Automatically Exploiting Symmetries in Constraint Programming

Arathi Ramani and Igor L. Markov
Department of EECS, University of Michigan
1301 Beal Avenue, Ann Arbor, MI 48109, USA
{ramania, imarkov}@eecs.umich.edu

Abstract

We introduce a framework for studying and solving a class of CSP formulations. The framework allows constraints to be expressed as linear and nonlinear equations, then compiles them into SAT instances via Boolean logic circuits. While in general reduction to SAT may lead to the loss of structure, we specifically detect several types of structure in high-level input and use them in compilation. Linearity is preserved by the use of pseudo-Boolean (PB) constraints in conjunction with a 0-1 ILP solver that extends common SAT-solving techniques. Symmetries are detected in high-level constraints by solving the graph automorphism problem on parse trees. Symmetry-breaking predicates are added during compilation. Our system generalizes earlier work [10; 2; 29] on symmetries in SAT and 0-1 ILP problems. Empirical evaluation is performed n instances of the social golfers and Hamming code generation problems. We show substantial speedups with symmetry-breaking, especially on unsatisfiable instances. In general, our runtimes with the specialized 0-1 ILP solver Pueblo [26] are competitive with results reported for ILOG Solver [28] in [15].

1 Introduction

Traditional constraint programming (CP) techniques such as generalized arc consistency (GAC) are frequently the methods of choice for hard problems arising in real-world applications. Well-known packages such as ECL^{i}PS^e [23] and ILOG Solver [28] offer powerful environments for constraint specification and solver deployment. These systems provide for the development of problem-specific solvers using the best available techniques for a given problem. Another option is *reduction* - a problem for which no solver is available can be reduced to another problem for which a solver does exist.

Boolean satisfiability (SAT) is commonly used in problem reductions, since it is widely-known and many SAT solvers are available in the public domain. Unfortunately, in most cases reduction-based methods are not competitive with CP approaches developed for a problem. While CP-based techniques can take advantage of problem-specific bounds to retain tighter control of the search, SAT solvers cannot. This disadvantage is mitigated to some extent by recent breakthroughs in SAT-solving. With new exact SAT solvers such as ZChaff [20], the size and scope of application-derived instances that can be solved has widened [21]. Unfortunately, many applications do not benefit from breakthroughs in SAT solving due to inefficiencies introduced during encoding. The CNF format used for SAT instances is very restrictive, and even encoding constraints with simple linear operations can result in a blowup in size. Another cause of inefficiency is the *loss of structure during problem reductions*. Examples of structure in constraints include *linearity* and *symmetry*.

The presence of symmetries slows down search due to the existence of redundant search paths. The work in [10] describes how symmetries can be detected in a SAT instance by reduction to graph automorphism and broken by adding lexicographic ordering constraints, called MinLex symmetry-breaking predicates (SBPs). The addition of MinLex SBPs accelerates SAT solvers. Linear "counting" constraints popular in applications are studied in [2]. These constraints can be efficiently expressed using ILP, where linear equations are allowed, but expressing them in CNF may be expensive. On the other hand, generic ILP solvers such as CPLEX are sometimes not competitive with leading-edge SAT solvers on Boolean constraints. Linearity can be preserved using 0-1 ILP, a problem closely related to SAT but with an ILP-like input format. Specialized techniques developed for SAT can be adapted to 0-1 ILP without paying any penalty for generality. Recently, several specialized 0-1 ILP solvers such as PBS [2], Galena [8] and Pueblo [26] have been introduced. Symmetry-breaking techniques from [10; 1] were extended to 0-1 ILP in [4]. In [15], the author proposes symmetry-breaking ordering constraints for CSPs with *matrix models*, but it is not clear how these constraints extend to SAT/0-1 ILP reductions.

This work contributes a framework for structure-aware compilation of a class of constraint programming problems by reduction to SAT and 0-1 ILP. We generalize techniques proposed in [10; 4] to detect symmetries in constraints via reduction to *graph automorphism*. Unlike earlier work, we detect symmetries in *high-level input* and add symmetry-breaking predicates to the original specification. Our system facilitates comparison of different encoding strategies and SAT reductions. This is useful since recent work [29;

5; 6] has shown that the encoding used can dramatically affect search speed. Our goals here are (1) to generalize earlier work on the detection of symmetries and linearity in SAT instances so that it is applicable to a more general class of high-level CSPs (2) to automate the tasks of reduction to SAT/0-1 ILP and structure detection (3) to use this framework to study whether using structure can improve the performance of reduction-based methods. Earlier work [10] detects and breaks symmetries after problems have been reduced to CNF. Our work generalizes these techniques by detecting structure *before reduction* and using it to produce more effective encodings. Our empirical results for the social golfer and Hamming code generation problems show that breaking symmetries during reduction vastly improves the performance of both SAT and 0-1 ILP solvers. On many instances, our runtimes are competitive with results reported using ILOG Solver [28] in [15]. Symmetries detected by our method can be used by *any* constraints solver, not just one that assumes reduction to SAT, since we detect symmetries in high-level input. While we add SBPs during preprocessing, there are several methods that focus on breaking declared symmetries during search [25; 13] that can make use of the symmetries we detect.

The rest of the paper is organized as follows. Section 2 discusses background and previous work. Section 3 explains how symmetries are detected and broken in high-level constraints. Section 4 discusses more comprehensive symmetry-breaking, with empirical results in Section 5. Section 6 concludes the paper. The details of compilation to SAT and 0-1 ILP and the encodings we use are discussed in the Appendix.

2 Background and Previous Work

Boolean Satisfiability (SAT). A SAT instance consists of a set of 0-1 variables V, and a set of clauses C, where each clause is a *disjunction of literals*. A literal is a variable or its complement. The SAT problem asks to find an assignment to the variables in V that satisfies all clauses in C, or prove that no such assignment exists.

0-1 ILP. 0-1 ILP allows a CNF formula to be augmented with Pseudo-Boolean (PB) constraints, or linear inequalities with integer coefficients of the form: $(a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b)$ where $a_i, b \in Z$, $a_i, b \neq 0$, and x_i are literals of Boolean variables.

