Proteus: A Hierarchical Portfolio of Solvers and Transformations

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

Cork Constraint Computation Centre
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 techniques for converting SAT problems into CSPs. In this paper, we leverage advances in both areas and present a novel hierarchical portfolio-based approach to CSP solving that does not rely purely on CSP solvers, but may convert a problem to SAT choosing a conversion technique and the accommodating SAT solver. Our experimental evaluation relies on competition CSP instances and uses eight CSP solvers, three SAT encodings and eighteen SAT solvers. We demonstrate that significant performance improvements can be obtained by considering alternative view-points of a combinatorial problem.

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 show that different SAT solvers have different performance 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 itself is not new. The solvers `Sugar`, `Azucar` and `CSP2SAT4J` are three examples for SAT-based CSP solving. `Sugar` [38] has been very competitive in recent CSP solver competitions. It encodes 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` [39] 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 to SAT is the best way of solving a problem, but considers multiple candidate encodings and solvers.

In contrast to our approach, `CSP2SAT4J` [26] 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, they use the support encoding. For all other constraints they use the direct encoding. Our approach does not have predefined rules but instead chooses the encoding and solver dynamically based on features of the problem to solve.

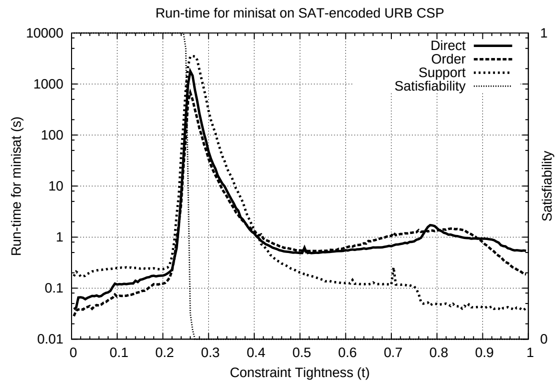
Our approach employs algorithm selection techniques to dynamically choose whether to translate to SAT, and if so, which SAT encoding and the solver to use. There has been a great deal of research in the area of algorithm selection and portfolios; we refer the reader to a recent survey of this work [24]. We note three contrasting example approaches to algorithm selection for the constraint satisfaction and satisfiability problems: CPHYDRA (CSP), SATZILLA (SAT), and ISAC (SAT). CPHYDRA [31] contains an algorithm portfolio of CSP solvers which partitions CPU-TIME between components of the portfolio in order to maximize the expected number of solved problem instances within a fixed time limit. SATZILLA [46], 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 [22] is a cluster-based approach that groups instances based on their features and then finds the best solver for each cluster.

Our 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.

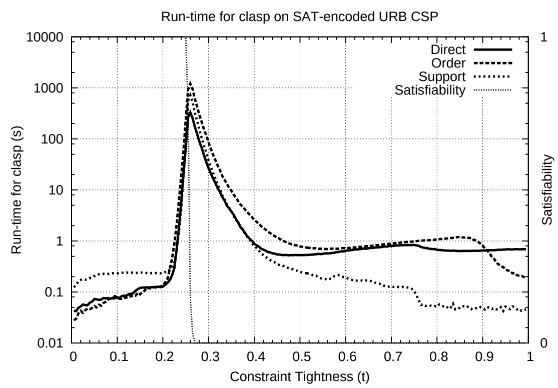
The remainder of this paper is organised as follows. Section 2 motivates the need to choose the representation and solver in combination. In Section 3 we summarise the necessary background on CSP and SAT to make the paper self-contained and present an overview of the main SAT encodings of CSPs. The detailed evaluation of our portfolio is presented in Section 4. We create a portfolio-based approach to CSP solving that employs eight CSP solvers, three SAT encodings and eighteen 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 the problem to be solved. For this experiment we considered uniform random binary CSPs with a fixed number of variables, domain size and number of constraints,



(a) Performance using `Minisat`.



(b) Performance using `Clasp`.

