A Classification-based Approach to Manage a Solver Portfolio for CSPs

Zeynep Kiziltan[1], Luca Mandrioli[1], Jacopo Mauro[1], and Barry O'Sullivan[2]

[1] Computer Science Department, University of Bologna, Italy.
{zeynep, mandriol, jmauro}@cs.unibo.it
[2] Cork Constraint Computation Centre, University College Cork, Ireland.
b.osullivan@4c.ucc.ie

Abstract. The utility of using portfolios of solvers for constraint satisfaction problems is well reported. We show that when runtimes are properly clustered, simple classification techniques can be used to predict the class of runtime as, for example, short, medium, long, time-out, etc. Based on runtime classifiers we demonstrate a dispatching approach to solve a set of problem instances in order to minimize the average completion time of each instance. We show that this approach significantly out-performs a well-known CSP solver and performs well against an oracle implementation of a solver portfolio.

1 Introduction

The past decade has witnessed a significant increase in the number of constraint solving systems deployed for solving constraint satisfaction problems (CSP). It is well recognized within the field of constraint programming that different solvers are better at solving different problem instances, even within the same problem class [3]. It has been shown in other areas, such as satisfiability testing [19] and integer linear programming [8], that the best on-average solver can be out performed by a portfolio of possibly slower on-average solvers. This selection process is usually performed using a machine learning technique based on feature data extracted from CSPs.

Three specific approaches that use contrasting approaches to portfolio management in CSP, SAT and QBF are CPHydra, SATzilla, and Acme, respectively. CPHydra is a portfolio of constraint solvers exploiting a case-base of problem solving experience [12]. CPHydra combines case-base reasoning of machine learning with the idea of partitioning CPU-TIME between components of the portfolio in order to maximize the expected number of solved problem instances within a fixed time limit. SATzilla [19] builds runtime prediction models using linear regression techniques based on structural features computed from instances of Boolean satisfiability problem. Given an unseen instance of the satisfiability problem, SATzilla selects the solver from its portfolio that it predicts to have the fastest running time on the instance. The Acme system is a portfolio approach to solve quantified Boolean formulae, i.e. SAT instances with some universally quantified variables [13].

In this paper we present a very different approach to managing a portfolio for constraint solving when the objective is to solve a set of problem instances so that the average completion time, i.e. the time at which we have either found a solution or proven that none exist, of each instance is minimized. This scenario arises in a context in which problem instances are submitted to, for example, a cloud-based solver which queues and solves instances in an autonomous fashion. This scenario is consistent with our long term objective which is to build an on-line service-based portfolio solver that receives CSPs from a user to solve, exploits multiple processing nodes to search for solutions and returns the answer as quickly as possible (see [7] for more details). In addition, there is a significant scheduling literature that focuses on minimizing average completion time, much of which is based around the use of dispatching heuristics [17].

The approach we propose in this paper is strongly inspired by dispatching rules for scheduling. Our approach is conceptually simple, but powerful. Specifically, we propose the use of classifier techniques as a basis for making high-level and qualitative statements about the solvability of CSP instances with respect to a given solver portfolio. We also use classifier techniques as a basis for a dispatching-like approach to solve a set of problem instances in a single processor scenario. We show that when runtimes are properly clustered, simple classification techniques can be used to predict the class of runtime as, for example, short, medium, long, time-out, etc. We show that this approach significantly outperforms a well-known general-purpose CSP solver and performs well against an oracle implementation of a portfolio.

The remainder of this paper is organized as follows. In Section 2 we summarize the requisite background on constraint satisfaction and machine learning required for this paper. Section 3 presents the large collection of CSP instances on which we base our study. We discuss the various classification tasks upon which our approach is based in Section 4, and evaluate the suitability of different representations and classification for these tasks in Section 5. We demonstrate the utility of our classification-based approach for managing a solver portfolio in Section 6. We discuss related work in Section 7 and conclude in Section 8.

2 Preliminaries

A *constraint satisfaction problem* (CSP) is defined by a finite set of variables, each associated with a domain of possible values that the variable can be assigned, and a set of constraints that define the set of allowed assignments of values to the variables [9]. The *arity* of a constraint is the number of variables it constrains. Given a CSP, the task is normally to find an assignment to the variables that satisfies the constraints, which we refer to as a *solution*.

Machine learning *"is concerned with the question of how to construct computer programs that automatically improve with experience"*. It is a broad field that uses concepts from computer science, mathematics, statistics, information theory, complexity theory, biology and cognitive science [10]. Machine learning can be applied to well-defined problems, where there is both a source of training

examples and one or more metrics for measuring performance. In this paper we are particularly interested in classification tasks. A classifier is a function that maps an *instance* with one or more discrete or continuous features to one of a finite number of classes [10]. A classifier is trained on a set of instances whose class is already known, with the intention that the classifier can transfer its training experiences to the task of classifying new instances.

