Why CP Portfolio Solvers Are (under)Utilized? Issues and Challenges

Roberto Amadini, Maurizio Gabbrielli, Jacopo Mauro

Why CP Portfolio Solvers Are (under)Utilized? Issues and Challenges*

Roberto Amadini[1], Maurizio Gabbrielli[1], and Jacopo Mauro[2]

[1] Department of Computer Science and Engineering/Lab. Focus INRIA, University of Bologna, Italy.
[2] Department of Informatics, University of Oslo.
{amadini, gabbri}@cs.unibo.it jacopom@ifi.uio.no

Abstract. It is well recognized that a single, arbitrarily efficient solver can be significantly outperformed by a *portfolio solver* exploiting a combination of possibly slower on-average different solvers. Despite the success of portfolio solvers within the context of solving competitions, they are rarely used in practice. In this paper we give an overview of the main limitations that hinder the practical adoption and development of portfolio solvers within the Constraint Programming (CP) paradigm, discussing also possible ways to overcome them and potential extensions outside the CP field.

1 Introduction

Solving combinatorial search problems is hard, and there exist nowadays plenty of techniques and constraint solvers for performing this task. It has become clear that different solvers are better when solving different problem instances, even within the same problem class. It has also been shown that a single, arbitrarily efficient solver can be significantly outperformed by using a *portfolio* of possibly on-average slower solvers.

Algorithm portfolios [25] can be seen as instances of the more general *Algorithm Selection* problem [57] where, as reported in [42], the algorithm selection is performed case-by-case for each problem to solve. Within the context of constraint solving, a portfolio approach enables to combine a number $m > 1$ of different constituent solvers $s_1, \ldots, s_m$ in order to create a globally better constraint solver, dubbed a *portfolio solver*. When a new, unseen problem p comes, the portfolio solver tries to predict the best constituent solver(s) $s_{i_1}, \ldots, s_{i_k}$ (with $1 \leq i_j \leq m$ for $j = 1, \ldots, k$) for solving p and then runs them on p. Properly selecting and scheduling the solvers is a crucial step for the performance of a portfolio solver, and it is usually performed by exploiting *Machine Learning* techniques based on features extracted from the problem p to solve.

We can safely say that portfolio approaches have proven to be particularly effective within the context of solving challenges. For instance, the SAT portfolio solvers 3S [38] and CSHC [46] won gold medals in the SAT Competition 2011 and 2013 respectively. SATZilla [72] won the SAT Challenge 2012. CPHydra [54] was the winner of the International Constraint Solver Competition 2008. The ASP portfolio solver claspfolio [17]

* Supported by the EU project FP7-644298 *HyVar: Scalable Hybrid Variability for Distributed, Evolving Software Systems*.

was gold medalist in different tracks of the ASP Competition 2009 and 2011. ArvandHerd [70] and IBaCoP [14] won some tracks in the Planning Competition 2014, where 29 out of 67 solvers were portfolio-based. Surprisingly enough, despite the remarkable results achieved in such challenges, portfolio solvers have been in general poorly adopted in the real word. So, a question naturally arises: why portfolio approaches are scarcely used outside the walls of solving competitions?

In this paper we tackle this problem by focusing in particular on portfolio approaches within the *Constraint Programming* (CP) paradigm, where the goal is to model and solve *Constraint Satisfaction Problems* (CSPs) as well as the more general *Constraint Optimisation Problems* (COPs). From this perspective the state of the art of CP portfolio solvers is still a raw fruit if compared, e.g., to the SAT field where a number of effective portfolio approaches have been developed and tested. As an example, the first and the only portfolio solver that won a MiniZinc Challenge [67] —the only still active competition for evaluating CP solvers— has been `sunny-cp` [6] in 2015. There are certainly a number of difficulties because of which many users prefer to take refuge in a more classical "single-solver" approach, rather than relying on portfolio solvers. However, we believe that in a not negligible number of cases a proper combination of different solvers might significantly improve the solving process. Our goal is therefore trying to reduce the obstacles that hinder the practical adoption and development of portfolio approaches. Among the various issues, we identified four main challenges for the future of CP portfolio solvers:

- *prediction model* (Sect. 2): what are the scientific and engineering issues that arise when building or using the prediction model responsible for the solver selection;
- *optimisation problem* (Sect. 3): how to apply the portfolio theory to COPs, being the state of the art in this field still in an embryonic stage;
- *parallelisation* (Sect. 4): how to exploit different processing units, possibly running in parallel more than one constituent solver;
- *utilisation* (Sect. 5): how to facilitate the practical use of portfolio solvers for solving generic CP problems.

In the rest of the paper we will explain in more detail these issues, by discussing possible ways to overcome them and providing also proposals for future directions, such as for example the extension of portfolio solving outside the CP field.

2 Prediction Model

With the term "*prediction model*" we refer to the set of data, knowledge and algorithms required to predict and run the best solver(s) for solving a new CP problem. In this section we focus in particular on three key components of the prediction model: the dataset of problems used to make (and test) the predictions, the features used to characterize each problem, and the algorithms used to perform the solver selection.

