SUNNY: a Lazy Portfolio Approach for Constraint Solving

ROBERTO AMADINI and MAURIZIO GABBRIELLI and JACOPO MAURO
Department of Computer Science and Engineering/Lab. Focus INRIA, University of Bologna, Italy.

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Within the context of constraint solving, a portfolio approach allows one to exploit the synergy between different solvers in order to create a globally better solver. In this paper we present SUNNY: a simple and flexible algorithm that takes advantage of a portfolio of constraint solvers in order to compute - without learning an explicit model - a schedule of them for solving a given Constraint Satisfaction Problem (CSP). Motivated by the performance reached by SUNNY vs. different simulations of other state of the art approaches, we developed `sunny-csp`, an effective portfolio solver that exploits the underlying SUNNY algorithm in order to solve a given CSP. Empirical tests conducted on exhaustive benchmarks of MiniZinc models show that the actual performance of `sunny-csp` conforms to the predictions. This is encouraging both for improving the power of CSP portfolio solvers and for trying to export them to fields such as Answer Set Programming and Constraint Logic Programming.

KEYWORDS: Algorithms Portfolio, Artificial Intelligence, Constraint Satisfaction, Machine Learning.

1 Introduction

Constraint Programming (CP) is a declarative paradigm that enables expressing relations between different entities in the form of constraints that must be satisfied. One of the main goals of CP is to model and solve *Constraint Satisfaction Problems* (CSP) (Mackworth 1977) as well as problems like the well-known Boolean satisfiability problem (SAT), Quantified Boolean Formula (QBF), Satisfiability Modulo Theories (SMT), and Answer-Set Programming (ASP). One of the more recent trends of CP - especially in the SAT field - is trying to solve a given problem by using a *portfolio* approach (Rice 1976; Gomes and Selman 2001).

A portfolio approach 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, dubbed a *portfolio solver*, that exploits a collection of $m > 1$ different constituent solvers $s_1, \dots, s_m$ in order to obtain a globally better CP solver. When a new unseen instance i comes, the portfolio solver tries to predict which are the best constituent solvers $s_1, \dots, s_k$ (with $1 \leq k \leq m$) 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* (ML) techniques. Exploiting the fact that different solvers are better at solving different problems, portfolios have proved to be particularly effective. For example, the overall winners of international solving competitions like the SAT Challenge and CSP Competition are often portfolio solvers.

Surprisingly, despite their proven effectiveness, portfolio solvers are rarely used in practice and usually applied only to the SAT field. Indeed there are only few applications outside the SAT world, such as, for example, Gebser et al. (2011) for ASP, OMahony et al. (2009) for CSP, and Hutter et al. (2012) for optimization problems like the Traveling Salesman Problem. On the other hand, if we exclude competitive scenarios such as the SAT Competitions, even the SAT portfolio solvers are actually underutilized. As pointed out also by Samulowitz et al. (2013), we think that one of the main reasons for this incongruity lies in the fact that state of the art portfolio solvers usually require a complex off-line training phase and they are not suitably structured to incrementally exploit new incoming information. For instance, a state of the art SAT solver like SATzilla (Xu et al. 2007) requires a model built exploiting a weighted Random Forest machine learning approach while 3S (Kadioglu et al. 2011), ISAC (Malitsky and Sellmann 2012), and CHSC (Malitsky et al. 2013) during their off-line phase cluster the instances of a training set or solve non-trivial combinatorial problems for computing an off-line schedule of the solvers. In order to use these solvers a suitable set of training instances must be found and, especially, the off-line phase has to be run on all the training samples. Moreover, new information (e.g., incoming problems, solvers, features) may imply a re-running of the whole off-line process.

In the literature some approaches avoiding a heavy off-line training phase have been studied, see for instance (Wilson et al. 2000; Pulina and Tacchella 2007; OMahony et al. 2009; Nikolic et al. 2009; Gebruers et al. 2004; Stern et al. 2010; Samulowitz et al. 2013). As pointed out in the comprehensive survey (Kotthoff 2012), these approaches are also referred as *lazy*. Unfortunately, to the best of our knowledge, among these lazy approaches only CPHydra (OMahony et al. 2009) was successful enough to win a solver competition in 2008. Recently, however, Amadini et al. (2013a) showed that some non-lazy approaches derived from SATzilla (Xu et al. 2007) and 3S (Kadioglu et al. 2011) have better performance than CPHydra on CSPs.

Our goal is, on one hand, bridging the gap between SAT and CSP portfolio solvers, and, on the other hand, encouraging the dissemination and the utilization in practice of portfolio solvers even in other growing fields such as, for example, the Answer Set Programming and Constraint Logic Programming paradigms where a few portfolio approaches have been studied (see for example Gebser et al. (2011)). As a first step in order to reach our goal, in this paper, we developed SUNNY and `sunny-csp`.

SUNNY is a lazy algorithm portfolio which exploits instances similarity to guess the best solver(s) to use. For a given instance i, SUNNY uses a *k-Nearest Neighbors* (k-NN) algorithm to select from a training set of known instances the subset $N(i,k)$ of the k instances closer to i. Then, it creates a schedule of solvers considering the smallest *sub-portfolio* able to solve the maximum number of instances in the neighborhood $N(i,k)$. The time allocated to each solver of the sub-portfolio is proportional to the number of instances it solves in $N(i,k)$. We performed a preliminary evaluation of the performance of SUNNY by exploiting a large benchmark of CSPs from each of which we extracted an exhaustive set of features (e.g., number of constraints, number of variables, domain sizes) used to estimate the instances similarity. Following the methodology of Amadini et al. (2013a), we measured the performance of SUNNY by comparing its expected behavior against the simulations of state of the art portfolio solvers, namely, CPHydra, 3S, and SATzilla.

