Constraints manuscript No.
(will be inserted by the editor)

SymChaff: Exploiting Symmetry in a Structure-Aware Satisfiability Solver

Ashish Sabharwal

the date of receipt and acceptance should be inserted later

Abstract This article presents a new low-overhead framework for representing and utilizing problem symmetry in propositional satisfiability algorithms. While many previous approaches have focused on symmetry extraction as a key component, the novelty in the proposed strategy lies in using high level problem description to pass on symmetry information to the SAT solver in a simple and concise form, in addition to the usual CNF formula. This information, comprising of the so-called symmetry sets and variable classes, captures variable semantics relevant to "complete multi-class symmetry" and is utilized dynamically to prune the search space. This allows us to address many limitations of alternative approaches like symmetry breaking predicates, implicit pseudo-Boolean representations, general group-theoretic methods, and ZBDDs. We demonstrate the efficacy of our technique through a solver called SymChaff, which achieves exponential speedup over DPLL-based SAT solvers on problems from both theory and practice, often by simply using natural tags or annotation in the problem specification.

Keywords Boolean satisfiability · SAT · SymChaff · complete multi-class symmetry · global symmetry · high-level representation · planning

1 Introduction

Propositional satisfiability or SAT is the classic NP-complete problem of determining whether or not a given Boolean formula has any satisfying assignments. This problem has proved to be of immense importance in both theory and practice. In recent years, many general purpose propositional satisfiability algorithms (SAT solvers) have been designed and shown to be very successful in handling and even outperforming specific tools on problems from many real-world domains including hardware verification [7, 56], automatic test pattern generation [33, 54], planning [30], and scheduling [26]. With a large community of

This work was primarily done while the author was at the University of Washington, Seattle. A preliminary version of it appeared at the 20th National Conference on Artificial Intelligence (AAAI), Pittsburgh, PA, 2005 [51] and also in the author's Ph.D. thesis [50].

A. Sabharwal
Department of Computer Science, Cornell University, Ithaca, NY 14853-7501, U.S.A.
E-mail: sabhar@cs.cornell.edu

researchers working towards a better understanding of SAT, it is not surprising that many competing SAT solvers have come into light, such as `Grasp` [38], `Relsat` [5], `SATO` [58], `zChaff` [40], `Berkmin` [25], `March-eq` [28], and `MiniSat` [16].

All of the above solvers fall into the category of systematic DPLL-based algorithms for formulas in conjunctive normal form (CNF). They build upon a basic branch-and-backtrack technique based on falsified clauses, due originally to Davis, Putnam, Logemann, and Loveland [12, 13], and provide logically sound proofs of unsatisfiability when the given formula has no solutions. They can be viewed as efficient practical implementations of the *clause learning proof system*, and hence of the *resolution* proof system [6]. By adding to the basic DPLL method features such as smart branch selection heuristics, conflict clause learning, random restarts, conflict-directed backjumping, and fast unit propagation using watched literals, these solvers have proved to be very effective in solving many challenging problems from a variety of domains of interest.

Despite the success, one aspect of many theoretical as well as real-world problems that we argue has not been fully exploited is the presence of *symmetry* or *equivalence* amongst the underlying objects. Symmetry can be defined informally as a mapping of a constraint satisfaction problem (CSP) onto itself that preserves its structure as well as its solutions. While we leave a formal definition to Section 2.2, the concept of symmetry in the context of SAT solvers and in terms of higher level problem objects is best explained through some examples of the many application areas where it naturally occurs. For instance, in FPGA (field programmable gate array) routing used in electronics design, all available wires or channels used for connecting two switch boxes are equivalent; in our design, it does not matter whether we use wire #1 between connector X and connector Y, or wire #2, or wire #3, or any other available wire. Similarly, in circuit modeling, all gates of the same "type" are interchangeable, and so are the inputs to a multiple fanin AND or OR gate; in planning, all identical boxes that need to be moved from city A to city B are equivalent; in multiprocessor scheduling, all available processors are equivalent; in cache coherency protocols in distributed computing, all available identical caches are equivalent. A key property of such objects is that when selecting k of them, we can choose, *without loss of generality*, any k. This without-loss-of-generality reasoning is what we would like to incorporate in an automatic fashion.