The basic framework of a prediction model is summarized in Figure 2. When a new, unseen problem p needs to be solved, the *feature vector* $FV = (F_1, \ldots, F_d)$ of p is firstly computed. Broadly speaking, FV is a collection of $d \geq 0$ numerical attributes that characterize the problem p (e.g., statistics over the problem structure). Then, a

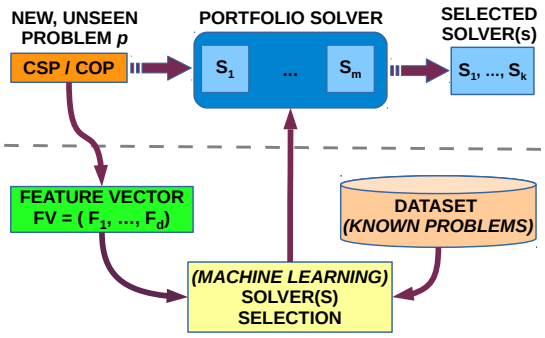

Fig. 1. Basic components of the prediction model.

subset $s_{i_1}, \ldots, s_{i_k}$ of $1 \leq i_j \leq m$ solvers of the portfolio $\{s_1, \ldots, s_m\}$ is selected and executed according to FV and to a dataset of already known problems on which the portfolio solver is trained. Note that, although the solver selection is usually performed by means of Machine Learning algorithms, the use of Machine Learning is not strictly necessary. For example, we could define a purely static prediction model that for every instance p always runs a schedule of solvers which is pre-computed *a priori*, regardless of p. In this case no prediction is needed.

The prediction should be transparent for the end user, i.e., the user should run the portfolio solver on p just like a regular, individual solver without worrying about the underlying structure of the model.

2.1 Dataset

The performance of a portfolio solver is strongly dependent on the choice of the set of problem instances used to perform the solver selection. The difficulties of choosing a suitable dataset are well recognized [57]. If we restrict ourselves to the CP field, the first issue is the lack of a standard language. Differently from SAT, ASP, and Planning, no standard format exists for specifying a CP problem. This problem affects individual CP solvers and, *a fortiori*, represents a major obstacle for defining and comparing portfolio solvers. Lately the CP community seems to converge on *MiniZinc* language [52] but, despite more than 8000 MiniZinc instances are publicly available, the standardisation process looks far from over. Other formats like XCSP [59] and Essence [20] are still in use, and *ad hoc* solver-specific languages are widely adopted. Even the natural language is used for the problem specification (e.g., see the well-known CSPLib library [23]).

Clearly we propose to converge to a common language, whether it be MiniZinc or any other. This should foster the definition of compilers/interpreters for switching from the standard language to the preferred target language (e.g., in [2] we introduced the `xcsp2mzn` compiler for converting an XCSP model to MiniZinc).

Assuming to have a standard language, even having standard datasets of problems is a desirable goal. This can be useful not only for making better predictions, but also for a

fair performance comparison of different solvers. The *Algorithm Selection Library* [13] is currently addressing this task by collecting and standardizing datasets coming from different algorithm selection scenarios. A good starting point for the creation of CP standard benchmarks might consist in using the instances of the MiniZinc Challenge, but also other choices might be equally justifiable (e.g., using the more extensive benchmarks of the International Constraint Solver Competition 2008/2009).

Having a too large dataset may also be counterproductive: including without criteria all the available instances is a poor choice, since it may create noise and hinder the predictions accuracy. A reasonable approach to construct a dataset consists in grouping the problems by their nature, difficulty, and origin and add few representatives per group into the dataset. Since such a classification may require a considerable human effort, a promising direction for future works concerns the automation of the dataset construction. For instance, in [31] a dataset of SAT problems is automatically generated by means of a clustering algorithm.

2.2 Features

The concept of feature is crucial for algorithm selection. Features are instance-specific attributes that characterize a given problem. Early as 1976, Rice stated that *"The determination of the best (or even good) features is one of the most important, yet nebulous, aspects of the algorithm selection problem"* [57]. Features can be categorized in static and dynamic [42]. In the context of CP solving, static features are computed off-line by parsing the input problem (e.g., statistics over the variables and the constraints of the problem). Dynamic features are instead collected by retrieving information at runtime (e.g., the number of propagation performed or nodes explored in the search tree).

A weakness of dynamic features regards the limited portability. Running a solver for short runs makes the features dependent on the architecture on which the solver is executed. This may distort the predictions, which may change when performed by different machines on the same problem.

From a problem specification it is possible to collect hundreds of features (see for instance [2, 9, 19, 36]) but, as for the dataset construction, care should be taken for avoiding to retrieve redundant and noisy knowledge. As also shown in [19, 44] usually a very small subset of features is really needed.

There is an extensive literature concerning the problem of *feature selection* [27], i.e, the problem of selecting the most significant features for a prediction model. Indeed, a common issue for many portfolio approaches consists in using features that do not properly characterize a given problem. Feature selection can be a very costly task that might result in negligible gains —or even deterioration— of performance when not properly performed. The selection procedure can be safely performed off-line, but turns out to be almost infeasible when the behaviour of the portfolio solver can not be simulated. For instance, in a COP setting the side effects of bounds communication are not predictable in advance [7] and thus can not be safely simulated. In these cases it is desirable to use a selection algorithm the more robust as possible w.r.t. redundant features.