The performance of SUNNY were promising: it overcame all the above mentioned approaches. We therefore developed `sunny-csp`, a new CSP portfolio solver built on the top of the SUNNY algorithm. Using the very same benchmarks and features sets, we compared the actual results of `sunny-csp` w.r.t. its expected performance, that is the ideal performance of the underlying SUNNY algorithm. Test results show that the difference between simulated and actual behavior is minimal, thus encouraging new insights, extensions and implementations of this simple but effective algorithm.

Paper structure. In Section 2 we describe the motivations and the algorithm underlying SUNNY while in Section 3 we explain the methodology and we report the results of our comparisons. In Section 4 we introduce the CSP portfolio solver `sunny-csp` as well as an empirical validation of its performance. In Section 5 we discuss the related literature while Section 6 concludes by discussing also future work.

2 SUNNY

One of the main empirical observations at the base of SUNNY is that usually combinatorial problems are extremely easy for some solvers and, at the same time, almost impossible to solve for others. Moreover, in case a solver is not able to solve an instance quickly, it is likely that such solver takes a huge amount of time to solve the instance. A first motivation behind SUNNY is therefore to select and schedule a subset of the solvers of a portfolio instead of trying to predict the "best" one for a given unseen instance. This strategy hopefully allows one to solve the same amount of problems minimizing the risk of choosing the wrong solver.

Another interesting consideration is that the use of large portfolios might not always lead to performance boost. In some cases the overabundance of solvers hinders the effectiveness of the considered approach. Indeed, selecting the best solver to use is more difficult when a big size portfolio is considered since there are more available choices. Despite the literature having examples of portfolios consisting of nearly 60 solvers (Nikolic et al. 2009), as pointed out by (Amadini et al. 2013a; Pulina and Tacchella 2009) usually the best results are obtained by adopting a relatively small portfolio (e.g., ten or even less solvers).

Another motivation for SUNNY is that - as witnessed for instance by the good performance reached by CPHydra, ISAC (Kadioglu et al. 2010), 3S, CSHC (Malitsky et al. 2013) - the 'similarity assumption', stating that similar instances will behave similarly, is often reasonable. It thus makes sense to use algorithms such as k-NN to exploit the closeness between different instances. As a side effect, this allows to avoid the off-line training phase that, as previously stated, makes the majority of the portfolio approaches rarely used in practice.

Starting from these assumptions we developed SUNNY, whose name is the acronym of:

- *SUb-portfolio*: for a given instance, we select a suitable sub-portfolio (i.e., a subset of the constituent solvers of the portfolio) to run;
- *Nearest Neighbor*: to determine the sub-portfolio we use a k-NN algorithm that extracts from previously seen instances the k instances that are the closest to the instance to be solved;
- *lazY*: no explicit prediction model is built off-line. In a nutshell, the underlying idea behind SUNNY is therefore to minimize the probability of choosing the wrong solvers(s) by exploiting instance similarities in order to quickly get the smallest possible schedule of solvers.

Listing 1: SUNNY Algorithm

```
 1  SUNNY(inst, solvers, bkup_solver, k, T, KB):
 2    feat_vect = getFeatures(inst, KB)
 3    similar_insts = getNearestNeigbour(feat_vect, k, KB)
 4    sub_portfolio = getSubPortfolio(similar_insts, solvers, KB)
 5    slots = $\sum_{s \in \texttt{sub\_portfolio}}$ getMaxSolved(s, similar_insts, KB, T) +
 6          (k − getMaxSolved(sub_portfolio, similar_insts, KB, T))
 7    time_slot = T / slots
 8    tot_time = 0
 9    schedule = {}
10    schedule[bkup_solver] = 0
11    for solver in sub_portfolio:
12      solver_slots = getMaxSolved(solver, similar_insts, KB, T)
13      schedule[solver] = solver_slots * time_slot
14      tot_time += solver_slots * time_slot
15    if tot_time < T:
16      schedule[bkup_solver] += T − tot_time
17    return sort(schedule, similar_insts, KB)
```

2.1 Algorithm

The pseudo-code of SUNNY is presented in Listing 1. SUNNY takes as input the problem `inst` to be solved, the portfolio of solvers `solvers`, a backup solver `bkup_solver`, [1] a parameter `k` (≥ 1) representing the size of the neighborhood to consider, a parameter `T` representing the total time available for running the portfolio solver, and a knowledge base `KB` of known instances for each of which we assume to know the features and the runtimes for every solver of the portfolio.

When a new unseen instance `inst` comes, SUNNY first extracts from it a proper set of features via the function `getFeatures` (line 2). This function takes as input also the knowledge base `KB` since the extracted features need to be preprocessed in order to scale them in the range $[-1,1]$ and to remove the constant ones. `getFeatures` returns the features vector `feat_vect` of the instance `inst`. In line 3, function `getNearestNeigbour` is used to retrieve the `k` nearest instances `similar_insts` to the instance `inst` according to a certain distance metric (e.g., Euclidean). Then, in line 4 the function `getSubPortfolio` selects the minimum subset of the portfolio that allows to solve the greatest number of instances in the neighborhood, by using the average solving time for tie-breaking.[2]

Once computed the sub-portfolio, we partition the time window $[0,\mathtt{T}]$ into `slots` equal time slots of size `T/slots`, where `slots` is the sum of the solved instances for each solver of the sub-portfolio plus the instances of `similar_insts` that can not be solved within the time limit `T`. In order to compute `slots`, we use the function `getMaxSolved(s, similar_insts, KB, T)` that returns the number of instances in `similar_insts` that a solver (or a portfolio of solvers) `s` is able to solve in time `T`. In lines 9-10 the associative array `schedule`, used to define the solvers schedules, is initialized. In particular, `schedule[s] = t` iff a time window of *t* seconds is allocated to the solver *s*.