CNF vs. 0-1 ILP. Recent work has shown that formulating problem instances as 0-1 ILP instead of SAT can result in accelerated search. Specialized 0-1 ILP solvers have been developed in [2; 8; 26], and have been shown to perform better than both leading-edge SAT solvers [20] and generic ILP solvers such as CPLEX on some 0-1 ILP formulas. However, this is not always the case. For an application, there can be several reductions to SAT, and some encodings are more difficult to solve than others. CNF encodings for circuit layout applications in [2] contain large numbers of symmetries, increasing their difficulty. In [29], Warners proposes an efficient encoding where a PB constraint is replaced by a linear number of CNF clauses. In [5], a tree-based linear conversion is proposed to translate 0-1 ILP constraints to CNF. More recently, [6] discusses a GAC-preserving encoding, with a solver modification that results in SAT instances that are solved faster than their 0-1 ILP counterparts. Our approach constructs a parse tree and instantiates Boolean circuits for addition, multiplication and subtraction. Most previous work performs reduction to SAT on a per-problem basis, but we provide a high-level specification language in which constraints can be easily expressed and conversion to SAT/0-1 ILP is automated for all problems. Given the impact that efficient encodings have on search speed, our framework is designed so that different encodings can be easily plugged in and used with our symmetry-breaking infrastructure. Our work is relevant to the recent NP–SPEC project [7], which aims to provide a formal specification language for all problems in NP, and translate them into SAT. However, [7] does not address symmetry and linearity during compilation.

Symmetry detection and breaking. A *symmetry* of a discrete object is a reversible transformation of its components that leaves the object unchanged, for example, permutations of graph vertices that map edges into edges. Symmetries occurring in a SAT instance indicate the presence of redundant search paths, and breaking symmetries can prune the search tree and reduce search time. Detection of symmetries in CNF formulas by reduction to graph automorphism is proposed in [10]. A graph is built from a CNF formula such that there is a one-one correspondence between symmetries of the formula and the graph. The graph automorphism software Nauty [17] is used to detect symmetries in the graph. The symmetry group induces an equivalence relation on the set of variable assignments for a CNF formula. *Lex-leader* symmetry-breaking predicates (MinLex SBPs) that allow only the *lexicographically smallest* assignment in an equivalence class are defined in [10] . A more efficient SBP construction is proposed in [3]. Symmetry detection via graph automorphism is extended to 0-1 ILP in [4]. Our work generalizes these methods to a broader class of problems that use integer coefficients, non-binary variables and non-linear operations. Instance symmetries are detected at a higher level, eliminating the risk that some symmetries may be obscured during reduction. In [15], the author defines high-level lexicographic (MinLex), anti-lexicographic (anti-Lex) and multiset ordering constraints for CSPs with matrix models that exhibit symmetry. However, row and column symmetries must first be identified in matrix models for individual problems and constraints designed accordingly. Our system allows symmetries to be automatically detected in any problem instance, not just a matrix model, and used by any solver. This functionality may be useful to methods that focus on declared symmetries during search. A modified search procedure that performs partial symmetry-breaking for matrix models is proposed in [25], where SBPs are specified for a stabilizer set that is a subgroup of the symmetry group. We find generators of the symmetry group using the graph automorphism program Saucy [11], and these generators can be used by the algorithms in [25] to compute SBPs. Another related work is [13], which takes as input some generators of the symmetry group and uses them to check for dominating elements in the search tree. Since our system automatically detects generators it may be applicable to such algorithms. At present, we use only MinLex SBPs from [10]. We have not yet studied other types of

SBPs such as those in [15]. Symmetries in linear programming problems have also been discussed in [18].

3 Symmetry Detection

Unlike earlier work [10; 4], which detects symmetries in SAT/0-1 ILP instances *after* reduction, we detect symmetries in high-level input using the parse tree created from the constraint specification. Symmetries in high-level input correspond directly to symmetries of the instance and can be used to prune the search tree by multiple solvers. Symmetries detected in a SAT instance can only be used by SAT solvers, or must be traced back to the original instance to understand their significance. This reconstruction may be difficult. Also, some symmetries may be obscured during reduction. For example, counting constraints are symmetric, but the most compact encodings for these constraints [29] use comparator circuits which are not symmetric. The methods we describe here efficiently detect symmetries in high-level constraints and add SBPs to eliminate redundant search paths.

Detecting symmetries in CNF and 0-1 ILP via graph automorphism was first proposed in [10]. We follow a similar approach for high-level symmetry detection. A parse graph is built from the constraints such that there is a one-to-one correspondence between the symmetries of the constraints and the graph symmetries. We describe the graph construction only for the arithmetic operators '+', '-', and '*', but it can be extended to include more arithmetic or logical operators by adding more colors. An example formula in our specification language and the corresponding graph construction are shown in Figure 1. The formula declares two 3-bit integers x_1 and x_2, and the constraint ${x_1}^2 + {x_2}^2 == 25$. The specification language we use is described in the Appendix. Vertex shapes in the figure indicate different colors. The figure shows the symmetry between vertices for x_1 and x_2.

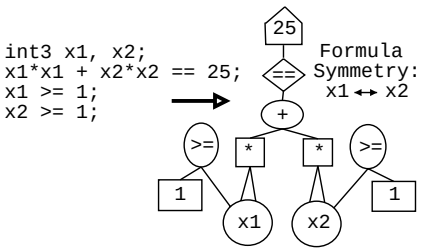

Figure 1: **Constraints declaration in our specification language and the corresponding parse graph construction. Vertices are shaped differently to indicate different colors.**

The graph construction is outlined as follows.

Step 1. Each binary variable x_i in a formula is represented by two positive and negative literal vertices, v_i and ${v_i}'$, which are given the same color. The vertices v_i and ${v_i}'$ are connected by an edge to ensure Boolean consistency. Each multi-bit variable x_j is represented by a single variable vertex v_j. A unique color is associated with each bit size.