The cost of feature extraction plays an important role. The time needed to compute the features should be minimized, since every second of feature computation is a second taken from the portfolio solver execution. For instance, extracting features based

on constraint graphs can be very time and space consuming. Beyond the "classic" scientific challenges relating to the feature identification and selection —which are typical of the Machine Learning field— as a future challenge we also propose to not overlook more practical problems like the computational tractability and the usability of features. In this respect we developed the feature extractor `mzn2feat-1.0` [51], that improves a previous version of `mzn2feat` [2] with the aim of being more portable, light-weight, flexible, and independent from the particular machine on which it is run. To the best of our knowledge, this is the only publicly available tool able to extract features from a generic MiniZinc model: of course, we welcome every other analogous tool for retrieving and selecting new, significant features.

2.3 Solver Selection

Selecting the best solver to run on a given problem is a critical issue, clearly related to the available dataset and the features considered for each problem of the dataset. In this context, classification techniques appear to be more robust than regression ones for the runtime prediction (e.g., see [48,55,71]). A common drawback of portfolio approaches is that typically a prediction model is built by first running each solver of the portfolio on every instance of the dataset. This task typically requires weeks of computations, and it is not very flexible and portable. For example, having a new (version of a) solver means re-running such solver on all the problems. Moreover, the runtime information computed during the training phase on a given machine may be no longer significant when the portfolio solver is run on another machine. An interesting direction to follow is shown in [65], where the prediction model is built by using only short runs of the constituent solvers on the training instances.

Another issue concerns the explicit construction of a prediction model. As pointed out also in [61], portfolio solvers usually require a complex off-line training phase for selecting the solver(s) to run. For example, SATzilla [71] uses a weighted Random Forest machine learning approach while CHSC [46] clusters the instances of the training set. Despite the proven effectiveness of these approaches, we think that a major challenge is to lighten as much as possible the training phase. In [3,54,61] the authors show that also "training-less" approaches can be competitive w.r.t. those that build an explicit prediction model. Another interesting direction to consider when building a prediction model is to use alternatives to the supervised learning approach. Techniques such as self-training and co-training already used for example for QBF [56] and ASP [47] may simplify the construction of the prediction model.

Of course, even the time needed for the on-line selection of the solvers to run must be considered. For example, a prediction model with a negligible building cost might be totally useless if selecting the solver(s) to run for a new instance takes an unreasonable amount of time. Also having a huge dataset may involve a time dilation in the solvers selection process. For instance, the time required by using an approach based on k-*Nearest Neighbour* [18] classification depends on the size of the dataset since it requires the scan of the whole dataset.

Note that being able to reduce the effort of building the prediction model allows one to quickly adapt the prediction to new unseen problems or to exploit new available solvers. This is of particular importance since real life applications usually focus on

solving a specific class of problems that may be not fully known in advance. Moreover, as the results of the Learning track of the Planning Competition 2014 [37] show, learning from new incoming instances may dramatically increase the performance.

3 Optimisation Problems

Optimisation problems are of great interest in many real life applications where we are interested in finding an optimal (or good enough) solution. If CSP portfolio solvers can draw inspiration from SAT approaches —possibly through an encoding into SAT— in a straightforward way, when dealing with COPs the matter becomes more complicated. Unlike CSPs, here the dichotomy solved/not solved is no longer suitable since a COP solver can provide *sub-optimal solutions* without finding the optimal one (or proving its optimality).

The first issue here is the lack of a universally accepted metric for measuring the performance of a COP solver, and therefore for building COP prediction models. Since for hard combinatorial problems it is often very difficult to complete the search in reasonable time, it is clear that the solution quality must be taken into account. What is less clear is *how* to do it. There are plenty of metrics, used in well known solving competitions, for evaluating the performance of COP solvers. In [49, 62] the solvers are ranked by using a lexicographic order over the solution quality, the number of the solved instances, and the solving time. The purse score described in [12] was used in the SAT Competition 2005, while a metric exploiting results aggregation and pair-wise comparisons between solvers is proposed in [22]. The MiniZinc Challenge uses instead a *Borda* count voting system: problems are treated like voters who rank the solvers. This approach is surely reasonable, but in our opinion has a disadvantage: it could overestimate small time differences in case of easy instances, as well as underrate big time differences in case of medium and hard instances. In [5, 7] we proposed and evaluated alternative metrics that take into account both the solution quality (i.e., the `score` metric) and the anytime performance of the solvers (i.e., the `area` metric) without relying on cross-comparisons between the solvers.

The possibility of producing sub-optimal solutions is a key factor worth investigating. We argue that collaborative strategies can be successfully adopted by COP portfolio solvers. Indeed, some solvers quickly find good sub-optimal solutions but fail to improve later, while others are slower but in the end find better solutions. In this setting a solver can exploit the objective function bounds found by another solver to reduce its search space, as done for instance in [7]. As an example, consider the behaviours of the solvers s_1 and s_2 in Figure 2 within 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. From a graphical point of view, this means in some way 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

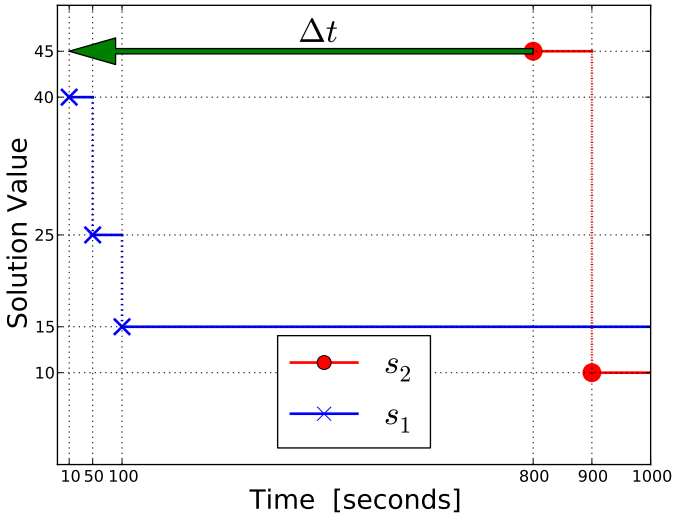

Fig. 2. Example of bound communication from s_1 to s_2.

