problem p and $\max W_p$ is the worst one. If $\mathcal{B}(s,p) = [(t_1,v_1),\ldots,(t_n,v_n)]$ is the behaviour of solver s on problem p, we define the *(solving) area* of s on p as:

$$\texttt{area}(s,p) = t_1 + \sum_{i=1}^{n} \left(0.25 + 0.5 \cdot \frac{\texttt{val}(s,p,t_i) - \min W_p}{\max W_p - \min W_p}\right)(t_{i+1} - t_i)$$

where $t_{n+1} = \texttt{otime}(s,p)$. As the name implies, `area` is a normalized measure of the area under a solver behaviour. This metric is similar to the *primal integral* [8] used for measuring impact of heuristics for MIP solvers, but differs since the primal integral assumes the optimal solution is known, while `area` also differentiates between finding and proving a solution optimal. The `area` measure

folds in a number of measures of the strength of an optimization algorithm: the quality of the best solution found, how quickly any solution is found, whether optimality is proven, and how quickly good solutions are found. Even though the ideal goal is to find the best objective value and hopefully proving its optimality, `area` allows us to discriminate much more between solvers, since we capture the tradeoff between speed and solution quality. Two solvers which eventually reach the same best solution (without proving optimality) are indistinguishable with the other measures, but we would almost certainly prefer the solver that finds the solution(s) faster. Furthermore, consider two solvers that prove optimality at the same instant $t < T$: while both will have $\texttt{otime} = t$, `area` will reward the solver in $[0, t]$ that finds better solutions faster.

Finally, we can now define the *best solver* of Σ for a given problem p as the solver $s \in \Sigma$ which minimizes (w.r.t. the lexicographic ordering) the set of triples $(1 - \texttt{score}(s, p), \texttt{otime}(s, p), \texttt{area}(s, p))$ i.e., the solver that finds the best solution within the time limit T, breaking ties using minimum optimization time first, and then minimum area (i.e., giving priority to the solvers that prove optimality in less time, or at least that quickly find sub-optimal solutions).

3.2 `TimeSplit` Algorithm

Our goal is now to find a suitable timesplit solver for instance p which can improve upon the best solver for p. The algorithm `TimeSplit` described with pseudo-code in Listing 1.1 encodes what was informally explained earlier (see Figure 1). Given as input a problem p, a portfolio $\Pi \subseteq \Sigma$, and the timeout T, the basic idea of `TimeSplit` is to start from the behaviour of the best solver $s_2 \in \Pi$ for p and then examine other solvers behaviours looking for the maximum ideal "left shift" toward another solver $s_1 \in \Pi \setminus \{s_2\}$. Then, starting from s_1, this process is iteratively repeated until no other shift is found. The best solver of Π is assigned to s_2 via function `best_solver` in line 2, while line 3 set the current schedule σ to $[(s_2, T)]$. In line 4 auxiliary variables are initialized: *tot_shift* keeps track of the sum of all the shifts identified, *max_shift* is the current maximum shift that s_2 can perform, *split_time* is the time instant from which s_2 will start its execution, while *split_solver* is the solver that has to be run before s_2 until *split_time* instant. The `while` loop enclosed in lines 5-18 is repeated until no more shifts are possible (i.e., $\mathit{max\_shift} = 0$). The three nested loops starting at lines 7-9 find two pairs (t_1, v_1) and (t_2, v_2) such that s_2 can virtually shift to another solver s_1, i.e., such that in the current solving window $[0, \mathit{split\_time}]$ we have that at time $t_1 < t_2$ solver s_1 finds a value v_1 better than or equal to v_2.

If the actual shift $\Delta t = t_2 - t_1$ is greater than *max_shift*, in lines 11-13 the auxiliary variables are updated accordingly. At the end of each such loop, if at least one shift has been detected ($\mathit{max\_shift} > 0$) the current schedule σ needs to be updated. In line 15, the allocated time of the current first solver of σ (i.e., s_2) is decreased by an amount of time *max_shift* + *split_time* (note that $\texttt{first}(\sigma)$ is a reference to the first element of σ, while `snd` returns the second element of a pair, i.e. the allocated time in this case). This is because *split_time* seconds will be allocated to *split_solver* (line 16: `push_front` inserts an element

on top of the list) while *max_shift* seconds corresponding to the ideal shift will be later donated to the 'overall' best solver of Π (i.e., the last solver of σ) via tot_shift variable. At this stage, the search for a new shift is restricted to the time interval $[0, split\_time]$ in which the new best solver s_2 will be *split_solver* (line 18). Once out of the `while` loop (no more shifts are possible) the total amount of all the shifts found is added to the best solver (line 19: $\texttt{last}(\sigma)$ is a reference to the last element of σ) and the final schedule is finally returned in line 20.

