Constraints (2014) 19:380–403
DOI 10.1007/s10601-014-9165-7

meSAT: multiple encodings of CSP to SAT

Mirko Stojadinović · Filip Marić

Published online: 17 June 2014

Abstract One approach for solving Constraint Satisfaction Problems (CSP) (and related Constraint Optimization Problems (COP)) involving integer and Boolean variables is reduction to propositional satisfiability problem (SAT). A number of encodings (e.g., direct, log, support, order) for this purpose exist as well as specific encodings for some constraints that are often encountered (e.g., cardinality constraints, global constraints). However, there is no single encoding that performs well on all classes of problems and there is a need for a system that supports multiple encodings. We present a system that translates specifications of finite linear CSP problems into SAT instances using several well-known encodings, and their combinations. We also present a methodology for selecting a suitable encoding based on simple syntactic features of the input CSP instance. Thorough evaluation has been performed on large publicly available corpora and our encoding selection method improves upon the efficiency of existing encodings and state-of-the-art tools used in comparison.

Keywords Encoding CSP to SAT; Algorithm portfolio; CSP; SAT

1 Introduction

Constraint satisfaction problems (CSP) (and related *Constraint optimization problems (COP)*) 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). A special class of constraint problems often encountered in applications are *finite linear CSP* [34]. *Global constraints* describe relations between a non-fixed number

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

M. Stojadinović (✉) · F. Marić
Faculty of Mathematics, University of Belgrade, Belgrade, Serbia
e-mail: mirkos@matf.bg.ac.rs

F. Marić
e-mail: filip@matf.bg.ac.rs

of variables and their purpose is to improve readability and efficiency of CSP solving. In the rest of this paper only finite linear CSP with global constraints will be considered. 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) [30]. In this paper we will consider solving CSP problems only by reductions to *propositional satisfiability problem (SAT)* [6]. In this approach CSP instances are translated to SAT instances (in conjunctive normal form) and modern efficient SAT solvers are used for finding solutions that are then converted back to solutions of the original CSP problems. In order to apply SAT solvers, a CSP instance must be first encoded as a SAT instance. 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 [41], the support encoding [15], the log encoding [14], the order encoding [35], the compact-order encoding [36], the log-support encoding [13]). Also, special attention has been put on encoding Boolean cardinality constraints [6] and specific global constraints (e.g., global *all-different* constraint [4]) and many different encoding schemes have been developed. 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 [16]). Also, there are several more general tools that reduce CSP to SAT (and the related SMT problem [6]) using one of several standard encodings (e.g., SPEC2SAT [9], FZNTINI [17], `fzn2smt` [7] Sugar [34], Azucar [37], URSA [19], URBIVA[22], BEE [23]). In *lazy clause generation* approach [28], 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 [6] solver, where each propagator is considered as a separate theory, and theory propagation is used to learn clauses.

It is known that there is no single encoding scheme suitable for all problems. It should be possible to formulate hybrid encodings that use and combine good aspects of several classic SAT encodings and can outperform them. When solving a CSP instance, one should try several SAT encodings and carefully choose which one to apply on a specific problem. If a multiprocessor machine is available, one could try to generate different encodings and process them in parallel until the best one is found. However, a much better approach is to somehow determine the encoding that would give the best results. The previous observations can be summarized in the following hypotheses analyzed in this work.

(H1) Different SAT encodings are suitable for different problems.
(H2) It is possible to formulate combinations of several classic SAT encodings so that these combinations outperform original encodings on many problems.
(H3) It is possible to automatically select a suitable SAT encoding based only on simple syntactic features of the input CSP instance. If families of syntactically similar instances are formed, then the selection can be trained only on easy instances.

Currently for solving CSP by reduction to SAT many different tools and, even more demanding, their different input languages must be used. We offer a single platform that supports translating finite linear CSP problems using various encodings into SAT. We are not aware that such a platform exists, although, ideas for it have been already presented in the literature [13].

Contributions of this work are the following.

- We describe details of all encodings used within our system and give some correctness proofs that were missing in the literature (Section 3).
- We present some modifications of existing encodings that improve their performance: we discuss *direct-support* (Section 3.1) and *direct-order* (Section 3.4) encodings and we present some novel encodings of global constraints (Section 4).
- We present a machine-learning methodology for automated instance-based selection of suitable encoding for a given CSP instance (Section 5).
- We present a single system *meSAT* (*Multiple Encodings of CSP to SAT*) that supports multiple encodings of Finite Linear CSP [35] into SAT and implements our instance-based selection methodology (Section 6).
- We present a thorough evaluation on several large publicly available corpora. Experimental results indicate that hypotheses (H1)-(H3) hold and that our implementation of specific encodings is comparable to state-of-the-art tools and when automated selection is used it outperforms them (Section 7). We also show that on real CSP corpora, it is possible to train the automated selection only on very easy instances significantly reducing the training time and still get good results on the whole corpus.