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 to exploit the remaining Δt seconds for finding better solutions or even proving the optimality of v^*. However, note that this *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).

The decision of switching between the solvers can be made statically, as done in [7], but also dynamically at run time. It may be counterproductive to stop a solver if it is actively producing new solutions while it is likely that it will not produce solutions if no solutions are produced so far. Of course, the behaviour of a solver depends on its nature and on the problem to be solved. We believe that interesting patterns can emerge by studying the problems and the solvers in a more "qualitative" way rather than performing only "quantitative" observations. From a practical point of view, *monitoring* the sub-optimal solutions found by the running solvers might be the starting point for the definition of more dynamic portfolios, where decisions are made during the search instead of relying on costly prediction models. We are currently examining this approach, that we consider promising especially in parallel settings.

Making tons of experiments on a large amount of data is certainly significant from a statistical point of view, but somehow hinders the understanding of what we are experimenting. Indeed, as mentioned in [42], often we observe and evaluate the performance of different algorithms without being able to give a full *explanation* for such performance. Hopefully, looking at the COP solvers behaviour more in depth could give us some explanation and hints on how to better combine the different solvers.

	CPX	FD	LazyFD	MIP	par-folio	obj-folio
Best Value Found	No Value	No Value	10	10	10	10
Optimisation Time [sec.]	Timeout	Timeout	Timeout	182.59	341.49	**16.71**

Table 1. Solvers outcomes on a bin-packing instance.

4 Parallelisation

Having a finite portfolio, its parallelisation would seem a trivial issue: you only need to run in parallel all the solvers. Unfortunately, often the number of the constituent solvers exceeds the number of available cores. Furthermore, even assuming to have fewer solvers than cores, it is likely that —due to synchronisation and memory contention issues— running in parallel all the solvers on the same multicore machine is actually different from running the same solvers on different machines [60].

In the SAT field parallel portfolios have been extensively studied. Usually, different configurations of the same solver are run simultaneously by enabling the sharing of learned clause between solvers [10,28,58]. Conversely, the parallelisation of CP solvers does not appear currently so fruitful. For example, in the MiniZinc Challenges 2014/15 the possibility of multiprocessing did not lead in general to remarkable performance gains despite the availability of 8 logical cores: the overall best single solver was the single-threaded solver Chuffed [15]. Except for some preliminary investigations done for the CPHydra and Numberjack solvers [29, 73] we are not aware of parallel CP portfolio solvers. This issue gives rise to interesting research perspectives. Specifically, parallelisation seems to be highly promising when applied to optimisation problems.

Let us consider as an example an instance of the *Bin Packing* problem taken from the `minizinc-1.6` benchmarks[3] when solved by using a portfolio of the solvers coming with the G12 MiniZinc distribution (viz., CPX, FD, LazyFD, and MIP) on a quadcore architecture within a timeout of 500 seconds. Table 1 shows the results achieved by the single solvers against two portfolio solvers: `par-folio` that just runs all the solvers in parallel and `obj-folio` that runs all the solver in parallel, restarting the solvers with the new bound of the objective function every time a better solution is found. The best solver of the portfolio is MIP, which is able to complete the search in 182.59 seconds. LazyFD finds in few seconds the optimal value 10, but never proves its optimality. FD and CPX instead do not find any solution. `par-folio` is remarkably worse than MIP (it takes almost twice the time to prove the optimality) witnessing that in practice running all the solvers in parallel does not mimic the sequential execution of the single solvers. `obj-folio` instead significantly *outperforms* the best solver: it quickly proves the optimality because in this case MIP is restarted by exploiting the value 10 found by LazyFD after few seconds.

Another interesting point concerns the solvers scheduling when the portfolio size m exceeds the number n of available cores. In the above example $m = n = 4$, but what if $n < m$? Is it better to select $k < n$ solvers for reducing the processor load, or to choose $k = n$ solvers to be run on all the available cores, or even scheduling $k > n$ solvers by properly splitting the solving time window? Furthermore, is it better to use

[3] The model file is `2DPacking.mzn` while the data file is `Class7_40_3.dzn`.

a static approach, where the solvers to run are decided in advance, or a dynamic one, where solvers are selected on-line according to the instance to be solved?

A major challenge is also predicting if, and when, restarting a solver is beneficial. Particular care must be taken in restarting a solver with new objective bounds, since interrupting the solver search means to lose the knowledge gained during the computation. This may be harmful for solvers that accumulate information during the search, e.g., lazy clause generation solvers [53].

Techniques like nogood learning and lazy clause generation have proven to be very effective, and in a parallel setting can gain additional benefits. Unfortunately, only few CP solvers use nogoods and there is no standard API to retrieve this knowledge. A standard protocol for extracting and sharing nogoods is hence desirable, since often the portfolio solver views its constituent solvers as "black-boxes" on which it has a very limited control. However, even without the control over the constituent solvers, it is possible to work directly on the problem to be solved. For instance, one can adopt work splitting techniques for dividing the original problem into a number of sub-problems, each of which assigned to a different solver.