The question of symmetry exploitation that we are interested in addressing arises when instances from domains such as the ones mentioned above are translated into CNF formulas to be fed to a SAT solver. A CNF formula consists of constraints over different kinds of variables that typically represent tuples of these high level objects (e.g., wires or boxes) and their interaction with each other. For example, during the problem modeling phase, we could have a Boolean variable $z_{w,c}$ that is TRUE if and only if the first end of wire w is attached to connector c. When this formula is converted into DIMACS format for a SAT solver, the *semantic meaning* of the variables, that, say, variable 1324 is associated with wire #23 and connector #5, is discarded. Consequently, in this translation, the global notion of the obvious interchangeability of the set of wire objects is lost, and instead manifests itself indirectly as a symmetry between the (numbered) variables of the formula and therefore also as a symmetry within the set of satisfying (or un-satisfying) variable assignments. These sets of symmetric satisfying and un-satisfying assignments artificially explode both the satisfiable and the unsatisfiable parts of the search space, the latter of which can be a challenging obstacle for a SAT solver searching for a satisfying assignment.

Example 1 For concreteness, we give one simple but detailed example of symmetry in SAT solvers. At the risk of appearing narrow in scope, we choose the *pigeonhole principle*,

PHP^n_m: given n pigeons and m holes, there is no one-to-one mapping of the pigeons to the holes when $n > m$. Translated into a CNF formula over variables $x_{i,j}$ denoting that pigeon i is mapped to hole j, this has two kinds of clauses. We use the notation $[p]$ to denote the set $\{1,2,\ldots,p\}$.

- Pigeon clauses: for $i \in [n]$, clause $(x_{i,1} \vee x_{i,2} \vee \ldots \vee x_{i,m})$ says that pigeon i is mapped to some hole, and
- Hole clauses: for $i \neq k \in [n]$, $j \in [m]$, hole clauses $(\neg x_{i,j} \vee \neg x_{k,j})$ say that no two pigeons are mapped to one hole.

This formula, despite being extremely simple to state, is a cornerstone of proof complexity research. Haken [27] used PHP^n_{n-1} to show the first ever exponential lower bound for the resolution proof system. Since then several researchers have improved upon and generalized his result to $m \ll n$, to other counting-based formulas, and to stronger proof systems. Needless to say, the results from the 2005 SAT competition [36] testify that the pigeonhole formulas still provide a class of hard instances for complete SAT solvers. Returning to the context of symmetry, PHP^n_m contains two natural sets of equivalent or symmetric objects, the n pigeons and the m holes. Accordingly, *all* variables $x_{i,j}$ in this formula are symmetric to each other to begin with. As we will see, it helps to distinguish between the "pigeon-symmetry" between $x_{i,j}$ and $x_{k,j}$, and the "hole-symmetry" between $x_{i,j}$ and $x_{i,\ell}$. Note that in the DIMACS format for SAT solvers, these mn variables lose their semantic meaning and are represented simply as the set $\{1,2,\ldots,mn\}$ of numbered variables.

Remark 1 While we use PHP^n_m as a motivation, we would like to remind the reader that the techniques we develop are generic and capable of handling symmetry in more complex forms that we will describe in due course. There are known techniques for dealing with the pigeonhole problem in SAT solvers, such as the use of cardinality constraints [8, 15] or compressed breadth-first search [41, 42]. However, such approaches either do not easily generalize or do not perform as well in the presence of complex forms of symmetry beyond this simple problem.

Current complete SAT solvers are unable to fully capitalize on such symmetry, as suggested by our experimental results.[1] The goal of this work is to develop a new general purpose technique towards this end and to empirically evaluate its effectiveness. While our approach involves adding structure to the problem description, we still stay within the realm of SAT solvers (as opposed to using, say, a constraint programming (CP) approach), thereby exploiting the many benefits of the CNF form and the advances in state-of-the-art SAT solvers.

When viewing complete SAT solvers as implementations of proof systems, the challenge here is to push the underlying proof system up in the weak-to-strong proof complexity hierarchy without incurring the significant cost that typically comes from large search spaces associated with complex proof systems. While most of the current SAT solvers implement subsets of the resolution proof system, our proposed solver, SymChaff, brings it up closer to *symmetric resolution*, a proof system known to be exponentially stronger than resolution [34, 55]. More critically, it achieves this in a time- and space-efficient manner.

A distinguishing aspect of our approach is that, in a sense, the semantic meaning of variables with respect to object symmetry is derived from a high level description and provided to SymChaff as part of the input. (We will see concrete examples of this later.) This

[1] There has been more success in exploiting symmetry in *domain-specific* algorithms and techniques. We are, however, interested in general purpose reasoning systems.

leads to several advantages. The high level description is typically very concise and reveals the structure of the problem much better than a large set of CNF clauses encoding the same problem. It is simple, in many cases almost trivial, for the problem designer to specify global symmetries at this level using straightforward "tagging" or annotation.

