Proteus: A Hierarchical Portfolio of Solvers and Transformations

Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry O'Sullivan

Insight Centre for Data Analytics
Department of Computer Science, University College Cork, Ireland
{b.hurley|l.kotthoff|y.malitsky|b.osullivan}@4c.ucc.ie

Abstract. In recent years, portfolio approaches to solving SAT problems and CSPs have become increasingly common. There are also a number of different encodings for representing CSPs as SAT instances. In this paper, we leverage advances in both SAT and CSP solving to present a novel hierarchical portfolio-based approach to CSP solving, which we call Proteus, that does not rely purely on CSP solvers. Instead, it may decide that it is best to encode a CSP problem instance into SAT, selecting an appropriate encoding and a corresponding SAT solver. Our experimental evaluation used an instance of Proteus that involved four CSP solvers, three SAT encodings, and six SAT solvers, evaluated on the most challenging problem instances from the CSP solver competitions, involving global and intensional constraints. We show that significant performance improvements can be achieved by Proteus obtained by exploiting alternative view-points and solvers for combinatorial problem-solving.

1 Introduction

The pace of development in both CSP and SAT solver technology has been rapid. Combined with portfolio and algorithm selection technology impressive performance improvements over systems that have been developed only a few years previously have been demonstrated. Constraint satisfaction problems and satisfiability problems are both NP-complete and, therefore, there exist polynomial-time transformations between them. We can leverage this fact to convert CSPs into SAT problems and solve them using SAT solvers.

In this paper we exploit the fact that different SAT solvers have different performances on different encodings of the same CSP. In fact, the particular choice of encoding that will give good performance with a particular SAT solver is dependent on the problem instance to be solved. We show that, in addition to using dedicated CSP solvers, to achieve the best performance for solving a CSP the best course of action might be to translate it to SAT and solve it using a SAT solver. We name our approach Proteus, after the Greek god Proteus, the shape-shifting water deity that can foretell the future.

Our approach offers a novel perspective on using SAT solvers for constraint solving. The idea of solving CSPs as SAT instances is not new; the solvers `Sugar`, `Azucar`, and `CSP2SAT4J` are three examples of SAT-based CSP solving. `Sugar` [29]

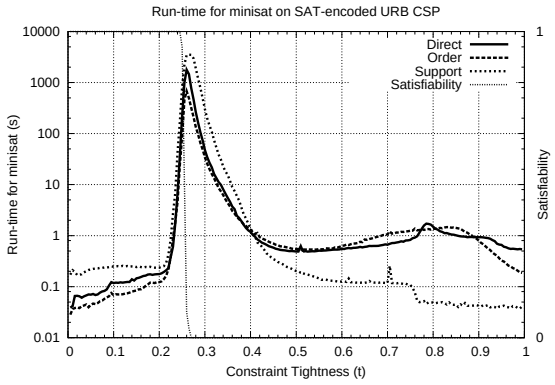
has been very competitive in recent CSP solver competitions. It converts the CSP to SAT using a specific encoding, known as the order encoding, which will be discussed in more detail later in this paper. `Azucar` [30] is a related SAT-based CSP solver that uses the compact order encoding. However, both `Sugar` and `Azucar` use a single predefined solver to solve the encoded CSP instances. Our work does not assume that conversion using a specific encoding to SAT is the best way of solving a problem, but considers multiple candidate encodings and solvers. `CSP2SAT4J` [21] uses the `SAT4J` library as its SAT back-end and a set of static rules to choose either the direct or the support encoding for each constraint. For intensional and extensional binary constraints that specify the supports, it uses the support encoding. For all other constraints, it uses the direct encoding. Our approach does not have predefined rules but instead chooses the encoding and solver based on features of the problem instance to solve.

Our approach employs algorithm selection techniques to dynamically choose whether to translate to SAT, and if so, which SAT encoding and solver to use, otherwise it selects which CSP solver to use. There has been a great deal of research in the area of algorithm selection and portfolios; we refer the reader to a recent survey of this work [20]. We note three contrasting example approaches to algorithm selection for the constraint satisfaction and satisfiability problems: CPHYDRA (CSP), SATZILLA (SAT), and ISAC (SAT). CPHYDRA [24] contains an algorithm portfolio of CSP solvers which partitions CPU-TIME between components of the portfolio in order to maximize the probability of solving a given problem instance within a fixed time limit. SATZILLA [34], at its core, uses cost-sensitive decision forests that vote on the SAT solver to use for an instance. In addition to that, it contains a number of practical optimizations, for example running a pre-solver to quickly solve the easy instances. ISAC [17] is a cluster-based approach that groups instances based on their features and then finds the best solver for each cluster. The Proteus approach is not a straightforward application of portfolio techniques. In particular, there is a series of decisions to make that affect not only the solvers that will be available, but also the information that can be used to make the decision. Because of this, the different choices of conversions, encodings and solvers cannot simply be seen as different algorithms or different configurations of the same algorithm.

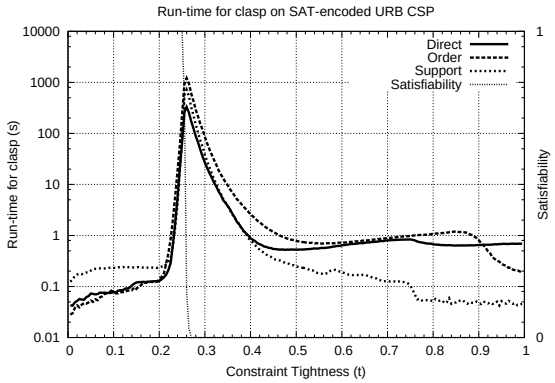
The remainder of this paper is organised as follows. Section 2 motivates the need to choose the representation and solver in combination. In Section 3 we summarise the necessary background on CSP and SAT to make the paper self-contained and present an overview of the main SAT encodings of CSPs. The detailed evaluation of our portfolio is presented in Section 4. We create a portfolio-based approach to CSP solving that employs four CSP solvers, three SAT encodings, and six SAT solvers. Finally, we conclude in Section 5.