5 Utilisation

As pointed out also in [61], a key reason for the lack of common adoption of portfolio solvers is their poor usability. The effort required to set up a portfolio solver is typically much higher than the cost of installing its constituent solvers. Building the prediction model is a hard work, partially justifiable by the fact that this process is performed off-line.

We believe that a major challenge for CP portfolio solvers relies in simplifying their installation and use, and that the best way to overcome this problem is to encourage the dissemination of (possibly open-source) portfolio solvers to be downloaded, installed, and used just like a regular individual solver.

For easing the everyday use, the installation of a portfolio solver should not be a nightmare. Wrapping all the software needed in a minimal virtual machine might be an idea: in this way the user just need to start the virtual machine and run the portfolio solver, instead of dealing with the installation of all the necessary components. Furthermore, following the current trend of cloud computing, it would be interesting to develop an "*online*" portfolio solver as a service to be installed and run in a public or private cloud. Some preliminary works for testing portfolios on the cloud have already been done [24, 43] and, as underlined in [40], this solution may have some advantages. The solving process is transparent for the end-user, which only needs the API for communicating with the server. Moreover, the internals of the portfolio solver can be studied and maintained directly in the cloud, by taking advantage of the emerging capabilities of cloud computing. An "immutable service" approach [21, 50] would enable to use the cloud resources to concurrently solve the incoming problems and update the prediction models.

The diffusion of CP portfolio solvers could have positive implications also for the individual solvers. Aside from the "image return" for a solver belonging to a successful portfolio, there are also technical aspects. For example, we realized that a lot of

solvers we tested have some bugs (e.g., only considering the MiniZinc Challenge 2014 there have been 24 wrong answers given by 5 different solvers). Portfolio solvers can be used for checking the reliability of a solver, by comparing its answer on a given problem against the answers given by each other solver of the portfolio. We believe that a portfolio solver should take into account the unreliability of its constituent solvers. Getting rid of a buggy solver may be too penalizing since it is often the case that the most promising solvers to include in a portfolio are the experimental ones, usually maintained by few people and not extensively tested. Where it is not possible to fix the bug, the verification *a posteriori* of the solution is preferable: the constituent solvers of the portfolio can be used for double-checking the solution. Unfortunately, the verification of a solution is sometimes computationally infeasible, especially when it comes to prove the unsatisfiability or the optimality. An alternative idea might be to encode the reliability in the prediction model, e.g., by associating to each solver a trust level.

5.1 SUNNY and `sunny-cp`

In order to facilitate and encourage the practical usage of CP portfolio solvers we developed `sunny-cp` [4], a parallel portfolio solver built on top of the SUNNY algorithm [3] able to solve generic CP problems encoded in MiniZinc language.

SUNNY is a lazy portfolio approach which exploits instances similarity to guess the best solver(s) to use. For a given problem instance p, SUNNY uses a k-Nearest Neighbours (k-NN) algorithm to select from a training set of known instances the subset $N(p, k)$ of the k problems closer to p. According to the $N(p, k)$ instances, SUNNY relies on three heuristics: h_{sel}, for *selecting* the most promising solvers to run; h_{all}, for *allocating* to each solver a certain runtime (the more a solver is promising, the more time is allocated); and h_{sch}, for *scheduling* the sequential execution of the solvers according to their presumed speed. These heuristics depend on the application domain. For example, for CSPs h_{sel} selects the smallest sub-portfolio $S \subseteq \Pi$ that solves the most instances in $N(p, k)$, by using the solving time for breaking ties. h_{all} allocates to each $s_i \in S$ a time t_i proportional to the instances that S can solve in $N(p, k)$, while h_{sch} sorts the solvers by increasing solving time in $N(p, k)$. For COPs the approach is analogous, but different performance metrics are used [5].

The first version of `sunny-cp` was sequential [6] and relied on eight solvers, viz. Chuffed, CPX, G12/CBC, G12/FD, G12/LazyFD, G12/Gurobi, Gecode, and MinisatID.[4] Then, we significantly improved it by adding more solvers (viz. Choco, iZplus, HaifaCSP, and OR-Tools) to its portfolio and especially by allowing their simultaneous execution and cooperation on multiple cores. This allowed `sunny-cp` to win the gold medal in the open category of MiniZinc Challenge 2015.

6 Related Work

The interest in algorithm selection and configuration is quite general and growing. It is outside the scope of the current paper to give a global overview of the plethora of

[4] `sunny-cp` attended the MiniZinc Challenge 2014 with respectable results (4$^{\text{th}}$ out of 18). It has also been awarded with an *honourable mention* by the challenge organizers.

portfolio approaches tried in the literature. For more comprehensive surveys, we refer the interested reader to [36, 42, 64].

As said earlier, portfolio solvers have proven their effectiveness in many international solving competitions. The SAT portfolio solvers 3S [38] and CSHC [46] won gold medals in SAT Competition 2011 and 2013 respectively. SATZilla [72] won the SAT Challenge 2012, CPHydra [54] the Constraint Solver Competition 2008, the ASP portfolio solver claspfolio [17] was gold medallist in different tracks of the ASP Competition 2009 and 2011, ArvandHerd [70] and IBaCoP [14] won some tracks in the Planning Competition 2014.