Step 2. For each constraint C_i, two vertices T_i and R_i represent the constraint type ($\leq, \geq, ==, !=$) and RHS value respectively. A unique color is associated with each constraint type and RHS value. The vertices T_i and R_i for a constraint C_i are connected by an edge. Additionally, for each C_i:

Step 2a. Variables/literals are grouped by the priority of operations in which they occur. Multiplication between variables or by coefficients has the highest priority. '+', '-' and '*' operators have distinct colors. Each distinct coefficient value in the formula is also given a unique color. Variables connected by a '*' operator are grouped under a single *coefficient vertex* that represents the product of their coefficients (if the product is unity, this vertex is omitted). This coefficient vertex is in turn attached to a *multiplication vertex*. Variables/literals not involved in multiplication operations are grouped by coefficient, with all variables having the same coefficient value connected to a common coefficient vertex.

Step 2b. After grouping multiplicative terms, we have single variables/literals or multiplicative groups connected by '+' or '-' operations. Variables/groups associated with a '+' sign are connected directly to the constraint type vertex T_i ('+' is the default operation, so there are no special vertices for it). Variables/groups associated with a '-' operation are connected to a *negation* vertex to indicate subtraction. The negation vertex is connected to the type vertex T_i.

Theorem 3.1. *Assume that a colored parse graph is constructed from a given formula of constraints as outlined above. Then, the symmetries of the constraints correspond one-to-one to the symmetries of the graph.*

Proof. We first prove that *a symmetry in the constraints is a symmetry in the parse graph.* Consider a formula with a set V of formula variables and a set C of constraints. Consider two variables, $v_1, v_2 \in V$, and let $C_1, C_2 \subset C$ be the sets of constraints that v_1 and v_2 occur in respectively. Let v_1 and v_2 be symmetric. Then, for every constraint c in C_1 there is a corresponding constraint in C_2 that is its symmetric image.

We construct a colored parse graph $G(X,E)$ for the formula where X is the set of vertices in the graph and E the set of edges. Let x_1 and x_2 be the vertices created for v_1 and v_2 respectively, and E_1 and E_2 be the edges incident on x_1 and x_2. Assume that x_1 and x_2 are *not* symmetric in the graph construction. For this to be true, it must be true that the edge sets E_1 and E_2 are not symmetric. Without loss of generality, assume there exists some edge $e \in E_1$ that does not have an image in E_2. From the graph construction rules, an edge can connect a variable vertex to one of the following: (i) a complementary literal (ii) a constraint type vertex (for addition with unit coefficient) (iii) a negation vertex (for subtraction with unit coefficient) (iv) a multiplication vertex (for multiplication with unit coefficients) and (v) a coefficient vertex that is connected to a multiplication/negation/constraint type vertex. In the first case, assume that e connects x_1 to a complementary literal vertex, and x_2 does not possess such an edge. Then, v_2 is not a binary variable, and it cannot be symmetric to v_1. In the second case, e indicates the presence of a constraint $c \in C_1$ where v_1 is added with a coefficient of 1. Since v_1 and v_2 are symmetric in the formula, there *must* be a constraint in C_2 that matches c. However, if such a constraint existed, there would be an edge representing it in E_2, symmetric to e. The same argument applies to cases (iii) and (iv). The only special case occurs in (v), when variables

are multiplied together with different coefficients. We use the product of all coefficient values as the resulting coefficient. This reflects the fact that multiplication is commutative, i.e. $(av_1)(bv_2) = (ab)(v_1)(v_2)$ and $(cv_3)(dv_2) = (cd)(v_1)(v_2)$, so if $ab = cd$ then the expressions are symmetric.

For the other direction, we note that symmetries in the parse graph can only exist between vertices of the *same color*. Additional vertices are created to represent operations, but they can never be mapped to variable vertices. Thus, the only spurious symmetries we need to consider are between *variable vertices of the same bit size*. It is clear that the proof for the forward direction can be reversed for this case, i.e. edge sets incident on both vertices must be symmetric and represent symmetric constraints in the formula.

□

Avoiding abstraction overhead. Our graph construction generalizes earlier work in [10; 4] for CNF and 0-1 ILP formulas. Often, generalization involves paying a performance penalty - in this case, dealing with a more expressive input format that includes non-linear constraints can introduce additional vertices. This penalty can be avoided by modifying the graph when special cases are detected. Consider the case where an instance contains *only* 0-1 ILP constraints with no non-linear operations and only 1-bit variables. IN this case, our construction is designed to mimic the construction in [4], and produce *exactly* the same graphs. For pure CNF formulas, some modification is required to produce graphs as compact as the specialized constructions from [10; 1]. Since there are no coefficients or RHS values, constructions in [10] and [1] use only two types (colors) of vertices: literal and clausal. A clause with > 2 literals is represented by a clausal vertex, connected to its literal vertices. Binary clauses are represented by an edge between both literals. Graphs created by our system require constraint type and RHS value vertices for each constraint. However, CNF formulas are easy to detect. A CNF formula involves *only* binary variables. All coefficients are unity. Clauses can be expressed in two ways: as the logical-or ("||") of literals, or as the additive constraint that the sum of literals must be ≥ 1. These characteristics can be tested for, and graph construction altered accordingly.

Symmetry-Breaking Predicates (SBPs). The parse graph is analyzed for symmetries using the efficient automorphism program Saucy [11], which returns generators of the symmetry group. We generate high-level lex-leader SBPs from the generators, and add them as constraints to the original instance. These SBPs are also compiled into SAT. For multi-bit variables, SBPs may be large and complex if a generator has several cycles (for a detailed description of cycles in a generator, and the resulting predicates, see [10]). We break only the first few (1 or 2) cycles in multiple-cycle generators for simplicity. For binary variables, we implement the efficient linear-sized SBP construction in [3] and add these SBPs to the CNF formula. The problems we test here all use matrix models with binary variables. The design of efficient SBPs for multi-bit variables is a direction for future research.

4 Comprehensive Symmetry Breaking

In this section, we discuss simple extensions to increase the system's coverage of symmetries.