2 Multiple Encodings and Solvers

To motivate our work, we performed a detailed investigation for two solvers to assess the relationship between solver and problem encoding with features of



(a) Performance using `MiniSat`.



(b) Performance using `Clasp`.

Fig. 1. `MiniSat` and `Clasp` on random binary CSPs.

the problem to be solved. For this experiment we considered uniform random binary (URB) CSPs with a fixed number of variables, domain size and number of constraints, and varied the constraint tightness. The constraint tightness t is a measure of the proportion of forbidden to allowed possible assignments to the variables in the scope of the constraint. We vary it from 0 to 1, where 0 means that all assignments are allowed and 1 that no assignments are part of a solution, in increments of 0.005. At each tightness the mean run-time of the solver on 100 random CSP instances is reported. Each instance contains 30 variables with domain size 20 and 300 constraints. This allowed us to study the performance of SAT encodings and solvers across the phase transition.

Figure 1 plots the run-time for `MiniSat` and `Clasp` on uniformly random binary CSPs that have been translated to SAT using three different encodings. Observe that in Figure 1(a) there is a distinct difference in the performance of `MiniSat` on each of the encodings, sometimes an order of magnitude. Before the phase transition, we see that the order encoding achieves the best performance

and maintains this until the phase transition. Beginning at constraint tightness 0.41, the order encoding gradually starts achieving poorer performance and the support encoding now achieves the best performance.

Notably, if we rank the encodings based on their performance, the ranking changes after the phase transition. This illustrates that there is not just a single encoding that will perform best overall and that the choice of encoding matters, but also that this choice is dependent on problem characteristics such as constraint tightness.

Around the phase transition, we observe contrasting performance for `Clasp`, as illustrated in Figure 1(b). Using `Clasp`, the ranking of encodings around the phase transition is direct $\succ$ support $\succ$ order; whereas for `MiniSat` the ranking is order $\succ$ direct $\succ$ support. Note also that the peaks at the phase transition differ in magnitude between the two solvers. These differences underline the importance of the choice of solver, in particular in conjunction with the choice of encoding – making the two choices in isolation does not consider the interdependencies that affect performance in practice.

In addition to the random CSP instances, our analysis also comprises 1493 challenging benchmark problem instances from the CSP solver competitions that involve global and intensional constraints. Figure 2 illustrates the respective performance of the best CSP-based and SAT-based methods on these instances. Unsurprisingly the dedicated CSP methods often achieve the best performance. There are, however, numerous cases where considering SAT-based methods has the potential to yield significant performance improvements. In particular, there are a number of instances that are unsolved by any CSP solver but can be solved quickly using SAT-based methods. The Proteus approach aims to unify the best of both worlds and take advantage of the potential performance gains.

3 Background

3.1 The Constraint Satisfaction Problem

Constraint satisfaction problems (CSP) are a natural means of expressing and reasoning about combinatorial problems. They have a large number of practical applications such as scheduling, planning, vehicle routing, configuration, network design, routing and wavelength assignment [26]. An instance of a CSP is represented by a set of variables, each of which can be assigned a value from its domain. The assignments to the variables must be consistent with a set of constraints, where each constraint limits the values that can be assigned to variables.

Finding a solution to a CSP is typically done using systematic search based on backtracking. Because the general problem is NP-complete, systematic search algorithms have exponential worst-case run times, which has the effect of limiting the scalability of these methods. However, thanks to the development of effective heuristics and a wide variety of solvers with different strengths and weaknesses, many problems can be solved efficiently in practice.

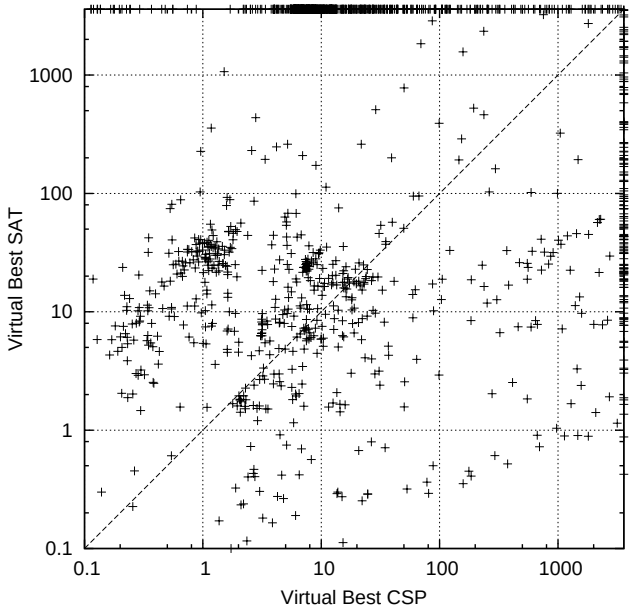


Fig. 2. Performance of the virtual best CSP portfolio and the virtual best SAT-based portfolio. Each point represents the time in seconds of the two approaches. A point below the dashed line indicates that the virtual best SAT portfolio was quicker, whereas a point above means the virtual best CSP portfolio was quicker. Clearly the two approaches are complementary: there are numerous instances for which a SAT-based approach does not perform well or fails to solve the instance but a CSP solver does extremely well, and vice-versa.

