Instance-based Selection of CSP Solvers using Short Training*

Mirko Stojadinović[1] and Filip Marić[2]

[1] Faculty of Mathematics,
University of Belgrade, Serbia
mirkos@matf.bg.ac.rs
[2] Faculty of Mathematics,
University of Belgrade, Serbia
filip@matf.bg.ac.rs

Abstract

Many different approaches for solving Constraint Satisfaction Problems (CSP) (and related Constraint Optimization Problems (COP)) exist. However, there is no single solver that performs well on all classes of problems and many portfolio approaches for selecting a suitable solver based on simple syntactic features of the input CSP instance are developed. In this paper we present a portfolio method for CSP based on k-nearest neighbors method. Unlike other existing portfolio approaches for CSP, our methodology is based on training with very short timeouts, thus significantly reducing overall training time. Still, thorough evaluation has been performed on large publicly available corpora and our portfolio method gives good results. The method improves upon the efficiency of single state-of-the-art tools used in comparison, and is comparable to classical methods that use long timeout during the training phase.

1 Introduction

Constraint satisfaction problems (CSP) (and related *Constraint optimization problems (COP)*) [2] over finite domains are wide classes of problems that include many problems relevant for real world applications (e.g., scheduling, timetabling, sequencing, routing, rostering, planning) [35]. Many different approaches for solving CSP problems exist (e.g., constraint propagation, backtracking search algorithms, local search methods, constraint logic programming, operation research methods, answer set programming) [35] and there are many state-of-the-art solvers that implement these approaches.

It has been recognized that there is no single solver nor single approach suitable for all problems. When solving a CSP instance, one should consider several solvers (and their configurations, if applicable) and carefully choose which one to apply on that specific instance. If a multiprocessor machine is available, one could try to run different solvers in parallel until one of them solves the problem. However, in many cases this is not possible and it is desirable to somehow guess the solver that would give the best results. *Portfolio approaches* that have been successfully used for SAT (e.g., [26, 27, 31, 32, 47]) but also for CSP (e.g., [1, 21, 24, 34]) assume that a number of different solvers is available and for each input instance these approaches select a solver that should be run. This choice is most often based on some syntactic characteristics of the instance to be solved and on the knowledge gained during previous runs of the available solvers on some other instances (*training instances*). When these data are gathered, usually some machine learning technique is applied.

*This work was partially supported by the Serbian Ministry of Science grant 174021 and by SNF grant SCOPES IZ73Z0_127979/1.

One of the problems when applying portfolio approaches is that a large number of instances needs to be solved during the training phase, usually with significant timeout value given for each instance. In consequence, the training phase usually takes a lot of CPU time and most often needs to be performed on a multi-processor machine. In this work we try to address this problem. Namely, in many scenarios (e.g., in solver competitions) instances are grouped into families sharing the same structure, but differing in size and difficulty (a family usually contains different instances of a single problem or several instances of the same origin). Our starting hypothesis for this work is that in such cases it suffices to try all available solvers only on some easier instances from each family, significantly reducing the overall training time. The main purpose of this work is to evaluate this hypothesis experimentally, i.e., to evaluate the effect of the training phase duration to the overall quality of a portfolio approach for CSP.

A very desirable characteristic of portfolio methods is their ability to generalize i.e., they should be able to show good results on instances that were not present in the training set (assuming that these share some common characteristics with the instances that were considered during training). Generalization strength of a portfolio approach is usually checked in two ways. The first is by running solvers on a test set of instances that does not overlap with the training set (e.g., the training and the test sets are two different corpora used in solver competitions). The second is by using techniques such as k-fold cross-validation [3] where all available instances are divided into k parts, and each part is used for testing with other k-1 parts used for training. However, in this work we also consider some practical scenarios where the user only wants to solve a specific, fixed set of instances as fast as possible using solvers that are on his disposal, and where generalization to other instances not from this set is not the primary concern. In this case, the training and the test set overlap, but still many techniques from standard portfolio approaches can be adapted and used.

A portfolio approach that we consider in this work is based on the k-nearest neighbors approach [32]. We consider only finite linear CSP with global constraints. *Finite linear CSP* [40] is a special class of constraint satisfaction problems that is often encountered in applications. *Global constraints* [4] describe relations between a non-fixed number of variables and their purpose is to improve readability and efficiency of CSP solving. We focus on solving CSP problems by *reduction to SAT* [6], *reduction to SMT* [6] and by using *lazy clause generation solvers* [33]. By using three different solving methods we hope to get greater diversity in the efficiency of these approaches, thus increasing the potential efficiency of portfolios. The set of solvers used in our portfolio is determined by the input format of CSP instances and global constraints that they use (e.g., solvers that do not support global constraints occurring in instances cannot be used).

In our previous work ([39]) we have applied the portfolio approach to SAT-based CSP solvers for selecting between different encodings that can be used. In the current work, we extend this by selecting between different available solvers.

Contributions of this work are the following.

- We present a machine-learning methodology for automated instance-based selection of suitable solver for a given CSP instance (Section 3).
- We present a thorough experimental evaluation on several large publicly available corpora using several state-of-the-art solvers, based on reduction to SAT, reduction to SMT and lazy clause generation (Section 4).
- We show that our approach gives good results even when used with a very fast training phase (so that the training does not require an advanced cluster computer but can be done on a single PC). By using reduction to SAT, reduction to SMT and lazy clause

generation we aim to unify the best from all three worlds and achieve significantly better results than by using a single type of solving method with different solvers/encodings.