Symmetries in Associative Expressions. Many of the operators that we support, such as '+' and '*' are associative, i.e. $x_1 + x_2 + x_3 = x_2 + x_3 + x_1$ and $(x_1 + x_2) + x_3 = x_1 + (x_2 + x_3)$. However, parse trees built from constraints often do not reflect this symmetry. In parsing, language rules are recursively matched. This imposes a non-symmetric structure on the parse tree. We avoid this non-symmetric structure by grouping all variables connected by an associative operation together. For example, given the expression $x_1 + x_2 + x_3$ the parser first matches $x_1 + x_2$ as a single group, and then matches x_1 and x_2 individually, which is not symmetric. Our construction treats all '+'s as a single '+' operation connecting a number of expressions, which may be either identifiers or multiplicative terms. Symmetry in associative operations can also be missed when nested parentheses are used. Our system currently does not support the nesting of expressions through the '(' and ')' operators, but can be easily extended to do so. Here we discuss symmetry-detection for this case. Detecting symmetries in associative operations has been addressed in the CGRASS system [12]. However, CGRASS detects symmetries in an ad-hoc way, by keeping track of the number and type of constraints a variable occurs in and matching these for different variables. Detection via graph automorphism is more comprehensive, and given efficient software such as Saucy, incurs hardly any overhead. Our method, like CGRASS, is not complete - it uses only the generators of the symmetry group found by Saucy. For complete symmetry-breaking, the full group would have to be reconstructed from the generators. This has been found to be very time-consuming [10], whereas using only generators is more efficient and often just as effective. CGRASS also undertakes simplification of constraints in other ways, which our system does not cover.

Consider the expressions $x_1 + (x_2 + x_3) + x_4$ and $x_1 + (x_2 + (x_3 + x_4))$, which are the same, but are evaluated differently due to parenthesization. The order of evaluation imposed by parentheses hides the symmetry between variables, since expressions enclosed within '()' symbols are treated as separate sub-expressions. Evaluating parenthesized expressions separately does not account for symmetries due to associativity of operations. However, it is possible to simplify high-level input so that such symmetry is preserved. We list simplification rules for the operators '+', '-' and '*'.

Rule 1. Nested () symbols must be simplified before the outermost () operation can be simplified.

Rule 2. If an expression within () symbols is flanked by '+' and '-' operations on the left and right sides, parentheses are unnecessary, e.g., in $\ldots + (x_1 + x_2) + \ldots$ the () operators can be ignored.

Rule 3. If an expression within () symbols is multiplied by a single term, the resulting expression can be evaluated, e.g., $x_2 * (x_1 + x_4)$ is written as $x_2 * x_1 + x_2 * x_4$. It is possible to simplify the parenthesized products, e.g. $(x_1 + x_2) * (x_3 + x_4)$ by implementing multiplication rules, but this may cause a size blowup in graphs for large expressions.

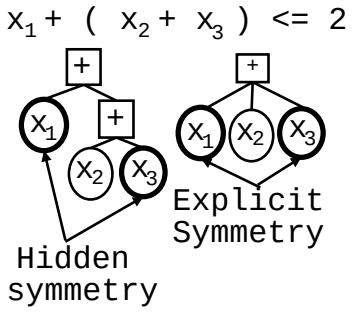

Figure 2: **Associative symmetry with parenthesized sub-expressions: x_1 and x_3 are symmetric but the original parse tree is asymmetric, since the sub-expression is represented with a separate node.**

The above list of rules can be extended further, but it already facilitates the detection of symmetries in simple associative expressions. This is illustrated in Figure 2, where x_1 and x_3 are symmetric, but the symmetry is not visible in the parse graph. With the proposed modifications the associative symmetry is preserved. Our system already implements this feature for '+' and '-' operations without parentheses, where we ignore the order in which the operations occur.

Value Symmetry. Our work so far detects *formula symmetries*, that are determined by the occurrence of variables in constraints. However, *value symmetries* that occur between the actual domains of variables can also be significant. Ordering constraints for declared value symmetries are discussed in [15], and [16] describes an algorithm to detect and break value symmetries during search. We discuss how our system may be extended to detect value symmetry.

Value symmetry can arise from *operators* that control the value of a variable, e.g. the *complement* operation on binary variables, i.e. $a' = 1 - a$. The mapping $a \leftrightarrow a'$ is known as a *phase shift symmetry*. The construction from [10] does not always account for phase-shift symmetries, but [1] proposes an improvement that detects phase-shift symmetries in almost all cases. For the non-binary case, such symmetries may arise in problems with a *cyclic* nature, for example, scheduling problems. Any scheduling solution for {Monday, Tuesday, Wednesday} can often be shifted to {Tuesday, Wednesday, Thursday}. Such shifts can also be described by an operator - if a variable's domain is a cyclic group modulo 4, we can say $a'' = (a+1)\%4$. Intuitively, the graph construction to represent a cyclic group of values is a cycle of vertices. However, if the domain size is > 2, this will result in spurious symmetries if all vertices are given the same color, since a'' can map to $(b'')''$, and so on. Each vertex in the cycle must be given a different color for this construction to work. This allows cyclic symmetries to be detected. The SBPs we use may need to be modified for this case, and our ongoing work is focused on proving correctness in SBP construction.

Giving each value in a variable's domain a different color prevents the detection of value symmetries between values in the domain of the same variable. A set of constraints satisfied when $a = 0$ may also be satisfied when $a = 2$. This type of symmetry-detection is addressed in [16]. Adapting our techniques to detect such symmetries is more difficult, since it may require the enumeration of variable and constraint values in the graph, resulting in very large and complex graphs. Another focus of our current work is developing efficient graph constructions for this case.

5 Empirical Results

We test our system on constraint programming problems with *matrix models* with row and/or column symmetries from [15]. Each problem is modeled using the constraints described in [15] and specified in our system's input language, followed by symmetry detection and compilation to SAT and 0-1 ILP. SBPs are added to the CNF or ILP instances. We use Saucy [11] to detect symmetries, ZChaff to solve SAT instances, and the new 0-1 ILP solver Pueblo [26] to solve 0-1 ILP instances. We show results for the balanced incomplete block design problem (BIBD), social golfer problem (SG) and Hamming code generation (HC) problems. Results here are obtained using a Intel Pentium processor processor at 1GHz for the SG and HC problems, and an Intel Xeon dual processor at 2 GHz. Both systems have 1GB of RAM and run RedHat Linux 9.0. ZChaff and Pueblo runtimes are the average of 3 starts. Timeout is set at 600 seconds. For BIBD instances, we use the Xeon processor at 2GHz to compare our encodings with those in [24]. For SG and HC instances, we use the 1GHz Pentium processor to allow runtime comparisons with [15]. Symmetry-breaking ordering constraints in [15] are implemented using ILOG Solver and tested on a 1 GHz Pentium processor running Windows XP. We note that [15] also reports a "number of failures" metric, which is the number of incorrect decisions made by Solver at nodes in the search tree. We do not have access to Solver and the SAT/0-1 ILP solvers we use do not report such a statistic. The SBPs we use are added as part of the instance and a SAT/0-1 ILP solver cannot distinguish between SBPs and regular constraints. Therefore, we cannot report a similar metric for our techniques, and runtime is the only comparable statistic[1]. However, we use exactly the same hardware as [15] so that runtime comparisons are fair. Since it is not possible for us to use Solver, we use results directly from [15].