3.2 The Satisfiability Problem

The satisfiability problem (SAT) consists of a set of Boolean variables and a propositional formula over these variables. The task is to decide whether or not there exists a truth assignment to the variables such that the propositional formula evaluates to *true*, and, if this is the case, to find this assignment.

SAT instances are usually expressed in conjunctive normal form (CNF). The representation consists of a conjunction of *clauses*, where each clause is a disjunction of *literals*. A literal is either a variable or its negation. Each clause is a logical *or* of its literals and the formula is a logical *and* of each clause. The following SAT formula is in CNF:

$$(x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_3 \vee x_4)$$

This instance consists of four SAT variables. One assignment to the variables which would satisfy the above formula would be to set $x_1 = \mathit{true}$, $x_2 = \mathit{false}$, $x_3 = \mathit{true}$ and $x_4 = \mathit{true}$.

SAT, like CSP, has a variety of practical real world applications such as hardware verification, security protocol analysis, theorem proving, scheduling, rout-

ing, planning, digital circuit design [5]. The application of SAT to many of these problems is made possible by transformations from representations like the constraint satisfaction problem. We will study three transformations into SAT that can benefit from this large collection of solvers.

The following sections explain the direct, support, and direct-order encodings that we use. We will use the following notation. The set of CSP variables is represented by the set $\mathcal{X}$. We use uppercase letters to denote CSP variables in $\mathcal{X}$; lowercase x_i and x_v refer to SAT variables. The domain of a CSP variable X is denoted $\mathrm{D}(X)$ and has size d.

3.3 Direct Encoding

Translating a CSP variable X into SAT using the *direct encoding* [32], also known as the *sparse encoding*, creates a SAT variable for each value in its domain: $x_1, x_2, \ldots, x_d$. If x_v is *true* in the resulting SAT formula, then $X = v$ in the CSP solution. This means that in order to represent a solution to the CSP, exactly one of $x_1, x_2, \ldots, x_d$ must be assigned *true*. We add an *at-least-one* clause to the SAT formula for each CSP variable as follows:

$$\forall X \in \mathcal{X} : (x_1 \vee x_2 \vee \ldots \vee x_d).$$

Conversely, to ensure that only one of these can be set to *true*, we add *at-most-one* clauses. For each pair of distinct values in the domain of X, we add a binary clause to enforce that at most one of the two can be assigned *true*. The series of these binary clauses ensure that only one of the SAT variables representing the variable will be assigned *true*, i.e.

$$\forall v, w \in \mathrm{D}(X) : (\neg x_v \vee \neg x_w).$$

Constraints between CSP variables are represented in the direct encoding by enumerating the conflicting tuples. For binary constraints for example, we add clauses as above to forbid both values being used at the same time for each disallowed assignment. For a binary constraint between a pair of variables X and Y, we add the conflict clause $(\neg x_v \vee \neg y_w)$ if the tuple $\langle X = v, Y = w \rangle$ is forbidden. For intensionally specified constraints, we enumerate all possible tuples and encode the disallowed assignments.

Example 1 (Direct Encoding). Consider a simple CSP with three variables $\mathcal{X} = \{X, Y, Z\}$, each with domain $\langle 1, 2, 3 \rangle$. We have an all-different constraint over the variables: alldifferent(X, Y, Z), which we represent by encoding the pairwise disequalities. Table 1 shows the complete direct-encoded CNF formula for this CSP. The first 12 clauses encode the domains of the variables, the remaining clauses encode the constraints between X, Y, and Z. There is an implicit conjunction between these clauses.

Table 1. An example of the direct encoding.

Domain Clauses	$(x_1 \vee x_2 \vee x_3)$ $(\neg x_1 \vee \neg x_2)$ $(\neg x_1 \vee \neg x_3)$ $(\neg x_2 \vee \neg x_3)$ $(y_1 \vee y_2 \vee y_3)$ $(\neg y_1 \vee \neg y_2)$ $(\neg y_1 \vee \neg y_3)$ $(\neg y_2 \vee \neg y_3)$ $(z_1 \vee z_2 \vee z_3)$ $(\neg z_1 \vee \neg z_2)$ $(\neg z_1 \vee \neg z_3)$ $(\neg z_2 \vee \neg z_3)$
$X \neq Y$	$(\neg x_1 \vee \neg y_1)$ $(\neg x_2 \vee \neg y_2)$ $(\neg x_3 \vee \neg y_3)$
$X \neq Z$	$(\neg x_1 \vee \neg z_1)$ $(\neg x_2 \vee \neg z_2)$ $(\neg x_3 \vee \neg z_3)$
$Y \neq Z$	$(\neg y_1 \vee \neg z_1)$ $(\neg y_2 \vee \neg z_2)$ $(\neg y_3 \vee \neg z_3)$

3.4 Support Encoding

The *support encoding* [9,18] uses the same mechanism as the direct encoding to encode CSP domains into SAT – each value in the domain of a CSP variable is encoded as a SAT variable which represents whether or not it takes that value. However, the support encoding differs on how the constraints between variables are encoded. Given a constraint between two variables X and Y, for each value v in the domain of X, let $S_{Y,X=v} \subset D(Y)$ be the subset of the values in the domain of Y which are consistent with assigning $X = v$. Either x_v is *false* or one of the consistent assignments from $y_1 \dots y_d$ must be true. This is encoded in the support clause

$$\neg x_v \vee \left(\bigvee_{i \in S_{Y,X=v}} y_i \right).$$

Conversely, for each value w in the domain of Y, a support clause is added for the supported values in X which are consistent with assigning $Y = w$.