- Our system *meSAT* [39] supports several encodings for reduction to SAT. In the current work we have extended it to support solving CSP instances by reduction to SMT.

Overview of the paper. In Section 2 we give some basic definitions, describe different solving methods and solvers, and present some of the most well-known portfolio approaches. We describe our portfolio approach in Section 3 and in Section 4 we present results of experimental evaluation of this approach. In Section 5 we draw some final conclusions and present ideas for further work.

2 Background

In this section we will give some background notions used by our system and we will analyze prior results in this area.

2.1 Finite Linear CSP

Definition 1. Linear expressions *over the set of integer variables V are algebraic expressions of the form $\sum_{k=1}^{n} a_k x_k$ where all x_k are variables from V and all a_k are integers.*

A Finite Linear CSP in CNF *is a tuple (V, L, U, B, S) where*

1. *V is a finite set of integer variables,*
2. *$L : V \mapsto \mathbb{Z}$ and $U : V \mapsto \mathbb{Z}$ are lower and upper bound of the integer variable x and these bounds determine the domain $D(x)$ of the variable,*
3. *B is a set of Boolean variables,*
4. *S is a finite set of clauses (over V and B). Clauses are formed as disjunctions of literals where literals are the elements of the union of the sets B, $\{\neg p \,|\, p \in B\}$ and $\{e \leq c \,|\, e$ is linear expression over V, $c \in \mathbb{Z}\}$.*

A Solution *of Finite Linear CSP in CNF is an assignment of Boolean values to Boolean variables and integer values to integer variables satisfying their domains such that when variables are replaced by the values, all clauses from S are satisfied.*

Example 1. *A solution of Finite Linear CSP problem $V = \{x_1, x_2, x_3\}$, $L = \{x_1 \mapsto 1, x_2 \mapsto 1, x_3 \mapsto 2\}$, $U = \{x_1 \mapsto 2, x_2 \mapsto 4, x_3 \mapsto 3\}$, $B = \{p\}$, $C = \{p \vee x_1 + x_3 \leq 4, \neg p \vee x_3 + (-1) \cdot x_1 \leq 0, x_1 \leq 1 \vee 2 \cdot x_2 \leq 4\}$ is the assignment $\{p \mapsto \bot, x_1 \mapsto 1, x_2 \mapsto 3, x_3 \mapsto 2\}$.*

In applications, the input syntax is usually modified so that it allows non-contiguous domains, formulae with arbitrary Boolean structure (not only CNF) and with literals formed by applying other arithmetic relations (e.g., $<$, $\geq$, $>$, $=$) and other arithmetic operations (e.g., integer division, modulo). All these formulae alongside the clauses described in Definition 1 are called *intensional constraints*. Another usual modification of the syntax is the usage of *extensional constraints* (sometimes called *user-defined relations*) that are defined by a table of allowed/disallowed assignments to the variables that they constrain. Both intensional and extensional constraints can be reduced to finite linear CSP in CNF form during preprocessing, but usually more efficient procedures are obtained if these are treated directly.

Example 2. *We give here an example of finite linear CSP specification in the* Sugar *input language [40] that system* meSAT *[39] also uses.*

```
(int x1 1 2) (int x2 1 4) (int x3 2 3)
(imp (>= (+ x1 (* 2 x3)) 3) (and (!= x1 x2) (< x3 (+ x1 x2))))
```

The example uses only intensional constraints. The first row declares the domains of the variables and the second row imposes constraint on these variables. One of the solutions to this problem is the assignment `x1 = 1, x2 = 2, x3 = 2`.

2.2 Global Constraints

A *global constraint*[1] is a constraint that captures a relation between a non-fixed number of variables. There are two main benefits from using global constraints. First, compared to encoding using low-level constraints (that can always be done), specifying problems using global constraints is simpler. This implies better readability of high-level problem specifications. Second, global constraints usually have some structure that can be exploited to solve problem instances more efficiently than by using low level constraints.

As examples, we will describe three global constraints that are frequently used.

The *all-different* constraint. The *all-different* constraint [45] requires that all of its arguments (expressions over integer variables and constants) have different values, i.e., *all-different* $(e_1, \ldots, e_n)$ specifies that $e_i \neq e_j$ for any $i \neq j$.

The *nvalue* constraint The *nvalue* constraint [4] requires that expressions $e_1, \ldots, e_n$ take a number of distinct values that is equal to the value of expression e. For example, the constraint $nvalue(\{x_1, x_2, x_3\}, 3)$ (where $e_i = x_i$ and $e = 3$) states that all three variables have to take three different values.

The *count* constraint. The count constraint [4] requires that the number of occurrences of the value of some specific expression e in the set of expressions $e_1, \ldots, e_n$ is in specific arithmetic relation $(=, \neq, \leq, <, \geq, >)$ with some expression n. For example, $count(\{x_1, x_2, x_3, x_4\},\ 5) > 3$ (where $e = 5$, $e_i = x_i$, the relation is $>$, and $n = 3$) specifies that the value 5 occurs more than 3 times in the set of variables $\{x_1, x_2, x_3, x_4\}$. This implies that all variables x_1, x_2, x_3 and x_4 need to take the value 5.