2 Background

In this section we will give some background notions used by our system and 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, $\{\neg p \mid p \in B\}$ and $\{e \leq c \mid e$ 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 usage of *extensional constraints* (sometimes called *user-defined relations*) that are defined by a table of allowed/disallowed assignments to the variables 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 [34] that we also use.

```
(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. Firstly, 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). Secondly, 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 Alldifferent constraint The *all-different* constraint [40] 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 number of distinct values equal to the value of expression e. For example, 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 different values.

The count constraint The count constraint [4] requires that the number of occurrences of a 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 for encoding CSP to SAT

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

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

that allows the compilation of NPSPEC specifications (when given together with input data) into SAT instances.

MiniZinc [24] is a constraint modeling language which is compiled by a variety of solvers to the low-level target language FlatZinc for which there exist many solvers. FZN-TINI [17] introduces a translation for FlatZinc constraint models, such that any satisfaction or optimization problem written in the language (not involving floating point numbers) can be automatically Booleanized and solved by one or more calls to a SAT solver. The related `fzn2smt` [7] tool is a compiler from FlatZinc to the SMT-LIB language[2]. Solvers `mzn-g12cpx` and `mzn-g12lazy` implement lazy clause generation and are included in MiniZinc distribution.

URSA family of tools (URSA, URBIVA, URSA MAJOR) [19, 22] introduce uniform reductions of C-like language specifications to SAT or 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.

Sugar is a constraint solver that solves finite linear CSPs by translating them into SAT by using order encoding method [35] and then solving SAT instances by several supported SAT solvers.

Azucar [37] is a successor of Sugar that uses the compact-order encoding [36] for translating finite linear CSP into SAT and is tuned for solving specific large domain sized CSP instances.

BEE [23] (Ben-Gurion University Equi-propagation Encoder) is a constraint specification language and a compiler to CNF based on the order-encoding [35], similar to Sugar, but applying several optimizations.

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 solvers 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 [42], ISAC [21] and ArgoSmArT [26, 27]). 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 its input features and the prediction model.

Algorithm portfolios have recently been applied to constraint satisfaction. CPHYDRA [29] 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. [20] is to use run-time classifiers (to categories “short”, “medium”, “long”) to minimize the average completion time of each instance. This portfolio uses features of CPHYDRA and SATZilla and the combination of two. Very recent work by Amadini et al. [1] compares efficiency of different portfolio approaches based on SAT portfolio techniques and machine learning algorithms.

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

3 Encoding finite linear CSP in CNF into SAT

Encodings of CSP into SAT have been extensively studied and described in the literature (e.g., [13–15, 35, 36, 41]). Finite Linear CSP can be encoded into SAT using several different popular encodings, determined by how integer variables are represented using auxiliary Boolean variables. In this Section we give descriptions of all encodings used within our system *meSAT*. We also describe the log encoding (it is not implemented within *meSAT*, but we describe it because of its significance). Each encoding must handle each type of constraint (and each arithmetical operation) present in the problem specification, but, for simplicity, we will only informally describe encodings and give examples of encoding some basic types of atomic constraints (e.g., those covered by the Finite Linear CSP definition, as well as those that involve equality or disequality of arithmetic expressions). We try to make the exposure uniform and to describe all details specific for our implementation. We also discuss hybrid *direct-order* and *direct-support* encodings. These encodings aim at reducing the generated instance size, as well as improving the propagation power. For the variation of order encoding that we use, we provide its correctness proof.

Boolean cardinality constraints When encoding CSP into SAT, the following class of constraints on Boolean variables $b_1, \ldots, b_n$ is often encountered:

$$b_1 + \ldots + b_n \;\; \# k, \quad k \in \mathbb{N}, \quad \# \in \{\leq, <, \geq, >, =\}.$$

These are called *Boolean cardinality constraints* and in recent years, several ways to encode them efficiently into SAT were proposed (sequential and parallel counters [32], cardinality networks [2], encoding of at-most-one constraint [10], pairwise cardinality networks [11], perfect hashing based encodings [5]). As described in Section 3.1, many other types of constraints can be easily reduced to Boolean cardinality constraints, and then are reduced to SAT using these efficient encodings.

3.1 Direct and support encodings

Direct encoding For each integer variable x_i and every value $v \in D(x_i)$ (i.e., between l_i and u_i), a Boolean variable $p_{i,v}$ is created, denoting that $x_i = v$. For example, if x_1 is between 3 and 5, the vector of variables $[p_{1,3}, p_{1,4}, p_{1,5}]$ is formed and its valuation [0, 1, 0] denotes that $x_1 = 4$. Exactly one of the $p_{i,v}$ variables needs to be true, and this mutual exclusion is achieved by imposing Boolean cardinality constraint $p_{i,l_i} + \ldots + p_{i,u_i} = 1$. A naive approach to express this is by using one at-least-one clause $p_{i,l_i} \vee \ldots \vee p_{i,u_i}$, and a quadratic number of at-most-one clauses of the form $\neg p_{i,v'} \vee \neg p_{i,v''}$, where $v', v'' \in D(x_i)$ and $v' \neq v''$. These are the *conflict* clauses. Note that if two Boolean variables p and q cannot be true at the same time, $\neg p \vee \neg q$ is a conflict clause that imposes this constraint.

For all other types of constraints in CSP, the direct encoding also prefers to use conflict clauses.

If the constraint $x_i \neq x_j$ is to be encoded, then for every value $v \in D(x_i) \cap D(x_j)$ the clause $\neg p_{i,v} \vee \neg p_{j,v}$ should hold.

If the constraint $x_i = x_j$ is to be encoded, then for each $v \in D(x_i), w \in D(x_j)$ and $v \neq w$, clause $\neg p_{i,v} \vee \neg p_{j,w}$ is imposed. A slightly better performance is achieved if these binary conflict clauses are imposed only for values $v, w \in D(x_i) \cap D(x_j)$, and if unit clauses $\neg p_{i,v}$ and $\neg p_{j,w}$ are imposed for all values $v \in D(x_i) \setminus D(x_j)$ and $w \in D(x_j) \setminus D(x_i)$.

If the constraint $x_i < x_j$ is to be encoded, then for each $v \in D(x_i), w \in D(x_j)$ and $w \leq v$, the clause $\neg p_{i,v} \vee \neg p_{j,w}$ is imposed. Again, a slightly better performance is achieved if conflict clauses are imposed only for $v, w \in D(x_i) \cap D(x_j)$ (without endpoints) and if the following unit clauses are imposed: $\neg p_{i,v}$ where $v \in D(x_i)$ such that $w \le v$ for all $w \in D(x_j)$ and $\neg p_{j,w}$ where $w \in D(x_j)$ such that $w \le v$ for all $v \in D(x_i)$.

Support encoding The Boolean variables $p_{i,v}$ are the same as in the direct encoding and the restriction that exactly one $p_{i,v}$ holds still remains. However, the constraints are encoded differently. For each constraint, instead of conflict clauses, the support encoding uses *support* clauses. For example, $p_{i,v_1} \rightarrow p_{j,v_2} \vee p_{j,v_3} \vee p_{j,v_4}$ is a support clause[3] stating that if x_i is v_1 then x_j must be v_2, v_3 or v_4.

If the constraint $x_i \neq x_j$ is to be encoded, where $x_i \in D(x_i) = [l_i, u_i]$, and $x_j \in D(x_j) = [l_j, u_j]$, then for each value $v \in D(x_i) \cap D(x_j)$, a clause $p_{i,v} \rightarrow p_{j,l_j} \vee \ldots \vee p_{j,v-1} \vee p_{j,v+1} \vee \ldots \vee p_{j,u_j}$ that states that $x_i = v$ supports all values of x_j different from v is imposed. Symmetrically, clauses stating that $x_j = w$ supports all values of x_i different from w are also imposed. No clauses are used for values not in $D(x_i) \cap D(x_j)$ as they support all values from the other variable domain.

If the constraint $x_i = x_j$ is to be encoded, then for every value $v \in D(x_i) \cap D(x_j)$ clauses obtained from $p_{i,v} \leftrightarrow p_{j,v}$ are imposed. For Boolean variables $p_{i,v}$ and $p_{j,w}$ corresponding to the values $v, w \notin D(x_i) \cap D(x_j)$, unit clauses $\neg p_{i,v}$ and $\neg p_{j,w}$ are imposed.

If the constraint $x_i < x_j$ is to be encoded, then for every value v a clause $p_{i,v} \rightarrow p_{j,w_1} \vee \ldots \vee p_{j,w_m}$ is imposed, where $W = \{w_1, \ldots, w_m\}$ is the set of all values in $D(x_j)$ strictly greater than v. If $W = D(x_j)$, the clause does not need to be imposed, and if $W = \emptyset$ it becomes unit $\neg p_{i,v}$. Symmetrically, clauses stating that $x_j = w$ supports all values of x_i strictly less than w are imposed.

Direct-support encoding The direct and support encodings have one property in common: Boolean representation of integer variables is the same. But each of these encodings translates some of the constraints to clauses containing less literals than the other. Based on the estimated size of generated encoding we propose combination of these two encodings that encodes integer variable in the same way as in the direct and support encodings, but for encoding the constraints, in some cases it uses the direct encoding, in some cases the support encoding, and in some cases constraints are partly encoded by the direct and partly by the support encoding. The "nature" of the constraint determines which one is used as part of the direct-support encoding: the direct encoding is better when encoding conflicts and the support encoding is better when encoding supports.

If the constraint $x_i \neq x_j$ is to be encoded, then the number of imposed clauses in the support encoding is approximately twice as large as the one in the direct encoding. Further, in the direct encoding, clauses always consist of two literals, while in the support encoding clauses consist of two or more literals. For these reasons, the direct-support encoding uses the direct encoding for this constraint.

If the constraint $x_i = x_j$ is to be encoded, then the direct encoding imposes $O(d_{x_i} \cdot d_{x_j})$ clauses and the support encoding imposes $O(\max(d_{x_i}, d_{x_j}))$ clauses. All generated clauses have two literals. The direct-support encoding uses the support encoding of this constraint, as it is smaller in size.

If the constraint $x_i < x_j$ is to be encoded, then for each value $v \in D(x_i)$ direct or support encoding is chosen so as to minimize the number of literals. If v is in conflict with at most one third of the values from $D(x_j)$, then clauses used in the direct encoding are imposed.

[3]Each implication $A \rightarrow B$ can be translated to clause $\neg A \vee B$.

Otherwise, v supports at most two thirds of the values from $D(x_j)$ and clauses used in the support encoding are imposed.

Encoding linear arithmetic For encoding linear arithmetic constraints using the direct, support and direct-support encodings, we used the following scheme (it is somewhat closer to the support encoding, but not all clauses are negative Horn clauses, i.e. clauses containing at most one negative literal [15]).

Assume that x_k is a fresh variable and its domain is restricted to the values that can be taken by the expression $x_k = ax_i$ or $x_k = x_i + x_j$.

If the constraint $x_k = ax_i$ is to be encoded, for an integer constant a, the clauses $p_{i,v} \rightarrow p_{k,av}$ are imposed. Also, the clauses $p_{k,u} \rightarrow p_{i,u/a}$ could be imposed, where u is in the domain of x_k and that is when u/a is an integer.

If the constraint $x_k = x_i + x_j$ is to be encoded, the clauses $p_{i,v} \wedge p_{j,w} \rightarrow p_{k,v+w}$, for each $v \in D(x_i), w \in D(x_j)$ are imposed. Also, clauses $p_{k,u} \wedge p_{i,v} \rightarrow p_{j,u-v}$ and $p_{k,u} \wedge p_{j,w} \rightarrow p_{i,u-w}$ could be imposed.

3.2 Order encoding

The order encoding used by Sugar is shown to be the most efficient for many types of problems, including Open-Shop Scheduling problem [35], two-dimensional strip packing problems [33], and test case generation [3]. This is since it is compact and has good propagation properties [36]. The original encoding has been described in [35]. Here we describe its slightly different variant (described only informally on the website of Sugar) and give its correctness proof.

Integer variable x_i with the domain between l_i and u_i is represented with Boolean variables $p_{i,l_i}, \ldots p_{i,u_i-1}$, where $p_{i,v}$ represents that $x_i \leq v$. For simplicity, variables p_{i,l_i-1} that is always false and p_{i,u_i} that is always true are also available. Connection between the variables is described by the following clauses, imposed for every $v \in \{l_i + 1, \ldots, u_i - 1\}$: $p_{i,v-1} \rightarrow p_{i,v}$.

We describe one general way to reduce constraints of the form $\sum_{k=1}^{n} a_k x_k \leq c$, where a_k and c are integers, and all $a_k \neq 0$, to a Boolean combination of constraints of the form $x_k \leq b_k$, for some integers b_k.