3 The International CSP Competition Dataset

We focused on demonstrating our approach on as comprehensive and a realistic set of problem instances as possible. Therefore, we constructed a comprehensive dataset of CSPs based on the various instances gathered for the annual International CSP Solver Competition[3] from 2006-2008. An advantage of using these instances is that they are publicly available in a standardized XML-based format called XCSP.[4] The first competition was held in 2005, and all benchmark problems were represented using extensional constraints only. In 2006, both intentional and extensional constraints were used. In 2008, global constraints were also added. Overall, there are *five categories of benchmark problem* in the competition: 2-ARY-EXT instances involving extensionally defined binary (and unary) constraints; N-ARY-EXT instances involving extensionally defined constraints, at least one of which is defined over more than two variables; 2-ARY-INT instances involving intensionally defined binary (and unary) constraints; N-ARY-INT instances involving intensionally defined constraints, at least one of which is defined over more than two variables; and, GLB instances involving any kind of constraints, including global constraints.

The competition required that any instance should be solved within 1800 seconds. Any instance not solved by this cut-off time was considered unsolved. To facilitate our analysis, we remove from the dataset any instance that could not have been solved by any of the solvers of our portfolio by the cut-off. In total, our data set contains around 4000 instances across these various categories.

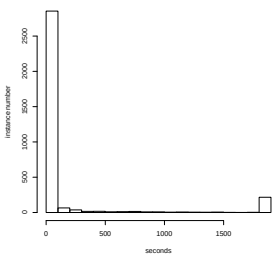
4 From Runtime Clustering to Runtime Classification

We show how clusters of runtimes can be used to define classification problems for a dispatching-based approach to managing an algorithm portfolio. While our focus here is not to develop the CPHydra system, we will, for convenience, use its constituent solvers and feature descriptions of problem instances to build our classifiers. We demonstrate our approach on a comprehensive and realistic set of problem instances.

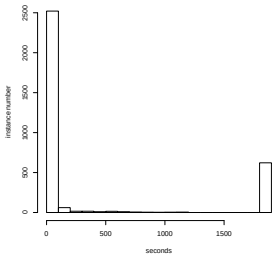
Based on the three solvers used in the 2008 CSP Solver Competition variant of CPHydra we present in Figures 1a, 1b, and 1c the runtime distributions for

[3] Competition web-site: `http://cpai.ucc.ie`

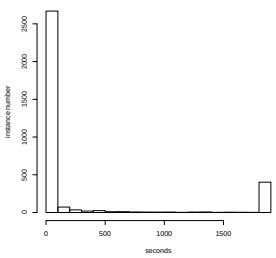
[4] `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`



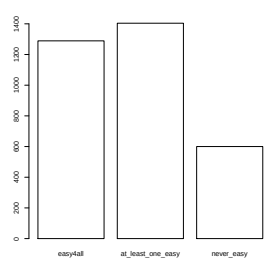
(a) Runtimes for Mistral.



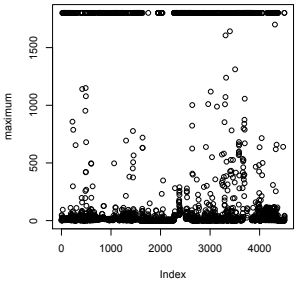
(b) Runtimes for Choco.



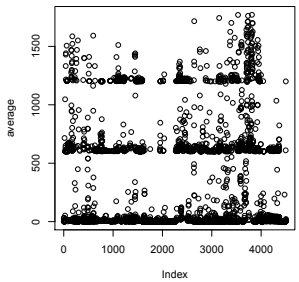
(c) Runtimes for Abscon.



(d) Occurrence of each class for $3C_{10}$ in the dataset, i.e. an instance is considered easy if it can be solved in at most 10 seconds.



(e) The maximum runtimes of the portfolio on each instance.



(f) The average runtimes of the portfolio on each instance.

Fig. 1: The performance of each solver in the portfolio on the dataset.

each of its solvers, Mistral, Choco, and Abscon respectively,[5] showing for every solver in the portfolio the number of instances of the data set solved in given time windows. Having removed from the dataset any instance that could not have been solved by any of these solvers within a 1800s time-limit, each instance is ensured to be solved by at least one solver. However, it is not the case that each solver finds that the same instances are either easy or hard. There are many instances, as we will show below, for which one of the solvers decides the instance quickly, while another solver struggles to solve it. Therefore we define a classification task that given a CSP instance returns the fastest solver for that instance.

Classification 1 (Fastest Solver Classification (FS)) *Given a CSP instance i and a CSP solver portfolio Π, the* FS *classification task is to predict which solver in Π gives the fastest runtime on i.*

[5] Visit the competition site for links to each of the solvers.