Balanced Incomplete Block Design Problem (BIBD). This problem asks to find $b > 0$ subsets of a set V of $v \geq 2$ elements such that each subset contains exactly k elements $(v > k > 0)$, each element appears in exactly $r > 0$ subsets, and each pair of elements appears together in exactly $\lambda > 0$ subsets. An instance is expressed as the 5-tuple (v, b, r, k, λ), and named `bibd(v,b,r,k,λ)` in the results table. We use the matrix model described in [15] (originally from [19]). We initially tested encodings with and without SBPs using ZChaff and Pueblo on the large instances used in [15] (originally from [9]). However, our observation on these instances was that adding MinLex SBPs actually affects performance negatively for the Pueblo solver (ZChaff is unable to solve most instances within the time limit, with or without SBPs). For satisfiable instances, this is not unusual and has been noted earlier in [10]. When there are several solutions, adding

[1] Due to lack of space, we cannot report both number of failures and runtime from [15]

SBPs may prevent some solutions from being found earlier in the search. This is borne out by our results on other problems. However, this does not explain the poor performance on unsatisfiable instances of this problem, which may be because MinLex SBPs are not useful in this case. In [15], several types of SBPs are tested, and the most effective SBPs for the BIBD problem anti-Lex ordering constraints. Since anti-Lex orderings are the reverse of MinLex orderings, they permit different assignments than MinLex, and may be more helpful in finding solutions for BIBD. However, we use this problem to illustrate the importance of efficient encodings. SAT encodings for the BIBD problem have been developed in [24], where the instances used are difficult for many SAT solvers, but are solved by CP solvers in a few minutes. These encodings are available at [14], with and without symmetry-breaking clauses from [24]. Table 1 shows a comparison of both encodings. The first column gives the instance parameters, followed by Saucy statistics for high-level symmetry-detection. This is followed by ZChaff and Pueblo runtimes for our encoding, and ZChaff runtimes for encodings from [24] with and without SBPs. Pueblo does not accept instances without 0-1 ILP constraints. Both Pueblo and ZChaff solve all instances with our encoding in a few seconds, but ZChaff times out on several instances from [24]. All instances possess symmetries, but Saucy runtimes are negligible.

Social Golfers (SG). This problem seeks to divide $g \times s$ golfers into g groups of size s for each of w weeks. Each golfer must play once a week. Any two golfers play in the same group at most once. A problem instance is described by its parameters (g,s,w) and is named `sg(g,s,w)` in the results table. We use the modified 3-D matrix model from [15], and the same instances used in [15]. Instances are tested on ZChaff and Pueblo with and without SBPs.

Results are shown in Table 2. The first column gives instance parameters (`sg` for SG instances).followed by the number of symmetry generators and runtime for Saucy. Next, we show approximate instance sizes and runtimes with and without SBPs. For SAT conversions, we show the number of variables and clauses. For 0-1 ILP instances we also show the number of PB constraints, which is the same as the number of high-level constraints in the instance specification. The best runtimes for a given instance are boldfaced. For this problem, adding SBPs speeds up Pueblo considerably on *unsatisfiable* benchmarks. For *all* cases where Pueblo is slower with SBPs, the instance is satisfiable. ZChaff is faster with SBPs for both SAT and UNSAT cases, but is not competitive with Pueblo. All instances possess large numbers of symmetries. The last column shows results reported in [15][2]. Pueblo is usually competitive with Solver results from [15] on SAT instances without the addition of SBPs. However, on UNSAT instances, SBPs are needed to make it competitive, and are effective in doing so. For the larger instances, Saucy runtimes are significant. This increases the overall time for our flow. However, [15] requires SBPs to be designed and implemented separately for individual problems. Our system is automated and generalized. Moreover, [15] reports results for four models of SBPs. Two of these are basic SBPs that assign values to a subset of the variables in an instance, thus forcing assignments that satisfy constraints on the remaining variables. The other two models use MinLex and anti-Lex constraints. Here, we report the best results among all models. Given an instance it may not be clear which model to use for best results until several have been tried. There is no model in [15] which consistently performs well for this problem. Our system uses only MinLex SBPs.

Hamming Code Generation (HC). This problem seeks to find $b-$bit code words to code n symbols, where the Hamming distance between two symbols is at least d. An instance is specified by the parameters (n,b,d). We use the matrix model from [15], and report results with and without symmetry-breaking in the last four rows of Table 2. The instances `hc(10,15,9)` and `hc(12, 20, 12)` are unsatisfiable, and the other two are satisfiable. [15] Results for the first two instances are available in [15], the last two are listed as N/A. We observe that symmetry-breaking is useful for both SAT and UNSAT instances, with greater benefit for UNSAT instances. Adding SBPs speeds up ZChaff in all cases, but it is not competitive with Pueblo and Solver. Results reported from [15] are the best out of several combinations of lexicographic and multiset-ordering SBPs. However, several of these combinations are not competitive with our results using Pueblo with SBPs.

Overall, the detection of structure - both linearity through 0-1 ILP and symmetries by the addition of SBPs - improves performance considerably for both Pueblo and ZChaff. For most unsatisfiable instances, the best results are obtained using Pueblo with SBPs added. For satisfiable instances, Pueblo is not improved by SBPs, and in some cases is actually slower. However, ZChaff benefits from SBPs for both SAT and UNSAT instances. This may be because SBPs have greater impact on variable orderings for Pueblo. In most cases Pueblo's results are competitive with results reported for Solver in [15] over a variety of symmetry-breaking ordering constraints. For the cases where Pueblo is faster with SBPs, the average speedup over its performance without SBPs is 83.2, not including timeouts for the no-SBP version. On satisfiable instances, the average slowdown with SBPs is 5.6, but it is much less than that in most cases and there are no timeouts with SBPs. Our system uses academic solvers whose source code and/or binaries are publicly available, but runtimes are comparable with those of Solver, a highly optimized commercial tool.