The loop enclosed between lines 11-14 assigns to each solver of the portfolio a number of time slots proportional to the number of instances that such solver can solve in the neighborhood. In

[1] 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).

[2] In order to compute the average solving time, we assign to an instance not solved within the time limit `T` a solving time of `T` seconds.

lines 15-16 the remaining time slots, corresponding to the unsolved instances, are allocated to the backup solver. Finally, line 18 returns the final schedule, obtained by sorting the solvers, in descending order, by number of solved instances in `similar_insts`.

Example

Let us suppose that `solvers` $= \{s_1, s_2, s_3, s_4\}$*,* `bkup_solver` $= s_3$*,* `T` $= 1800$ *seconds,* `k` $= 5$*,* `similar_insts` $= \{p_1, ..., p_5\}$*, and the run-times of the problems defined by* `KB` *as listed in Table 1. The minimum size sub-portfolios that allow to solve the most instances are* $\{s_1, s_2, s_3\}$*,* $\{s_1, s_2, s_4\}$*, and* $\{s_2, s_3, s_4\}$*. Indeed, each of this portfolios can solve 4 instances. SUNNY selects* `sub_portfolio` $= \{s_1, s_2, s_4\}$ *since it has a lower average solving time (1270.4 sec., to be precise). Since* s_1 *and* s_4 *solve 2 instances,* s_2 *solves 1 instance and* p_1 *is not solved by any solver within* `T` *seconds, the time window* $[0, \mathtt{T}]$ *is partitioned in* $2+2+1+1 = 6$ *slots: 2 assigned to* s_1 *and* s_4*, 1 slot to* s_2*, and 1 to the backup solver* s_3*. After sorting in descending order the solvers by number of solved instances in the neighborhood (using the solving time for tie-breaking) we get the schedule illustrated in Table 2.*

	p_1	p_2	p_3	p_4	p_5
s_1	T	T	**3**	T	**278**
s_2	T	**593**	T	T	T
s_3	T	T	**36**	**1452**	T
s_4	T	T	T	**122**	**60**

Table 1: Runtimes (in seconds). T indicates solver timeouts.

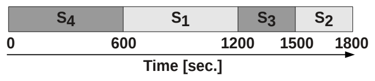


Table 2: Resulting schedule of the solvers.

Clearly, the proposed algorithm features a number of degrees of freedom. For instance, the underlying k-NN algorithm depends on the quality of the features extracted (and possibly filtered), on the choice of the k parameter, and on the distance metric adopted.

A potential weakness of SUNNY is that it could become impracticable for large portfolios. Indeed, in the worst case, the complexity of `getSubPortfolio` is exponential w.r.t. the portfolio size. However, the computation of this function is almost instantaneous for portfolios containing up to 15 solvers and, as later detailed, from an empirical point of view the sizes of the sub-portfolios are small for reasonable values of k.

Finally, let us note that the assignment of the uncovered instances of $N(p, k)$ with the backup solver allows the assignment of some slots to (hopefully) the most reliable solver. This choice obviously biases the schedule toward the backup solver but experimental results have proven the effectiveness of this approach.

3 Validation

Taking as baseline the methodology and the results of (Amadini et al. 2013a) in this section we present the main ingredients and the procedure that we have used for conducting our experiments and for evaluating the portfolio approaches, as well as the obtained experimental results. We would like to point out that, in order to reduce the computational costs, the results of this Section are based on simulations. We computed the running times of all the solvers of the portfolio just once and used this information to evaluate the performance of every approach on every instance of the test set. To conduct the experiments we used Intel Dual-Core 2.93GHz computers with 3 MB of CPU cache, 2 GB of RAM, and Ubuntu 12.04 operating system. For keeping track of the solving times we considered the CPU time by exploiting the Unix `time` command.

3.1 Solvers, Dataset, and Features

In order to evaluate the sensitivity of the proposed approaches w.r.t. the use of different solvers, we built portfolios of different size by considering all the publicly available and directly usable solvers of the MiniZinc Challenge 2012, namely: BProlog, Fzn2smt, CPX, G12/FD, G12/Lazy-FD, G12/MIP, Gecode, izplus, MinisatID, Mistral and OR-Tools. We used all of them with their default parameters, their global constraint redefinitions when available, and keeping track of their performances within a timeout of $T = 1800$ seconds.[3]

To conduct our experiments on a dataset of instances as realistic and large as possible we considered the dataset used in Amadini et al. (2013b). This dataset was obtained by combining 1650 instances gathered from the MiniZinc 1.6 benchmarks, 6 instances from the MiniZinc challenge 2012, and 6944 instances from the International Constraint Solver Competitions (ICSC) of 2009, discarding the "easiest" instances (i.e., those solved by Gecode in less than 2 seconds) and the "hardest" ones (i.e., those for which the features extraction has required more than $T/2 = 900$ seconds). The benchmark thus obtained consisted of 4642 instances (3538 from ICSC, 6 from MiniZinc Challenge 2012, and 1098 from MiniZinc 1.6 benchmarks).

For every instance of the benchmark we have extracted a set of 155 different features by exploiting the tool `mzn2feat` (Amadini et al. 2014a). Among the extracted features, 144 were *static*, i.e., obtained by parsing the source problem instance while 11 were *dynamic*, i.e., obtained by running the Gecode solver for a short run of 2 seconds. For further details about the benchmark and the features we refer the interested reader to Amadini et al. (2013b).