From Figure 1 it is clear that there are many instances that can be solved easily or not at all. To capture this property we introduce a classification task called 3C which is defined over a solver portfolio as follows.

Classification 2 (3C$_k$) *Given a CSP instance i and a CSP solver portfolio Π, the* 3C$_k$ *classification task is to predict whether i: (a) can be solved by all solvers in Π in at most k seconds; (b) can be solved by at least one solver, but not all, in Π in at most k seconds; or (c) takes more than k seconds to solve with each solver in Π.*

The number of instances in our CSP dataset in each class is presented in Figure 1d. Note that while many instances were easy (i.e. solvable within 10 seconds) for all solvers, a larger number were easy for some, but not all (the middle stack in the histogram). We consider two additional classifiers related to the performance of the portfolio as a whole. We compute the maximum and the average time required by each solver in the portfolio to solve each instance. The maximum times are presented in Figure 1e, in which the x axis lists the index of each instance and the y-axis represents the maximum run-time. Note that a time-limit of 1800 seconds was applied on the dataset which gives the upper bound of the maximum solving time and which is why there are a number of instances presenting across the top of the plot. For applying this classifier, we consider only two intervals of running time according to the data presented in Figure 1e: at most 1500 seconds and greater than 1500 seconds.

Classification 3 (MaxC$_{ks}$) *Given a CSP instance i and a CSP solver portfolio Π, the* MaxC$_{ks}$ *classification task is to predict which interval of running-times in ks that instance i can be solved using the worst performing solvers from Π.*

Similarly, the average times are presented in Figure 1f. In this plot we note that there are three distinct classes of runtimes: instances that take on average between 0-600 seconds, between 601-1200, and more than 1200. Again, this division is influenced by the fact that an instance's maximum solving time is at most 1800 seconds.

Classification 4 (AvgC$_{ks}$) *Given a CSP instance i and a CSP solver portfolio Π, the* AvgC$_{ks}$ *classification task is to predict which interval of running-times in ks that instance i can be solved taking the average solving times for each of the solvers in Π.*

To complement the AvgC classifier, we will also make use of a classifier that considers the variance, or spread, of runtimes across the constituent solvers of a portfolio over a given instance. We refer to this classifier as Spread.

Classification 5 (Spread$_k$) *Given a CSP instance i and a CSP solver portfolio Π, the* Spread$_k$ *classification task is to predict whether the difference across the runtimes of the constituent solvers is at most (or at least) k.*

For applying this classifier, we consider the difference of at most 100 seconds, based on the given runtimes.

The classifiers presented in this section define a very expressive qualitative language to describe the expected performance of a solver portfolio on a given CSP instance. For example, we can make statements like *"this instance is easy for all solvers in the portfolio"*, or *"this instance is easy for some, but not all solvers, but the average running time is low and has low variation"*. This contrasts with all current approaches to managing a solver portfolio. As our empirical results will demonstrate, this approach is also very powerful in term of efficient solving.

5 Experiments in Runtime Classification

In this section, we experiment with the various classification problems discussed in the previous section. To do so, we first establish "good" features to represent CSPs starting from the features used in CPHydra. The objective of these experiments is to show that accurate classifiers for solver runtimes can be generated, and that these can be successfully used to build effective dispatching heuristics for managing a solver portfolio for CSPs. For space reasons, we only focus on the 3C, AvgC, and MaxC classifiers. The experimental data set is based on the 2008 International CSP Solving Competition[6]. We consider a portfolio comprising three solvers: Abscon, Choco and Mistral. Running times for each of these solvers are available from the CSP competition's web-site. A time-out on solving time is imposed at 1800 seconds. We exclude from the dataset the CSP instances that cannot be solved in that amount of time and also some other instances due to the reason that we will explain later. In total, our final dataset, upon which we run our experiments comprises 3293 CSP instances.

Knowledge Representation and Classifiers. Since the selection of good features has a significant impact on the classifier performances, we investigate which ones are more suitable to capture problem hardness. In particular, we consider three feature-based representations of the CSP instances in our dataset: SATzilla features representing each CSP instance encoded into SAT, those features used by CPHydra(with some modifications), and the combination of the two.

SATzilla uses a subset of the features introduced by [11]: starting from 84 features they discard those computationally expensive and too instable to be of any value. At the end they consider only 48 features that can be computed in less then a minute (for more information see [18]). In this work we are able to use directly these features simply translating each competition CSP instance into SAT using the Sugar solver[7] and then using SATzilla to extract its feature description. In some (but few) cases, the encoding of a CSP instance into SAT requires an excessive amount of time (i.e. more than a day). In order to make a

[6] Competition web-site: `http://cpai.ucc.ie`

[7] `http://bach.istc.kobe-u.ac.jp/sugar/`

fair comparison between the set of features, we simply dropped such instances from the dataset.