All results here use problems with matrix models, which frequently possess large numbers of symmetries by construction. While row and column symmetries can be detected manually in a matrix model, our system provides a way to detect and break these symmetries automatically without having to give it any knowledge of the problem semantics. Moreover, it is not restricted to matrix models, and may be used for problems that are likely to have symmetry, but for which matrix models do not exist. It is also applicable in cases where added constraints may disrupt the symmetry in matrix models, e.g. for instances with "customized" requirements. For example, in the social golfer problem, we can add the constraint that certain pairs of golfers must *never* be in the same

[2]Results in [15] are on a logarithmic scale, so our numbers are not exact, but all runtimes are rounded *down* for fairness.

Instance Name	Symmetry Stats			Our Encoding				Encoding in [24]	
	Symm.	Gen.	Saucy Time	W. SBPs		W/o. SBPs		W. SBPs	W/o. SBPs
				ZChaff	Pueblo	ZChaff	Pueblo	ZChaff	ZChaff
bibd(7,7,3,3,1)	2.54e7	12	0	0.08	**0**	0.01	**0**	0.29	T/O
bibd(6,10,5,3,2)	2.61e9	14	0	0.54	**0**	0.03	**0**	54.24	T/O
bibd(7,14,6,3,2)	4.39e14	19	0.01	0.38	**0.01**	1.25	**0.01**	T/O	T/O
bibd(9,12,4,3,1)	1.73e14	19	0.02	0.64	**0.01**	1.89	0.013	T/O	T/O
bibd(8,14,7,4,3)	3.51e15	20	0.02	0.72	0.01	1.57	**0**	T/O	T/O

Table 1: **ZChaff results and Saucy statistics for BIBD instances using our encodings and those in [24], with and without SBPs. T/O indicates timeout at 600s. Pueblo is not tested on encodings in [24], since they are not available as 0-1 ILP.**

Instance Params	Saucy Stats		Size with SBPs							Size w/o SBPs							Sol ver
	G.	Tm.	CNF - ZChaff			0-1 ILP - Pueblo				CNF - ZChaff			0-1 ILP - Pueblo				
			Var.	Cl.	Tm.	Var.	Cl.	PB	Tm.	Var.	Cl.	Tm.	Var.	Cl.	PB	Tm.	
sg(2,5,4)	16	0.02	6311	33K	0.06	1694	1361	141	**.003**	6139	32K	0.12	1522	721	141	0.01	.01
sg(2,6,4)	18	0.02	9076	48K	0.14	2418	1835	178	**.006**	8868	46K	0.15	2210	1057	178	0.01	0.1
sg(2,7,4)	20	0.03	12K	65K	0.31	3270	2373	219	**0.01**	12041	63894	0.14	3026	1457	219	0.02	5
sg(2,8,5)	24	0.07	22K	125K	1.25	5320	3761	300	**0.02**	22K	123K	0.89	4962	2401	300	**0.02**	30
sg(3,5,4)	25	0.09	26K	155K	2.27	5645	4138	249	**0.05**	26K	152K	T/O	5222	2521	249	7.54	0.5
sg(3,6,4)	28	0.14	37K	221K	1.63	8072	5629	321	**0.09**	37K	219K	T/O	7562	3673	321	25.7	0.4
sg(3,7,4)	31	0.21	51K	299K	7.7	10K	7336	402	**0.17**	50K	296K	120	10K	5041	402	24.8	0.5
sg(4,5,4)	34	0.30	70K	430K	11.5	13K	9115	382	0.25	69K	426K	T/O	12K	6081	382	T/O	**0.2**
sg(4,6,5)	42	0.75	134K	837K	T/O	23K	15K	556	**0.5**	132K	831K	T/O	22K	11K	556	T/O	2
sg(4,7,4)	42	0.79	135K	829K	T/O	25K	16K	634	**0.62**	134K	824K	T/O	24K	12K	634	T/O	5
sg(4,9,4)	50	1.75	221K	1.35M	T/O	42K	25K	950	**1.41**	220K	1.34M	T/O	40K	20K	950	T/O	2.5
sg(5,4,3)	33	0.26	64K	394K	17.1	12K	8502	340	0.37	64K	391K	315	11K	5701	340	**0.07**	0.1
sg(5,5,4)	43	0.89	145K	911K	300	25K	16K	540	1.3	144K	906K	T/O	24K	12K	540	1.17	**0.9**
sg(5,7,4)	53	2.79	281K	1.76M	T/O	50K	30K	915	**1.8**	279K	1.75M	T/O	48K	23K	915	T/O	7
sg(5,8,3)	53	2.3	250K	1.51M	107	48K	29K	1050	1.76	248K	1.51M	T/O	47K	23K	1050	T/O	**0.6**
sg(6,4,3)	40	0.61	118K	733K	496	21K	14K	456	0.86	117K	729K	T/O	20K	9937	456	**0.47**	0.5
sg(6,5,3)	46	1.25	182K	1.13M	T/O	33K	20K	651	1.9	181K	1.12M	T/O	31K	15K	651	1.02	**0.6**
sg(6,6,3)	52	2.51	260K	1.61M	T/O	47K	28K	882	2.57	259K	1.60M	T/O	46K	22K	882	**0.1**	50
sg(7,5,3)	54	3.06	301K	1.89M	T/O	52K	32K	847	3.85	299K	1.88M	T/O	50K	24K	847	**1.9**	1K
sg(7,5,5)	68	11.4	551K	3.55M	T/O	87K	54K	1015	59.2	547K	3.53M	T/O	84K	41K	1015	37	**20**
hc(10,15,9)	38	0.07	32K	206K	93.4	5842	3762	45	**0.59**	32K	205K	T/O	5552	2701	45	T/O	7.2
hc(10,10,5)	28	0.04	19K	122K	T/O	3892	2487	45	22.2	19K	121K	T/O	3702	1801	45	T/O	**0.4**
hc(10,15,8)	38	0.07	32K	206K	T/O	5842	3762	45	**275**	32K	205K	T/O	5552	2701	45	286	N/A
hc(12,20,12)	50	0.19	66K	426K	T/O	11K	7023	66	**2.77**	65K	10K	424K	5281	10K	66	T/O	N/A