2.3 Systems used for modeling and solving CSP

Modeling languages. Before solving, a constraint satisfaction problem must be somehow specified and many modeling languages for this purpose exist. MiniZinc [30] is a constraint modeling language which is compiled by a variety of solvers to the low-level target language FlatZinc for which many solvers exist. XCSP [36] is an XML-like low-level format used in several CSP solving competitions. Instances of this format can be directly translated to Sugar input format, which is even simpler as it does not use any tags.

[1] A catalogue of global constraints [4] is available online: `http://www.emn.fr/z-info/sdemasse/gccat`

Reduction to SAT. *Propositional satisfiability problem (SAT)* [6] is the problem of deciding if there is a truth assignment under which a given propositional formula (in conjunctive normal form) evaluates to true. It is a canonical NP-complete problem [12] and it holds a central position in the field of computational complexity. When using reduction to SAT, CSP instances are encoded as SAT instances and modern efficient satisfiability solvers are used for finding solutions that are then converted back to the solutions of the original CSP problems.

A fundamental design choice when encoding finite domain constraints into SAT concerns the representation of integer variables. Several different encoding schemes have been proposed and successfully used in various applications (e.g., the direct encoding [46], the support encoding [18], the log encoding [17], the order encoding [41], the compact-order encoding [42], the log-support encoding [16]). Apart from these standard encoding methods, many custom, problem-specific encodings and corresponding tools have been devised for various applications (e.g., solving resource-constrained project scheduling problem [19]). Also, there are several more general tools that reduce CSP to SAT using one or more of several standard encodings (e.g., Spec2SAT [9], FznTini [20], Sugar [40], Azucar [43], ursa [23], bee [29], *meSAT* [39]).

NPSpec [8] is a PROLOG-like declarative modeling language. Each problem specification consists of a database of facts and specification of constraints. Spec2SAT is an application that allows the compilation of NPSpec specifications (when given together with input data) into SAT instances.

FznTini [20] introduces a translation for FlatZinc constraint models, such that any satisfaction or optimization problem written in FlatZinc (not involving floating point numbers) can be automatically Booleanized and solved by one or more calls to a SAT solver.

Sugar is a constraint solver that solves finite linear CSPs by translating them into SAT by using order encoding method [41] and then solving SAT instances by several supported SAT solvers.

Azucar [43] is a successor of Sugar that uses the compact-order encoding [42] for translating finite linear CSP into SAT. It is tuned for solving specific large domain sized CSP instances. Log encoding is a special case of compact-order encoding so Azucar can also use this encoding when reducing to SAT.

ursa family of tools (ursa, URBiVA, ursa major) [23, 28] introduce uniform reductions of C-like language specifications to SAT or to different SMT theories. The translation has a precise semantics, communication with SAT/SMT solvers is done using their APIs and finding all models is supported.

bee [29] (Ben-Gurion University Equi-propagation Encoder) is a constraint specification language and a compiler to CNF based on the order-encoding [41], similar to Sugar, but applying several optimizations.

meSAT [39] (Multiple Encodings of CSP to SAT) is a system using different encodings and their combinations. The supported encodings are: direct [46], support [18], direct-support [39], order [41] and direct-order [39].

Reduction to SMT. *Satisfiability modulo theories (SMT)* [6] is a research field concerned with the satisfiability of formulae with respect to some decidable background theory (or combination of them). Some of these theories are *Linear Integer Arithmetic*, *Integer Difference Logic*, *Linear Real Arithmetic*, etc. There are several systems that solve CSP problems by reduction to SMT.

`fzn2smt` [7] tool is a compiler from FlatZinc modeling language to the SMT-LIB language[2] that is a standard input language of SMT solvers.

[2] `http://www.smtlib.org`

URSA family of tools, as stated in previous paragraph, introduces reduction to different SMT theories.

Lazy clause generation. In *lazy clause generation* approach [33], finite domain propagation engine is combined with SAT solver: propagators are mapped into clauses and passed to SAT solver, which uses unit propagation and then returns information obtained back to the engine. In contrary to the eager approach, clauses are not generated a priori but are constructed and given to the SAT solver during the solving phase. The lazy propagation approach can be viewed as a special form of Satisfiability Modulo Theories solver, where each propagator is considered as a separate theory, and theory propagation is used to learn clauses. Solvers `mzn-g12cpx` and `mzn-g12lazy` (included in the MiniZinc G12 distribution) implement lazy clause generation.

2.4 Solver selection for SAT and CSP

The instance-based algorithm selection problem has been widely studied in the SAT community. Based on the characteristics of the input instance, either some parameters of a single solver are tuned, or one of several available solvers (so called solver portfolio) is selected to be applied on an instance. The most successful results are based on machine-learning techniques (e.g., SATZilla [47], ISAC [27], ArgoSmArT [31, 32], Non-Model-Based Algorithm Portfolios for SAT [26]). Each SAT instance is characterized by a set of its features (most of them are purely syntactic and extracted from the CNF representation). Usually, a training corpus is solved by different SAT solvers (or a single solver configured by different parameters) and a prediction model is formed. When a new instance is to be solved, the most suitable solver is chosen, based on input features of instance and the prediction model.