For the second feature representation, we started from the 36 features of CPHydra. Whilst the majority of them are syntactical, the remaining are computed by collecting data from short runs of the Mistral solver. Among the syntactical features, worth mentioning are the number of variables, the number of constraints and global constraints, the number of constants, the sizes of the domains and the arity of the predicates. The dynamic features instead take into account the number of nodes explored and the number of propagations done by Mistral with a time limit of 2 seconds. When we extracted the CPHydra features using our dataset we noticed that two of them (viz. the logarithm of the number of constants and the logarithm of the number of extra values) were constant. Since constant features are not useful for discriminating between different problems, we discarded these two features. Inspired by Nudelman et al [11], we considered additional features like the ratio of the number constraints over the number variables, and the ratio of the number of variables over the number of constraints. Moreover, we added features representing an instance's *variable graph* and *variable-constraint graph*. In the former, each variable is represented by a node with an edge between pairs of nodes if they occur together in at least one constraint. In the latter, we construct a bipartite graph in which each variable and constraint is represented by a node, with an edge between a variable node v and a constraint node c if v is constrained by c. From these graphs, we extract the average and standard deviation of the node degrees and take their logarithm. With these, the total amount of features we consider are 42.

The third feature-based description of the CSP instances, which we refer to as Hylla, is simply the concatenation of the two feature descriptions discussed above. We consider a variety of classifiers, implemented in publicly available tools RapidMiner[8] and WEKA.[9] Our SVM classifier is hand-tuned to the specific tasks considered in this paper according to the best parameters found using RapidMiner; however, it is only applied to the 3C and AvgC tasks because it appeared to be problematic to tune for the MaxC task. The other WEKA classifiers are used with their default settings.

Results. The results of the runtime classification tasks are presented in Tables 1– 3. Three alternative feature descriptions, as discussed earlier, are compared; these are denoted as CPHydra, Hylla, and SATzilla, in the tables. We compare the performance of various classifiers on each of the three classification tasks (3C, MaxC, and AvgC). A 10-fold cross-validation is performed. The performance of each classifier, on each representation, on each classification task are measured in terms of the classification accuracy and the κ-statistic. The latter measures the relative improvement in classification over a random predictor.

[8] `http://rapid-i.com/content/view/181/196/`
[9] `http://www.cs.waikato.ac.nz/ml/weka/`

Table 1: Classification accuracy and κ-statistics for the 3C classifier.

Classifier	(CPHydra)	(Hylla)	(SATzilla)	(CPHydra)	(Hylla)	(SATzilla)
trees.J48	83.83	82.52 •	79.70 •	0.75	0.73	0.68 •
meta.MultiBoostAB	84.75	84.91	82.19 •	0.76	0.76	0.72 •
trees.RandomForest	85.34	85.14	82.39 •	0.77	0.77	0.72 •
functions.LibSVM	85.63	84.68	82.62 •	0.77	0.76 •	0.73 •
lazy.IBk	83.69	83.53	78.71 •	0.74	0.74	0.67 •
bayes.NaiveBayes	61.70	62.74	52.49 •	0.37	0.44 ○	0.31 •
meta.RandomCommittee	84.99	85.44	82.57 •	0.76	0.77	0.73 •
rules.OneR	72.89	72.89	69.16 •	0.57	0.57	0.51 •

○, • statistically significant improvement or degradation over CPHydra .

Table 2: Classification accuracy and κ-statistics for the AvgC classifier.

Classifier	(CPHydra)	(Hylla)	(SATzilla)	(CPHydra)	(Hylla)	(SATzilla)
trees.J48	84.18	83.35	82.42 •	0.63	0.61	0.58 •
meta.MultiBoostAB	85.14	85.10	83.95 •	0.65	0.65	0.62 •
trees.RandomForest	85.37	85.53	84.42	0.65	0.65	0.62 •
functions.LibSVM	84.99	84.24	83.67 •	0.64	0.63	0.60 •
lazy.IBk	83.45	83.13	81.39 •	0.62	0.61	0.57 •
bayes.NaiveBayes	64.96	53.81 •	41.69 •	0.25	0.22	0.12 •
meta.RandomCommittee	85.03	85.32	84.25	0.65	0.65	0.62
rules.OneR	78.97	78.97	75.75 •	0.45	0.45	0.34 •

○, • statistically significant improvement or degradation CPHydra:

Table 3: Classification accuracy and κ-statistics for the MaxC classifier.