An interesting property of the support encoding is that if a constraint has no consistent values in the corresponding variable, a unit-clause will be added, thereby pruning the values from the domain of a variable which cannot exist in any solution. Also, a solution to a SAT formula without the *at-most-one* constraint in the support encoding represents an arc-consistent assignment to the CSP. Unit propagation on this SAT instance establishes arc-consistency in optimal worst-case time for establishing arc-consistency [9].

Example 2 (Support Encoding). Table 2 gives the complete support-encoded CNF formula for the simple CSP given in Example 1. The first 12 clauses encode the domains and the remaining ones the support clauses for the constraints. There is an implicit conjunction between clauses.

3.5 Order Encoding

Unlike the direct and support encoding, which model $X = v$ as a SAT variable for each value v in the domain of X, the order encoding (also known as the regular encoding [2]) creates SAT variables to represent $X \leq v$. If X is less than or equal to v (denoted $x_{\leq v}$), then X must also be less than or equal to $v + 1$

Table 2. An example of the support encoding.

Domain Clauses	$(x_1 \vee x_2 \vee x_3)$ $(\neg x_1 \vee \neg x_2)$ $(\neg x_1 \vee \neg x_3)$ $(\neg x_2 \vee \neg x_3)$ $(y_1 \vee y_2 \vee y_3)$ $(\neg y_1 \vee \neg y_2)$ $(\neg y_1 \vee \neg y_3)$ $(\neg y_2 \vee \neg y_3)$ $(z_1 \vee z_2 \vee z_3)$ $(\neg z_1 \vee \neg z_2)$ $(\neg z_1 \vee \neg z_3)$ $(\neg z_2 \vee \neg z_3)$
$X \neq Y$	$(\neg x_1 \vee y_2 \vee y_3)$ $(\neg x_2 \vee y_1 \vee y_3)$ $(\neg x_3 \vee y_1 \vee y_2)$ $(\neg y_1 \vee x_2 \vee x_3)$ $(\neg y_2 \vee x_1 \vee x_3)$ $(\neg y_3 \vee x_1 \vee x_2)$
$X \neq Z$	$(\neg x_1 \vee z_2 \vee z_3)$ $(\neg x_2 \vee z_1 \vee z_3)$ $(\neg x_3 \vee z_1 \vee z_2)$ $(\neg z_1 \vee x_2 \vee x_3)$ $(\neg z_2 \vee x_1 \vee x_3)$ $(\neg z_3 \vee x_1 \vee x_2)$
$Y \neq Z$	$(\neg y_1 \vee z_2 \vee z_3)$ $(\neg y_2 \vee z_1 \vee z_3)$ $(\neg y_3 \vee z_1 \vee z_2)$ $(\neg z_1 \vee y_2 \vee y_3)$ $(\neg z_2 \vee y_1 \vee y_3)$ $(\neg z_3 \vee y_1 \vee y_2)$

$(x_{\leq v+1})$. Therefore, we add clauses to enforce this consistency across the domain as follows:

$$\forall_v^{d-1} : (\neg x_{\leq v} \vee x_{\leq v+1}).$$

This linear number of clauses is all that is needed to encode the domain of a CSP variable into SAT in the order encoding. In contrast, the direct and support encodings require a quadratic number of clauses in the domain size.

The order encoding is naturally suited to modelling inequality constraints. To state $X \leq 3$, we would just post the unit clause $(x_{\leq 3})$. If we want to model the constraint $X = v$, we could rewrite it as $(X \leq v \wedge X \geq v)$. $X \geq v$ can then be rewritten as $\neg X \leq (v-1)$. To state that $X = v$ in the order encoding, we would encode $(x_{\leq v} \wedge \neg x_{\leq v-1})$. A conflicting tuple between two variables, for example $\langle X = v, Y = w \rangle$ can be written in propositional logic and simplified to a CNF clause using De Morgan's Law:

$$\neg((x_{\leq v} \wedge x_{\geq v}) \wedge (y_{\leq w} \wedge y_{\geq w}))$$
$$\neg((x_{\leq v} \wedge \neg x_{\leq v-1}) \wedge (y_{\leq w} \wedge \neg y_{\leq w-1}))$$
$$\neg(x_{\leq v} \wedge \neg x_{\leq v-1}) \vee \neg(y_{\leq w} \wedge \neg y_{\leq w-1})$$
$$(\neg x_{\leq v} \vee x_{\leq v-1} \vee \neg y_{\leq w} \vee y_{\leq w-1})$$

Example 3 (Order Encoding). Table 3 gives the complete order-encoded CNF formula for the simple CSP specified in Example 1. There is an implicit conjunction between clauses in the notation.

3.6 Combining the Direct and Order Encodings

The direct encoding and the order encoding can be combined to produce a potentially more compact encoding. A variable's domain is encoded in both representations and clauses are added to chain between them. This gives flexibility in the representation of each constraint. Here, we choose the encoding which gives the most compact formula. For example, for inequalities we use the order encoding since it is naturally suited, but for a (dis)equality we would use the direct encoding. This encoding is referred to as direct-order throughout the paper.

Table 3. An example of the order encoding.