Listing 1.1: `TimeSplit` Algorithm.

```
1  TimeSplit(p, Π, T):
2     s_2 = best_solver(p, Π, T)
3     σ = [(s_2, T)]
4     tot_shift = 0  ;  max_shift = 1  ;  split_time = T  ;  split_solver = s_2
5     while max_shift > 0:
6        max_shift = 0
7        for (t_2, v_2) in {(t, v) ∈ B(s_2, p) | t ≤ split_time}:
8           for s_1 in Π \ {s_2}:
9              for (t_1, v_1) in {(t, v) ∈ B(s_1, p) | t < t_2 ∧ v ≤ v_2}:
10                if t_2 − t_1 > max_shift:
11                   max_shift = t_2 − t_1
12                   split_time = t_1
13                   split_solver = s_1
14       if max_shift > 0:
15          first(σ).snd −= max_shift + split_time
16          push_front(σ, (split_solver, split_time))
17          tot_shift += max_shift
18          s_2 = split_solver
19    last(σ).snd += tot_shift
20    return σ
```

3.3 `TimeSplit` Evaluation

In order to experimentally verify the correctness of our assumptions of the behaviour of timesplit solvers, we tested `TimeSplit` by considering a portfolio Π constructed from the solvers of the MiniZinc 1.6 suite [21] (i.e., CPX, G12/FD, G12/LazyFD, and G12/MIP) with some additional solvers disparate in their nature, namely: Gecode [11] (CP solver), MinisatID [10] (SAT-based solver), Chuffed (Lazy Clause CP solver), and G12/Gurobi (MIP solver). We retrieved and filtered an initial dataset Δ of 4864 MiniZinc COPs from MiniZinc 1.6 benchmarks and the MiniZinc Challenges 2012/13 and then ran `TimeSplit` using a solving timeout of $T = 1800$ seconds. In particular, we ran and compared two versions of the algorithm: the original one and a variant (denoted `TS-2` in what follows) in which we imposed a maximum size of 2 solvers for each schedule σ. This is because splitting $[0, T]$ in too many slots can be counterproductive in practice: excessive fragmentation of the time window may produce time slots that are too short to be useful. Once executed these algorithms, in order to evaluate their significance we discarded all the "degenerate" instances for which the

	score	proven	otime	area
VBS	**82.40%**	**34.73%**	1298.67	478.05
TimeSplit	80.49%	33.67%	1263.74	347.91
TS-2	80.60%	33.89%	**1259.98**	**343.97**

Table 1: Average performances.

	VBS	TimeSplit	TS-2
VBS	—	222	232
TimeSplit	**373**	—	40
TS-2	364	13	—

Table 2: Pairwise Comparisons.

potential total shift was minimal (less than 5 seconds). We then ended up with a reduced dataset $\Delta' \subset \Delta$ of 596 instances. We ran timesplit solvers defined by the schedule returned by each algorithm on every instance of Δ'. In addition, we added as a baseline the *Virtual Best Solver* (VBS) i.e. a fictitious portfolio solver that always choose the best solver for every instance according to a given metric. Finally, we evaluated and compared the average performance in terms of the above mentioned metrics: score, proven, otime, area.

Table 1 shows the average results for each approach. As can be seen, the performances are rather close. On average, VBS is still the best solver if we focus on score metric (i.e., considering only the values found at the time limit T). Regarding proven and otime metrics, we can observe a substantial equivalence: VBS is slightly better in terms of percentage of optima proven, while it is worse than TimeSplit and TS-2 if we consider the average time to prove optimality. Conversely, looking at area the situation appears to be more clearly defined: on average, VBS is substantially worse than both TimeSplit and TS-2. This means that, even if the virtual behaviour does not always occur, often the time splitting we propose is able to find good partial solutions more quickly than the best solver of Π. Focusing just on the two versions of TimeSplit, we can also note that these are substantially equivalent: this confirms the hypothesis that limiting the algorithm to schedule only two solvers is a reasonable choice (TS-2 seems slightly better than TimeSplit on average). Indeed, among all the instances of Δ', only for 53 of them TimeSplit has produced a schedule with more than two solvers.