Table 2: **Results for social golfers and Hamming code generation problems. Best results for a given instance are boldfaced. T/O indicates timeout at 600s. The last column shows results from [15]. 'K' and 'M' in instance sizes indicate multiples of one thousand and one million.For UNSAT instances, using Pueblo with SBPs generally performs best. For SAT instances Pueblo is slowed down by SBPs, however ZChaff benefits from SBPs even on SAT instances. All runtimes are in seconds. N/A in the last two rows indicates that results for these instances are not shown in [15].**

group. The present matrix model has symmetry along all three dimensions - groups, weeks and golfers. Adding pairwise constraints for specific golfers would leave only partial symmetry between golfers, which poses more effort for manual identification of symmetries. However, with our method added constraints can be analyzed and surviving symmetries detected without any modification. Even if row/column symmetry between certain rows and columns is destroyed, we can still detect symmetries that exist between specific variables in these rows and/or columns automatically. We also hope to identify problems that can be analyzed using our system, but for which matrix models are not applicable.

6 Conclusion

We present an integrated framework for studying and solving a class of CSPs by reduction to SAT and 0-1 ILP. The framework provides for the specification of constraints in a high-level language and automatic compilation into SAT. Specialized methods for SAT have improved considerably over the last 10 years, but these improvements do not necessarily apply to more sophisticated domains because SAT encodings are not always possible and may introduce inefficiencies due to the loss of *structure* in problem reductions. Our system automatically detects certain types of structure, such as linearity and symmetries during compilation and uses them to produce more efficient encodings. Linearity is preserved through the use of 0-1 ILP, a comparatively more sophisticated problem with specialized solvers that can use leading-edge techniques for SAT solving.

We extend earlier work on symmetry-detection in SAT and

0-1 ILP [10; 4] to a more general class of CSPs that may use non-binary variables and non-linear operations. Symmetries are detected in high-level input by solving the graph automorphism problem on parse trees. MinLex symmetry-breaking predicates (SBPs) from [10] are added to the resulting SAT/0-1 ILP encodings. Other work [15] has focused on symmetry-breaking ordering constraints for known or declared symmetries in generalized CSPs, but we detect and break symmetries automatically. Empirically, we evaluate our system on the balanced incomplete block design (BIBD), social golfers (SG) and Hamming code generation (HC) problems. We detect large numbers of symmetries in all instances, and show that breaking symmetries produces substantial speedups for the 0-1 ILP solver Pueblo [26] on unsatisfiable instances of the SG and HC problems. When symmetry-breaking is useful on unsatisfiable instances, the average speedup is 83.2 over the no-SBPs case (not including several timeouts). For satisfiable instances, there is a small slowdown with SBPs, but no timeouts. For CNF reductions, the SAT solver ZChaff [20] exhibits speedups for both satisfiable and unsatisfiable instances when symmetries are broken. Overall, CNF reductions are not competitive with 0-1 ILP reductions. A somewhat surprising observation is that on many satisfiable instances, Pueblo is slowed down by the addition of symmetry-breaking predicates (SBPs). This may be because adding SBPs to satisfiable instances prevents some solutions from being found by Pueblo. More effective SBPs need to be developed for this case. Overall, our runtimes for Pueblo with SBPs added are competitive with Solver runtimes reported in [15] on unsatisfiable instances of the SG and HC problems. We also show that our circuit-based CNF encodings for the BIBD problem are more efficient than those proposed in [24]. In general, our system facilitates the comparison of different SAT encodings, since any encoding can be plugged into our framework and automatically tested on several instances. This is useful since encodings often have a huge impact on search speed [29; 2; 5; 6]. Symmetries detected in high-level input can be used by *any* constraints solver and by other methods that add SBPs for declared symmetries during search [25; 13]. Moreover, SBPs can be added to a SAT/0-1 ILP reduction even if the actual encoding used obscures symmetries, since they are detected before reduction. We provide an extensible framework that can be easily modified to include other types of constraints and operations, and discuss two such extensions for symmetries due to *associative operations* and *value symmetries*. We plan to release code in the public domain to facilitate experimentation with different problems and encodings. At present, more information on this project, and contact addresses for source code, binaries and sample input files are available at [27].

Our current and future work is focused on extending our system to allow more comprehensive coverage of symmetries, e.g. symmetries in associative expressions and value symmetries briefly discussed in Section 4. We plan to extend our compiler to allow more operations and different types of constraints, and to support more OPL-like [22] syntax. Another direction is the development of efficient SBPs for non-binary variables and of symmetry-breaking constraints that are more effective on satisfiable instances.

References

[1] F. A. Aloul, A. Ramani, I. L. Markov, K. A. Sakallah, "Solving Difficult SAT Instances In The Presence of Symmetry", *IEEE Transactions on CAD*, vol. 22(9), pp. 1117-1137, 2003.

[2] F. A. Aloul, A. Ramani, I. L. Markov, K. A. Sakallah, "Generic ILP versus Specialized 0-1 ILP: An Update", *in Proceedings of the International Conference on Computer-Aided Design*, pp. 450-457, 2002.

[3] F. A. Aloul, I. L. Markov, K. A. Sakallah, "Shatter: Efficient Symmetry-Breaking for Boolean Satisfiability", *in Proc. Intl. Joint Conf. on AI*, pp. 271-282, 2003.

[4] F. A. Aloul, A. Ramani, I. L. Markov, K. A. Sakallah, "Symmetry-Breaking for Pseudo-Boolean Formulas", *in Proceedings of the Asia-South Pacific Design Automation Conference*, pp. 884-887, 2004.

[5] O. Bailleux, Y. Boufkhad, "Efficient CNF Encoding of Boolean Cardinality Constraints", *Proceedings of Principles and Practice of Constraint Programming*, pp. 109-122, 2003.

[6] O. Bailleux, Y. Boufkhad, "Full CNF Encoding: The Counting Constraints Case", in *7th Intl. Conf. on Theory and Applications of SAT Testing*, 2004.