Algorithm portfolios have recently been applied to constraint satisfaction. CPHYDRA [34] is an algorithm portfolio for CSP that uses case-based reasoning to determine how to solve an unseen problem instance by exploiting a case base of problem solving experience. The superiority of the portfolio over each of its constituent solvers is demonstrated using challenging benchmark problem instances from the most recent CSP Solver Competition. Another approach by Kiziltan et al. [24] uses run-time classifiers (categories are: "short", "medium" and "long") to minimize the average completion time of each instance. This portfolio uses features of CPHYDRA and SATZilla and the combination of two. Work by Amadini et al. [1] compares efficiency of different portfolio approaches based on SAT portfolio techniques and machine learning algorithms. Hurley et al. [21] presents a hierarchical portfolio approach named *Proteus* where for the instance to be solved, first, the type of solving is chosen. In case of reduction to SAT, the encoding is chosen and then also the solver to be used. The instances used in the experiments of that paper contain only several types of constraints (e.g., global constraints do not occur in these instances).

3 Instance-based Solver Selection

In this section, we describe a portfolio approach that automatically selects solver to be used, based on the features of the constraints used in CSP input instance.

Our approach is a modification of the portfolio approach ArgoSmArT-*k*-*NN*, introduced by Nikolić et al. [32]. In the original formulation, the portfolio chooses between different SAT solving methods (different SAT solvers, or a single SAT solver with different setups) when solving a SAT instance. Instead, we choose between different CSP solvers when solving a CSP instance. The instance-based solver selection consists of *training phase* and *testing phase*.

First, a training set of instances is fixed and a *training phase* is conducted. For each instance from the training set, its features are extracted and different CSP solvers are applied with a given time limit. For each such instance and the applied solver the PAR10 score (penalty) [22] is calculated — solving time if the instance is solved within a given time limit, or the time limit multiplied by 10, otherwise.

Let us fix the number of neighbors (k) and the distance measure (d). For a fixed training instance, k-nearest neighbors (k-*NN*) from the training set are found (with respect to the extracted features of training instances and a fixed distance measure). For each solver, the sum of PAR10 scores of k nearest neighbors is calculated (for the fixed instance). The solver with the minimal sum is selected for this instance and we call PAR10 score of this solver *individual score*. The sum of individual scores of all training instances is called *total score*.

Different number of neighbors (k) and distance measures (d) are considered in training phase. The combination giving the best total score (the smallest number) is used in *testing phase*. When a new instance (*test instance*) is to be solved, its features are extracted and k-nearest neighbors (k-*NN*) from the training set are found. The solver with the minimum individual score for this instance is selected to solve it (off course, the aim is to choose this solver so that it is the best solver for this instance). In both the training and the testing phase if the same (minimal) scores are obtained for several solvers then the one of them is selected, based on some fixed priority. Note that the time limit used when calculating PAR10 score does not have to be the same as the one used for solving the test instance.

In some k-*NN* approaches it can be useful to weight the contributions of the neighbors, so that the nearer neighbors contribute more to the score than the more distant ones. However, in our approach weights are not used, and the distance of neighboring instances is used only to determine the set of k-nearest neighbors.

Unlike some other approaches (e.g., [24]) that use features of the generated SAT instances, we use only features extracted from the original CSP formulation. We considered 70 different features[3] divided in several groups: features related to the number and the percentage of the constraints of different types — intensional (e.g., percentage of intensional constraints among all the constraints), extensional, global (e.g., average arity of global constraints), as well as for each specific type of constraint (e.g., number of arithmetic constraints, number of multiplications, sum of domains of variables involved in multiplications, number of *all-different* constraints), features related to the sizes of the domains of integer variables for all variables in the instance (e.g., average domain size), and for the variables included in each different type of constraint, features related to the number of all variables and variables with non-contiguous domains, etc.

Example 3. *We give here a simple CSP instance and calculate some of its features.*

```
(int x1 0 3)(int x2 0 4)(int x3 1 5)
(alldifferent x1 x2 x3)
(<= 6 (+ x1 x2))
```

Sum of the sizes of the domains of variables involved in addition or subtraction is 9 (`x1` *can take 4 and* `x2` *can take 5 values). Number of occurrences of global constraints is 1 (the occurrence of all-different constraint). Number of occurrences of intensional constraints is 2 (one addition and one comparison). The average arity of global constraints is 3 (the only global constraint has 3 operands). Percentage of global constraints considering all constraints is 33% (all constraints are the all-different constraint, comparison and addition).*

[3] A detailed description of all 70 features is available at `http://jason.matf.bg.ac.rs/~mirkos/Mesat.html`

4 Experimental Results

In order to show the efficiency of automated solver selection, we conducted an experimental evaluation. For reduction to SAT, we used direct encoding implemented in *meSAT* [39], order encoding implemented in Sugar [40], and log and compact-order encoding implemented in Azucar [43]. In all experiments and tools, SAT solver Minisat 2.2 [14] was used for solving generated CNF instances. We extended *meSAT* to enable reduction of CSP instances to SMT-LIB language[4]. The translation of most constraints is straightforward, and only global constraints are decomposed to more simpler constraints. Solver Yices [13] was used for solving generated SMT-LIB instances. Lazy clause generation solvers `mzn-g12lazy` and `mzn-g12cpx` were also used [33].

Instances. We used three corpora of CSP problems: (i) CPAI09 containing all problems used in Fourth International CSP Solver Competition[5] that use global constraints, (ii) MiniZinc containing 29 problems from MiniZinc corpus[6] also encoded to use global constraints[7], and (iii) instances of the *Dominating Queens* problem, described in the global constraints catalogue[8]. For each problem, these corpora include several instances that differ in the size of the problem and specific input data. We used two formats of input files: MiniZinc language [30] and Sugar input language [40].