Table 2 shows instead how many times the approach on the i-th row is better than the one on the j-th column. In this case we can note that TimeSplit and TS-2 perform better than VBS: indeed, in the cases in which the score is the same for both the approaches, often the timesplit solvers take less time to find a (partial) solution. Note that for 375 problems (62.92% of Δ') *at least one* between TimeSplit and TS-2 is better than the VBS. Let Δ^* be the set of such instances, and considering the base $\underline{\sigma(p)}$ of each schedule $\sigma(p)$ returned by the best approach between TimeSplit and TS-2 for each instance of $p \in \Delta^*$, we noticed an interesting fact: the time allocated to $\underline{\sigma(p)}$ is usually pretty low. Figure 2 reports the distribution of such each times. As can be seen, almost all the times are concentrated in the lower part of the graph: even if the maximum value is 1363 seconds, the mean is less than a minute (54.18 seconds to be precise) while the median value is significantly lower (9 seconds).

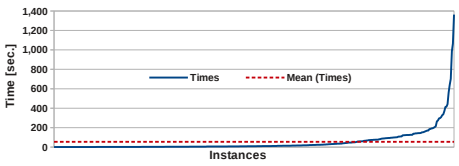


Fig. 2: Times allocated to $\underline{\sigma(p)}$.

4 Timesplit Portfolio Solvers

The results of Section 3.3 show that in a non-negligible number of cases the benefits of using a timesplit solver are tangible. Unfortunately, in such experiments for every instance we already knew the corresponding runtimes of each solver of the portfolio. The main motivation of this work is instead to try to predict and run the best timesplit solver for a new unseen instance. Regrettably, given runtime prediction of a solver is a non-trivial task, predicting the detailed solver behaviour on a new test instance is even harder. Indeed, in our case we can not simply limit ourselves to guess the best solver for a new instance, but we should instead predict a suitable timesplit solver $[(s_1, t_1), \ldots, (s_k, t_k)]$. Moreover, even if in most cases the `TimeSplit` algorithm works pretty well, on the others we noticed a considerable number of instances for which this algorithm is ineffective (or even harmful). Therefore, a successful strategy should be able not only to predict a proper timesplit solver, but also to distinguish between the instances for which the timesplit is actually useful and those where it is counterproductive. Furthermore, another interesting observation that has emerged from the results of Section 3.3 is that often for the "significant" timesplit solvers is sufficient to run the base of the schedule for a relatively low number of seconds in order to allow an effective improvement of the best solver.

On the basis of these observations and motivations, what we propose is therefore a generic and hybrid framework that we called *Timesplit Portfolio Solver* (`TPS`). When a new instance p arrives, we compute and run on p a corresponding timesplit solver $\texttt{TPS}(p) = [(\mathbb{S}, C), (\mathbb{D}(p), T - C)]$, where $[(\mathbb{S}, C)]$ is a *static* timesplit solver pre-computed off-line that will run for $C < T$ seconds, while for the remaining $T - C$ seconds we execute a *dynamic* timesplit solver $[(\mathbb{D}(p), T - C)]$ computed on-line by means of a given prediction algorithm $\mathbb{D}(p)$

The underlying idea of `TPS` is to exploit for the first C seconds a fixed schedule calculated *a priori*, whose purpose is to produce as many good sub-optimal solutions as possible. If after C seconds the optimality is still not proven, in the remaining $T - C$ seconds the algorithm $\mathbb{D}(p)$ tries to predict which is the best (timesplit) solver for p, that will be executed taking advantage of any upper bound provided by $\mathbb{S}$. Since `TPS` is a general model that can be arbitrarily specialized, in the rest of the Section we explain in more detail what choices we made and what algorithms we used to define and evaluate (variants of) `TPS`.

4.1 Static Splitting

Drawing inspiration from what was done in [16] for SAT problems, we decided to compute a static schedule of solvers according to the outcomes of `TimeSplit` on a given set of training instances. While in [16] the authors solve a Resource Constrained Set Covering Problem (RCSCP) in order to get a schedule that maximizes the number of training instances that can be solved within a time limit of $C = 180$ seconds, in our case the objective is different. What we would like is indeed to compute a schedule that may act as a good base for the solver(s) who will be executed in the remaining $T - C$ seconds. To do this, we first identify by

means of `TimeSplit` algorithm the set Δ^* of all the training instances for which a timesplit solver outperforms the `VBS`. Let $\sigma(p) = [(s_{p,1}, t_{p,1}), \ldots, (s_{p,k}, t_{p,k})]$ be the schedule returned by `TimeSplit` on each $p \in \Delta^*$. We look for a schedule $\mathbb{S}$ that maximizes the number of time slots $t_{p,i} \in \underline{\sigma(p)}$ for $i = 1, \ldots, k-1$ that are *covered*, that is the portfolio solver allocates at least $t_{p,i}$ seconds to solver $s_{p,i}$. Again, note that we consider the base $\underline{\sigma(p)}$ instead of $\sigma(p)$ since at this stage we are not interested in choosing the best solver: we want to determine an effective timesplit solver able to quickly find suitable sub-optimal solutions. However, a nice side-effect of this approach is that it also may be able to solve quickly those instances that are extremely difficult for some solvers but very easy for others.