Following what is usually done by the majority of current approaches, we removed all the constant features and we scaled their values in the range [-1, 1]. In this way we ended up with a reduced set of 114 features.

3.2 Portfolios Composition, Approaches, and Evaluation

After having run every solver on each instance of the benchmark, by using a timeout of $T = 1800$ seconds and by keeping track of all the runtimes, we built fixed portfolios of different size $m = 2, \dots, 11$. More specifically, for $m = 2, \dots, 11$ the portfolio composition was computed by considering the portfolio of size m which maximized the number of potential solved instances (possible ties were broken by minimizing the average solving time). Among all the constituent solvers, we elected MinisatID (DeCat 2013) as a *backup solver*, since it is the one that solved the greatest number of instances within the time limit T.

For each of such portfolios we then compared the performances of SUNNY against some of the most effective portfolio approaches. As done in Amadini et al. (2013a), we reproduced the approaches of SATzilla, 3S, and CPHydra. Note that, since 3S and SATzilla are portfolio approaches tailored for SAT, we had to reimplement these approaches, adapting them to our purposes.[4] In order to reproduce the CPHydra approach, we used its original algorithm without

[3] We decided to use the timeout used in the International Constraint Solver Competitions.

[4] The reproduction of SATzilla used in this paper differs from the original version (Xu et al. 2012) because ties during

any parameter tuning. However, since CPHydra does not scale very well w.r.t. the size of the portfolio,[5] we did not consider for the simulations the time taken to compute the schedule of the solvers to use. Thus, the results of CPHydra presented in this paper can be considered only an upper bound of its real performances.

In order to validate and test every approach on each portfolio, by following a common practice we used a 5-repeated 5-fold cross validation (Arlot and Celisse 2010). The benchmark was randomly partitioned in 5 disjoint folds $\Delta_1, \dots, \Delta_5$ treating in turn one fold Δ_i, for $i = 1, \dots, 5$, as the test set and the union of the remaining folds $\bigcup_{j \neq i} \Delta_j$ as the training set. To avoid possible *overfitting* problems we repeated the random generation of the folds for 5 times, thus obtaining 25 different training sets (consisting of about 3714 instances each) and 25 test sets (consisting of about 928 instances). For every instance of every test set we then computed the solving strategy proposed by the particular portfolio approach and we simulated it using a time cap of 1800 seconds. For estimating the solving time we have taken into account also the time needed for extracting the features. We finally evaluated the performances of every approach in terms of Average Solving Time (AST) and Percentage of Solved Instances (PSI). When a portfolio strategy was not able to solve an instance, we set its solving time to the time cap T.

3.3 Test Results

Before comparing SUNNY w.r.t. other approaches, we performed a sensitivity analysis by tuning the k parameter. As depicted in Fig. 1a the robustness of SUNNY is reflected by the fact that varying the value of k in $[5, 20]$ does not entail a huge impact in performance (i.e., less than 1% of solved instances). The peak performances ($PSI = 77.81\%$) are reached with $k = 16$, while for $k > 20$ we observed a gradual performance degradation. For instance, by using $k = 25, 50, 100, 250, 500$ we get a maximum PSI of $77.59, 77.55, 77.3, 77.16, 76.56$, and 72.63 respectively (see Figure 1b).

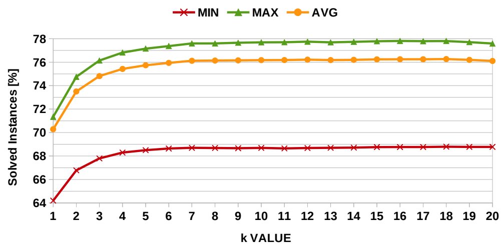
We then selected the version of SUNNY with $k = 16$ as the best one and we compared it against 3S, SATzilla, and CPHydra approaches in terms of Percentage of Solved Instances (PSI) and Average Solving Time (AST). As a baseline, we considered the performances of the *Virtual Best Solver* (VBS), an 'oracle' portfolio solver which always selects the best solver for a given instance. As shown in Fig. 2, SUNNY turns out to be the best among all other approaches. Indeed, it can reach peaks of 77.81% solved instances against the 76.86% of 3S, the 75.85% of SATzilla, and the 73.21% of CPHYdra. If compared with the Single Best Solver (SBS), i.e., the best constituent solver in terms of number of solved instances,[6] SUNNY is able to close up to the 93.38% of the gap between it and the VBS. Moreover, the performances of SUNNY do not degrade at the increase of the portfolio size. Indeed the best performance of SUNNY, both in terms of PSI and AST, is reached using a portfolio of 11 solvers.

As depicted in Fig. 3a, the size of the sub-portfolios computed by SUNNY is usually small:

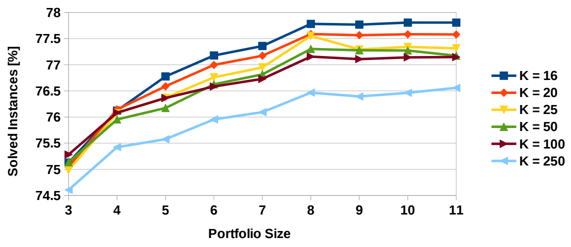
solvers comparison are broken by selecting the solver that in general solves the largest number of instances. Moreover no presolver selection or other parameters tuning was performed. To reproduce 3S, instead of using the original column generation method (Kadioglu et al. 2011), we solved the scheduling problem imposing an additional constraint that requires every solver to be run for an integer number of seconds. No parameter tuning was performed even in this case. For more details we defer the interested reader to Amadini et al. (2013a).