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

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 difference. Before the phase transition we see that the order encoding achieves the best performance on these instances and maintains this even at 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 2207 benchmarks from the CSP solver competitions. Figure 2 illustrates the respective performance of CSP-based and SAT-based methods on these instances. Unsurprisingly the dedicated CSP methods often achieve the best performance. There are, however, 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 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

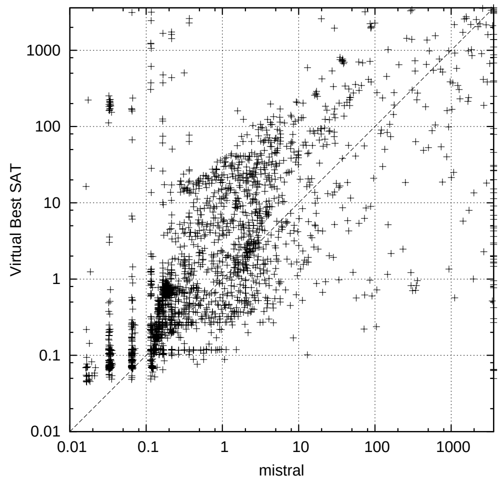
3.1 The Constraint Satisfaction Problem

Constraint satisfaction problems (CSP) are a natural means expressing and reasoning about combinatorial problems that are present in everyday life. They have a large number of practical applications such as scheduling, planning, vehicle routing, configuration, network design, routing and wavelength assignment [34]. 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.

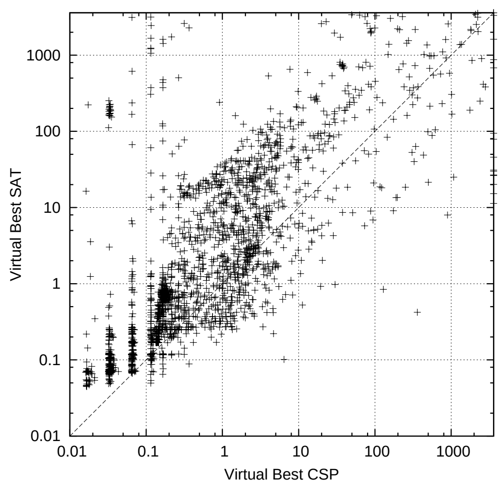
Finding a solution to a CSP is typically done using systematic search based on backtracking. Because the general problem is NP-complete, systematic search algorithms have exponential worst-case run times, which has the effect of limiting the scalability of these methods. However, the development of effective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

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



(a) Mistral and the virtual best SAT-based approach, showing that Mistral can be outperformed by an approach that selects the best SAT encoding/solver combination.



(b) Virtual best CSP portfolio and the virtual best SAT-based portfolio, which shows that even if one selects the best per-instance CSP solver, a SAT-based approach that selects the best encoding/solver combination can be superior on specific instances.

Fig. 2. Performance of CSP solvers against a SAT-based approach that selects the best encoding/solver combination.

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, routing, 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 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 $\mathrm{D}(X)$.

3.3 Direct Encoding

Translating a CSP variable X into SAT using the *direct encoding* [42], 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_1 is *true* in the resulting SAT formula, then $X = 1$ 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 distinct pair of 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 extensionally specified constraints, we simply encode the conflicts directly using the respective SAT variable for each assignment. 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 $\langle 1, 2, 3\rangle$. The constraints between the variables prevent each pair of variables from taking the same value: $X \neq Y$, $X \neq Z$, and $Y \neq Z$. 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* [14, 23] 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 clause

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

There is a clause for each value in the domain of Y that lists the values in X which are consistent with the respective assignment.

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 under 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 [14].

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.

Table 2. An example of the support encoding.

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

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 [**?**]) 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 $v+1$ ($x_{\leq v+1}$). 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 under 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 $X = v$ under 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:

$$\neg((x_{\le v} \wedge x_{\ge v}) \wedge (y_{\le w} \wedge y_{\ge w}))$$
$$\neg((x_{\le v} \wedge \neg x_{\le v-1}) \wedge (y_{\le w} \wedge \neg y_{\le w-1}))$$
$$\neg(x_{\le v} \wedge \neg x_{\le v-1}) \vee \neg(y_{\le w} \wedge \neg y_{\le w-1})$$
$$(\neg x_{\le v} \vee x_{\le v-1} \vee \neg y_{\le w} \vee y_{\le w-1})$$

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.