For each $p \in \Delta^*$, we define $\nabla_p = \{(s_{p,i}, t) \mid (s_{p,i}, t_{p,i}) \in \underline{\sigma(p)},\ t_i \leq t \leq C\}$ as the set of all the pairs $(s_{p,i}, t)$ that *cover* the time slot $t_{p,i}$ within C seconds. Named $\Pi^* = \bigcup_{p \in \Delta^*} \{s \in \Pi : (s,t) \in \nabla_p\}$ the set of the solvers of the portfolio that appear in at least a ∇_p, and fixed $C = T/10$, we solve the following problem:

$$
\begin{aligned}
\min \left[(C+1) \sum_{p \in \Delta^*} y_p + \sum_{s \in \Pi^*} \sum_{t \in [0,C]} t\, x_{s,t} \right] \qquad & s.t. \\
y_p + \sum_{(s,t) \in \nabla_p} x_{s,t} \geq 1 \qquad & \forall p \in \Delta^* \\
\sum_{s \in \Pi^*} \sum_{t \in [0,C]} t\, x_{s,t} \leq C & \\
y_p,\ x_{s,t} \in \{0,1\} \qquad & \forall p \in \Delta^*,\ \forall s \in \Pi^*,\ \forall t \in [0,C]
\end{aligned}
$$

For each pair (s,t) there is a binary variable $x_{s,t}$ that will be equal to one if and only if in $\mathbb{S}$ the solver s will run for t seconds. For each problem p, the binary variable y_p will be one if and only if $\mathbb{S}$ cannot cover any time slot of $\underline{\sigma(p)}$. Constraint $y_p + \sum x_{s,t} \geq 1$ imposes that instance p is covered (possibly setting $y_p = 1$ in the worst case) while $\sum t\, x_{s,t} \leq C$ ensures that $\mathbb{S}$ will not exceed the time limit C. The objective is thus to minimize the number of uncovered instances first (by means of $C+1$ coefficient for each y_p), and the total time of $\mathbb{S}$ then (using t coefficient for each $x_{s,t}$).

Note that the solution of the problem defines an allocation $\xi = \{(s,t) : x_{s,t} = 1\}$ and not actually a schedule: we still have to define the execution order of the solvers. Since the interaction between different solvers is not easily predictable, and neither generalizable, we decided to use a simple and reasonable heuristic: we get the schedule $\mathbb{S}$ by sorting each $(s,t) \in \xi$ by increasing allocated time t.

4.2 Dynamic Splitting

Once defined the static part of `TPS`, we want to determine an algorithm $\mathbb{D}(p)$ able to predict for a new unseen instance p a proper (timesplit) solver to run for $T - C$ seconds after $[(\mathbb{S}, C)]$. Inspired by the results of [4], we made use of the SUNNY algorithm [3, 4]. The reasons behind this choice are essentially two. First, even if originally designed for CSP portfolios [3], the adaption of SUNNY for COPs turns out to perform well according to the results of [4]. Second, SUNNY is not

	p_1	p_2	p_3	Total
s_1	$(\mathbf{1}, 150)$	$(0.25, 1000)$	$(\mathbf{0.75}, 1000)$	$(\mathbf{2}, \textit{2150})$
s_2	$(0, 1000)$	$(\mathbf{1}, 10)$	$(0, 1000)$	$(\mathbf{1}, \textit{2010})$
s_3	$(1, 100)$	$(0.75, 1000)$	$(0.7, 1000)$	$(2.45, \textit{2100})$
s_4	$(0.75, 1000)$	$(0.75, 1000)$	$(0.25, 1000)$	$(1.75, 3000)$

Table 3: Pairs (`score`, `otime`) of each solver s_i for every problem p_j.

limited to predict a single solver but selects instead a schedule of the constituent solvers: in other terms, it implicitly returns a timesplit solver.