[5] CPHydra needs to solve a NP-hard problem to decide the schedule of solver to use. Computing the schedule can take, in few cases, more than half an hour for portfolios with more than 8 solvers.

[6] The SBS for the selected benchmark was MinisatID having a PSI of 51.62% and an AST of 950.51 seconds.



(a) Minimum, maximum, and average of the PSI reached by SUNNY on all the considered portfolios, by ranging the k parameter in $[1,20]$.



(b) PSI of SUNNY with $k = 16, 20, 25, 50, 100, 250$.

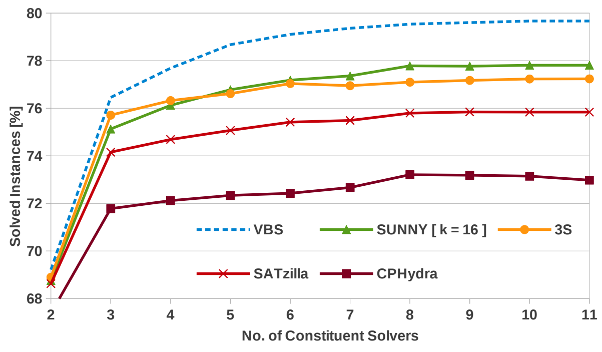
Fig. 1: PSI of SUNNY varying the k parameter.

their average is between 1.21 and 1.44. This means that it can be reasonable to limit the sub-portfolio size without compromising the overall performances. A further study of this issue is left as a future work.

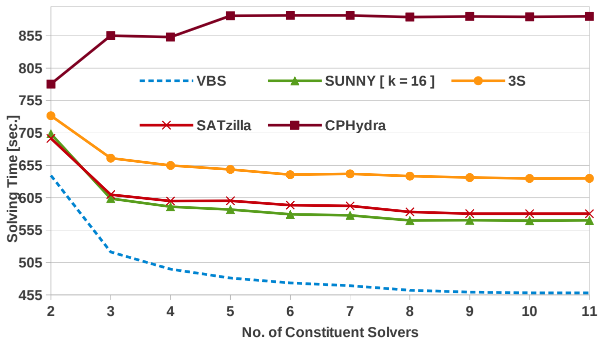
We believe that the reasons behind the good performance of SUNNY lie in the quality of the features extracted as well as in the selection criteria of the schedules it computes. For example, Fig. 3b shows the comparison of SUNNY against two other baselines: KNN and EQU. KNN is the portfolio approach that for every instance selects the solver that solves more instances in the neighborhood, using the solving time for tie-breaking. EQU is instead an approach that simply allocates to every constituent solver of the portfolio an equal amount of time (Pulina and Tacchella 2009). Fig. 3b shows that, fixing the neighborhood size to 16, the performances of these two approaches are significantly worse w.r.t. to the peak performances of SUNNY.

4 `sunny-csp`

The results so far described are based on simulations. We decided to simulate the approaches because running every portfolio solver on each fold/repetition would take a tremendous amount of time (i.e., every single approach needs to be validated on $4642 * 5 = 23210$ instances). Clearly, it is legitimate to ask if these simulations are faithful. Discrepancies may arise from factors not considered in the simulation like, for instance, the presence of memory leaks, solvers erratic behaviors, or solver faults. Moreover, since some of the features are dynamic, executing again

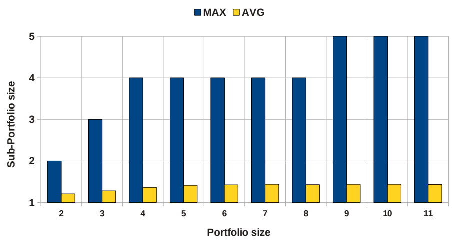


(a) Percentage of Solved Instances.

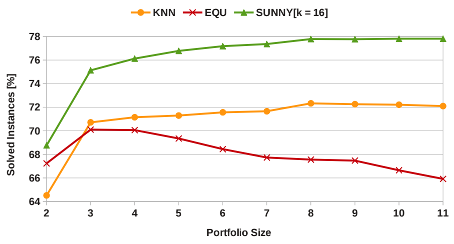


(b) Average Solving Time.

Fig. 2: Performances of SUNNY, 3S, SATzilla, and CPHYdra.



(a) Average and maximum sub-portfolio sizes of SUNNY ($k = 16$).



(b) Performances of KNN and EQU against SUNNY ($k = 16$).

Fig. 3: SUNNY sub-portfolio sizes and comparison against KNN and EQU.

the feature extraction process may lead to slightly different features that may cause a deviance on the expected performance.

Driven by these motivations, we therefore decided to develop and test `sunny-csp`: a CSP portfolio built on the top of the SUNNY algorithm. `sunny-csp` essentially uses the pseudo-code of Listing 1 exploiting the features extractor described in Amadini et al. (2014a). Taking as reference the results of Section 3, `sunny-csp` implements SUNNY by setting $k = 16$, by exploiting a portfolio of 11 solvers (i.e., all the solvers listed in paragraph 3.1), and by using MinisatID as a backup solver. If a scheduled solver prematurely terminates its execution, `sunny-csp` assigns the remaining time to the next scheduled solver if any, otherwise it iteratively selects the solver not yet executed that solves the greatest number of instances of the training set, until either the instance is solved or the timeout expires. In order to measure and compare the actual performance of `sunny-csp` w.r.t. its simulated performance we used the same training sets, the same timeout, and the same machines adopted for the simulations.