Classifier	(CPHydra)	(Hylla)	(SATzilla)	(CPHydra)	(Hylla)	(SATzilla)
trees.J48	89.61	89.08	87.99 •	0.73	0.72	0.69 •
meta.MultiBoostAB	90.19	90.33	89.47	0.75	0.75	0.73
trees.RandomForest	90.35	90.70	89.90	0.75	0.76	0.74
lazy.IBk	89.18	89.00	87.87 •	0.73	0.72	0.70 •
bayes.NaiveBayes	71.37	67.30 •	54.26 •	0.31	0.34	0.18 •
meta.RandomCommittee	90.28	90.64	90.10	0.75	0.76	0.74
rules.OneR	82.98	82.98	77.87 •	0.54	0.54	0.38 •

○, • statistically significant improvement or degradation over CPHydra .

Differences in performance are tested for statistical significance using a Paired t-Test at a 95% confidence level. The performance on the CPHydra feature set is used as a baseline. In each table, values that are marked with a ○ represent performances that are statistically significantly better than CPHydra, while those marked with a • represent performances that are statistically significantly worse.

In summary, both classification accuracies and κ values are high across all three tasks. Interestingly, combining both CPHydra and SATzilla features improves performance in only one κ value, and without any significant improvement in classification accuracy. The CPHydra feature set thus gives rise to the best overall performance. Based on these promising results, we consider in the next section the utility of using these classifiers as a basis for managing how a solver portfolio can be used to solve a collection of CSP instances.

6 Scheduling a Solver Portfolio

We now consider a solver portfolio and demonstrate the utility of our classification-based approach for its management via some experimental results.

Portfolio Construction and its Management. The portfolio is composed of three solvers previously introduced: Mistral, Choco and Abscon. It is designed to solve a collection of CSP instances as quickly as possible. Specifically, it tries to minimize the average *completion time* of each instance; the completion time of an instance is the time by which it is solved. One would wish to minimize average completion time, if for example a CSP solver was deployed as a web-service and instances were being added to a queue for solving. We assume all CSP instances are known at the outset. This setting is similar to that used in the international SAT competitions and also relevant in the context of the International CSP Competition.

Minimizing average completion time can be achieved by solving each problem in increasing order of difficulty, i.e. by using the well-known *shortest processing time* heuristic. In a solver portfolio context this corresponds to solving an instance with the fastest available solver for it, and ordering each instance by the corresponding solving time. This give us the most basic oracle (perfect) approach to minimize average completion time. As it is unlikely to have such perfect information, the portfolio can be *managed* via an *instance selector* and a *solver selector* whose purpose are to predict, respectively, the correct order of the instances from easiest to hard and the fastest solver on a given instance.

The classifiers developed previously can help to manage the portfolio. Consider for instance Figure 1d in which we see that the instances are grouped in three: i) those that can be solved by all solvers in 10 seconds; ii) those that can be solved by at least one solvers in 10 seconds; iii) those that are solved by all solvers in more than 10 seconds. The figure exhibits a rather balanced distribution, especially between the first two classes. This suggests that 3C can provide a good basis for distinguishing the easy instances from the hard ones, and that the classifiers AvgC, Spread, and MaxC could be used to break ties between the instances of the second and third classes. We exemplify this approach in Figure 2. Given two classifiers C_1 and C_2 with 3 and 2 classes respectively, an instance ordering $C_1 \prec C_2$ would mean that the instances are first divided according the predictions of C_1 resulting in three classes, and then those in each class of C_1 would be further divided according to the predictions of C_2, resulting in six classes in total. The ordering of the instances then would be from the left most to the right most leaf in the tree.

Oracles, Baselines and Experimental Methodology. In our experiments, we compare the quality of our classifiers for managing a portfolio of solvers with various oracle-based management strategies and simple baseline strategies. In particular we consider the following instance selectors to order the instances:

- oracle: orders the instances based on their solving time, shortest first;
- BC (best classifier): orders the instances based on the Spread classifier and then uses the 3C, AvgC, MaxC classifiers for tie-breaking; following the notation introduce above this instance selector might be defined as Spread $\prec$ 3C $\prec$ AvgC $\prec$ MaxC. As we will show later, this selector is the best classifier-based instance selector amongst all the possible instance selectors built using the classifiers defined in Section 4;

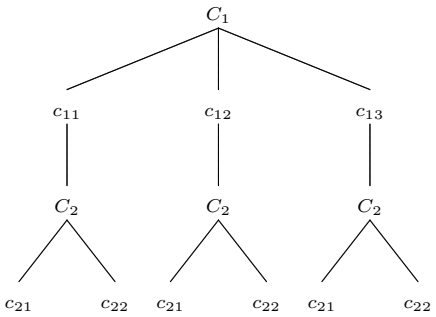


Fig. 2: Instance ordering using classifiers C_1 (dividing in 3 classes) and C_2 (dividing in 2 classes).