Table 3. An example of the order encoding.

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

3.6 Algorithm Portfolios

The Algorithm Selection Problem [33] is to select the most appropriate algorithm for solving a particular problem. It is especially relevant in the context of algorithm portfolios [16,20], 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 [46], ISAC [22] or CPHydra [31]. 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 series of decisions are usually referred to as hierarchical models. [44] and [17] 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 [15], who first predict 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

We implemented a tool to translate a CSP instance specified in XCSP format [35] into SAT (CNF). At present, it supports encoding inequality and binary extensional constraints using the direct, support and order encoding.

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 10 regression, 26 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 competition.[1] Of these, we consider the instances that contain either inequality or binary extensional constraints that our tool can translate to SAT. Altogether, we use 2,207 instances from the Graph Colouring, Random, Quasi-random, Black Hole, Quasi-group Completion, Quasi-group with Holes, Langford, Towers of Hanoi and Pigeon Hole problem classes.

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 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.

In addition to the CSP features, we computed the 54 SAT features used by SATZILLA [46] 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 8 complete CSP solvers:

- `Abscon` [27],
- `Choco` [40],
- `CSP4J` [26],
- `Gecode` [12],
- `Mistral` [19],
- `PCS` [41],
- `SAT4J` [25] and
- `Sugar` [38].

Satisfiability Solvers. We considered the following 18 complete SAT solvers:

[1] CSP solver competition instances
`http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`

- Minisat [10],
- cryptominisat [37],
- glucose [1],
- lingeling [4],
- clasp [11],
- picosat [3],
- precosat [3],
- kcnfs [9],
- mxc [6],
- riss [28],
- sat4j [25],
- glueminisat [30],
- qutersat [43],
- contrasat [13],
- rsat [32],
- march_rw [29],
- MPhaseSAT64 [8] and
- cirminisat [36].

In addition to these solvers, we include Minisat with variable elimination turned off. During our initial experimentation we found that some SAT-encoded formulas are very large but trivially solvable. Minisat was wasting time eliminating variables when it could very quickly solve the instance without having to do so. This gives a total of 19 SAT solvers.

Learning Algorithms. We evaluate a number of regression, classification, and clustering algorithms using WEKA [18]. All algorithms, unless otherwise stated use the default parameters. The regression algorithms we used were AdditiveRegression, LinearRegression, PaceRegression, REPTree, M5Rules, M5P, SMOreg, SVM ε and SVM ν. The classification algorithms were AdaBoost, 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, SimpleLogistic, SVM with radial basis function and SVM with sigmoid basis function. For clustering, we considered EM, FarthestFirst, and SimplekMeans. The FarthestFirst and SimplekMeans algorithms require the number of clusters to be given as input. We evaluated with multiples of 1 through 10 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.

4.2 Portfolio and Solver Results

The performance of each of the 19 SAT solvers was evaluated on the three SAT encodings of 2,207 CSP competition benchmarks with a time-out of 1 hour. Each of the 8 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. This data will be made available online.

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 SVM regression ν with CSP features at the root node to choose SAT or CSP, J48 with CSP features is used to select the CSP solver, M5P with CSP features is used to select the SAT encoding, IBk with 5 neighbours with CSP features selects the SAT solver for the direct encoded instance, SVM regression

Table 4. Performance of the learning algorithms for the hierarchical approach. We show the top 10 results due to space constraints. 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	54	2207
Proteus	252	2194
J48 with CSP features	335	2189
SVM nu log with CSP features	365	2187
Category Bests	433	2183
M5P with CSP features	565	2175
IBk with 5 neighbours with CSP features	616	2172
IBk with 5 neighbours with all features	753	2168
J48 with all features	770	2168
SVM nu log with all features	1058	2150
M5P with all features	1091	2148

ν with CSP features selects the SAT solver for the order encoded instance, and IBk with 5 neighbours with CSP features is used to choose the SAT solver for the support encoded instance. The hierarchical tree of specific machine learning approaches we found to deliver the best overall performance 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). The improvements we achieve over other approaches are similarly impressive. The results conclusively demonstrate that