Table 3 reports a comparison between the simulated (i.e., ideal) performance of SUNNY and the actual performance of `sunny-csp` for each repetition and fold in terms of PSI and AST. As one can see, the real performance are very close to the expected ones. In particular, `sunny-csp` solves on average only 0.07% instances less than predicted (see Table 4). There are cases in which it is also better than expected (see repetition 4 of Table 4). Looking at the AST statistics (Table 5), we noted a substantial equivalence between the two approaches (even if on average `sunny-csp` is slightly better, a mean difference of 1.78 seconds appears quite insignificant).

Table 3: `sunny-csp` ideal and actual performance for each repetition and fold.

		IDEAL		**ACTUAL**		**ACTUAL − IDEAL**	
Rep.	**Fold**	**PSI**	***AVG***	**PSI**	***AVG***	**PSI**	***AVG***
1	1	75.78		75.24		−0.54	
1	2	78.69		78.15		−0.54	
1	3	78.77		78.77		0.00	
1	4	78.13		78.23		0.11	
1	5	78.23	*77.92*	78.13	*77.70*	−0.11	*−0.22*
2	1	78.90		79.22		0.32	
2	2	76.64		76.53		−0.11	
2	3	78.77		78.77		0.00	
2	4	78.56		77.80		−0.75	
2	5	76.08	*77.79*	76.08	*77.68*	0.00	*−0.11*
3	1	78.79		78.69		−0.11	
3	2	76.32		76.32		0.00	
3	3	78.56		78.13		−0.43	
3	4	78.13		78.66		0.54	
3	5	77.37	*77.83*	77.16	*77.79*	−0.22	*−0.04*
4	1	78.04		78.04		0.00	
4	2	75.35		75.78		0.43	
4	3	78.66		78.56		−0.11	
4	4	78.23		78.23		0.00	
4	5	78.02	*77.66*	77.91	*77.70*	−0.11	*0.04*
5	1	77.93		78.04		0.11	
5	2	78.36		78.47		0.11	
5	3	77.05		76.94		−0.11	
5	4	78.56		78.45		−0.11	
5	5	77.26	*77.83*	77.05	*77.79*	−0.22	*−0.04*
		Total Average	***77.81***	***Total Average***	***77.73***	***Total Average***	***−0.07***

Table 4: Percentage of Solved Instances.

		IDEAL		**ACTUAL**		**ACTUAL − IDEAL**	
Rep.	**Fold**	**AST**	***AVG***	**AST**	***AVG***	**AST**	***AVG***
1	1	610.03		618.39		8.37	
1	2	561.75		567.91		6.16	
1	3	545.77		549.88		4.11	
1	4	550.57		545.54		−5.04	
1	5	571.98	*568.02*	572.29	*570.80*	0.31	*2.78*
2	1	564.72		559.29		−5.43	
2	2	577.05		581.86		4.81	
2	3	540.97		537.76		−3.21	
2	4	557.87		565.39		7.52	
2	5	608.41	*569.80*	602.52	*569.36*	−5.89	*-0.44*
3	1	561.18		557.79		−3.39	
3	2	609.80		607.50		−2.30	
3	3	564.97		568.77		3.80	
3	4	556.06		542.47		−13.60	
3	5	560.65	*570.53*	555.87	*566.48*	−4.78	*-4.05*
4	1	566.71		571.09		4.38	
4	2	599.00		589.31		−9.68	
4	3	551.58		549.03		−2.55	
4	4	581.12		578.40		−2.71	
4	5	552.34	*570.15*	549.28	*567.42*	−3.06	*-2.73*
5	1	571.06		566.74		−4.32	
5	2	558.51		557.11		−1.41	
5	3	580.12		579.47		−0.66	
5	4	581.40		570.17		−11.23	
5	5	568.84	*571.99*	564.09	*567.51*	−4.75	*-4.47*
		Total Average	***570.10***	***Total Average***	***568.32***	***Total Average***	***−1.78***

Table 5: Average Solving Time.

In conclusion we have shown that `sunny-csp` performance are basically equivalent to the expected peak performance of SUNNY. In particular, focusing on the individual performance of the constituent solvers we can argue that `sunny-csp` is on average able to solve 26.11% instances more than the Single Best Solver, with an average solving time of 382.59 seconds less.

The source code of `sunny-csp`, as well as of the scripts we used to conduct the experiments, is fully available and downloadable at `http://www.cs.unibo.it/~amadini/iclp_2014.zip`.

5 Related Work

Portfolios approaches have been used in different areas ranging from SAT to optimization problems like Mixed Integer Programming, Scheduling, Most Probable Explanation (MPE) and Travel Salesman Problem (TSP). For a survey of the different portfolio approaches studied in the literature we defer the interested reader to the comprehensive surveys by Smith-Miles (2008),Kotthoff (2012),Hutter et al. (2012) or more specifically to Amadini et al. (2013a) for CSPs, to Amadini et al. (2014b) for Constraint Optimization Problems, and to Gebser et al. (2011) for ASP.

Here we focus just on the most promising sequential approaches (i.e., winners of SAT and CSP competitions) not considering, for instance, parallel portfolio approaches where more than one solver is run concurrently.