Instances from the first corpus were automatically converted from the original input language to MiniZinc by the converter `xcsp2zinc` available on MiniZinc page[9] and to Sugar input format by the converter included in the Sugar distribution. Ten instances of problem *nengfa* could not be converted by `xcsp2zinc` and they are omitted from the experiments.

Instances from the second corpus are already in MiniZinc input format and use original input data (specified in .dzn files from MiniZinc corpora). New problem descriptions in Sugar input language were made (.mzn files were not directly used due to different types of constraints supported by these tools).

The third corpus is formed in order to include instances with *nvalue* constraint in experiments. For this purpose, the instances of the Dominating Queens problem are used. Specification of the problem is the same for MiniZinc and Sugar input language.

Experimental environment. All tests were performed on a multiprocessor machine with AMD Opteron(tm) CPU 6168 on 1.9Ghz with 2GB of RAM per CPU, running Linux. Testing timeout was 600 seconds for each instance (for total time including selecting the solver where needed, encoding where needed, and solving).

As different solvers are suitable for different problems, there is a good motivation to use some instance-based solver selection scheme, and we applied the approach described in Section 3. We compare our instance based selection scheme to the (i) *oracle* (or virtual best) method that would select the best solver for each instance (this method is not feasible in practice since it makes perfect decisions and for each instance it must guess the optimal solver before trying

[4]The source code of our implementation and the instances used in experiments (but without third-party solvers, due to specific licensing) are available online from: `http://jason.matf.bg.ac.rs/~mirkos/Mesat.html`

[5]`http://www.cril.univ-artois.fr/CPAI09`

[6]`http://www.minizinc.org`

[7]We chosen these instances for our previous experiments with system *meSAT*. We plan to use more instances in our future experiments, not only the ones containing global constraints.

[8]`http://www.emn.fr/z-info/sdemasse/gccat/Cnvalue.html#uid22241`

[9]`http://www.minizinc.org`

Solving method	Solver	# (out of 1379)	Time (minutes)
Reduction to SAT	*meSAT* (direct)	945	4755
	Sugar (order)	1051	3862
	Azucar (compact-order)	959	4713
	Azucar (log)	916	5278
Reduction to SMT	Yices	771	6456
Lazy clause generation	`mzn-g12cpx`	707	7002
	`mzn-g12lazy`	845	5824
	best-fixed	1051	3862
	oracle	1191	2200

Table 1: Summary results of experimental evaluation on all instances; the encoding used for reduction to SAT is given in parentheses; # denotes the number of solved instances; *best-fixed* – the single solver achieving the best results (Sugar in this case), *oracle* – the best solver for the instance.

to solve it, which is impossible to implement) and (ii) the *best-fixed* method – one solver that gives the best overall performance.

We used 70 features, described in Section 3, all extracted only from instance input files. The time used for the feature extraction is small (about 0.05 seconds in average on all instances).

Given a training set, all its instances are solved using each of the included solvers in a given training timeout, and then the optimal parameters (the number of neighbors k and the distance measure d) are selected in the way described in Section 3. All combinations of k (ranging from 1 to 20) and 4 different distance measures ([44]) are tried on the training set and the ones generating the best score are declared optimal.

Training phase results (solving times of individual training instances, optimal value of parameter k, and optimal distance measure d) are used to select the solver for a given test instance. If the same (best) scores are obtained for more solvers, then the priority of choosing equals the ordering of solvers when looking total results on all instances used in experiments. When the solver is selected, it is invoked for that test instance, given a testing timeout of 600 seconds.

For later in discussion we applied all the solvers on all the instances with the timeout of 600 seconds. The total results are shown in Table 1. The separate results for different corpora are not shown, but only aggregate results. The number of solved instances and total solving time (in minutes) are shown (for each unsolved instance 600 seconds are added to the total time). Mean time spent on an instance is directly computable from the total time and the number of instances. The results indicate that there are many instances easily solved by these state-of-the-art solvers. Still, the difference between the best-fixed and the oracle is 140 instances, so, it makes sense to apply some portfolio approach.

The effect of training time on the portfolio effectiveness. In this group of experiments the training is done on random portions of the corpus and the testing is done on the rest (i.e., the cross-validation [3] approach is used for evaluation). Therefore, 5-fold cross-validation is used, and this method is denoted by $auto_{cv}$. With this method, the training and the test set never overlap, and we can estimate the potential generalization strength of our approach. The corpus was divided in 5 equal parts, testing on each part after training on other 4 parts, and reporting the total results for all 5 parts. To estimate the effect of training time on the portfolio effectiveness, the experiment is repeated several times, with different values of timeouts used