```
procedure order_encode
input: constraint $\sum_{k=1}^{n} a_k x_k \leq c$, such that all $a_k \neq 0$
output: set of clauses over atoms of the form $x_k \leq b_k$, for some integers $b_k$.
if $n = 1$ then
    if $a_1 > 0$ then return $\{x_1 \leq \lfloor c/a_1 \rfloor\}$
    if $a_1 < 0$ then return $\{\neg(x_1 \leq \lceil c/a_1 \rceil - 1)\}$
else
    select some $x_i$
    if $a_i > 0$ then
        return $\bigcup_{v \in D(x_i)} \left(x_i \leq v - 1 \circledvee \text{order\_encode} \left(\sum_{k=1,k\neq i}^{n} a_k x_k \leq c - a_i v\right)\right)$
    if $a_i < 0$ then
        return $\bigcup_{v \in D(x_i)} \left(\neg(x_i \leq v) \circledvee \text{order\_encode} \left(\sum_{k=1,k\neq i}^{n} a_k x_k \leq c - a_i v\right)\right)$
```

The operation $l \circledvee S$ denotes adding a literal l to all clauses in the set S, i.e., $l \circledvee S = \{l \vee c.\, c \in S\}$.

The choice of the variable x_i in the procedure is arbitrary, but for efficiency it is good to choose the x_i with the smallest domain (and with the smallest absolute value of its coefficient in case of equal-sized domains) as such choice reduces the size of generated SAT instance.

Example 3 Let us describe how to encode the constraint $3x_1 + 5x_2 \leq 14$, if $D(x_1) = [0, 5]$ and $D(x_2) = [0, 3]$. We choose the variable x_2 since it has a smaller domain. Since $x_2 \geq 0$, then $3x_1 \leq 14 - 5 \cdot 0$, so the clause $x_1 \leq 4$ holds. If $x_2 \geq 1$, then $3x_1 \leq 14 - 5 \cdot 1$, so $x_1 \leq 3$. Therefore, either $\neg(x_2 \geq 1)$ (equivalent to $x_2 \leq 0$, since x_2 is integer) or $x_1 \leq 3$ holds, so the clause $x_2 \leq 0 \vee x_1 \leq 3$ holds. Similarly, if $x_2 \geq 2$, then $3x_1 \leq 14 - 5 \cdot 2$, so $x_1 \leq 1$. Therefore, the clause $x_2 \leq 1 \vee x_1 \leq 1$ holds. Finally, if $x_2 \geq 3$, then $3x_1 \leq 14 - 5 \cdot 3$, so $x_1 \leq -1$, but this is always false so the clause $x_2 \leq 2$ holds.

The next theorem establishes the correctness of the order encoding. The final transition to the Boolean level (introducing Boolean variables $p_{i,v}$) is trivial and left to the reader.

Theorem 1 *Let x_i be integer variables with domains $D(x_i)$, let C be the constraint $\sum_{k=1}^{n} a_k x_k \leq c$, for integer $a_k \neq 0$ and integer c, and let $S =$ order_encode (C) be the set of first-order clauses obtained by applying the order encoding procedure on C. Then, the tuple $(v_1, \ldots, v_n)$ satisfying all the domains of x_i, satisfies C iff it satisfies all the clauses in S.*

Proof The proof is by induction on n.

If $n = 1$, then the constraint C is of the form $a_1 x_1 \leq c$.

If $a_1 > 0$, then $S = \{x_1 \leq \lfloor c/a_1 \rfloor\}$. It holds that $\lfloor c/a_1 \rfloor \leq c/a_1$. If v_1 satisfies C, then $a_1 v_1 \leq c$, i.e., $v_1 \leq c/a_1$, but since v_1 is integer $v_1 \leq \lfloor c/a_1 \rfloor$, so v_1 satisfies S. If v_1 satisfies S, then $v_1 \leq \lfloor c/a_1 \rfloor \leq c/a_1$, so $a_1 v_1 \leq c$, so v_1 satisfies C.

If $a_1 < 0$, then $S = \{\neg(x_1 \leq \lceil c/a_1 \rceil - 1)\} \equiv \{x_1 > \lceil c/a_1 \rceil - 1\}$. If v_1 satisfies C, then $a_1 v_1 \leq c$, so $v_1 \geq c/a_1$. Since it holds that $\lceil c/a_1 \rceil - 1 < c/a_1 \leq \lceil c/a_1 \rceil$, it holds that $v_1 > \lceil c/a_1 \rceil - 1$, so v_1 satisfies S. If v_1 satisfies S, then $v_1 > \lceil c/a_1 \rceil - 1$, and since v_1 is integer then $v_1 \geq \lceil c/a_1 \rceil \geq c/a_1$, and v_1 satisfies C.

Assume the inductive hypothesis that for all $n' < n$, the encoding of all constraints of the form $\sum_{k=1}^{n'} a'_k x_k \leq c'$ for integer $a'_k \neq 0$ and integer c' is valid.

If x_i is fixed, for each v in the domain of x_i, let S_v denote the set of clauses obtained from the constraint $C_v \equiv \sum_{k \neq i} a_k x_k \leq c - a_i v$ (i.e., $S_v =$ order_encode (C_v)).

In the first direction we assume that $\vec{v} = (v_1, \ldots, v_n)$ satisfies C, and all the domains of x_i and then show that $\vec{v}$ satisfies S. It holds that $\sum_{k=1}^{n} a_k v_k \leq c$. Let x_i be the chosen variable. The value v_i is in the domain of x_i.

Assume that $a_i > 0$. Then, the set S consists of clauses from S_v, for each v, extended by the literal $x_i \leq v - 1$. For each $v \geq v_i + 1$, $v_i \leq v - 1$ is true, so all the clauses in S obtained from S_v are satisfied. If $v \leq v_i$, then $c \geq \sum_{k \neq i} a_k x_k + a_i v_i \geq \sum_{k \neq i} a_k x_k + a_i v$, so the tuple obtained from $\vec{v}$ by removing v_i satisfies C_v. By the inductive hypothesis it satisfies all the clauses of S_v, and they remain satisfied when extended by the literal $x_i \leq v - 1$.

Assume that $a_i < 0$. Then, the set S consists of clauses from S_v, for each v, extended by the literal $\neg(x_i \leq v)$. For each $v < v_i$, $\neg(v_i \leq v)$ is true, so all the clauses in S obtained from S_v are satisfied. If $v \geq v_i$, then $c \geq \sum_{k \neq i} a_k x_k + a_i v_i \geq \sum_{k \neq i} a_k x_k + a_i v$ (since $a_i < 0$), so the tuple obtained from $\vec{v}$ by removing v_i satisfies C_v. By the inductive hypothesis it satisfies all the clauses of S_v, and they remain satisfied when extended by the literal $\neg(x_i \leq v)$.

In the opposite direction we assume that the tuple $\vec{v}$ satisfies S and all the domains of variables x_k and show that it satisfies C. Let x_i be a chosen variable. Then, v_i is in the domain of x_i.

Assume that $a_i > 0$. It suffices to consider only clauses obtained from S_v, for $v = v_i$. These are clauses of S_{v_i}, extended by the literal $x_i \leq v_i - 1$. But $v_i \leq v_i - 1$ is false, so all the clauses of S_{v_i} must be satisfied by the tuple obtained from $\vec{v}$ by removing v_i. By the inductive hypothesis, the constraint $C_v \equiv \sum_{k \neq i} a_k x_k \leq c - a_i v_i$ is satisfied by the same tuple, so $\vec{v}$ also satisfies C.

Assume that $a_i < 0$. Again, it suffices to consider only clauses obtained from S_v, for $v = v_i$. These are clauses of S_{v_i}, extended by the literal $\neg(x_i \leq v_i)$. But $\neg(v_i \leq v_i)$ is false, so all the clauses of S_{v_i} must be satisfied by the tuple obtained from $\vec{v}$ by removing v_i. By the inductive hypothesis, the constraint $C_v \equiv \sum_{k \neq i} a_k x_k \leq c - a_i v_i$ is satisfied by the same tuple, so $\vec{v}$ also satisfies C. □

The constraint $x_i \neq x_j$ is equivalent to $x_i - x_j \leq -1 \vee x_j - x_i \leq -1$. These two linear constraints are encoded separately and their disjunction is encoded by the Tseitin transformation [39].

The constraint $x_i = x_j$ is equivalent to $x_i - x_j \leq 0 \wedge x_j - x_i \leq 0$. These two linear constraints are encoded separately and trivially combined.

The constraint $x_i < x_j$ is equivalent to the linear constraint $x_i - x_j \leq -1$.

3.3 Log encoding

This encoding corresponds to representation of machine integers. Each integer variable is encoded with the same number n of Boolean variables (i.e., bits). The bit-width n is chosen so that the maximal value in the domain of each variable can be represented (e.g., 32 which corresponds to standard machine-word width). For each integer variable x_i, the Boolean variable $p_{i,k}$ is true iff the k-th bit of the binary representation of the value assigned to x_i is 1. For variables taking negative values twos complement representation should be used (note that these are rarely encountered in CSP problems so the description of the log encoding is usually limited to unsigned variables). If the upper bound of an unsigned variable is drastically smaller than 2^n, then most of the bits of greater significance are set to false in the beginning. If the upper bound for an unsigned variable is not a power of two, excess values must be excluded. For each such value v which is not in the domain of x_i, and is represented with binary digits $(v_{n-1}, \ldots, v_0)$ $(v = \sum_{k=0}^{n-1} 2^k v_k)$, the disjunction $\bigvee_{k=0}^{n-1} v_k \oplus p_{i,k}$ is imposed. Symbol $\oplus$ denotes exclusive disjunction and $v_k \oplus p_{i,k}$ evaluates to true if these two bits take different values, and to false if they take same values.