[7] M. Cadoli et. al, "NP-SPEC: An Executable Specification Language for Solving All Problems in NP", *Proceedings of Practical Aspects of Declarative Languages*, pp.16-30, 1999.

[8] D. Chai, A. Kuehlmann, "A fast pseudo-boolean constraint solver", *in Proceedings of the Design Automation Conference*, pp.830-835, 2003.

[9] C. H Colbourn, J. H. Dinitz, "The CRC Handbook of Combinatorial Designs", CRC Press, 1996.

[10] J. Crawford, M. Ginsburg, E. M. Luks, A. Roy, "Symmetry-breaking predicates for search problems", *in Proc. of the Intl. Conf. on Principles of Knowledge Representation and Reasoning*, pp. 148-159, 1996.

[11] P. Darga, "SAUCY Man Page", `http://vlsicad.eecs.umich.edu/BK/SAUCY/`

[12] A.M. Frisch, I. Miguel, T. Walsh, "Cgrass: A System for Transforming Constraint Satisfaction Problems", *Jt. Workshop of ERCIM/CologNet area on Constr. Solving and Constr. Logic Prog.*, pp. 23-26, 2002.

[13] I. P. Gent, W. Harvey, T. Kelsey, S. Linton, "Generic SBDD using Computational Group Theory", *in Principles and Practice of Constr. Prog.*, pp. 333-347, 2003.

[14] I. P. Gent, T. Walsh, B. Selman, CSPLib Problem Library for Constraints; `http://www.csplib.org`

[15] Z. Kiziltan, "Symmetry Breaking Ordering Constraints", *Doctoral Thesis*, Uppsala University, 2004.

[16] A.Lal, B. Choueiry, "Dynamic Detection and Exploitation of Value Symmetries for Non-Binary Finite CSPs", *Workshop on Symmetry in CSPs*, 2003.

[17] B. McKay, "Practical Graph Isomorphism", *Congressus Numerantium*, vol. 30, pp. 45-87, 1981.

[18] F. Margot, "Exploiting Orbits in Symmetric ILP", Mathematical Programming Ser. B 98, pp. 3-21, 2003.

[19] P. Meseguer and C. Torras, "Solving strategies for highly symmetric CSPs", *in Proceedings IJCAI*, pp. 400-405, 1999.

[20] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, "Chaff: Engineering an Efficient SAT Solver", *in Proc. Design Automation Conf.*, pp. 530-535, 2001.

[21] G. Nam, F. Aloul, K. Sakallah, R. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints", *in Proc. of the Intl. Symposium on Physical Design*, pp. 222-227, 2001.

[22] P. van Hentenryck, "The OPL Optimization Programming Language", the MIT Press, 1999.

[23] The ECL^{i}PS^e Team, "The ECL^{i}PS^e Constraint Logic Programming System": `http://www.icparc.ic.ac.uk/eclipse/`

[24] S. D. Prestwich, "Balanced Incomplete Block Design as Satisfiability", *in 12th Irish Conference on Artificial Intelligence and Cognitive Science* , 2001.

[25] J. F. Puget, "Symmetry Breaking Using Stabilizers", *Principles and Practice of Constraints Prog.*, pp. 585-599, 2003.

[26] H. Sheini, "Pueblo": PB constraint solver; `http://www.eecs.umich.edu/~hsheini/pueblo`

[27] A. Ramani and I.L. Markov, "GSymEx": Generic Symmetry Extraction for Constraint Programming Problems; `http://vlsicad.eecs.umich.edu/BK/GSymEx/`

[28] ILOG Solver, `http://www.ilog.com/products/solver/`

[29] J. P. Warners, "A Linear-Time Transformation of Linear Inequalities into Conjunctive Normal Form", *in Information Proc. Letters*, vol. 68(2), pp. 63-69, 1998.

Appendix: Compilation into SAT/0-1 ILP

Below, we describe how constraints are translated into CNF and 0-1 ILP. We use a C-like language for high-level constraint specification, and a customized parser that builds a parse tree for the system of constraints. Compilers for SAT and 0-1 ILP then walk the parse tree and translate the constraints into CNF/0-1 ILP formulas. The formulas are handed to SAT/0-1 ILP solvers and solutions are translated back into a form that is meaningful to the original problem. The input language uses C-like syntax to declare variables and specify constraints. Variables are specified as unsigned integers of varying bit sizes, e.g. `int1` represents a 1-bit (binary) variable, etc. The mathematical operators allowed are addition (+), subtraction (-) and multiplication (*). Relational operators may be $<=$, $>=$, $==$, and $!=$ (not-equal constraint). *Complement* notation is allowed to express the negative literal for a binary variable ($x1'$ for $x1$). Numeric constants are allowed as coefficients or as the right-hand-side (RHS) value of equations. Division is not presently supported. The compiler also does not support the use of nested parentheses or unary negation but can be easily extended to do so. Support for more sophisticated language constructs, e.g., those used by OPL [22], may be added in the future. An example of constraint declaration in the input language is shown in Figure 1 in Section 3.

To compile into SAT, Boolean "circuits" are instantiated to carry out mathematical operations. An $n-$bit variable is represented by n binary variables in the CNF instance plus a sign bit, which is necessary to perform subtraction with 2's complement notation. The size of the CNF circuits depends on the operation to be performed. Ripple-carry adders are instantiated for addition operations, and subtraction is performed using 2's complement representation. Both adder and subtractor circuits are linear in the input size. Multiplication is implemented using circuits for Booth's algorithm which are quadratic in the input size. Comparison against RHS values uses a linear comparator circuit. There are some built-in optimizations, e.g. smaller circuits for 1-bit addition and subtraction. 1-bit multiplication uses an AND gate. Circuits with a constant as input are partially evaluated. For compilation into 0-1 ILP, linearity is preserved by stating '+' and '-' operations directly as 0-1 ILP constraints. Inequalities ($\leq$, $\geq$, $==$) are also directly expressed in 0-1 ILP, with no need for comparator circuits. Coefficients can be directly written and not multiplied. Multiplication between variables uses CNF clauses, but multiplier outputs can be added/subtracted as part of a linear constraint.