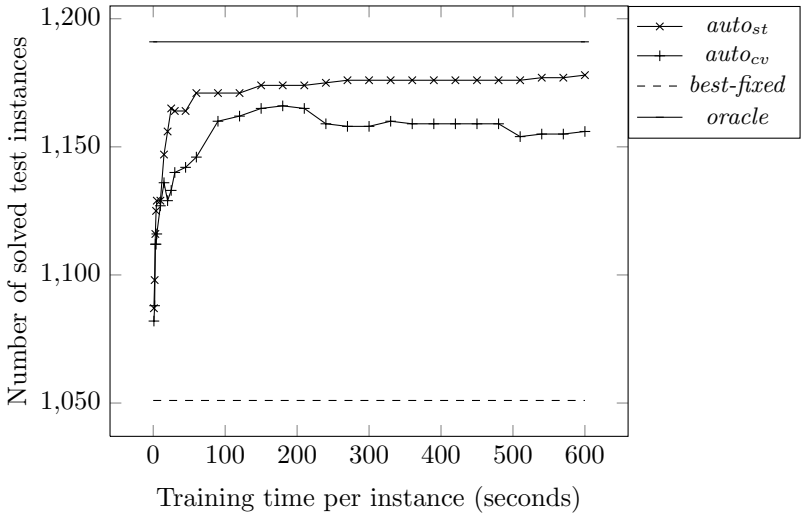

Figure 1: Summary results of experimental evaluation on all instances of $auto_{cv}$ – automatically selected solver using cross validation, $auto_{st}$ – automatically select solver based on short training, *best-fixed* solver and *oracle*; each mark on curve represents one conducted testing based on training with the corresponding timeout.

for training, ranging from 1 to 600 seconds (but all the time the corpus was partitioned to the same 5 parts). The training is done 5 times independently and each time different parameters (k and d) are obtained and used for testing. The obtained value of k ranges from 5 to 19, with bigger values used more often. No distance measure d turned out to be significantly better than the others.

Training timeout	1	5	10	30	60	90	120	300	600
Solved instances	1082	1116	1136	1140	1146	1160	1162	1158	1156
Total training time	145	622	1114	2733	4916	6981	8977	20239	37893
Total solving time	3467	3020	2759	2764	2696	2429	2540	2597	2609
Total time	3612	3642	3873	5497	7612	9410	11517	22836	40502

Table 2: Summary results of experimental evaluation on all instances of $auto_{cv}$ method using different timeouts.

Table 2 shows the obtained results. These results are plotted in Figure 1. The figure shows the results for the $auto_{cv}$ method (but also for the $auto_{st}$ method that is described later). The figure clearly shows that in the beginning there is a sharp increase in the number of solved instances with the increase of training timeout, but after certain threshold the curve stabilizes. E.g., for training timeouts of 600 and 90 seconds the number of solved instances is quite similar, while the overall training time reduces from 28 to 6.5 days.

This demonstrates that in the CSP domain and on corpora that are similar to ours it is plausible to design and use a portfolio approach where timeouts during training are significantly lower than expected timeouts during exploitation.

Training timeout	1	2	3	4	5	10	15	30	300	600
Solved instances	1087	1098	1116	1125	1129	1129	1147	1164	1176	1178
Total training time	145	275	396	512	622	1114	1546	2733	20239	37893
Total solving time	3427	3302	3135	2941	2859	2744	2625	2497	2390	2393
Total time	3572	3577	3531	3453	3481	3858	4171	5230	22629	40286

Table 3: Summary results of experimental evaluation on all instances of $auto_{st}$ method using different timeouts.

Obtaining the best results on a fixed set of instances. In some cases users aim to solve as many instances from a single fixed set of instances in the given total time (e.g., they have several days before they must deliver the final results) with the given set of available solvers. A straightforward way would be to choose the single solver that is expected to give the best results (e.g., the one that gave the best results on some solver competition) and apply it on all instances. However, this can be improved using the similar techniques that were used for usual portfolio approaches. Due to the nature of the problem (only a single fixed corpus is relevant), the generalization of the portfolio to other corpora is not important, so the overlapping between the training and the test set of instances can be tolerated.

In our experiments we performed both training and testing on the same set of instances (the union of our three corpora), and we denote this approach by $auto_{st}$. When applying this method, all instances from the corpus are initially solved with the given training timeout. Parameters k and d are obtained in training phase as described in Section 3. The obtained parameters are used for each test instance to select k-nearest neighbors from the training set. Note that as training and test set are equal, the nearest neighbor from the training set is always the same as the test instance for which the solver is selected. Again, we used different training timeouts and in this scenario we focused on using very short timeout values. The results are shown in Table 3 and plotted in Figure 1. The same trend presented for $auto_{cv}$ method also applies – the total number of solved instances rapidly increases for small training timeouts and then stabilizes.

It seems that $auto_{st}$ outperforms $auto_{cv}$ method, but this can be expected as the same instances that were solved during training are used in testing. In practice, solving time of $auto_{st}$ method can be even smaller as some easy instances are already solved during training within the given training timeout and there is no need to solve them again during testing. However, Table 3 presents the results where the selected solver was started again for each test instance.

In a practical scenario, central quality measures are the number of solved instances and the total time spent for both training and testing – one should try to maximize the number of solved instances but keep the total time spent within the given time limits (e.g., so that both training and exploitation can be performed on a PC that user has on his disposal). Surprisingly, our experiments show that even for extremely small training timeouts, the number of solved test instances is greater than for the best-fixed solver (e.g., for the training timeout of only 1 second, the number of solved instances increases from 1051 to 1087, while the total time reduces from 3862 to 3572 minutes – 145 minutes are used for training and 3427 are used for testing).