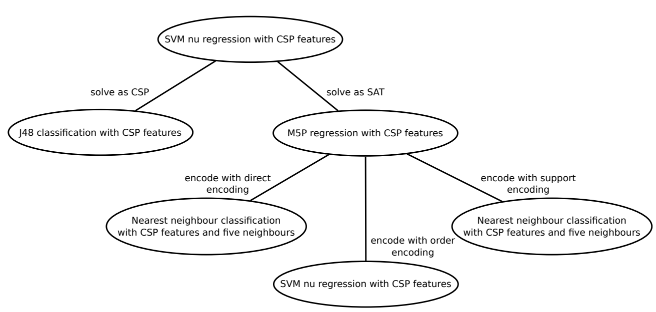


Fig. 3. Overview of the machine learning models used in the hierarchical approach.

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 19 SAT solvers is and the 8 CSP solvers for a total of 65 solvers. We show the top 10 results due to space constraints.

Classifier	Mean PAR10	Number Solved
VBS	22	2207
FarthestFirst1n with CSP features	380	2186
SVM rbf with CSP features	403	2185
IBk10 with CSP features	412	2184
lm with CSP features	424	2184
PaceRegression with CSP features	426	2184
REPTree with CSP features	435	2183
LinearRegression with CSP features	439	2183
MultilayerPerceptron with CSP features	443	2182
FarthestFirst1n with all features	453	2186

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 and the SAT features in addition. The additional overhead of computing SAT features is worthwhile in some cases though, for example for IBk with 5 neighbours, J48, SVM ν and M5P, as seen in the lower half of Table 4.

We also compare the hierarchical approach to that of a flattened setting with a single portfolio of all solvers and encoding solver combinations. That is, the flattened portfolio includes all possible combinations of the 3 encodings and the 19 SAT solvers and the 8 CSP solvers for a total of 65 solvers. Table 5 shows these results. The clustering algorithm FarthestFirst gives the best performance using this approach. However, it is significantly worse than the performance achieved by 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. It is interesting to note that the actual performance of Proteus is competitive with these oracle approaches, and outperforms the other approaches we present in this table.

Table 6. Virtual best performances ranked by PAR10 score.

Method	Mean PAR10	Number Solved
VB Proteus	54	2207
VB SAT	111	2207
VB CSP	224	2197
Proteus	252	2194
VB CPHydra	326	2191
VB Order Encoding	969	2156
VB Direct Encoding	1450	2125
VB Support Encoding	2333	2070

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 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 of those systems could theoretically achieve. Therefore, the strength of the Proteus approach is very convincing.

5 Conclusions

In this paper 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. By 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 the significant performance improvements Proteus achieves empirically on a large set of diverse benchmarks with 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.

References

1. Audemard, G., Simon, L.: Glucose: a solver that predicts learnt clauses quality. SAT Competition (2009)
2. Berre, D.L., Parrain, A.: The SAT4J library, release 2.2. JSAT 7(2-3), 59–6 (2010)
3. Biere, A.: P{re,i}cosatsc'09. SAT Competition (2009)
4. Biere, A.: Lingeling. SAT Race (2010)
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. Bregman, D.R.: The SAT Solver MXC, Version 0.99. SAT Competition (2009)
7. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the really hard problems are. In: 12th International Joint Conference on Artificial Intelligence. pp. 331–337. IJCAI'91, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1991)
8. Chen, J.: MPhaseSAT64. SAT Competition (2011)
9. Dequen, G., Dubois, O.: kcnfs. solver description. SAT Competition (2007)
10. Een, N., Sörensson, N.: Minisat. `http://minisat.se` (2010)
11. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: clasp: A conflict-driven answer set solver. In: In LPNMR '07. pp. 260–265. Springer (2007)
12. Gecode Team: Gecode: Generic Constraint Development Environment (2006), `http://www.gecode.org`
13. Gelder, A.V.: Contrasat - A Contrarian SAT Solver. JSAT 8(1/2), 117–122 (2012)
14. Gent, I.P.: Arc Consistency in SAT. In: ECAI. pp. 121–125 (2002)
15. 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)
16. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence 126(1-2), 43–62 (2001)
17. Haim, S., Walsh, T.: Restart strategy selection using machine learning techniques. In: SAT '09: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing. pp. 312–325. Springer-Verlag, Berlin, Heidelberg (2009)
18. 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)
19. Hebrard, E.: Mistral, a Constraint Satisfaction Library. In: Proceedings of the Third International CSP Solver Competition (2008)
20. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science 275(5296), 51–54 (1997)
21. Kadioglu, S., Malitsky, Y., Sabharwal, A., Samulowitz, H., Sellmann, M.: Algorithm Selection and Scheduling. In: Principles and Practice of Constraint Programming '11. pp. 454–469. Springer-Verlag, Berlin, Heidelberg (2011)
22. 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)
23. 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`