Domain Clauses	$(\neg x_{\leq 1} \vee x_{\leq 2})$ $(\neg x_{\leq 2} \vee x_{\leq 3})$ $(x_{\leq 3})$ $(\neg y_{\leq 1} \vee y_{\leq 2})$ $(\neg y_{\leq 2} \vee y_{\leq 3})$ $(y_{\leq 3})$ $(\neg z_{\leq 1} \vee z_{\leq 2})$ $(\neg z_{\leq 2} \vee z_{\leq 3})$ $(z_{\leq 3})$
$X \neq Y$	$(\neg x_{\leq 1} \vee \neg y_{\leq 1})$ $(\neg x_{\leq 2} \vee x_{\leq 1} \vee \neg y_{\leq 2} \vee y_{\leq 1})$ $(\neg x_{\leq 3} \vee x_{\leq 2} \vee \neg y_{\leq 3} \vee y_{\leq 2})$
$X \neq Z$	$(\neg x_{\leq 1} \vee \neg z_{\leq 1})$ $(\neg x_{\leq 2} \vee x_{\leq 1} \vee \neg z_{\leq 2} \vee z_{\leq 1})$ $(\neg x_{\leq 3} \vee x_{\leq 2} \vee \neg z_{\leq 3} \vee z_{\leq 2})$
$Y \neq Z$	$(\neg y_{\leq 1} \vee \neg z_{\leq 1})$ $(\neg y_{\leq 2} \vee y_{\leq 1} \vee \neg z_{\leq 2} \vee z_{\leq 1})$ $(\neg y_{\leq 3} \vee y_{\leq 2} \vee \neg z_{\leq 3} \vee z_{\leq 2})$

3.7 Algorithm Portfolios

The Algorithm Selection Problem [25] is to select the most appropriate algorithm for solving a particular problem. It is especially relevant in the context of algorithm portfolios [11,16], where a single solver is replaced with a set of solvers and a mechanism for selecting a subset to use on a particular problem.

Algorithm portfolios have been used with great success for solving both SAT and CSP instances in systems such as SATZILLA [34], ISAC [17] or CPHYDRA [24]. Most approaches are similar in that they relate the characteristics of a problem to solve to the performance of the algorithms in the portfolio. The aim of an algorithm selection model is to provide a prediction as to which algorithm should be used to solve the problem. The model is usually induced using some form of machine learning.

There are three main approaches to using machine learning to build algorithm selection models. First, the problem of predicting the best algorithm can be treated as a classification problem where the label to predict is the algorithm. Second, the training data can be clustered and the algorithm with the best performance on a particular cluster assigned to it. The cluster membership of any new data decides the algorithm to use. Finally, regression models can be trained to predict the performance of each portfolio algorithm in isolation. The best algorithm for a problem is chosen based on the predicted performances.

Our approach makes a series of decisions – whether a problem should be solved as a CSP or a SAT problem, which encoding should be used for converting into SAT, and finally which solver should be assigned to tackle the problem. Approaches that make a series of decisions are usually referred to as hierarchical models. [33] and [12] use hierarchical models in the context of a SAT portfolio. They first predict whether the problem to be solved is expected to be satisfiable or not and then choose a solver depending on that decision. Our approach is closer to [10], which first predicts what level of consistency the `alldifferent` constraint should achieve before deciding on its implementation.

To the best of our knowledge, no portfolio approach that potentially transforms the representation of a problem in order to be able to solve it more efficiently exists at present.

4 Experimental Evaluation

4.1 Setup

The hierarchical model we present in this paper consists of a number of layers to determine how the instance should be solved. At the top level, we decide whether to solve the instance using as a CSP or using a SAT-based method. If we choose to leave the problem as a CSP, then one of the dedicated CSP solvers must be chosen. Otherwise, we must choose the SAT encoding to apply, followed by the choice of SAT solver to run on the SAT-encoded instance.

Each decision of the hierarchical approach aims to choose the direction which has the potential to achieve the best performance in that sub-tree. For example, for the decision to choose whether to solve the instance using a SAT-based method or not, we choose the SAT-based direction if there is a SAT solver and encoding that will perform faster than any CSP solver would. Whether this particular encoding-solver combination will be selected subsequently depends on the performance of the algorithm selection models used in that sub-tree of our decision mechanism. For regression models, the training data is the best performance of any solver under that branch of the tree. For classification models, it is the label of the sub-branch with the virtual best performance.

This hierarchical approach presents the opportunity to employ different decision mechanisms at each level. We consider 6 regression, 19 classification, and 3 clustering algorithms, which are listed below. For each of these algorithms, we evaluate the performance using 10-fold cross-validation. The dataset is split into 10 partitions with approximately the same size and the same distribution of the best solvers. One partition is used for testing and the remaining 9 partitions as the training data for the model. This process is repeated with a different partition considered for testing each time until every partition has been used for testing. We measure the performance in terms of PAR10 score. The PAR10 score for an instance is the time it takes the solver to solve the instance, unless the solver times out. In this case, the PAR10 score is ten times the timeout value. The sum over all instances is divided by the number of instances.

Instances. In our evaluation, we consider CSP problem instances from the CSP solver competitions [1]. Of these, we consider all instances defined using global and intensional constraints that are not trivially solved during 2 seconds of feature computation. We also exclude all instances which were not solved by any CSP or SAT solver within the time limit of 1 hour. Altogether, we obtain 1,493 non-trivial instances from problem classes such as Timetabling, Frequency Assignment, Job-Shop, Open-Shop, Quasi-group, Costas Array, Golomb Ruler, Latin Square, All Interval Series, Balanced Incomplete Block Design, and many others. This set includes both small and large arity constraints and all of the

global constraints used during the CSP solver competitions: all-different, element, weighted sum, and cumulative.