The overall training time increases approximately linearly with the increase of the training timeout. On the other hand, the overall solving time of all test instances sharply decreases with the increase of the training timeout and then stabilizes. Therefore, their sum exhibits the behavior plotted in Figure 2. One can see that the total time is minimal for the training timeout of 4 seconds. For the training timeout of 10 seconds, the total time is approximately

the same as the best-fixed method, but 78 more instances are solved. We argue that in practical scenario it makes sense to use training timeout of approximately 60 seconds as in that case the total number of solved instances is reached and the total time is still reasonable (e.g., if we allowed twice as much time for training, the overall time would significantly increase from 7346 to 11415 minutes, and no new instances would be solved).

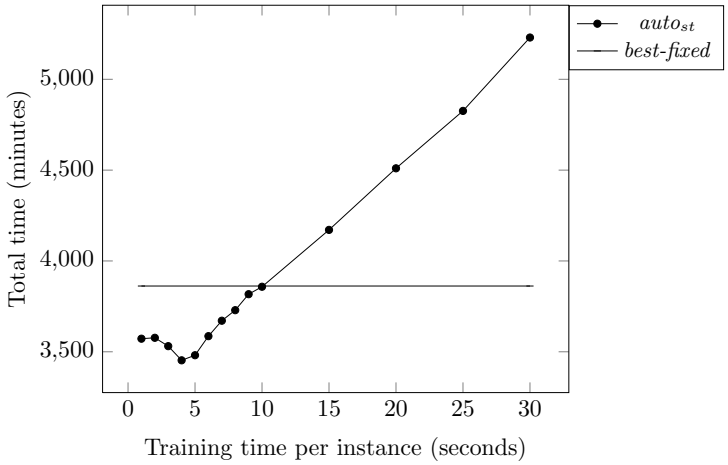

Figure 2: Summary results of experimental evaluation on all instances of $auto_{st}$ and *best-fixed* method; each mark on curve represents one conducted testing based on training with the corresponding timeout.

5 Conclusions and Further Work

In this paper, we have presented a machine-learning methodology for automated instance-based selection of suitable solver for a given CSP instance. We have conducted thorough experimental evaluation on several large publicly available corpora using several state-of-the-art solvers, based on reduction to SAT, reduction to SMT and lazy clause generation. We have focused on the effects of the training phase duration to overall portfolio quality, under the assumption that the corpus is organized into families of instances of the same structure, differing only in size and hardness.

We have tested the quality of our portfolio method using the standard cross-validation technique, for different training timeout values. We have shown that our $auto_{cv}$ approach significantly improves each single constituent solver and gives good generalization results even when used with a very fast training phase (so that the training does not require an advanced cluster computer but can be done on a single PC).

In some scenarios, solving a single fixed set of instances within the given time limit with the given set of solvers is the only practical concern. We have addressed this problem and developed a method denoted by $auto_{st}$ that uses a short training phase on that set of instances (so the training and the exploitation instances overlap) before solving each instance by using the solver selected based on the results of the training phase. Experimental results indicate that our $auto_{st}$ method is very stable and that number of solved instances gradually increases when increasing the training timeout value, but then saturates and from some point increasing

the training timeout values does not affect the number of solved instances. Therefore, one can choose a training timeout value depending on the time he has available and significantly improve the results of every single fixed solver.

Since very small timeouts (e.g., 1s, 5s, or 10s) already improve the number of solved instances, a simple, but robust solver might be obtained if all solvers are run as schedule where each solver has 1 (or 5, or 10) seconds to solve the given problem, and otherwise the finally selected solver is assigned the full run time.

Instead of only choosing solvers, their configuration options can also be selected automatically. For example, our system *meSAT* [39] supports multiple encodings of CSP to SAT (order, direct, support), and the Bee system [29] offers somewhat similar possibilities. Similarly, solvers such as Minisat++ [38] offer 3 different options when encoding PB constraints (adders, BDDs, Sorters), and when selecting Sorters based technique there is the whole phase of selecting which base to use (see, for example, [11, 15]). Our prior experiments with the system *meSAT* [39] have shown that choosing a suitable SAT encoding can be automated, with very promising results. In our further work, it would be good to consider configuration options of a diverse set of solvers (such as the one used in this paper), and with a solver also to choose its suitable configuration.

In our future work a larger and more diverse set of instances should be included in experiments, not only the ones containing global constraints. Also, the set of the used solvers should be extended. It would be also interesting to see if other classifying methods, different from *k-NN*, give similar results. We also plan to compare the efficiency of our portfolio methods for CSP with the existing ones.

Although it would be interesting to know if our conclusions transfer to SAT portfolios as well, we expect worse results (i.e., smaller improvements for short training times), due to the richer structure of CSP instances.

5.1 Acknowledgments

The authors wish to thank Mladen Nikolić for valuable comments on the first versions of this manuscript.

References

[1] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. An empirical evaluation of portfolios approaches for solving csps. In Carla P. Gomes and Meinolf Sellmann, editors, *CPAIOR*, volume 7874 of *Lecture Notes in Computer Science*, pages 316–324. Springer, 2013.

[2] Krzysztof R. Apt. *Principles of constraint programming*. Cambridge University Press, 2003.

[3] Sylvain Arlot, Alain Celisse, et al. A survey of cross-validation procedures for model selection. *Statistics surveys*, 4:40–79, 2010.

[4] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. Global constraint catalog. Technical report, SICS, 2005.

[5] Christian Bessiere, editor. *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*. Springer, 2007.

[6] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[7] Miquel Bofill, Josep Suy, and Mateu Villaret. A system for solving constraint satisfaction problems with smt. In Ofer Strichman and Stefan Szeider, editors, *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 300–305. Springer, 2010.