SUNNY is a new lazy algorithm portfolio originally tailored for CSPs: given a CSP p and a portfolio Π, it uses a k-NN algorithm to select from a set of training instances a subset $N(p, k)$ of the k instances closer to p according to the Euclidean distance. Then, on-the-fly, it computes a schedule of solvers by considering the smallest sub-portfolio $S \subseteq \Pi$ able to solve the maximum number of instances in the neighborhood $N(p, k)$ and by allocating to each solver of S a time proportional to the number of solved instances in $N(p, k)$. In [4] SUNNY was adapted in order to deal with COPs: this variant selects the sub-portfolio $S \subseteq \Pi$ that maximizes the `score` in the neighborhood and allocates to each solver a time in $[0, T]$ proportional to its total score in $N(p, k)$. In particular, while in the CSP version SUNNY allocates to a *backup solver*[4] an amount of time proportional to the number of instances not solved in $N(p, k)$, in the COP version it assigns to it a slot of time proportional to $k-h$ where h is the maximum `score` achieved by the sub-portfolio S. While for CSPs the final schedule is obtained by sorting the solvers by increasing solving time, for COPs the sorting is done by using increasing `otime`. In a nutshell, the underlying idea behind SUNNY is to minimize the probability of choosing the wrong solvers(s) by exploiting instance similarities in order to get the smallest possible schedule of solvers. Padding the uncovered instances of $N(p, k)$ with the backup solver has the purpose of filling the "gray area" between the best sub-portfolio found and a virtual solver always able to find the optimal solution with the (hopefully) most reliable solver of Π. Of course, this is an arbitrary choice that biases the schedule toward the backup solver. But experimental results have proven the effectiveness of this approach.

Example 1. Let us suppose that $\Pi = \{s_1, s_2, s_3, s_4\}$, the backup solver is s_3, $T = 1000$ seconds, $k = 3$, $N(p, k) = \{p_1, p_2, p_3\}$, and the scores/optimization times are defined as listed in Table 3. The minimum size sub-portfolio that reaches the highest score $h = 1 + 1 + 0.75 = 2.75$ is $\{s_1, s_2\}$. On the basis of the sum of the scores reached by s_1 and s_2 in $N(p, k)$ (resp. 2 and 1, see the last column of Table 3) we then determine a slot size of $t = T/(2+1+(k-h)) = 307.69$ seconds and assign $t_1 = 2 * t = 615.38$ seconds to s_1 and $t_2 = 1 * t = 307.69$ seconds to s_2. The remaining $t_3 = 1000 - (t_1 + t_2) = 76.93$ seconds are finally allocated to the backup solver s_3. The final schedule $[(s_2, t_2), (s_3, t_3), (s_1, t_1)]$ is then obtained by sorting s_1, s_2, s_3 by their total optimization time in $N(p, k)$ (i.e., 2010, 2100, and 2150 respectively: see the last column of Table 3).

[4] A backup solver is a special solver of the portfolio (typically, its *single best solver*) aimed to handle exceptional circumstances (e.g.,premature failures of other solvers).

5 Empirical Evaluation

In order to measure the performances of the `TPS` described in Sections 4.1 and 4.2, in the following referred to as `sunny-tps`, we considered a solving timeout of $T = 1800$ seconds, a threshold of $C = 180$ seconds, the portfolio $\Pi = \{$Chuffed, CPX, G12/FD, G12/LazyFD, G12/Gurobi, G12/MIP, Gecode, MinisatID$\}$ and the dataset Δ of 4864 MiniZinc instances introduced in Section 3.3.

We evaluated `sunny-tps` using a 10-fold cross validation [6]: Δ was randomly partitioned in 10 disjoint folds $\Delta_1, \ldots, \Delta_{10}$ treating in turn one fold Δ_i as the test set TS_i $(i = 1, \ldots, 10)$ and the union of the remaining folds $\bigcup_{j \neq i} \Delta_j$ as the training set TR_i. For each training set TR_i we then computed a corresponding static schedule $[(\mathbb{S}_i, 180)]$ as explained in Section 4.1, and for every instance $p \in \mathrm{TS}_i$ we computed and executed the timesplit solver $[(\mathbb{S}_i, 180), (\mathbb{D}_i(p), 1620)]$ where $\mathbb{D}_i(p)$ is the schedule returned by SUNNY algorithm for problem p using a reduced solving window of $T - C = 1620$ seconds.