For the SAT-based approaches, Numberjack [15] was used to translate a CSP instance specified in XCSP format [27] into SAT (CNF).

Features. A fundamental requirement of any machine learning algorithm is a set of representative features. We explore a number of different feature sets to train our models: *i*) features of the original CSP instance, *ii*) features of the direct-encoded SAT instance, *iii*) features of the support-encoded SAT instance, *iv*) features of the direct-order-encoded SAT instance and *v*) a combination of all four feature sets. These features are described in further detail below.

We computed the 36 features used in CPHYDRA for each CSP instance using `Mistral`; for reasons of space we will not enumerate them all here. The set includes static features like statistics about the types of constraints used, average and maximum domain size; and dynamic statistics recorded by running `Mistral` for 2 seconds: average and standard deviation of variable weights, number of nodes, number of propagations and a few others. Instances which are solved by `Mistral` during feature computation are filtered out from the dataset.

In addition to the CSP features, we computed the 54 SAT features used by SATZILLA [34] for each of the encoded instances and different encodings. The features encode a wide range of different information on the problem such as problem size, features of the graph-based representation, balance features, the proximity to a Horn formula, DPLL probing features and local search probing features.

Constraint Solvers. Our CSP models are able to choose from 4 complete CSP solvers:

- `Abscon` [22],
- `Choco` [31],
- `Gecode` [8], and
- `Mistral` [14].

Satisfiability Solvers. We considered the following 6 complete SAT solvers:

- `clasp` [7],
- `cryptominisat` [28],
- `glucose` [3],
- `lingeling` [4],
- `riss` [23], and
- `MiniSat` [6].

Learning Algorithms. We evaluate a number of regression, classification, and clustering algorithms using WEKA [13]. All algorithms, unless otherwise stated use the default parameters. The regression algorithms we used were LinearRegression, PaceRegression, REPTree, M5Rules, M5P, and SMOreg. The classification algorithms were BayesNet, BFTree, ConjunctiveRule, DecisionTable, FT, HyperPipes, IBk (nearest neighbour) with 1, 3, 5 and 10 neighbours, J48, J48graft, JRip, LADTree, MultilayerPerceptron, OneR, PART, RandomForest, RandomForest with 99 random trees, RandomTree, REPTree, and SimpleLogistic. For clustering, we considered EM, FarthestFirst, and SimplekMeans. The FarthestFirst and SimplekMeans algorithms require the number of clusters to

Table 4. Performance of the learning algorithms for the hierarchical approach. The 'Category Bests' consists of the hierarchy of algorithms where at each node of the tree of decisions we take the algorithm that achieves the best PAR10 score for that particular decision.

Classifier	Mean PAR10	Number Solved
VBS	97	1493
Proteus	1774	1424
M5P with csp features	2874	1413
Category Bests	2996	1411
M5Rules with csp features	3225	1398
M5P with all features	3405	1397
LinearRegression with all features	3553	1391
LinearRegression with csp features	3588	1383
MultilayerPerceptron with csp features	3594	1382
lm with csp features	3654	1380
RandomForest99 with csp features	3664	1379
IBk10 with csp features	3720	1377
RandomForest99 with all features	3735	1383

be given as input. We evaluated with multiples of 1 through 5 of the number of solvers in the respective data set given as the number of clusters. The number of clusters is represented by $1n$, $2n$ and so on in the name of the algorithm, where n stands for the number of solvers.

We use the LLAMA toolkit [19] to train and test the algorithm selection models.

4.2 Portfolio and Solver Results

The performance of each of the 6 SAT solvers was evaluated on the three SAT encodings of 1,493 CSP competition benchmarks with a time-out of 1 hour and limited to 2GB of RAM. The 4 CSP solvers were evaluated on the original CSPs. Our results report the PAR10 score and number of instances solved for each of the algorithms we evaluate. The PAR10 is the sum of the runtimes over all instances, counting 10 times the timeout if that was reached. Data was collected on a cluster of Intel Xeon E5430 Processors (2.66Ghz) running CentOS 6.4. This data is available online.[1]

The performance of a number of hierarchical approaches is given in Table 4. The hierarchy of algorithms which produced the best overall results for our dataset involves M5P regression with CSP features at the root node to choose SAT or CSP, M5P regression with CSP features to select the CSP solver, LinearRegression with CSP features to select the SAT encoding, LinearRegression with CSP features to select the SAT solver for the direct encoded instance, LinearRegression with CSP features to select the SAT solver for the direct-order

[1] `http://4c.ucc.ie/~bhurley/proteus/`

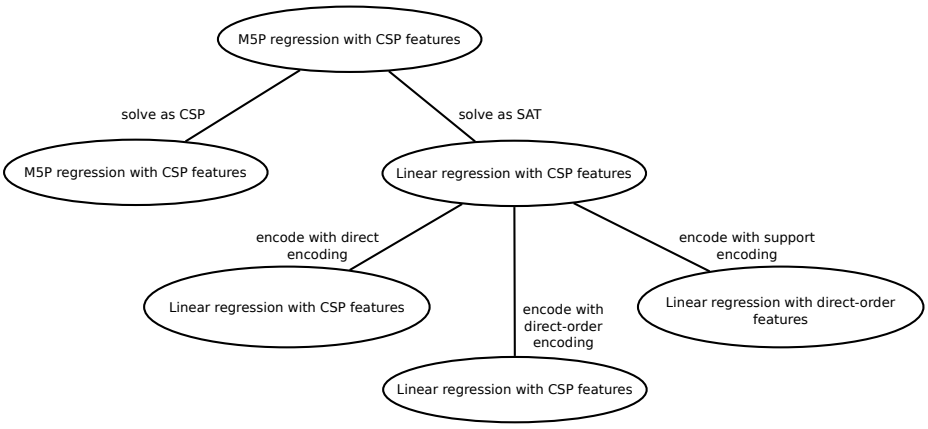


Fig. 3. Overview of the machine learning models used in the hierarchical approach.

encoded instance, and LinearRegression with the direct-order features to select the SAT solver for the support encoded instance. The hierarchical tree of specific machine learning approaches we found to deliver the best overall performance on our data set is labelled Proteus and is depicted in Figure 3.