If the constraint $x_i \neq x_j$ is to be encoded then the disjunction $\bigvee_{k=0}^{n-1} p_{i,k} \oplus p_{j,k}$ is imposed.

If the constraint $x_i = x_j$ is to be encoded, then the clauses obtained from $p_{i,k} \leftrightarrow p_{j,k}$ are generated for every bit index k.

If the constraint $x_i < x_j$ is to be encoded, then the variables $d_0, \ldots, d_{n-1}$ are introduced, where the value of d_k is true if and only if constraint is satisfied concerning only the bits with indices $0, \ldots, k$ of variables x_i and x_j. The following clauses are encoded: $d_0 \leftrightarrow \neg p_{i,0} \wedge p_{j,0}$, $d_k \leftrightarrow (d_{k-1} \wedge \neg p_{i,k} \wedge \neg p_{j,k}) \vee (d_{k-1} \wedge p_{i,k} \wedge p_{j,k}) \vee (\neg p_{i,k} \wedge p_{j,k})$, where k is index of bit taking values from 1 to $n-2$, and $d_{n-1} \leftrightarrow (d_{n-2} \wedge \neg p_{i,n-1} \wedge \neg p_{j,n-1}) \vee (d_{n-2} \wedge p_{i,n-1} \wedge p_{j,n-1}) \vee (p_{i,k} \wedge \neg p_{j,k})$. Unit clause d_{n-1} is also generated.

Arithmetical operations are encoded depending on the representation of negative numbers (if the encoding uses twos complement representation, then the same encoding of arithmetic operations can be used as for the unsigned values).

3.4 Direct-order encoding

A drawback of the order encoding is that it is not suitable for encoding some global constraints. Recent studies [23, 28] have shown that for some problems the best results are obtained if the order encoding is combined with the direct encoding of the *all-different* constraint. In this approach, the variables involved in *all-different* constraints are both encoded using the direct and the order encoding, and transition clauses are also generated.

We use extension of this approach and describe *direct-order*[4] encoding. It is essentially the same as that used by Minizinc [24]. This encoding uses direct encoding of specialized global constraints described in Section 4 and for all the other types of constraints it uses the order encoding. Since global constraints are based on the direct encoding, for their integer variables another set of Boolean variables and Boolean constraints corresponding to the direct encoding is defined (they are encoded by the Boolean variables and constraints corresponding to the order encoding). Assume that for integer variable x_i the order encoding creates Boolean variables $p_{i,v}$ and the direct encoding creates Boolean variables $p'_{i,v}$. Translation clauses connecting these two families of variables are obtained from: $p'_{i,v} \leftrightarrow \neg p_{i,v-1} \wedge p_{i,v}$, for each $v \in D(x_i)$. Note that mutual exclusions on the direct encoding need not be imposed, as the order literals do this.

4 Encoding global constraints

In this section we will describe how to encode the global constraints. As suggested in the system BEE [23], the *all-different* constraint can be much better encoded using the direct (and support) than using the order encoding. We build upon this idea and describe specific encodings for several global constraints (based on the direct and support integer representations). For some global constraints (e.g., the *count* constraint) in some special cases efficient encoding can be defined but in other cases it is hard to define an efficient encoding. The former occurrences will be called *specialized* global constraint occurrences (e.g., special case of the *count* constraint), and the latter occurrences will be called *non-specialized* global constraints occurrences (e.g., non-special case of the *count* constraint). Occurrences of other global constraints will be called *general* global constraints (e.g., all occurrences of constraints like *lex_less*, *element*, etc.).

The count constraint The constraint $count\ (e_1, \ldots, e_n, v)\ op\ c$, described in Section 2.2 is usually encoded by reducing to simpler constraints:

$$(\mathit{if}\ e_1 = v\ \mathit{then}\ 1\ \mathit{else}\ 0) + \ldots + (\mathit{if}\ e_n = v\ \mathit{then}\ 1\ \mathit{else}\ 0)\ op\ c,$$

and this constraint is translated to constraints $s_1 + \ldots + s_n\ op\ c$ and $((e_i = v) \rightarrow (s_i = 1)) \wedge ((e_i \neq v) \rightarrow (s_i = 0)), i \in \{1, \ldots, n\}$, where $s_1, \ldots, s_n$ are fresh variables.

[4]This encoding could be called *support-order* of *direct-support-order* as well, as constraints encoded by the direct encoding are the same for the support encoding

Specialized occurrences of the count constraint When modeling CSP problems usually the special case of this constraint is used: both v and c are constants (and not expressions as it is permitted in general case) and $e_1, \ldots, e_n$ are variables. This special case of constraint can be translated to SAT by the direct/support encoding[5] in a special way. For simplicity, let us assume that the domain of all variables is $[l, u]$. These variables are represented with Boolean variables (each row represents one integer variable):

$$\begin{matrix} p_{1,l} & \cdots & p_{1,u} \\ \vdots & \ddots & \vdots \\ p_{n,l} & \cdots & p_{n,u} \end{matrix}$$

The special case of *count* constraint can be simply encoded to SAT by imposing the Boolean cardinality constraint $p_{1,v} + p_{2,v} + \ldots + p_{n,v}$ op c (if the domains of the variables are not the same, then for every integer variable x_i such that v is not in its domain, the Boolean variable $p_{i,v}$ is excluded from the sum).

The order encoding does not use Boolean cardinality constraints and therefore can not use this special translation (even if it used Boolean cardinality constraints translation would not be as straightforward as in the case of the direct/support encoding).

Other global constraints Other global constraints also have special cases that are usually used for modeling CSP. From 11 global constraints supported by the Sugar input language, both the general case of *all-different* constraint and special cases of 4 other constraints (*global_cardinality*, *global_cardinality_with_costs*, *nvalue*, *cumulative*) can be handled efficiently by the direct/support encoding. Many other global constraints from global constraints catalogue can also be handled in a special way by the direct/support encoding. Generally, the direct/support encoding can implement global constraints more efficiently whenever it can be encoded by imposing cardinality constraints on Boolean variables representing the same value taken by different integer variables.

Many of these global constraints can be reduced to the special case of the *count* constraint, so by improving its encoding, other constraints are improved as well. For example, note that the encoding of the *all-different* constraint can be reduced to encoding of the special case of the *count* constraint. Namely, for each value from the union of the domains of the *all-different* arguments, a *count* constraint can be introduced specifying that the number of occurrences of that value in the set of expressions that are arguments of *all-different* is less than or equal to 1.

5 Instance-based encoding selection

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

Our approach is a modification of the portfolio approach introduced by Nikolić et al. [27]. In the original formulation, first, a training set of instances is fixed, and for each instance from the training set different SAT solving methods (different SAT solvers, or a single SAT solver with different setups) are applied with a given time limit. For each such instance its PAR10 score (penalty) [18] is calculated — solving time if the instance is solved within the time limit, or the time limit multiplied by 10, otherwise. When all training data is gathered

[5]The same translation is used for both of these encodings for all global constraints

and a new instance is to be solved, its features are extracted and k-nearest neighbors (*k-NN*) from the training set are found (with respect to the extracted features of training instances and a fixed distance function). The PAR10 score of each SAT solving method is calculated as a sum of scores for each neighboring instance, and the encoding with the minimal score is selected. If the same (minimal) scores are obtained for several SAT solving methods then the one of them is selected, based on some fixed priority.