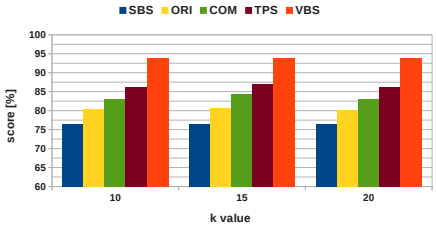
Note that, for computing $\mathbb{D}_i(p)$, SUNNY has to retrieve the k instances of TR_i closest to p. In order to do so, a proper set of *features* has to be extracted from p (and each instance of TR_i). Instead of using the whole set of 155 features extracted by the `mzn2feat` tool described in [2,5] (as in [4]) we decided to select a proper subset of them by exploiting the new extractor `mzn2feat-1.0`.[5] This tool is a new version of `mzn2feat` designed to be more portable, light-weight, flexible, and independent from the particular machine on which it is run as well as from the specific global redefinitions of a given solver. Indeed, `mzn2feat-1.0` does not compute features based on graph measures (since this process could be very time/space consuming), solver specific features (like global constraint redefinitions) and dynamic features (to decouple the extractor from a particular solver and from the given machine on which it is executed). In more detail, `mzn2feat-1.0` extracts in total 95 features: the variables (27), domains (18), constraints (27), and solving (11) features are exactly the same of `mzn2feat`; the objective features (8) are the 12 objective features of [5] except the 4 features that involve graph measures; finally, the global constraints features are just 4 and no longer bound to the Gecode solver, namely: the number of global constraints n, the number of different global constraints m, the ratio m/n and the ratio n/c where c is the total number of constraints of the problem. We finally removed all the constant features and scaled them in [-1, 1], obtaining thus a reduced set of 88 features.

As in Section 2, we evaluated the average performance of `sunny-tps` in terms of `score`, `proven`, `otime`, `area` by varying the neighbourhood size k in $\{10, 15, 20\}$. Finally, we compared `sunny-tps` vs. the following approaches:

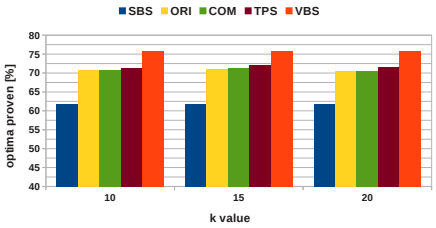
- `SBS`: is the overall Single Best Solver of Π according to the given metric; [6]
- `VBS`: is the Virtual Best Solver of Π defined as in Section 3.3;

[5] Available at `http://www.cs.unibo.it/~amadini/mzn2feat-1.0.tar.bz2`

[6] Regarding the `score`, `otime`, and `area` metrics the single best solver of Π turned out to be CPX, while for the `proven` metric it is Chuffed. According to these results, we elected CPX as the backup solver of Π.



(a) `score` results (in percent).



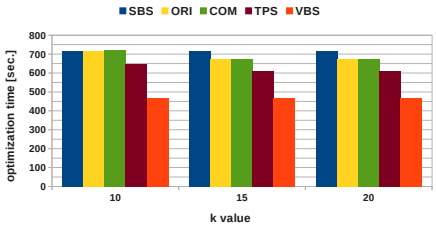
(b) `proven` results (in percent).

- `sunny-ori`: is the original SUNNY algorithm evaluated in [4], that is a portfolio solver in which the selected solvers are executed independently (i.e., without any bounds communication) in the time window $[0, T]$ without the "warm start" provided by $\mathbb{S}$ for the first C seconds;
- `sunny-com`: is a portfolio solver that acts basically as `sunny-ori`, with the only exception that solvers execution is not independent: the best value found by a solver within its time slot is subsequently exploited by the following solver of the schedule.

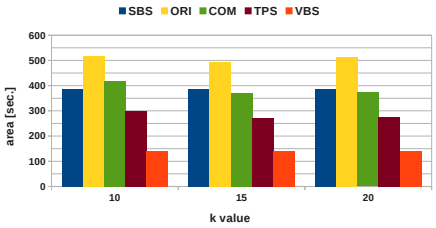
The universe of all the (timesplit) solvers we used for our empirical evaluation was therefore $\Sigma = \Pi \cup \{\texttt{SBS}, \texttt{VBS}, \texttt{sunny-com}, \texttt{sunny-ori}, \texttt{sunny-tps}\}$. It is worth nothing that, while in [1, 4] the evaluation was based on *simulations* of the portfolio approaches according to the already computed behaviours of every solver of Π on every instance of Δ, in this work all the approaches have been *actually* run and evaluated. Indeed, as shown also in Section 3.3, in this case we can not make use of simulations since the side effects of bounds communication are unpredictable in advance.