We would like to point out that in many solver competitions the difference between the top few solvers is fewer than 10 additional instances solved. In the 2012 SAT Challenge for example, the difference between the first and second place single solver was only 3 instances and the difference among the top 4 solvers was only 8 instances. The results we present in Table 4 are therefore very significant in terms of the gains we are able to achieve.

Our results demonstrate the power of Proteus. The performance it delivers is very close to the virtual best (VBS), that is the best performance possible if an oracle could identify the best choice of representation, encoding, and solver, on an instance by instance basis. The improvements we achieve over other approaches are similarly impressive. The results conclusively demonstrate that having the option to convert a CSP to SAT does not only have the potential to achieve significant performance improvements, but also does so in practice.

An interesting observation is that the CSP features are consistently used in each of the top performing approaches. One reason for this is that it is quicker to compute only the CSP features instead of the CSP features, then converting to SAT and computing the SAT features in addition. The additional overhead of computing SAT features is worthwhile in some cases though, for example for LinearRegression, which is at its best performance using all the different feature sets. Note that for the best tree of models (cf. Figure 3), it is better to use the features of the direct-order encoding for the decision of which solver to choose for a support-encoded SAT instance despite the additional overhead.

We also compare the hierarchical approach to that of a flattened setting with a single portfolio of all solvers and encoding solver combinations. The flattened portfolio includes all possible combinations of the 3 encodings and the 6 SAT

Table 5. Ranking of each classification, regression, and clustering algorithm to choose the solving mechanism in a flattened setting. The portfolio consists of all possible combination of the 3 encodings and the 6 SAT solvers and the 4 CSP solvers for a total of 22 solvers.

Classifier	Mean PAR10	Number Solved
VBS	97	1493
Proteus	1774	1424
LinearRegression with all features	2144	1416
M5P with csp features	2315	1401
LinearRegression with csp features	2334	1401
lm with all features	2362	1407
lm with csp features	2401	1398
M5P with all features	2425	1404
RandomForest99 with all features	2504	1401
SMOreg with all features	2749	1391
RandomForest with all features	2859	1386
IBk3 with csp features	2877	1378

solvers and the 4 CSP solvers for a total of 22 solvers. Table 5 shows these results. The regression algorithm LinearRegression with all features gives the best performance using this approach. However, it is significantly worse than the performance achieved by the hierarchical approach of Proteus.

4.3 Greater than the Sum of its Parts

Given the performance of Proteus, the question remains as to whether a different portfolio approach that considers just CSP or just SAT solvers could do better. Table 6 summarizes the virtual best performance that such portfolios could achieve. We use all the CSP and SAT solvers for the respective portfolios to give us VB CSP and VB SAT, respectively. The former is the approach that always chooses the best CSP solver for the current instance, while the latter chooses the best SAT encoding/solver combination. VB Proteus is the portfolio that chooses the best overall approach/encoding. We show the actual performance of Proteus for comparison. Proteus is better than the virtual bests for all portfolios that consider only one encoding. This result makes a very strong point for the need to consider encoding and solver in combination.

Proteus outperforms four other VB portfolios. Specifically, the VB CPHYDRA is the best possible performance that could be obtained from that portfolio if a perfect choice of solver was made. Neither SATZILLA nor ISAC-based portfolios consider different SAT encodings. Therefore, the best possible performance either of them could achieve for a specific encoding is represented in the last three lines of Table 6.

These results do not only demonstrate the benefit of considering the different ways of solving CSPs, but also eliminate the need to compare with existing portfolio systems since we are computing the best possible performance that any

Table 6. Virtual best performances ranked by PAR10 score.

Method	Mean PAR10	Number Solved
VB Proteus	97	1493
Proteus	1774	1424
VB CSP	3577	1349
VB CPHydra	4581	1310
VB SAT	17373	775
VB DirectOrder Encoding	17637	764
VB Direct Encoding	21736	593
VB Support Encoding	21986	583

of those systems could theoretically achieve. Proteus impressively demonstrates its strengths by significantly outperforming oracle approaches that use only a single encoding.

5 Conclusions

We have presented a portfolio approach that does not rely on a single problem representation or set of solvers, but leverages our ability to convert between problem representations to increase the space of possible solving approaches. To the best of our knowledge, this is the first time a portfolio approach like this has been proposed. We have shown that, to achieve the best performance on a constraint satisfaction problem, it may be beneficial to translate it to a satisfiability problem. For this translation, it is important to choose both the encoding and satisfiability solver in combination. In doing so, the contrasting performance among solvers on different representations of the same problem can be exploited. The overall performance can be improved significantly compared to restricting the portfolio to a single problem representation.

We demonstrated empirically the significant performance improvements Proteus can achieve on a large set of diverse benchmarks using a portfolio based on a range of different state-of-the-art solvers. We have investigated a range of different CSP to SAT encodings and evaluated the performance of a large number of machine learning approaches and algorithms. Finally, we have shown that the performance of Proteus is close to the very best that is theoretically possible for solving CSPs and significantly outperforms the theoretical best for portfolios that consider only a single problem encoding.