[8] Marco Cadoli, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile. NP-SPEC: An executable specification language for solving all problems in np. In Gopal Gupta, editor, *PADL*, volume 1551 of *Lecture Notes in Computer Science*, pages 16–30. Springer, 1999.

[9] Marco Cadoli and Andrea Schaerf. : Compiling problem specifications into sat. *Artif. Intell.*, 162(1-2):89–120, 2005.

[10] Alessandro Cimatti and Roberto Sebastiani, editors. *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*. Springer, 2012.

[11] Michael Codish, Yoav Fekete, Carsten Fuhs, and Peter Schneider-Kamp. Optimal base encodings for pseudo-boolean constraints. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2011.

[12] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *STOC*, pages 151–158. ACM, 1971.

[13] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006.

[14] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[15] Yoav Fekete and Michael Codish. Simplifying pseudo-boolean constraints in residual number systems.

[16] Marco Gavanelli. The log-support encoding of csp into sat. In Bessiere [5], pages 815–822.

[17] Allen Van Gelder. Another look at graph coloring via propositional satisfiability. *Discrete Applied Mathematics*, 156(2):230–243, 2008.

[18] Ian P. Gent. Arc consistency in sat. In Frank van Harmelen, editor, *ECAI*, pages 121–125. IOS Press, 2002.

[19] Andrei Horbach. A boolean satisfiability approach to the resource-constrained project scheduling problem. *Annals OR*, 181(1):89–107, 2010.

[20] Jinbo Huang. Universal booleanization of constraint models. In Peter J. Stuckey, editor, *CP*, volume 5202 of *Lecture Notes in Computer Science*, pages 144–158. Springer, 2008.

[21] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry OSullivan. Proteus: A hierarchical portfolio

of solvers and transformations. In *Integration of AI and OR Techniques in Constraint Programming*, pages 301–317. Springer, 2014.

[22] 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.

[23] Predrag Janicic. Uniform reduction to sat. *Logical Methods in Computer Science*, 8(3), 2010.

[24] Zeynep Kiziltan, Luca Mandrioli, Barry OSullivan, and Jacopo Mauro. A classification-based approach to manage a solver portfolio for csps. In *Proceedings of the 22nd Irish Conference on Artificial Intelligence and Cognitive Science, AICS-2011*, 2011.

[25] Oliver Kullmann, editor. *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*. Springer, 2009.

[26] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Non-model-based algorithm portfolios for sat. In Sakallah and Simon [37], pages 369–370.

[27] Yuri Malitsky and Meinolf Sellmann. Instance-specific algorithm configuration as a method for non-model-based portfolio generation. In Nicolas Beldiceanu, Narendra Jussien, and ric Pinson, editors, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, volume 7298 of *Lecture Notes in Computer Science*, pages 244–259. Springer Berlin Heidelberg, 2012.

[28] Filip Maric and Predrag Janicic. Urbiva: Uniform reduction to bit-vector arithmetic. In Jürgen Giesl and Reiner Hähnle, editors, *IJCAR*, volume 6173 of *Lecture Notes in Computer Science*, pages 346–352. Springer, 2010.

[29] Amit Metodi and Michael Codish. Compiling finite domain constraints to sat with bee. *TPLP*, 12(4-5):465–483, 2012.

[30] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Bessiere [5], pages 529–543.

[31] Mladen Nikolić, Filip Marić, and Predrag Janičić. Instance-based selection of policies for sat solvers. In Kullmann [25], pages 326–340.

[32] Mladen Nikolić, Filip Marić, and Predrag Janičić. Simple algorithm portfolio for sat. *Artificial Intelligence Review*, pages 1–9, 2011.

[33] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[34] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science, AICS-2008*, 2008.

[35] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[36] Olivier Roussel and Christophe Lecoutre. Xml representation of constraint networks: Format xcsp 2.1. *CoRR*, abs/0902.2362, 2009.

[37] Karem A. Sakallah and Laurent Simon, editors. *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*. Springer, 2011.

[38] Niklas Sörensson and Niklas Eén. Minisat 2.1 and minisat++ 1.0sat race 2008 editions. *SAT*, page 31, 2009.

[39] Mirko Stojadinović and Filip Marić. meSAT: Multiple Encodings of CSP to SAT. *Constraints*, 2014, doi: 10.1007/s10601-014-9165-7.

[40] Naoyuki Tamura and Mutsunori Banbara. Sugar: A csp to sat translator based on order encoding. In *Proceedings of the third constraint solver competition*, pages 65–69, 2008.

[41] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear csp into sat. *Constraints*, 14(2):254–272, 2009.

[42] Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. A compact and efficient sat-encoding of finite domain csp. In Sakallah and Simon [37], pages 375–376.

[43] Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. Azucar: A sat-based csp solver using compact order encoding - (tool presentation). In Cimatti and Sebastiani [10], pages 456–462.

[44] Andrija Tomovic, Predrag Janicic, and Vlado Keselj. n-gram-based classification and unsupervised hierarchical clustering of genome sequences. *Computer Methods and Programs in Biomedicine*, 81(2):137–153, 2006.

[45] Willem Jan van Hoeve. The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015, 2001.

[46] Toby Walsh. Sat v csp. In Rina Dechter, editor, *CP*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000.

[47] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.