5.1 Test Results

The average `score` results (in percent) by varying the k parameter are reported in Figure 3a. The plot clearly shows a common pattern: `SBS`, `sunny-ori`, `sunny-com`, `sunny-tps`, and `VBS` are respectively sorted by increasing score for every value of k. In general, we can see a rather sharp separation between the various approaches: this witnesses the effectiveness of bounds communication for reaching a better score or, in other terms, for improving the objective value (possibly proving its optimality). For example, the percentage difference between `sunny-ori` and `sunny-com` ranges between 2.83% and 3.45%. Furthermore, running the static schedule for the first 180 seconds (and therefore shrinking the dynamic schedule of `sunny-com` in the remaining 1620 seconds) seems to be advantageous: `sunny-tps` is always better than `SBS`, `sunny-ori`, and `sunny-com`. The peak performance (86.91%) is reached with $k = 15$, but the difference with $k = 10$ and 20 is minimal (0.73% and 0.59% respectively). Considering $k = 15$, `sunny-tps` has an average score higher than `SBS` by 10.55%, and lower than `VBS` by 6.9%. Moreover, in 82 cases (1.69% of Δ) it scores better than `VBS`.



(c) `otime` results (in seconds).



(d) `area` results (in seconds).

When considering the `proven` metric (Figure 3b) the performance difference between the different SUNNY approaches is not so pronounced. Indeed, `sunny-ori`, `sunny-com`, and `sunny-tps` are pretty close: for every k, the percentage difference between the worst and the best SUNNY approach ranges between 0.45% and 1.13%. In this case we can say that the remarkable difference in performance between the portfolio solvers and the `SBS` is mainly due to the SUNNY algorithm rather than the bounds communication. In other words, passing the bound is not so effective if we just focus on proving optimality. A possible explanation is that communicating an upper bound can be useful to find a better solution (see Figure 3a) but ineffective when it comes to prove optimality. In these cases probably the time needed by a solver to compute information for completing the search process can not be offset by the mere knowledge of an objective bound. Nonetheless, the plot shows how the "warm start" provided by the static schedule is helpful: in fact, the performance of `sunny-tps` is always better than the other approaches. The peak performance ($k = 15$) is 72.06%, about 10.36% more than `SBS` and only 3.74% less than `VBS`. For 27 instances (0.56% of Δ) `sunny-tps` is able to prove the optimality while `VBS` is not.

Let us now focus on optimization time. In Figure 3c we see, in contrast to all the `score` and `proven` results that appear to be pretty robust by varying k, a slight discrepancy between $k = 10$ and $k > 10$. This delay time in proving optimality is due to the scheduling order of the constituent solvers. However, for $k = 15$ the results improve and for $k = 20$ are substantially the same. The peak performance is achieved with $k = 15$ (272.61 seconds), 105.07 less than `SBS` and 145.9 more than `VBS`; in 53 cases (1.09% of Δ) `sunny-tps` is able to prove the optimality in less time than `VBS`.

The `area` results depicted in Figure 3d clearly show the benefits of bounds communication. First, note that `sunny-ori` is always worse than `SBS`: this is because each solver scheduled by `sunny-ori` is executed independently, and therefore for every solver the search is always (re-)started from scratch without exploiting previously found solutions. `sunny-com` significantly improves `sunny-ori`, even if its average area is very close to `SBS` (even worse for $k = 10$). On the other hand, the fixed schedule run by `sunny-tps` often allows one to quickly find partial solutions and thus to noticeably outperform both `sunny-ori` and `sunny-com`. Like the `otime` metric, the average area is not so close to `VBS` (the peak perfor-

mance, with $k = 15$, is 272.61 seconds: 132.77 seconds more than `VBS`, and 114.9 less than `SBS`), but `sunny-tps` outperforms `VBS` in 110 cases (2.26% of Δ).

6 Conclusions and Future Work

In this work we addressed the problem of boosting optimization by exploiting the sequential cooperation of different COP solvers. Exploiting the fact that finding good solutions early can significantly improve optimization solvers, we first provided a proper `TimeSplit` algorithm that relies on the behaviour of different solvers on an instance for determining a good timesplit solver for this instance (i.e., ideally able to outperform the best solver of a portfolio). Our results show that on average the actual timesplit solver does perform similarly to (and sometimes even better than) the Virtual Best Solver of the portfolio. We therefore exploited the results of `TimeSplit` in order to define the Timesplit Portfolio Solver (`TPS`), a generic and hybrid framework that combines a static schedule (computed off-line and run for a limited time) as well as a dynamic schedule (computed on-line by means of a proper prediction algorithm and run in the remaining time) for solving a new unseen instance by exploiting the bounds communication between the scheduled solvers. In particular, on the one hand, we determined the static schedule by solving a Set Covering problem according to the results of `TimeSplit` on a set of training instances, and, on the other, we defined the dynamic selection by exploiting the SUNNY algorithm [3, 4]. Empirical results shows that this idea can be beneficial and sometimes even able to outperform the Virtual Best Solver according to different metrics that we introduced (namely, `score`, `proven`, `otime`, and `area`) in order to evaluate the performance of different (portfolio) solvers.