CPHydra (OMahony et al. 2009) is currently the only CSP solver which uses a portfolio approach. For the feature extraction it uses the code of Mistral, one of its constituent solvers, extracting from every instance 36 static and dynamic features. A k-nearest neighbor algorithm is used in order to compute a schedule of solvers which maximizes the chances of solving an instance within a time-out of 1800 seconds. The schedule is computed solving an optimization problem that is NP-hard. Nevertheless, CPHydra was able to win the 2008 International CSP Solver Competition. Unfortunately, a direct comparison between CPHydra and `sunny-csp` is not immediately possible since CPHydra process only instances encoded in the (old) input format XCSP, it uses solvers not maintained anymore, and it does not provide an API to change the training sets used for compute the schedule. Despite the similarities between CPHydra and `sunny-csp`, there are however some notable differences. Apart for the simulated performance gap in terms of PSI (`sunny-csp` can solve almost 5% instances more than the simulation of CPHydra), CPHydra does not try to minimize the AST since - conversely to SUNNY - it doesn't use any heuristic to determine the order of the solvers to be run.[7]

SATzilla (Xu et al. 2007) is a SAT solver that relies on runtime prediction models to select the solver that (hopefully) has the fastest running time on a given problem instance. Its last version (Xu et al. 2012), which consistently outperforms the previous ones, uses a weighted random forest approach provided with an explicit cost-sensitive loss function punishing misclassifications in direct proportion to their impact on portfolio performance. SATzilla won the 2012 SAT Challenge in the Sequential Portfolio Track.

3S (Kadioglu et al. 2011) is instead a SAT solver that conjugates a fixed-time static solver schedule with the dynamic selection of one long-running component solver. 3S solves the scalability issues of CPHydra because the scheduling is computed offline and covers only 10% of the time limit. If a given instance is not yet solved after the short runs, a designated solver is chosen at runtime (using a k-nearest neighbors algorithm) and executed. 3S originally used a portfolio of 21 SAT solvers and was the best-performing dynamic portfolio at the International SAT Competition 2011.

More recently, a brand new portfolio approach was proposed by Malitsky et al. (2013). This portfolio improves 3S by using its schedule for 10% of the available time and then runs for the remaining time a solver selected based on a Cost-Sensitive Hierarchical Clustering (CSHC). The CSHC solver won 2 gold medals in the last SAT competition 2013. Clearly, an evaluation of CSHC - properly adapted to CSP - is in our interest for its comparison w.r.t. SUNNY. Unfortunately, this was not immediately possible since the code of CSHC is not publicly available.

[7] Conversely to the other approaches, the AST of CPHydra is not anti-correlated to its PSI.

The ASP field present a lot of similarities w.r.t. the state of research in the CSP field. Despite the development of portfolios is not as well studied as in SAT, there exist some portfolio approaches. Maratea et al. (2012) developed ME-ASP, a multi-engine solver for propositional ASP programs that apply ML techniques in order to inductively choose the "best" constituent solver to be run. Analogously on what done by the configuration tool ISAC (Kadioglu et al. 2010) for SAT problems, some works try instead to exploit algorithms portfolio to optimally tune the parameters of ASP solvers. In particular, in (Gebser et al. 2011) the solver clasp is improved by using Support Vector Regression for choosing a good configuration. A framework that allows to learn and use domain-specific heuristics for choice-point selection is instead presented in Balduccini (2011) for improving the performance of SMODEL solver.

Related to SUNNY is also the work performed by Hoos et al. (2012) that devise an approach that takes advantage of the modeling and solving capacities of ASP to automatically determine a schedule from existing benchmarking data without rely on any domain-specific features.

6 Conclusions and Extensions

In this work we presented SUNNY, a simple yet effective algorithm portfolio that, without explicitly building and learning any specific model, allows one to compute a sequential schedule of the constituent solvers of a portfolio for solving a given CSP.

Despite its simplicity, preliminary results have shown that this approach is promising and can even outperform the state of the art CSP portfolio techniques. To get a more realistic comparison we implemented `sunny-csp`, a CSP portfolio solver that exploits the SUNNY algorithm in order to solve a given MiniZinc instance by using a portfolio of 11 solvers. Empirical evidences confirm both the effectiveness of SUNNY and the potential benefits of using a portfolio of solvers: indeed, `sunny-csp` greatly outperforms each of its constituent solvers.

In view of the promising preliminary results, the possible extensions of SUNNY are manifold. First, its flexibility and usability may naturally lead to build effective portfolios also outside the CSP domain. In fact, SUNNY may be easily adapted to other domains such as SAT and ASP. As an example, SUNNY performs well even when adapted to Constraint Optimization Problems (Amadini et al. 2014b). Moreover, even if in this work we focused only on sequential portfolios, the scheduling-based approach of SUNNY is well suited for the construction of parallel portfolios. An interesting future direction may be to consider the impact of choosing different features and/or distance metrics for the *k*-NN algorithm.

Constraint Logic Programming (CLP) can exploit the speed up of the search of solution provided by SUNNY. Indeed, by adapting the same techniques described in Cipriano et al. (2008), from a CLP it is possible to derive a MiniZinc instance that can be solved by `sunny-csp`.

Another interesting direction is to dynamize SUNNY algorithm in order to make it an online portfolio approach. Currently, SUNNY is indeed based on a static and pre-computed knowledge base. A possible extension consists in allowing the dynamic update of the knowledge base, thus exploiting new incoming information such as new instances, solvers, or features.

Finally, we are planning to improve the usability and the portability of `sunny-csp`, extending it to the resolution of also optimization problems in order to possibly enrolling it to a MiniZinc Challenge.

References

AMADINI, R., GABBRIELLI, M., AND MAURO, J. 2013a. An Empirical Evaluation of Portfolios Approaches for Solving CSPs. In *CPAIOR*. Lecture Notes in Computer Science, vol. 7874. Springer.

AMADINI, R., GABBRIELLI, M., AND MAURO, J. 2013b. Features for Building CSP Portfolio Solvers. *CoRR abs/1308.0227*.

AMADINI, R., GABBRIELLI, M., AND MAURO, J. 2014a. An Enhanced Features Extractor for a Portfolio of Constraint Solvers. In *SAC*.