In this work, we make a general decision to encode the entire problem using a particular encoding. A natural extension would be to mix and vary the encoding depending on attributes of the problem. An additional avenue for future work would be to generalize the concepts in this paper to other problem domains where transformations, like CSP to SAT, exist.

Acknowledgements. This work is supported by Science Foundation Ireland (SFI) Grant 10/IN.1/I3032 and FP7 FET-Open Grant 284715. The Insight Centre for Data Analytics is supported by SFI Grant SFI/12/RC/2289.

References

1. CSP Solver Competition Benchmarks. http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html (2009)
2. Ansótegui, C., Manyà, F.: Mapping Problems with Finite-Domain Variables into Problems with Boolean Variables. In: The 7th International Conference on Theory and Applications of Satisfiability Testing — SAT'04 (2004)
3. Audemard, G., Simon, L.: Glucose 2.3 in the SAT 2013 Competition. Proceedings of SAT Competition 2013 p. 42 (2013)
4. Biere, A.: Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. Proceedings of SAT Competition 2013 (2013)
5. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (February 2009)
6. Een, N., Sörensson, N.: Minisat 2.2. http://minisat.se (2013)
7. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: clasp: A conflict-driven answer set solver. In: Logic Programming and Nonmonotonic Reasoning 2007. pp. 260–265. Springer (2007)
8. Gecode Team: Gecode: Generic Constraint Development Environment (2006), http://www.gecode.org
9. Gent, I.P.: Arc Consistency in SAT. In: Proceedings of the 15th European Conference on Artificial Intelligence — ECAI'2002. pp. 121–125 (2002)
10. Gent, I.P., Kotthoff, L., Miguel, I., Nightingale, P.: Machine learning for constraint solver design – a case study for the alldifferent constraint. In: 3rd Workshop on Techniques for implementing Constraint Programming Systems (TRICS). pp. 13–25 (2010)
11. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence 126(1-2), 43–62 (2001)
12. Haim, S., Walsh, T.: Restart strategy selection using machine learning techniques. In: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing — SAT'09. pp. 312–325. Springer-Verlag, Berlin, Heidelberg (2009)
13. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: An update. SIGKDD Explor. Newsl. 11(1), 10–18 (Nov 2009)
14. Hebrard, E.: Mistral, a Constraint Satisfaction Library. In: Proceedings of the Third International CSP Solver Competition (2008)
15. Hebrard, E., O'Mahony, E., O'Sullivan, B.: Constraint Programming and Combinatorial Optimisation in Numberjack. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010. pp. 181–185 (2010)
16. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science 275(5296), 51–54 (1997)
17. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: ISAC – Instance-Specific Algorithm Configuration. In: Coelho, H., Studer, R., Wooldridge, M. (eds.) ECAI. Frontiers in Artificial Intelligence and Applications, vol. 215, pp. 751–756. IOS Press (2010)
18. Kasif, S.: On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks. Artificial Intelligence 45(3), 275–286 (Oct 1990), http://dx.doi.org/10.1016/0004-3702(90)90009-O

19. Kotthoff, L.: LLAMA: leveraging learning to automatically manage algorithms. Tech. Rep. arXiv:1306.1031, arXiv (Jun 2013), http://arxiv.org/abs/1306.1031
20. Kotthoff, L.: Algorithm Selection for Combinatorial Search Problems: A Survey. AI Magazine (2014), to appear
21. Le Berre, D., Lynce, I.: CSP2SAT4J: A Simple CSP to SAT Translator. In: Proceedings of the Second International CSP Solver Competition (2008)
22. Lecoutre, C., Tabary, S.: Abscon 112, Toward more Robustness. In: Proceedings of the Third International CSP Solver Competition (2008)
23. Manthey, N.: The SAT Solver RISS3G at SC 2013. Proceedings of SAT Competition 2013 p. 72 (2013)
24. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using Case-based Reasoning in an Algorithm Portfolio for Constraint Solving. Proceeding of the 19th Irish Conference on Artificial Intelligence and Cognitive Science (2008)
25. Rice, J.R.: The algorithm selection problem. Advances in Computers 15, 65–118 (1976)
26. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming. Foundations of Artificial Intelligence, Elsevier, New York, NY, USA (2006)
27. Roussel, O., Lecoutre, C.: XML Representation of Constraint Networks: Format XCSP 2.1. CoRR abs/0902.2362 (2009)
28. Soos, M.: Cryptominisat 2.9.0 (2011)
29. Tamura, N., Tanjo, T., Banbara, M.: System Description of a SAT-based CSP Solver Sugar. In: Proceedings of the Third International CSP Solver Competition. pp. 71–75 (2009)
30. Tanjo, T., Tamura, N., Banbara, M.: Azucar: A SAT-Based CSP Solver Using Compact Order Encoding — (Tool Presentation). In: Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing — SAT'12. pp. 456–462. Springer (2012)
31. choco team: choco: an Open Source Java Constraint Programming Library (2008)
32. Walsh, T.: SAT v CSP. In: Principles and Practice of Constraint Programming — CP 2000. vol. 1894, pp. 441–456. Springer-Verlag (2000)
33. Xu, L., Hoos, H.H., Leyton-Brown, K.: Hierarchical hardness models for SAT. In: Principles and Practice of Constraint Programming — CP'07. pp. 696–711 (2007)
34. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based Algorithm Selection for SAT. Journal of Artificial Intelligence Research pp. 565–606 (2008)