- LR (linear regression): orders the instances based on the expected solving time, shortest first. The expected solving time is computed predicting the logarithm of the solving time using the ridge regression algorithm implemented in RapidMiner; the ridge parameter was set at the default lever, i.e. $1.0 * 10^{-8}$. The features used as input of this algorithm where those used by CPHydra, defined in Section 5;
- random: orders the instances randomly.

To sort the solvers to execute we consider the following:

- oracle: uses the best solver only;
- Fs: uses the solver predicted by the Fs classifier first for 1800 seconds. In the case of time-out, we randomly switch to another solver for 1800 seconds. In case this second solver time-outs we then run the remaining solver in the portfolio;
- SLR (Simple linear regression): uses the solver having the smallest expected solving time for 1800 seconds. The expected solving time is computed using ridge regression similar to the implementation of the LR selector. Note that SLR selector is a simple attempt to simulate the linear regression approach to select solvers a lá SATzilla.
- SLR$^+$: which uses first the solver having the smallest expected solving time for 1800 seconds. In case of time-out, we switch to the solver having the second smallest expected solving time. This solver is run for 1800 seconds and, in case it time-outs, we run the remaining solver in the portfolio. The expected solving time is computed using ridge regression similar to the implementation of the LR selector;
- CPHydra: simulates the performance that would be obtained from CPHydra[10]. The solver scheduling algorithm of CPHydra is used to partition the time window of 1800 seconds among the three solver of the portfolio. We then sim-

[10] This simulation was necessary to avoid running CPHydra several million times in order to perform the experiments.

ulate what would have happened if Mistral was used first, Choco for second and Abscon for last; [11]
- Mistral: uses only Mistral for 1800 seconds;
- Random: tries the solvers in a sequence in random order, every one of them with a timeout of 1800 seconds.

As noted earlier, our benchmarks are composed of instances that can be solved by at least one solver by the cut-off. Whilst the SLR, CPHydra, and Mistral solver selectors do not guarantee that all instances will be solved, all the other selector can make such a claim. In particular Mistral is not able to solve 7.871% of instances whilst using the SLR and CPHydra solver selectors 3.83% and 0.91% of the instances were not solved, respectively.

In the experiments, we adopt a simple random split validation approach to evaluate the quality of our classifiers for managing a portfolio of solvers. We use the instances described in Section 3. Given a management strategy, we execute it (via simulation) 1000 times. A single run consists in random split of the dataset into a testing set (1/5 of the instances) and a training set (the remaining 4/5). We use the random forest algorithm (RapidMiner) for learning the models. For every run, we compute the overall average solving times and report just the average of the average solving times across 1000 runs. Differences in performance are tested for statistical significance using a Paired t-test at a 95% confidence level.

Results. First we built instance selectors considering all possible orderings of the classifiers described in Section 4 and using Fs as a solver selector. Table 4 presents a subset of our results where we report some of the best instance selectors. The best one is $stdv < 3C < avg < max$ that as previously mentioned is the instance selector that sorts the instances first using the Spread classifier and then using 3C, AvgC, MaxC in that order for tie-breaking. We selected this classifier, denoted as BC for short, for further comparisons against other approaches.

Even though BC turned out to be the best classifier-based instance selector in terms of average performance, it is not statistically better than several of the other configurations, as indicted in Table 4. For example the instance selector $stdv < 3C < avg$ is not statistically worse than BC. All the classifier-based instance selectors not shown in the table are statistically worse than BC.

The next experiments show what happens when we vary the instance selector having fixed the solver selector. In Table 5 we compare these results when the oracle, the random and the Fs solver selector are used. When the oracle or the Fs solver selector are chosen the oracle instance selector was, of course, the best since it has full information about which solver is best for each instance. The random instance selector is ranked last while the instance selector based on classifiers was better that the LR selector. When, instead, the random solver selector was chosen we have only one inversion: the BC selector is better than the oracle. This rather surprising result can be easily explained by the fact

[11] The execution order of the solvers was chosen according to their average solving times, smallest first.

Table 4: Performance of classification-based portfolios. ∘, • indicate statistically significant improvement or degradation over instance selector BC.

instance sel.	solver sel.	average
BC= $stdv < 3C < avg < max$	Fs	13158.59
$stdv < 3C < avg$	Fs	13178.91
$stdv < 3C < max < avg$	Fs	13193.21
$3C < stdv < avg < max$	Fs	13195.21
$3C < avg < stdv < max$	Fs	13198.94
$3C < stdv < max < avg$	Fs	13206.20
$3C < avg < max < stdv$	Fs	13214.93
$3C < max < avg < stdv$	Fs	13225.41
$3C < avg < stdv$	Fs	13232.20
$3C < stdv < avg$	Fs	13255.44 •
$3C < max < stdv < avg$	Fs	13275.99 •
$3C < avg < max$	Fs	13427.87 •
$3C < max < avg$	Fs	13480.43 •
...	...	

that the oracle instance selector is the best one only when the solver selector always picks the fastest solver for that instance. If, instead, the solver to use is chosen randomly the best oracle would be the one that sorts the instances according to the average solving time of all the solvers for that instance. With the support of these results we argue that classification-based instance selectors are not only better than selectors based on the prediction of expecting times via linear regression, but they also adapt nicely to scenarios where the solver selector is far from optimal.