AMADINI, R., GABBRIELLI, M., AND MAURO, J. 2014b. Portfolio Approaches for Constraint Optimization Problems. In *LION*.

ARLOT, S. AND CELISSE, A. 2010. A survey of cross-validation procedures for model selection. *Statistics Surveys 4*, 40–79.

BALDUCCINI, M. 2011. Learning and using domain-specific heuristics in asp solvers. *AI Commun. 24,* 2, 147–164.

CIPRIANO, R., DOVIER, A., AND MAURO, J. 2008. Compiling and executing declarative modeling languages to gecode. In *ICLP*. Lecture Notes in Computer Science, vol. 5366. Springer, 744–748.

CSP Competition 2009. Third International CSP Solver Competition 2008. `http://www.cril.univ-artois.fr/CPAI09/`.

DECAT, B. 2013. KRR Software: MinisatID. `http://dtai.cs.kuleuven.be/krr/software/minisatid`.

GEBRUERS, C., GUERRI, A., HNICH, B., AND MILANO, M. 2004. Making Choices Using Structure at the Instance Level within a Case Based Reasoning Framework. In *CPAIOR*. Lecture Notes in Computer Science, vol. 3011. Springer, 380–386.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., SCHAUB, T., SCHNEIDER, M. T., AND ZILLER, S. 2011. A Portfolio Solver for Answer Set Programming: Preliminary Report. In *LPNMR*. Lecture Notes in Computer Science, vol. 6645. Springer, 352–357.

GOMES, C. P. AND SELMAN, B. 2001. Algorithm portfolios. *Artif. Intell. 126,* 1-2, 43–62.

HOOS, H., KAMINSKI, R., SCHAUB, T., AND SCHNEIDER, M. T. 2012. aspeed: ASP-based Solver Scheduling. In *ICLP (Technical Communications)*. 176–187.

HUTTER, F., XU, L., HOOS, H. H., AND LEYTON-BROWN, K. 2012. Algorithm Runtime Prediction: The State of the Art. *CoRR abs/1211.0906*.

KADIOGLU, S., MALITSKY, Y., SABHARWAL, A., SAMULOWITZ, H., AND SELLMANN, M. 2011. Algorithm Selection and Scheduling. In *CP*. Lecture Notes in Computer Science, vol. 6876. Springer.

KADIOGLU, S., MALITSKY, Y., SELLMANN, M., AND TIERNEY, K. 2010. ISAC - Instance-Specific Algorithm Configuration. In *ECAI*. Frontiers in Artificial Intelligence and Applications, vol. 215. IOS Press.

KOTTHOFF, L. 2012. Algorithm Selection for Combinatorial Search Problems: A Survey. *CoRR abs/1210.7959*.

MACKWORTH, A. K. 1977. Consistency in Networks of Relations. *Artif. Intell. 8,* 1, 99–118.

MALITSKY, Y., SABHARWAL, A., SAMULOWITZ, H., AND SELLMANN, M. 2013. Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. In *IJCAI*. IJCAI/AAAI.

MALITSKY, Y. AND SELLMANN, M. 2012. Instance-Specific Algorithm Configuration as a Method for Non-Model-Based Portfolio Generation. In *CPAIOR*. Lecture Notes in Computer Science, vol. 7298. Springer.

MARATEA, M., PULINA, L., AND RICCA, F. 2012. The Multi-Engine ASP Solver me-asp. In *JELIA*. Lecture Notes in Computer Science, vol. 7519. Springer, 484–487.

NIKOLIC, M., MARIC, F., AND JANICIC, P. 2009. Instance-Based Selection of Policies for SAT Solvers. In *SAT*. Lecture Notes in Computer Science, vol. 5584. Springer, 326–340.

OMAHONY, E., HEBRARD, E., HOLLAND, A., NUGENT, C., AND OSULLIVAN, B. 2009. Using case-based reasoning in an algorithm portfolio for constraint solving. *AICS 08*.

PULINA, L. AND TACCHELLA, A. 2007. A Multi-engine Solver for Quantified Boolean Formulas. In *CP*. Lecture Notes in Computer Science, vol. 4741. Springer, 574–589.

PULINA, L. AND TACCHELLA, A. 2009. A self-adaptive multi-engine solver for quantified boolean formulas. *Constraints 14,* 1, 80–116.

RICE, J. R. 1976. The Algorithm Selection Problem. *Advances in Computers 15*, 65–118.

SAMULOWITZ, H., REDDY, C., SABHARWAL, A., AND SELLMANN, M. 2013. Snappy: A simple algorithm portfolio. In *SAT*. Lecture Notes in Computer Science, vol. 7962. Springer, 422–428.

SAT Challenge. SAT Challenge 2012. `http://baldur.iti.kit.edu/SAT-Challenge-2012/`.

SMITH-MILES, K. 2008. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv. 41,* 1.

STERN, D. H., SAMULOWITZ, H., HERBRICH, R., GRAEPEL, T., PULINA, L., AND TACCHELLA, A. 2010. Collaborative expert portfolio management. In *AAAI*. AAAI Press.

WILSON, D., LEAKE, D., AND BRAMLEY, R. 2000. Case-Based Recommender Components for Scientific Problem-Solving Environments. In *In Procs. of the 16th International Association for Mathematics and Computers in Simulation World Congress*.

XU, L., HUTTER, F., HOOS, H. H., AND LEYTON-BROWN, K. 2007. SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT. In *CP*. Lecture Notes in Computer Science, vol. 4741. Springer.

XU, L., HUTTER, F., SHEN, J., HOOS, H., AND LEYTON-BROWN, K. 2012. SATzilla2012: Improved algorithm selection based on cost-sensitive classification models. Solver description, SAT Challenge 2012.