24. Kotthoff, L.: Algorithm Selection for Combinatorial Search Problems: A Survey. CoRR abs/1210.7959 (2012)
25. Le Berre, D., Parrain, A.: The SAT4J library, release 2.2 system description. Journal on Satisfiability, Boolean Modeling and Computation 7, 59–64 (2010)
26. Le Berre, D., Lynce, I.: CSP2SAT4J: A Simple CSP to SAT Translator. In: Proceedings of the 2nd International CSP Solver Competition (2008)
27. Lecoutre, C., Tabary, S.: Abscon 112, Toward more Robustness. In: Proceedings of the Third International CSP Solver Competition (2008)
28. Manthey, N.: Solver submission of riss 1.0 to the SAT Competition 2011. SAT Competition (2011)
29. Mijnders, S., de Wilde, B., Heule, M.J.H.: Symbiosis of Search and Heuristics for Random 3-SAT. In: Mitchell, D., Ternovska, E. (eds.) Proceedings of the Third International Workshop on Logic and Search (LaSh 2010) (2010), `http://www.st.ewi.tudelft.nl/~marijn/publications/ALDS_LaSh.pdf`
30. Nabeshima, H., Iwanuma, K., Inque, K.: Glueminisat 2.2.5. SAT Competition (2011)
31. 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)
32. Pipatsrisawat, K., Darwiche, A.: RSat 2.0: SAT Solver Description. Tech. Rep. D–153, Automated Reasoning Group, Computer Science Department, UCLA (2007)
33. Rice, J.R.: The algorithm selection problem. Advances in Computers 15, 65–118 (1976)
34. Rossi, F., Beek, P.v., Walsh, T.: Handbook of Constraint Programming. Foundations of Artificial Intelligence, Elsevier, New York, NY, USA (2006)
35. Roussel, O., Lecoutre, C.: XML Representation of Constraint Networks: Format XCSP 2.1. CoRR abs/0902.2362 (2009)
36. Sonobe, T.: cir minisat. SAT Competition (2011)
37. Soos, M.: Cryptominisat 2.9.0 (2011)
38. Tamura, N., Tanjo, T., Banbara, M.: System Description of a SAT-based CSP Solver Sugar. In: Proceedings of the 3rd International CSP Solver Competition. pp. 71–75 (2009)
39. Tanjo, T., Tamura, N., Banbara, M.: Azucar: A SAT-Based CSP Solver Using Compact Order Encoding — (Tool Presentation). In: SAT 2012. pp. 456–462. Springer (2012)
40. choco team: choco: an Open Source Java Constraint Programming Library. SAT Competition (2010)
41. Veksler, M., Strichman, O.: A Proof-Producing CSP Solver. In: AAAI (2010)
42. Walsh, T.: SAT v CSP. In: Principles and Practice of Constraint Programming — CP 2000. vol. 1894, pp. 441–456. Springer-Verlag (2000)
43. Wu, C.A., Lin, T.H., Lee, C.C., Huang, C.Y.R.: QuteSAT: a robust circuit-based SAT solver for complex circuit structure. In: DATE '07. pp. 1313–1318. EDA Consortium (2007)
44. Xu, L., Hoos, H.H., Leyton-Brown, K.: Hierarchical hardness models for SAT. In: CP. pp. 696–711 (2007)
45. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: Evaluating component solver contributions to portfolio-based algorithm selectors. In: SAT. pp. 228–241 (2012)
46. 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)