We modify the original approach, consider CSP instances, and instead of choosing a SAT solving method we choose a SAT encoding that should be applied. Unlike some other approaches (e.g., [20]) that use features of the generated SAT instances, we use only features extracted from the original CSP formulation. We considered 70 different features divided in several groups: features related to the number and the percentage of constraints of different types — intensional, extensional, global, as well as for each specific type of constraint (e.g., arithmetic, *all-different*), features related to the sizes of the domains of integer variables for all variables in the instance, 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 4 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).

6 System description

The system *meSAT*[6] is implemented in C++. The system has a flexible architecture (illustrated on Fig. 1). There are several different input languages for CSP problems that have recently became popular (e.g., MiniZinc [24], XCSP [31], Sugar [34]). Therefore, we decided not to invent another input format, and *meSAT* currently supports input format of the tool Sugar. This syntax was selected since it is rather low-level and all its constructs can be translated to SAT. Furthermore, there exists a tool that converts CSP specifications written in XCSP 2.1 format [31] used in Fourth International CSP solver competition[7] into the Sugar syntax. An input specification is parsed into an abstract syntax tree (AST). The abstract syntax tree is traversed by different SAT encoders (forming a class hierarchy). During the traversal either SAT encodings are generated or features of instance are collected. Ways of encoding each kind of constraint can be defined for all encodings simultaneously, for groups of similar encodings and for some specific encodings.

[6]The source code with examples and instances used in experiments (but without third-party solvers, due to specific licensing) is available online from: http://argo.matf.bg.ac.rs

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

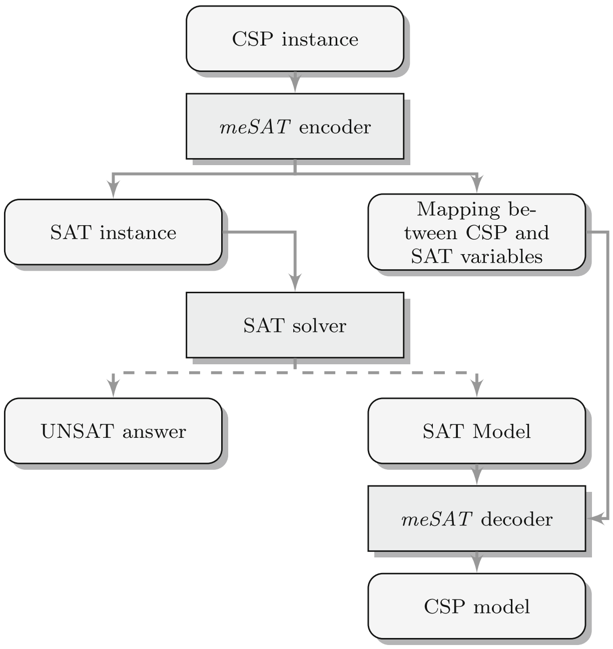

Fig. 1 *meSAT* overview

When reducing to SAT, either only the output DIMACS[8] file can be generated or one of the supported SAT solvers (minisat, clasp) can be automatically invoked and a satisfiable valuation, if exists, is translated back to the solution of the problem. If COP is solved, the solver is repeatedly invoked for different values of integer variable that needs to be maximized/minimized until an optimal value is found. The next value is selected by simple binary search algorithm. The instance is considered solved only if the optimum value is proved within the time limit.

The following encodings are supported: direct, support, direct-support, order, direct-order. Boolean cardinality constraints can be encoded to SAT either by using *sequential unary counters* [32] or *cardinality networks* [2] and for at-most-one constraints the encoding proposed by Chen [10] is used.

7 Experimental results

In order to show the advantages of supporting multiple encodings as well as automated encoding selection, we have conducted an experimental evaluation and tested our hypotheses given in Section 1. We have used all encodings available in *meSAT*, sequential unary counters for Boolean cardinality constraints [32], and Chen's encoding of the at-most-one constraints [10].

[8] ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.dvi

Instances We used three corpora of CSP problems: (i) CPAI09 containing all problems used in Fourth International CSP Solver Competition[9] that use global constraints, (ii) MiniZinc containing 29 problems from MiniZinc corpus[10] also encoded to use global constraints, and (iii) instances of the *Dominating Queens* problem, described in the global constraints catalogue[11]. The reason for including only instances that use global constraints is that in all preliminary experiments the order encoding proved to be the best encoding for instances that do not use global constraints: propagations are faster and generated instances are smaller. 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 [24] and Sugar input language [34].

Instances from the first corpus were automatically converted from the original input language to MiniZinc by the converter `xcsp2zinc` available on MiniZinc page[12] 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 contains only instances of the Dominating Queens problem. A lot of focus in *meSAT* has been put on efficient encoding of global constraints. For example, the *nvalue* constraint can be very efficiently encoded when the direct or support encodings are used. However, no problems in first two corpora include this constraint, so to test the efficiency of *meSAT* on this kind of constraint we included the Dominating Queens problem. Specification of the problem is the same for MiniZinc and Sugar input language.

Interesting instances In our preliminary experiments we noticed that many instances are easy for most of the solvers, and that there are instances too hard for all solvers. So in most of the experiments we focus on "interesting" instances, the ones that are solved within a given 600 seconds timeout by at least one, but not all three solving methods: direct-support, order and direct-order encoding implemented within *meSAT* system. Choosing the best encoding for them is very important, since these are the only instances that affect the total number of solved instances in the instance-based encoding selection. In order to increase their number, we generated instances of greater size for problems of the second corpus where most of the instances were too easy. To make the distribution of interesting instances among problems slightly more uniform than in original corpora, for problems that had more than 30 interesting instances we randomly selected 30 instances that were used in experiments.

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. In all experiments and tools, SAT solver Minisat 2.2 [12] was used for solving generated CNF instances. Timeout was 600 seconds for each instance (for total time including selecting the encoding where needed, encoding, and solving). In all tables, # denotes the total number of instances. In each cell the number of instances solved is given. In most tables separate

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

[10] http://www.minizinc.org

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

[12] http://www.minizinc.org

results for different corpora are not shown, but only aggregate results. The total solving time (in minutes) is shown in parentheses (for each unsolved instance 600 seconds are added to the total time). Mean time spent on a instance is directly computable from the total time and the number of instances. Since in some cases less than 50 % of instances are solved, median time is not shown. In each row, cell entry corresponding to the best solver is printed bold.

Table 1 shows experimental results for all instances.[13] Table 2 shows experimental results for interesting instances. These results are discussed in the following subsections.

7.1 Comparison of *meSAT* with other state-of-the-art tools

We compare the results of our *meSAT* system with the results of some other state-of-the-art tools. We compare it with system Sugar-v2-0-0, its successor Azucar-v0.2.3 [37] (used in two different configurations; Azucar-m2 that implements the compact-order encoding for $m = 2$ and is default configuration and Azucar-log that implements the log encoding), and two lazy clause generation solvers included in MiniZinc 1.6 distribution: `mzn-g12cpx` and `mzn-g12lazy`.