Apart from CPHydra and SUNNY, there are only few other approaches that can deal with CSPs. In [8,9] Machine Learning techniques are used to enhance the performances of a single CSP solver by dynamically adapting its search heuristics. These works lists an extensive set of features to train and improve the heuristics model through Support Vector Machines. Proteus [33] is a recent CSP portfolio approach that does not rely purely on CSP solvers, but may decide to encode a CSP problem instance into SAT, by selecting an appropriate encoding and a corresponding SAT solver.

Regarding optimisation problems, we can say that COP portfolios are mostly developed just for some specific optimisation problems like Knapsack, Most Probable Explanation, Set Partitioning, Travel Salesman Problem [26, 36, 69]. The only COP solver we are aware of is presented in [7] using an adaptation of the SUNNY algorithm. An empirical evaluation of different portfolio approaches applied to COPs was performed in [5].

Surprisingly enough, only a few portfolio solvers are parallel and even fewer are the dynamic ones selecting on-line the solvers to run. We are aware of only two dynamic and parallel portfolio solvers that attended a solving competition, namely p3S [45] (in the SAT Challenge 2012) and IBaCoP2 [14] (in the Planning Competition 2014). Apart from a preliminary investigation about CPHydra parallelisation [73], the only parallel and dynamic CP portfolio solver able to deal with also COPs is `sunny-cp` [4]. The parallelisation of portfolio solvers is a hot topic which is drawing some attention in the community. For instance, parallel extensions of well-known sequential portfolio approaches are studied in [32], while in [30] ASP techniques are used for computing a static schedule of solvers which can even be executed in parallel.

Finally, a number of tools are being developed in order to improve portfolio solvers usability. `snappy` [61] is a simple and training-less algorithm portfolio which relies on a nearest neighbours prediction mechanism. LLAMA (Leveraging Learning to Automatically Manage Algorithm) [41] is instead a framework that facilitates the exploration of different portfolio techniques on any problem domain, by supporting the most common solver selectors and possibly combining them.

7 Conclusions and Extensions

Portfolio approaches have been extensively studied, and successfully used in solving competitions. In this paper we discussed the main challenges that, in our view, need to be tackled for spreading the use of portfolio approaches in Constraint Programming. We identified in particular four main aspects: the prediction model used for the solver

selection, the treatment of optimisation problems, the parallelisation of execution, and the actual usability of CP portfolio solvers.

We already performed some preliminary investigations, and we are currently working on the implementation of some ideas we proposed. In particular, we are working to further improve the `sunny-cp` solver.

Clearly, portfolio solvers are not a *panacea* and there are contexts in which their use is unnecessary. For instance, when a given solver of the portfolio strongly dominates all the others it might be preferable switching to other related techniques such as *Algorithm Configuration* [34,35,39] for properly tuning the parameters of the dominant solver. Scenarios like this are not uncommon in real life applications, where the focus is on solving a specific problem (or class of problems) rather than different problems disparate in their nature.

Having more and better datasets and solvers is of course welcome for our purposes. We would like to encourage the CP community to submit new problems and solvers to international solving competitions like the MiniZinc Challenge. To advance the state of the art and bridge the current gaps, it would be nice to have a number of CP portfolio entrants (maybe running in a dedicated track). This somehow would go against the —surprising in our opinion— direction taken in the SAT competition 2014, where only portfolio approaches consisting of at most two core algorithms were allowed.

We conclude the paper by discussing some possible extensions of the portfolio approach beyond the CP paradigm. As mentioned, Algorithm Portfolios can be viewed as particular instances of the Algorithm Selection framework where we are interested in predicting case-by-case the best algorithm (not necessarily a constraint solver) for any new, unseen problem to be solved. The generality of this framework allows its instantiation to different paradigms such as Boolean Satisfiability (SAT), Answer-Set Programming (ASP), Quantified Boolean Formula (QBF), or even for solving different instances of the same problem, e.g., the Container Pre-marshalling Problem [13].

A natural, yet unexplored, target for portfolio solvers is certainly the *Constraint Logic Programming* (CLP) field. On the one hand, from a CLP specification is possible to derive a CP problem to be solved by CP portfolio solvers like for instance what done in [16]. On the other hand, a CLP solver can provide interfaces for dealing with different CP problem specifications (e.g., the Zinc library of SICStus Prolog [63] that allows to solve both FlatZinc and MiniZinc models) and therefore be embedded into a portfolio solver.

The portfolio framework also enables to consider program transformation techniques that may speed up the solving process. A possible approach consists in splitting the input problem into different, maybe overlapping sub-problems and to assign the sub-problems to the different constituent solvers. This might be advantageous especially when solvers are running simultaneously. Furthermore, the input model can even be enriched by adding redundant constraints (e.g., bounds, nogoods or other clauses learned by a solver during the search) for narrowing the search space. We remark that program transformation techniques are not uncommon in constraint solving and in particular there exists a lot of work proposing different techniques for encoding a CP problem into a SAT problem [1, 11, 33, 66, 68]. In this setting, portfolio approaches can be used for predicting whether and how to compile a CP model into SAT.