In Table 6 we fixed the instance selector varying the solver selectors. In particular we present what happens when the oracle, random and BC selectors are chosen.[12] As already mentioned, in Table 6 when the Mistral, CPHydra, and SLR solvers were used some instances were not solved. Hence, in these cases the results indicate only a lower bound. When the oracle and random instance selectors are considered the results are similar and present just one inversion amongst the rankings. Indeed, the oracle solver selector is the best one, followed by Fs, SLR, and SLR$^+$. The use of the portfolio-based solver CPHydra outperforms Mistral when the random instance selector is chosen; it is worse in the other cases. When the BC instance selector is chosen the SLR solver selector ranks higher than Fs, and SLR$^+$ becomes the worst solver selector, excluding the random solver selector that is always the worst one. We would like to underline that in this case even if SLR has lower average solving times it cannot solve all instances. Thus for a fair comparison the Fs should be compared against the SLR$^+$ solver selector that enhances the SLR selector allowing us to solve all instances in the dataset.

[12] here we do not present all the possible combinations of instance selectors and solver selectors for space reasons

Table 5: Performance of classification-based portfolios. ∘, • indicate statistically significant improvement or degradation over instance selector BC.

instance sel.	solver sel.	average
oracle	oracle	1242.67 ∘
BC	oracle	4215.63
LR	oracle	6171.37 •
random	oracle	15412.25 •

instance sel.	solver sel.	average
BC	random	52159.37
oracle	random	61919.52 •
LR	random	78560.97 •
random	random	156312.27 •

instance sel.	solver sel.	average
oracle	fs	6666.42 ∘
BC	fs	13159.58
LR	fs	17309.40 •
random	fs	36614.87 •

Based on these results it seems that approaches based on the use of classifiers for ordering instances and selecting solvers are reasonable and, compared against approaches based on linear regression, they allow an improvement of the average solving times or the resolution of more instances.

7 Related Work

The three closest approaches to solver portfolio management are CPHydra, SATzilla, and Acme. CPHydra, using a CBR system for configuring a set of solvers to maximize the chances of solving an instance in 1800 seconds, was overall winner of the 2008 International CSP Solver Competition. Gebruers et al. [2] also use case-based reasoning to select solution strategies for constraint satisfaction. In contrast, SATzilla [19] relies on runtime prediction models to select the solver from its portfolio that (hopefully) has the fastest running time on a given problem instance. In the International SAT Competition 2009, SATzilla won all three major tracks of the competition. An extension of this work has focused on the design of solver portfolios [18]. The Acme system is a portfolio approach to solve quantified Boolean formulae, i.e. SAT instances with some universally quantified variables [13]. Streeter et al. [15] use optimization techniques to produce a schedule of solvers that should be executed in a specific order, for specific amounts of time, in order to maximize the probability of solving the given instance.

In [4], a classification-based algorithm selection for a specific CSP is studied. Given an instance of the bid evaluation problem (BEP), the objective is to be able to decide a-priori whether an Integer Programming (IP) solver, or an hybrid one between IP and CP (HCP) will be the best. Such a selection is done on the basis of the instance structure which is determined via (a subset of) 25 static features derived from the constraint graph [8]. These features are extracted on a set of training instances and the corresponding best approach is identified.

Table 6: Performance of classification-based portfolios. ∘, • indicate statistically significant improvement or degradation over solver selector Fs.

instance sel.	solver sel.	average
oracle	oracle	1242.67 ∘
oracle	Fs	6666.42
oracle	SLR	> 8763.23 •
oracle	SLR$^+$	11610.11 •
oracle	mistral	> 14051.78 •
oracle	CPHydra	> 15133.02 •
oracle	random	61919.52 •

instance sel.	solver sel.	average
random	oracle	15412.25 ∘
random	Fs	36614.87
random	SLR	> 43006.82 •
random	SLR$^+$	54180.92 •
random	CPHydra	> 62236.14 •
random	mistral	> 69899.70 •
random	random	156312.27 •

instance sel.	solver sel.	average
BC	oracle	4215.63 ∘
BC	SLR	> 12702.69 ∘
BC	Fs	13159.58
BC	mistral	> 13459.71 •
BC	CPHydra	> 16298.59 •
BC	SLR$^+$	17043.58 •
BC	random	52159.37 •

The resulting data is then given to a classification algorithm that builds decision trees. Our objective in this paper is not only to be able to predict the best solver for a given instance but also to choose the right instance at the right time to minimize the average finishing time of the set of instances. Consequently we develop multiple classifiers and utilize them so as to predict their order of difficulty. Moreover, our features are general-purpose and our approach works for any CSP in the XCSP format with any of the related solvers. Furthermore, we take into account a set of dynamic features which provide complementary information.