Results given in Tables 1 and 2 indicate that our system *meSAT* significantly outperforms other solvers used in comparison. The exception is Sugar (the winning solver on several CSP competitions), where *meSAT* was better only when the direct-order encoding is used. Since the difference is not so big, this might be attributed to random variations in SAT solving time[14]. The efficiency of *meSAT* can be significantly improved if instance-based encoding selection is used, as described in Section 7.3, and then it clearly outperforms even Sugar.

7.2 Comparison of different encodings within *meSAT*

Next, we analyze the efficiency of different encodings implemented within *meSAT*: direct (d), support (s), direct-support (ds), order (o), and the direct-order (do). The aim is to analyze the performance of our hybrid encodings (direct-support and direct-order) and compare them to their constituent encodings (direct, support, direct-support and order) on the interesting instances. Due to the lack of space we did not show results of direct and support encoding in Table 2, therefore we give these results (only total numbers) in Table 3.

Results indicate that the direct-support encoding slightly outperforms both the direct and the support encoding, although this could be attributed to random variations in SAT solving time. As differences are minor, the separate results for the direct and the support encoding will not be further considered. Results also indicate that the direct-order encoding outperforms both the order and the direct-support encoding. All this supports our hypothesis (H2) to some extent.

The cumulative results given in Table 3 might indicate that the order-based encodings (the direct-order encoding can be considered to be order-based since in many cases it exactly matches with the order encoding) significantly outperform the direct-support based encodings. However, results for separate problems (given in Table 2) show that there are problems that are much better suited for the direct-support encoding (e.g., M/cars, M/carseq, M/knights, D/dominatingQueens). Table 4 shows the number of interesting instances for

[13] More detailed results are available online from: http://argo.matf.bg.ac.rs

[14] SAT solving time always has some amount of unpredictability and can be affected by trivial changes of the input instances (e.g., changes to the order of clauses), so there is a small amount of randomness in all experimental results that include SAT solving. A statistical approach to comparing SAT solvers is given by Nikolić [25].

Table 1 Summary results of experimental evaluation on all instances of five solvers: *meSAT* (ds – direct-support encoding, o – order encoding, do – direct-order encoding, auto – automatically selected encoding based on the syntactic features of the instance: auto$_{easy}$ – train on easy instances, auto$_{cv}$ – cross validation, oracle – the best encoding for the instance), Sugar, Azucar, `mzn-g12cpx` and `mzn-g12lazy`

		meSAT						Sugar	Azucar		`mzn-g12`	
corpus	#	ds	o	do	auto$_{cv}$	auto$_{easy}$	oracle	Sugar	m2	log	cpx	lazy
CPAI09	583	359 (2402)	488 (1099)	486 (1104)	486 (1094)	487 (1103)	492 (1055)	**494 (1053)**	448 (1503)	428 (1722)	332 (2609)	348 (2638)
MiniZinc	770	583 (2100)	562 (2516)	579 (2266)	**609 (1852)**	599 (1940)	612 (1811)	554 (2572)	509 (2965)	486 (3309)	371 (4170)	492 (2974)
DQueens	26	**23 (76)**	2 (240)	20 (89)	**23 (76)**	**23 (76)**	25 (60)	3 (236)	2 (246)	2 (247)	4 (224)	5 (211)
Total	1379	965 (4578)	1052 (3855)	1085 (3459)	**1118 (3023)**	1109 (3120)	1129 (2926)	1051 (3862)	959 (4713)	916 (5278)	707(7002)	845 (5824)

Table 2 Experimental evaluation for the interesting instances divided in problem categories. For each problem initial letter denotes the corpus (C for CPAI09, M for MiniZinc, D for Dominating Queens). The same set of solvers as in Table 1 was used

		meSAT												Sugar	Azucar	`mzn-g12`							
problem	#	ds		o		do		auto$_{cv}$		auto$_{easy}$		oracle		Sugar		m2		log		`cpx`		`lazy`	
C/allsquares	16	0	(160)	16	(48)	15	(47)	15	(47)	16	(48)	16	(45)	**16**	**(7)**	11	(71)	9	(93)	0	(160)	0	(160)
C/bibd	16	0	(160)	**16**	**(23)**	**16**	**(23)**	**16**	**(23)**	**16**	**(23)**	16	(22)	15	(16)	15	(25)	13	(55)	3	(149)	1	(151)
C/CabinetStart1	30	0	(300)	30	(16)	30	(14)	30	(14)	30	(14)	30	(13)	30	(16)	30	(9)	**30**	**(3)**	30	(5)	30	(17)
C/compet08	2	**2**	**(0)**	0	(20)	**2**	**(0)**	**2**	**(0)**	**2**	**(0)**	2	(0)	1	(11)	0	(20)	0	(20)	0	(20)	0	(20)
C/costasArray	2	**2**	**(10)**	1	(11)	0	(20)	0	(20)	0	(20)	2	(8)	1	(16)	0	(20)	0	(20)	0	(20)	0	(20)
C/latinSquare	1	0	(10)	**1**	**(0)**	0	(10)	0	(10)	0	(10)	1	(0)	**1**	**(0)**	0	(10)	0	(10)	0	(10)	0	(10)
C/magicSquare	6	0	(60)	**6**	**(11)**	4	(25)	**6**	**(11)**	4	(25)	6	(9)	5	(25)	4	(23)	3	(38)	0	(60)	0	(60)
C/ortholatin	1	0	(10)	0	(10)	1	(8)	0	(10)	1	(8)	1	(8)	1	(4)	**1**	**(1)**	1	(8)	0	(10)	0	(10)
C/pseudoGLB	30	0	(300)	**30**	**(8)**	**30**	**(8)**	**30**	**(8)**	**30**	**(8)**	30	(8)	30	(15)	29	(30)	24	(93)	28	(35)	26	(125)
C/QG	1	0	(10)	1	(7)	1	(6)	0	(10)	1	(6)	1	(6)	**1**	**(2)**	**1**	**(2)**	0	(10)	0	(10)	0	(10)
M/bacp	5	0	(50)	5	(13)	5	(8)	5	(8)	5	(8)	5	(8)	5	(14)	4	(26)	3	(34)	**5**	**(1)**	**5**	**(1)**
M/bibd	2	2	(10)	0	(20)	2	(9)	2	(9)	2	(9)	2	(9)	**2**	**(1)**	1	(10)	1	(10)	1	(11)	1	(11)
M/cars	9	**9**	**(21)**	0	(90)	5	(74)	**9**	**(21)**	**9**	**(21)**	9	(21)	4	(80)	0	(90)	1	(84)	0	(90)	0	(90)
M/carseq	8	8	(25)	0	(80)	5	(69)	8	(25)	8	(25)	8	(25)	3	(71)	0	(80)	1	(79)	0	(80)	**8**	**(1)**
M/costas-array	4	4	(3)	0	(40)	4	(26)	4	(3)	4	(3)	4	(3)	0	(40)	**4**	**(1)**	0	(40)	0	(40)	0	(40)
M/debruijnbinary	2	0	(20)	2	(1)	**2**	**(0)**	2	(1)	2	(1)	2	(0)	2	(1)	2	(1)	**2**	**(0)**	**2**	**(0)**	**2**	**(0)**
M/golfers	2	1	(10)	2	(5)	0	(20)	1	(14)	1	(10)	2	(1)	2	(2)	2	(10)	1	(16)	1	(10)	**2**	**(1)**
M/golomb	1	0	(10)	1	(7)	**1**	**(4)**	**1**	**(4)**	**1**	**(4)**	1	(4)	1	(9)	1	(9)	0	(10)	**1**	**(4)**	**1**	**(4)**
M/knights	14	**14**	**(41)**	0	(140)	4	(126)	**14**	**(41)**	**14**	**(41)**	14	(41)	0	(140)	0	(140)	0	(140)	12	(20)	5	(91)
M/langford	2	0	(20)	2	(2)	**2**	**(0)**	**2**	**(0)**	**2**	**(0)**	2	(0)	**2**	**(0)**	2	(3)	2	(15)	2	(3)	2	(3)
M/latinsquares	10	0	(100)	**10**	**(5)**	0	(100)	**10**	**(5)**	0	(100)	10	(5)	9	(22)	0	(100)	0	(100)	4	(66)	5	(67)
M/magicseq	2	0	(20)	2	(3)	2	(3)	2	(3)	2	(3)	2	(3)	**2**	**(1)**	**2**	**(1)**	2	(2)	**2**	**(1)**	**2**	**(1)**
M/nsp	2	**2**	**(0)**	0	(20)	**2**	**(0)**	**2**	**(0)**	**2**	**(0)**	2	(0)	1	(15)	2	(4)	2	(7)	0	(20)	0	(20)

Table 2 (continued)

		meSAT												Sugar	Azucar	`mzn-g12`							
problem	#	ds		o		do		auto$_{cv}$		auto$_{easy}$		oracle		Sugar		m2		log		`cpx`		`lazy`	
M/QCP	1	**1**	**(1)**	0	(10)	**1**	**(1)**	**1**	**(1)**	**1**	**(1)**	1	(1)	1	(5)	0	(10)	0	(10)	0	(10)	0	(10)
M/queens	9	8	(39)	0	(90)	**9**	**(32)**	**9**	**(32)**	**9**	**(32)**	9	(32)	2	(74)	3	(70)	3	(72)	0	(90)	0	(90)
M/searchstress	2	0	(20)	2	(14)	0	(20)	0	(20)	0	(20)	2	(14)	2	(4)	0	(20)	0	(20)	**2**	**(0)**	**2**	**(0)**
M/steiner	2	**2**	**(0)**	1	(11)	0	(20)	**2**	**(0)**	2	(1)	2	(0)	1	(10)	1	(11)	1	(14)	**2**	**(0)**	1	(10)
M/stilllife	2	0	(20)	2	(9)	**2**	**(8)**	2	(9)	**2**	**(8)**	2	(8)	0	(20)	0	(20)	0	(20)	0	(20)	0	(20)
M/talentscheduling	1	0	(10)	**1**	**(1)**	1	(2)	1	(2)	**1**	**(1)**	1	(1)	1	(4)	1	(10)	0	(10)	1	(6)	0	(10)
D/dominatingQueens	23	**21**	**(66)**	0	(230)	18	(79)	**21**	**(66)**	**21**	**(66)**	23	(50)	1	(225)	0	(230)	0	(230)	2	(213)	3	(201)
TOTAL	208	76	(1507)	131	(945)	164	(763)	**197**	**(419)**	188	(518)	208	(348)	142	(849)	116	(1057)	99	(1263)	98	(1165)	96	(1255)

Table 3 Summary results of experimental evaluation of different encodings within *meSAT* system on interesting instances

corpus	#	d	s	ds	o	do
Total	208	61 (1608)	71 (1528)	76 (1507)	131 (945)	**164 (763)**

each encoding where it was the best and it shows that for each encoding there are many instances for which it is the best. This clearly supports our hypothesis (H1) that different encodings are suitable for different problems.

7.3 Evaluation of instance-based encoding selection