We see this work a cornerstone for portfolio approaches to solving Constraint Optimization Problems. Clearly bounds communication should be taken into account in this case. It is natural to think of extensions to this work, for example, one may try to maximize the ideal shift by considering all the constituent solvers instead of focusing just on improving the best one. Moreover, the nature of `TPS` naturally allows one to instantiate its generic schema with new algorithms and techniques (perhaps by simply adapting the most successful portfolio approaches of the SAT and CSP fields). For instance, one might study how to select the set of features to improve solver selection. Note that, contrary to [1, 4], for example, significant experimentation is required since it is clearly not predictable in a deterministic way what the side effects of transmitting bounds from a solver to another are. Finally, other interesting directions concerns the study of parallel timesplit solvers as well as the communication of not only the objective bounds but also other additional information (such as for instance cuts which is common in SAT portfolios).

Acknowledgments NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. This work was partially supported by Asian Office of Aerospace Research and Development grant 12-4056.

References

1. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An Empirical Evaluation of Portfolios Approaches for Solving CSPs. In *CPAIOR*, 2013.
2. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Features for Building CSP Portfolio Solvers. *CoRR*, abs/1308.0227, abs/1308.0227, 2013.
3. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY: a Lazy Portfolio Approach for Constraint Solving. `http://www.cs.unibo.it/~amadini/iclp_2014.pdf`, In *ICLP*, 2014.
4. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Portfolio Approaches for Constraint Optimization Problems. `http://www.cs.unibo.it/~amadini/lion_2014.pdf`, In *LION*, 2014.
5. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An Enhanced Features Extractor for a Portfolio of Constraint Solvers. `http://www.cs.unibo.it/~amadini/sac_2014.pdf`, In *SAC*, 2014.
6. S. Arlot and A. Celisse. A survey of cross-validation procedures for model selection. *Statistics Surveys*, 4:40–79, 2010.
7. Gilles Audemard, Benot Hoessen, Sad Jabbour, Jean-Marie Lagniez, and Cdric Piette. PeneLoPe, a Parallel Clause-Freezer Solver. In *SAT Challenge 2012*.
8. Timo Berthold. Measuring the impact of primal heuristics. *Operations Research Letters*, 41(6):611–614, 2013.
9. Tom Carchrae and J. Christopher Beck. Low-knowledge algorithm control. In *AAAI*, pages 49–54, 2004.
10. Broes DeCat. KRR Software: MinisatID. `http://dtai.cs.kuleuven.be/krr/software/minisatid`, 2013.
11. GECODE - An open, free, efficient constraint solving toolkit. `http://www.gecode.org`.
12. Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 2001.
13. Haipeng Guo and William H. Hsu. A machine learning approach to algorithm selection for NP-hard optimization problems: a case study on the MPE problem. *Annals OR*, 156(1):61–82, 2007.
14. Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT Solver. *JSAT*, 6(4):245–262, 2009.
15. Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm Runtime Prediction: The State of the Art. *CoRR*, abs/1211.0906, 2012.
16. Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm Selection and Scheduling. In *CP*, 2011.
17. Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC - Instance-Specific Algorithm Configuration. In *ECAI*, 2010.
18. Lars Kotthoff. Algorithm Selection for Combinatorial Search Problems: A Survey. *CoRR*, abs/1210.7959, 2012.
19. Alan K. Mackworth. Consistency in Networks of Relations. *Artif. Intell.*, 1977.
20. Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Boosting sequential solver portfolios: Knowledge sharing and accuracy prediction. In *LION*, pages 153–167. Springer, 2013.
21. Minizinc version 1.6. `http://www.minizinc.org/download.html`.
22. Eoin OMahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry OSullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. *AICS 08*, 2009.
23. John R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 1976.

24. Olivier Roussel. ppfolio. http://www.cril.univ-artois.fr/~roussel/ppfolio/.
25. Kate Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 41(1), 2008.
26. Orestis Telelis and Panagiotis Stamatopoulos. Combinatorial Optimization through Statistical Instance-Based Learning. In *ICTAI*, 2001.
27. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-brown. Hydra-MIP: Automated Algorithm Configuration and Selection for Mixed Integer Programming. In *RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 2011.