Also related to our work is the instance-specific algorithm configuration tool ISAC [6]. Given a highly parameterized solver for a CSP instance, its purpose is to tune the parameters based on the characteristics of the instance. Again, such characteristics are determined via static features and extracted from the training instances. Then the instances are clustered using the g-means algorithm, the best parameter tuning for each cluster is identified, and a distance threshold is computed which determines when a new instance will be considered as close enough to the cluster to be solved with its parameters. The fundamental difference with our approach is that instances that are likely to prefer the same solver are grouped with a clustering algorithm based on their features. We instead do not use any clustering algorithm. We create clusters ourselves according to the observed performance of the solvers on the instances and predict which cluster an instance belongs based on its features using classification algorithms.

8 Conclusions

We have presented a novel approach to managing a portfolio for constraint solving. We proposed the use of classifier techniques as a basis for making high-level and qualitative statements about the solvability of CSP instances with respect to a given solver portfolio. We showed how these could then be used for solving a collection of problem instances. While this approach is conceptually very simple, we demonstrated that using classifiers to develop dispatching rules for a solver portfolio is very promising. The code for computing the CPHydra features and the simulator is available at `www.cs.unibo.it/~jmauro/cpaior_2012.html`.

The work presented here is a first step towards the ambitious goal of developing an on-line service-based portfolio solver that receives CSPs from a user, exploits multiple processing nodes to search for solutions as quickly as possible. In the future, we will study distributed strategies for systems having more than one processor. In this setting, using more than one solver in parallel for the same instance could also be useful. For instance, consider running all solvers in parallel for instances that are difficult for all solvers but one, and running only the best solver for all the other instances. This strategy looks promising since it will not waste resources running all the solvers for all the instances and, at the same time, it minimizes the risk of choosing the wrong solver for some instances.

In this paper, our classes of run times were manually extracted. As part of future work we consider to automate this task using clustering techniques. In addition, we will investigate the benefit of using automatic algorithm tuning tools like GGA [1] and ParamILS [5] to train a larger portfolio of solvers. It has been observed in ISAC and Hydra that additional performance benefits can be achieved with solvers that have been expressly tuned for a particular subset of problem types.

Finally we would like to exploit the solving statistics (e.g. solving times, memory consumption) obtained at run time to improve on-the-fly the predictions of the models. This goal has been already considered for the QSAT domain [14]. We plan to follow similar ideas using on-line machine learning techniques [16].

References

1. Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *CP*, pages 142–157, 2009.
2. Cormac Gebruers, Brahim Hnich, Derek G. Bridge, and Eugene C. Freuder. Using cbr to select solution strategies in constraint programming. In *ICCBR*, pages 222–236, 2005.
3. Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artif. Intell.*, 126(1-2):43–62, 2001.
4. Alessio Guerri and Michela Milano. Learning techniques for automatic algorithm portfolio selection. In *ECAI*, pages 475–479, 2004.
5. 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.

6. Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. Isac - instance-specific algorithm configuration. In *ECAI*, pages 751–756, 2010.
7. Zeynep Kiziltan and Jacopo Mauro. Service-oriented volunteer computing for massively parallel constraint solving using portfolios. In *CPAIOR*, pages 246–251, 2010.
8. Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *CP*, pages 556–572, 2002.
9. Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.
10. Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.
11. Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding random sat: Beyond the clauses-to-variables ratio. In *CP*, pages 438–452, 2004.
12. 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.
13. Luca Pulina and Armando Tacchella. A multi-engine solver for quantified boolean formulas. In *CP*, pages 574–589, 2007.
14. Luca Pulina and Armando Tacchella. Time to learn or time to forget? strengths and weaknesses of a self-adaptive approach to reasoning in quantified boolean formulas. In *CP Doctoral Program*, 2008.
15. Matthew J. Streeter, Daniel Golovin, and Stephen F. Smith. Combining multiple heuristics online. In *AAAI*, pages 1197–1203, 2007.
16. Vladimir Vovk, Alex Gammerman, and Glenn Shafer. *Algorithmic Learning in a Random World*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
17. Richard J. Wallace and Eugene C. Freuder. Supporting dispatchability in schedules with consumable resources. *J. Scheduling*, 8(1):7–23, 2005.
18. Lin Xu, Holger Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *AAAI*, 2010.
19. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. : The design and analysis of an algorithm portfolio for sat. In *CP*, pages 712–727, 2007.