As different encodings are suitable for different problems, there is a good motivation to use some instance-based encoding selection scheme, and we applied the approach described in Section 5. We compare our instance based selection scheme to the (i) *oracle* (or virtual best) method that would select the best encoding for each instance (this method is not feasible in practice since it makes perfect decisions and for each instance it must guess the optimal encoding before trying to solve it, which is impossible to implement) and (ii) the *best fixed* method – one encoding that gives the best overall performance.

Currently, selection is done among three types of encodings: direct-support, order, and direct-order. As already noted, the direct-support encoding is slightly more efficient than both the direct and the support encoding so these two are not used. The log-based encodings (e.g., log, compact-order) are not currently supported by our system, but we plan to incorporate them in our future work.

We used 70 features, all extracted only from instance input files (e.g., average domain size, number of multiplications, sum of domains of variables involved in multiplications, average arity of specialized occurrences of global constraints, percentage of intensional constraints among all the constraints). 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 three included encodings in a given timeout, and then the optimal parameters (the number of neighbors k and the distance measure d) are selected in the following way. For each fixed k and d the total score is determined by the *leave-one-out* procedure (the total score is the sum of individual scores of all instances, where the individual score for each instance is its score when using the encoding selected by applying the *k-NN* selection procedure with parameters k and d and looking at the rest of the training set). All combinations of k (ranging from 1 to 20) and 4 different distance measures ([38]) 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 function d) are used to select the encoding for a given test instance. If the same (best) scores are obtained for more encodings, then the direct-order encoding has maximum priority of being selected and the direct-support encoding has the

Table 4 Numbers of interesting instances where each encoding achieved the best result

#	ds	o	do
208	59	64	**85**

lowest priority. When the encoding is selected, *meSAT* is invoked for that test instance using the selected encoding, given a timeout of 600 seconds.

We have considered two different approaches for choosing training and testing sets — training on random portions of the corpus and testing on the rest (so called cross-validation) and training only on the easy instances and testing on the rest.

Cross validation In the first approach we have used 5-fold cross-validation, dividing the corpus in 5 equal parts, testing on each part after training on other 4 parts, and reporting the total results for all 5 folds. This method ($auto_{cv}$) enables *meSAT* to solve 33 instances more than when using single fixed encoding. Although this does not seem much considering all 1379 instances, this is only due to a very high number of instances solved by all methods (889) and the number of hard instances not solved by any method (295). Considering only interesting instances (the only ones that can affect the total number of solved instances), gives more clear picture. Table 2 shows that the *best-fixed* method (the direct-order encoding) solves 164 instances, that $auto_{cv}$ solves 197 and that the theoretical optimum *oracle* solves 208 instances. Further, the total solving time is almost halved (from 763 to 419 minutes). We consider this to be a significant improvement. The biggest improvement was achieved on MiniZinc corpus, while on the CPAI09 corpus, improvement is small (as the $auto_{cv}$ solved the same number of instances as the *best-fixed* method, but in slightly shorter time).

Training on easy instances In the second approach, we wanted to test the possibility to reduce the overall training time (that is measured in CPU days when training on all kinds of instances) by training only on the easy instances and testing on the whole corpus.

As the easy instances are not known a priori, a preparation phase had to be conducted and it consisted of running *meSAT* with the time limit of 5 seconds on all instances using each of the 3 encodings. From 1379 instances, 543 were solved by all encodings within the time limit — these were considered to be easy instances and were used for training. Since the time limit is low, the maximal time for preparation is only 1379 × 3 × 5s i.e., only about 6 hours and it could be performed on a standard PC (in fact, it took only 3.5 hours, as many instances are solved under these 5s). Solving 543 easy instances is done in less than 13 minutes, but, as these are not known a priori, the whole time including preparation must be considered.

After the training phase the testing phase was conducted on the whole corpus. In the tables, this method is denoted by $auto_{easy}$. Note that, in order to have uniform layout for Tables 1 and 2, the reported totals are given for the whole corpus (so there is an overlap between the training and the test set). However, the training instances are so easy that they are all solved by each method and the best and the worst encoding on all of them together differ by less than 5 minutes and this time is negligible for the total results (where the encodings differ by hundreds of minutes). Moreover, when considering only interesting instances, there is no overlap between the training and the test set.

Although not as effective as when training on both easy and hard instances, training only on easy instances also managed to improve upon the best fixed encoding. This is due to the nature of considered corpora where instances are grouped into families (each coming from a single problem) and each family contains very similar instances differing only in their size, for which the optimal encoding does not depend on the size, but on the structure of the instance. Looking carefully at Table 2, it can be seen that the main difference between $auto_{easy}$ and $auto_{cv}$ can be attributed to a single problem (M/latin_squares where $auto_{cv}$

solved 10 instances more). The reason for this is that the best encoding when solving with smaller timeout (180s) is the direct-order encoding, while the best encoding when solving with bigger timeout (600s) is the order encoding. This was the only problem with such behavior in our corpus, so we suppose that these kinds of problems (where training only on the easy instances fails) are rare. We have tried to increase the timeout in the preparation phase (5s) and to include a bit harder instances, but this prolonged the preparation and the training phase (what we wanted to avoid in the first place), and did not improve the results significantly.

All these results supports our hypothesis (H3).

8 Conclusions and further work

In this paper we have presented the system *meSAT* that supports encoding finite linear CSP using various SAT encodings. We have evaluated the direct-support and the direct-order encoding that combine existing encodings and slightly outperform them. We have described all these encodings in a uniform manner discussing their correctness where necessary. Experimental evaluation given in this paper supports the claim that there is no single encoding suitable for all types of problems and that one could benefit from trying different encodings and solvers.

We have developed a simple instance-based selection scheme that chooses one among different encodings and shows improvement over the fixed encodings used. The scheme is based on machine-learning and follows the ArgoSmArT-knn approach [27]. The main difference with the other portfolio approaches is that instead of selecting the tool that should be applied on the already encoded SAT instance, we select the encoding that would give the best results. We have demonstrated that it is possible to train only on the very easy instances (thus to have a very fast training phase), to achieve some generalization and get better results on the whole corpus including larger and harder instances than by using any fixed encoding. Including harder instances in the training set further improves the results, as there could be some problems where the optimal encodings for easy and hard instances differ (still, there were not many problems of this kind in our corpus).