References

1. Ignasi Abío and Peter J. Stuckey. Encoding Linear Constraints into SAT. In *CP*, volume 8656 of *LNCS*, pages 75–91. Springer, 2014.
2. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An Enhanced Features Extractor for a Portfolio of Constraint Solvers. In *SAC*, pages 1357–1359. ACM, 2014.
3. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY: a Lazy Portfolio Approach for Constraint Solving. *TPLP*, 14(4-5):509–524, 2014.
4. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. A Multicore Tool for Constraint Solving. In *IJCAI*, pages 232–238. AAAI Press, 2015.
5. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Portfolio approaches for constraint optimization problems. *AMAI*, pages 1–18, 2015.
6. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. SUNNY-CP: a sequential CP portfolio solver. In *SAC*, pages 1861–1867. ACM, 2015.
7. Roberto Amadini and Peter J. Stuckey. Sequential Time Splitting and Bounds Communication for a Portfolio of Optimization Solvers. In *CP*, volume 8656 of *LNCS*, pages 108–124. Springer, 2014.
8. Alejandro Arbelaez, Youssef Hamadi, and Michele Sebag. Online heuristic selection in constraint programming. In *SoCS*, 2009.
9. Alejandro Arbelaez, Youssef Hamadi, and Michèle Sebag. Continuous Search in Constraint Programming. In *ICTAI*, pages 53–60. IEEE Computer Society, 2010.
10. Gilles Audemard, Benot Hoessen, Sad Jabbour, Jean-Marie Lagniez, and Cdric Piette. PeneLoPe, a Parallel Clause-Freezer Solver. In *SAT Challenge*, pages 43–44, 2012.
11. Pedro Barahona, Steffen Hölldobler, and Van-Hau Nguyen. Representative Encodings to Translate Finite CSPs into SAT. In *CPAIOR*, volume 8451 of *Lecture Notes in Computer Science*, pages 251–267. Springer, 2014.
12. Daniel Le Berre and Laurent Simon. Preface to the Special Volume on the SAT 2005 Competitions and Evaluations. *JSAT*, 2(1-4), 2006.
13. Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Thomas Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger H. Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. Aslib: A benchmark library for algorithm selection. *CoRR*, abs/1506.02465, 2015.
14. Isabel Cenamor, Tomás de la Rosa, and Fernando Fernández. IBACOP and IBACOP2 Planner. `http://www.plg.inf.uc3m.es/~icenamor/files/IBaCoPPlanner.pdf`, 2014.
15. Geoffrey Chu, Maria Garcia de la Banda, and Peter J. Stuckey. Automatically Exploiting Subproblem Equivalence in Constraint Programming. In *CPAIOR*, volume 6140 of *LNCS*, pages 71–86. Springer, 2010.
16. Raffaele Cipriano, Agostino Dovier, and Jacopo Mauro. Compiling and executing declarative modeling languages to gecode. In *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 744–748. Springer, 2008.
17. claspfolio. `http://www.cs.uni-potsdam.de/claspfolio/`.
18. Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2Nd Edition)*. Wiley-Interscience, 2000.
19. Chris Fawcett, Mauro Vallati, Frank Hutter, Jörg Hoffmann, Holger H. Hoos, and Kevin Leyton-Brown. Improved features for runtime prediction of domain-independent planners. In *ICAPS*. AAAI, 2014.
20. Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A Constraint Language for Specifying Combinatorial Problems. *Constraints*, 13(3):268–306, 2008.
21. FullContact. How We Used Immutable Servers to Simplify Our Cloud Infrastructure. `http://www.fullcontact.com/blog/immutable-servers-benefits/`, 2014.
22. Allen Van Gelder. Careful Ranking of Multiple Solvers with Timeouts and Ties. In *SAT*, volume 6695 of *LNCS*, pages 317–328. Springer, 2011.

23. Ian P. Gent and Toby Walsh. Csp_{lib}: A Benchmark Library for Constraints. In *CP*, volume 1713 of *LNCS*, pages 480–481. Springer, 1999.
24. Daniel Geschwender, Frank Hutter, Lars Kotthoff, Yuri Malitsky, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm Configuration in the Cloud: A Feasibility Study. In *LION*, volume 8426 of *LNCS*, pages 41–46. Springer, 2014.
25. Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001.
26. 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.
27. Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.
28. Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. ManySAT: a Parallel SAT Solver. *JSAT*, 6(4):245–262, 2009.
29. E. Hebrard, E. O'Mahony, and B. O'Sullivan. Constraint Programming and Combinatorial Optimisation in Numberjack. In *CPAIOR-10*, volume 6140 of *LNCS*, pages 181–185. Springer-Verlag, May 2010.
30. Holger H. Hoos, Roland Kaminski, Marius Thomas Lindauer, and Torsten Schaub. aspeed: Solver scheduling via answer set programming. *TPLP*, 2015.
31. Holger H. Hoos, Benjamin Kaufmann, Torsten Schaub, and Marius Schneider. Robust Benchmark Set Selection for Boolean Constraint Solvers. In *LION*, volume 7997 of *LNCS*, pages 138–152. Springer, 2013.
32. Holger H. Hoos, Marius Thomas Lindauer, and Frank Hutter. From Sequential Algorithm Selection to Parallel Portfolio Selection. In *LION*, 2015.
33. Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan. Proteus: A Hierarchical Portfolio of Solvers and Transformations. In *CPAIOR*, volume 8451 of *LNCS*, pages 301–317. Springer, 2014.
34. Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *LION*, volume 6683 of *LNCS*, pages 507–523. Springer, 2011.
35. Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An Automatic Algorithm Configuration Framework. *J. Artif. Intell. Res. (JAIR)*, 36:267–306, 2009.
36. Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm Runtime Prediction: The State of the Art. *CoRR*, abs/1211.0906, 2012.
37. International planning competition. `http://ipc.icaps-conference.org/`.
38. Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm Selection and Scheduling. In *CP*, volume 6876 of *LNCS*. Springer, 2011.
39. Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. ISAC - Instance-Specific Algorithm Configuration. In *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2010.
40. Zeynep Kiziltan and Jacopo Mauro. Service-oriented volunteer computing for massively parallel constraint solving using portfolios. In *CPAIOR*, volume 6140 of *LNCS*, pages 246–251. Springer, 2010.
41. Lars Kotthoff. LLAMA: Leveraging Learning to Automatically Manage Algorithms. *CoRR*, abs/1306.1031, 2013.
42. Lars Kotthoff. Algorithm Selection for Combinatorial Search Problems: A Survey. *AI Magazine*, 35(3):48–60, 2014.
43. Lars Kotthoff. Reliability of computational experiments on virtualised hardware. *J. Exp. Theor. Artif. Intell.*, 26(1):33–49, 2014.
44. Christian Kroer and Yuri Malitsky. Feature Filtering for Instance-Specific Algorithm Configuration. In *ICTAI*, pages 849–855. IEEE, 2011.
45. Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Parallel SAT Solver Selection and Scheduling. In *CP*, volume 7514, pages 512–526. Springer, 2012.

46. Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm Portfolios Based on Cost-Sensitive Hierarchical Clustering. In *IJCAI*. IJCAI/AAAI, 2013.
47. Marco Maratea, Luca Pulina, and Francesco Ricca. Multi-engine ASP solving with policy adaptation. *Journal of Logic and Computation*, 2013.
48. Marco Maratea, Luca Pulina, and Francesco Ricca. A multi-engine approach to answer-set programming. *TPLP*, 14(6):841–868, 2014.
49. Max-SAT 2013. `http://maxsat.ia.udl.cat/introduction/`.
50. Kief Morris. Immutable server web page. `http://martinfowler.com/bliki/ImmutableServer.html`, 2013.
51. mzn2feat-1.0 web page. `http://www.cs.unibo.it/~amadini/mzn2feat-1.0.tar.bz2`.
52. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a Standard CP Modelling Language. In *CP*, 2007.
53. Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
54. Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. *AICS 08*, 2009.
55. Luca Pulina and Armando Tacchella. A Multi-engine Solver for Quantified Boolean Formulas. In *CP*, volume 4741 of *LNCS*, pages 574–589. Springer, 2007.
56. Luca Pulina and Armando Tacchella. A self-adaptive multi-engine solver for quantified boolean formulas. *Constraints*, 14(1):80–116, 2009.
57. John R. Rice. The Algorithm Selection Problem. *Advances in Computers*, 15:65–118, 1976.
58. Olivier Roussel. ppfolio. http://www.cril.univ-artois.fr/~roussel/ppfolio/.
59. Olivier Roussel and Christophe Lecoutre. XML Representation of Constraint Networks: Format XCSP 2.1. *CoRR*, abs/0902.2362, 2009.
60. Ashish Sabharwal and Horst Samulowitz. Insights into Parallelism with Intensive Knowledge Sharing. In *CP*, volume 8656, pages 655–671. Springer, 2014.
61. Horst Samulowitz, Chandra Reddy, Ashish Sabharwal, and Meinolf Sellmann. Snappy: A simple algorithm portfolio. In *SAT*, volume 7962 of *LNCS*, pages 422–428. Springer, 2013.
62. SAT Challenge 2012. `http://baldur.iti.kit.edu/SAT-Challenge-2012/`.
63. Zinc Interfacelibrary(zinc). `https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/lib_002dzinc.html#lib_002dzinc`.
64. Kate Smith-Miles. Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Comput. Surv.*, 41(1), 2008.
65. Mirko Stojadinovic and Filip Maric. Instance-based Selection of CSP Solvers using Short Training. In *Pragmatics of SAT*, 2014.
66. Mirko Stojadinovic and Filip Maric. meSAT: multiple encodings of CSP to SAT. *Constraints*, 19(4):380–403, 2014.
67. Peter J. Stuckey, Ralph Becket, and Julien Fischer. Philosophy of the MiniZinc challenge. *Constraints*, 15(3):307–316, 2010.
68. Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling Finite Linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
69. Orestis Telelis and Panagiotis Stamatopoulos. Combinatorial Optimization through Statistical Instance-Based Learning. In *ICTAI*, pages 203–209, 2001.
70. Richard Anthony Valenzano, Hootan Nakhost, Martin Müller, Jonathan Schaeffer, and Nathan R. Sturtevant. ArvandHerd: Parallel Planning with a Portfolio. In *ECAI*, pages 786–791, 2012.
71. L. Xu, F. Hutter, J. Shen, H. Hoos, and K. Leyton-Brown. SATzilla2012: Improved Algorithm Selection Based on Cost-sensitive Classification Models. SAT Challenge, 2012.
72. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based Algorithm Selection for SAT. *JAIR*, 32:565–606, 2008.
73. Xi Yun and Susan L. Epstein. Learning Algorithm Portfolios for Parallel Execution. In *ION*, volume 7219, pages 323–338. Springer, 2012.