Many instances are relatively easy for our system, so the margin between the best-fixed and the oracle solver is rather tight (only 44 instances). However, if only interesting instances are considered the same margin does not look that tight. Our selection scheme manages to make a good selection for most of these instances and the overall total solving time decreases.

Our future plans are to extend our system by incorporating other state-of-the-art SAT encodings (e.g., [13, 14, 36]) and by enabling translations to other satisfiability problems (e.g, pseudo-boolean constraints, SMT). As new encodings of Boolean cardinality constraints were invented in recent years (e.g. [5, 11]), we plan to implement and evaluate them. We also plan to investigate how the set of encodings can be expanded to handle nonlinear arithmetic (e.g., multiplication, division). Currently, optimization problems are solved using a naive binary search and we plan to implement more advanced methods (e.g. branch-and-bound or unsatisfiable-core based). All these possible parameters should be included in instance-based selection.

Out of 1379 instances, 62 of them were solved by other solvers but not solved by *meSAT* (Sugar solved 12 instances that *meSAT* did not, Azucar-m2 13, Azucar-log 9, `mzn-g12cpx` 25, and `mzn-g12lazy` 43). This shows how much improvement could theoretically be possible if these tools were included in the selection. Therefore, we also plan to apply our instance based selection methodology outside the system *meSAT* and to select between different tools and implementations of SAT encodings.

Acknowledgements The authors wish to thank Mladen Nikolić, Predrag Janičić, and anonymous reviewers for valuable comments on the first versions of this manuscript. We also wish to thank Naoyuki Tamura for some clarifications about the order encoding and the Sugar system, Peter Stuckey for providing information about MiniZinc, and Emmanuel Hebrard for providing some information about global constraints.

References

1. Amadini, R., Gabbrielli, M., Mauro, J. (2013). An empirical evaluation of portfolios approaches for solving CSPs. In *CoRR 2013, LNCS 7874* (pp. 316–324). Springer.
2. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E. (2009). Cardinality networks and their applications. In *SAT 2009, LNCS 5584* (pp. 167–180). Springer.
3. Banbara, M., Matsunaka, H., Tamura, N., Inoue, K. (2010). Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. In *LPAR (Yogyakarta), LNCS 6397* (pp. 112–126). Springer.
4. Beldiceanu, N., Carlsson, M., Rampon, J.-X. (2005). Global constraint catalog Technical report. *SICS*.
5. Ben-Haim, Y., Ivrii, A., Margalit, O., Matsliah, A. (2012). Perfect hashing and CNF encodings of cardinality constraints. In *SAT 2012, LNCS 7317* (pp. 397–409). Springer.
6. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T., (eds.) (2009). *Handbook of satisfiability, volume 185 of frontiers in artificial intelligence and applications*. IOS Press.
7. Bofill, M., Suy, J., Villaret, M. (2010). A system for solving constraint satisfaction problems with smt. In *SAT 2010, LNCS 6175* (pp. 300–305). Springer.
8. Cadoli, M., Palopoli, L., Schaerf, A., Vasile, D. (1999). NP-SPEC: An executable specification language for solving all problems in NP. In *PADL 1999, LNCS 1551* (pp. 16–30). Springer.
9. Cadoli, M., & Andrea, S. (2005). SPEC2SAT: Compiling problem specifications into SAT. *Artificial Intelligence*, *162*(1–2), 89–120.
10. Chen, J. (2010). A new SAT encoding of the at-most-one constraint. In *Proceedings of the 9th international workshop on constraint modelling and reformulation*.
11. Codish, M., & Zazon-Ivry, M. (2010). Pairwise cardinality networks. In *LPAR (Dakar), LNCS 6355* (pp. 154–172). Springer.
12. Eén, N., & Sörensson, N. (2003). An extensible SAT-solver. In *SAT 2003, LNCS 2919* (pp. 502–518). Springer.
13. Gavanelli, M. The log-support encoding of CSP into SAT. In *CP 2007, LNCS 4741* (pp. 815–822).
14. Van Gelder, A. (2008). Another look at graph coloring via propositional satisfiability. *Discrete Applied Mathematics*, *156*(2), 230–243.
15. Gent, I.P. (2002). Arc consistency in SAT. In *ECAI 2002* (pp. 121–125). IOS Press.
16. Horbach, A. (2010). A boolean satisfiability approach to the resource-constrained project scheduling problem. *Annals OR*, *181*(1), 89–107.
17. Huang, J. (2008). Universal booleanization of constraint models. In *CP 2008, LNCS 5202*, (pp. 144–158). Springer.
18. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T. (2009). ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research (JAIR)*, *36*, 267–306.
19. Janičić, P. (2012). Uniform reduction to SAT. *Logical Methods in Computer Science*, *8*(3), 1–39.
20. Kiziltan, Z., Mandrioli, L., Barry, OS., Mauro, J. (2011). 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.
21. Malitsky, Y., & Sellmann, M. (2012). Instance-specific algorithm configuration as a method for non-model-based portfolio generation. In *CPAIOR 2012, LNCS 7298* (pp. 244–259). Springer.
22. Marić, F., & Janičić, P. (2010). URBIVA: Uniform reduction to bit-vector arithmetic. In *IJCAR 2010, LNCS 6173* (pp. 346–352). Springer.
23. Metodi, A., & Codish, M. (2012). Compiling finite domain constraints to SAT with BEE. *TPLP*, *12*(4–5), 465–483.
24. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G. Minizinc: Towards a standard CP modelling language. In *CP 2007, LNCS 4741* (pp. 529–543).

25. Nikolić, M. (2010). Statistical methodology for comparison of sat solvers. In *SAT 2010, LNCS 6175* (pp. 209–222). Springer.
26. Nikolić, M., Marić, F., Janičić, P. (2009). Instance-based selection of policies for SAT solvers. In *SAT 2009, LNCS 5584* (pp. 326–340).
27. Nikolić, M., Marić, F., Janičić, P. (2011). Simple algorithm portfolio for SAT. *Artificial Intelligence Review*, *40*(4), 457–465.
28. Ohrimenko, O., Stuckey, P.J., Codish, M. (2009). Propagation via lazy clause generation. *Constraints*, *14*(3), 357–391.
29. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B. (2008). 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.
30. Rossi, F., van Beek, P., Walsh, T., (eds.) (2006). *Handbook of constraint programming*. Elsevier.
31. Roussel, O., & Lecoutre, C. (2009). XML representation of constraint networks: format XCSP 2.1. *CoRR, abs/0902.2362*.
32. Sinz, C. (2005). Towards an optimal CNF encoding of boolean cardinality constraints. In *CP 2005, LNCS 3709* (pp. 827–831). Springer.
33. Soh, T., Inoue, K., Tamura, N., Banbara, M., Nabeshima, H. (2010). A SAT-based method for solving the two-dimensional strip packing problem. *Fundamental Information*, *102*(3–4), 467–487.
34. Tamura, N., & Banbara, M. (2008). Sugar: A CSP to sat translator based on order encoding. In *Proceedings of the third constraint solver competition* (pp. 65–69).
35. Tamura, N., Taga, A., Kitagawa, S., Banbara, M. (2009). Compiling finite linear CSP into SAT. *Constraints*, *14*(2), 254–272.
36. Tanjo, T., Tamura, N., Banbara, M. (2011). A compact and efficient SAT-encoding of finite domain CSP. In *SAT 2011, LNCS 6695* (pp. 375–376). Springer.
37. Tanjo, T., Tamura, N., Banbara, M. Azucar: A SAT-based CSP solver using compact order encoding - (tool presentation). In: *SAT 2012, LNCS 7317* (pp. 456–462).
38. Tomovic, A., Janicic, P., Keselj, V. (2006). n-gram-based classification and unsupervised hierarchical clustering of genome sequences. *Computer Methods and Programs in Biomedicine*, *81*(2), 137–153.
39. Tseitin, G.S. (1983). On the complexity of derivation in propositional calculus. In *Automation of reasoning 2: Classical papers on computational logic 1967–1970* (pp. 466–483). Springer.
40. van Hoeve, W.J. (2001). The alldifferent constraint: A survey. *CoRR*, cs.PL/0105015.
41. Walsh, T. (2000). SAT vs CSP. In Dechter, R. (ed.), *CP 2000, LNCS 1894* (pp. 441–456). Springer.
42. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K. (2008). Satzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research (JAIR)*, *32*, 565–606.