We note that while symmetries are annotated manually in the empirical evaluation presented in this work, this is not a necessity by any means. If one prefers to compute these symmetry annotations in an automated manner, off-the-shelf graph isomorphism tools can be used directly on the high level problem description. For instance, for planning problems, the high level description is typically in the form of the standard STRIPS language [18], whose basic building blocks are already the "objects" under consideration, such as trucks or locations or wires. In this case, a graph isomorphism tool can be used to identify sets of objects that are syntactically identical in the STRIPS specification, by analyzing the bipartite graph formed from these objects on one side and all predicates that the objects appear in (in either the initial state or the goal state) on the other side. Since a STRIPS description with a few dozen objects often gets translated into a CNF formula with thousands of variables and tens of thousands of clauses (cf. Section 4, Tables 1 and 2), it is reasonable to expect graph isomorphism tools to be substantially more scalable when run on the concise STRIPS description rather than on the corresponding CNF encoding. We leave such automated annotation for future work.

The class of symmetries that our framework captures are referred to here as complete multi-class global symmetries. Informally, we associate each variable semantically with the "objects" it refers to (e.g, *load-*P_i*-onto-*T_j*-at-*L_k*-time-*t in a logistics planning problem may refer to package P_i, truck T_j, location L_k, and time point t). When several packages are annotated as symmetric (because, for instance, they have the same source and destination), the variables that refer to these packages are also inferred to have a symmetry (depending also on the other objects these variables refer to). While such complete multi-class global symmetries may seem somewhat restrictive at first glance, our empirical evaluation shows that working only with such kinds of symmetry can often be more beneficial than trying to exhaustively identify and exploit all types of symmetry.

Our experimental results demonstrate that `SymChaff`, built on top of the popular SAT solver `zChaff` [40], is able to achieve empirical (and sometimes provable) exponential speed-up on many unsatisfiable and satisfiable formulas from a variety of problem domains from theory and practice. The highlight of the empirical evaluation are the planning formulas, on which tremendous performance gains are obtained by simply annotating their standard PDDL description with semantic tags signifying object symmetry.

The rest of the article is organized as follows. We first give a brief overview of the main idea and discuss related work on symmetries in SAT and CSPs. Section 2 provides the necessary background in DPLL-based solvers, pseudo-Boolean constraints, symmetry in our context, and many-sorted first order logic. Section 3 defines a new framework of complete multi-class symmetry, and describes in detail our representation and the key features of `SymChaff`. Section 4 discusses the experimental setup with a set of symmetric problems from both theory and applications. It reports experimental results and also gives an example of how symmetry information needed by `SymChaff` can be automatically derived from straightforward symmetry annotation in planning problem specifications. Finally, Section 5 summarizes the work and suggests future directions.

1.1 Main Idea

Our strategy involves a new technique for representing and dynamically maintaining symmetry information for DPLL-based satisfiability solvers. Our framework as presented applies to both CNF and pseudo-Boolean formulations (i.e., inequalities over Boolean variables) of problems. However, the current implementation of SymChaff supports only CNF.

We focus on symmetries that are present in the initial formula (as opposed to symmetries that arise dynamically after certain variable values have been fixed). Many symmetric problems of interest have a concise description in what is called many-sorted first order logic constraints. Although it sounds complex, this description can be easily specified by the problem designer and almost as easily inferred automatically. We use one specific syntax in our .sym file representation, which is an additional input to SymChaff besides the usual CNF formula. SymChaff uses symmetry sets that are created from the "sorts" of universally quantified variables in the first order logic constraints. (Sorts in logic are like "types" in programming languages.) These symmetry sets are used to partition variables into classes and to maintain and utilize symmetry information dynamically.

Two key features of SymChaff are multiway branching and symmetric learning. While it is natural to choose a variable and branch two ways by setting it to TRUE and FALSE, this is not necessarily the best option when k variables, $x_1, x_2, \ldots, x_k$, are known to be arbitrarily interchangeable. Specifically, it only matters *how many* of the x_i are set to FALSE at any time, and not which exact ones. The same applies to more complex symmetries where multiple classes of variables *simultaneously* depend on an index set $I = \{1, 2, \ldots, k\}$ and can be arbitrarily interchanged in parallel within their respective classes. We formalize this as a k-complete multi-class symmetry with index set I, and handle it using a $(k+1)$-*way branch-point* based on I. This multi-way branching maintains completeness of the search and shrinks the search space by as much as roughly 2^k. The index sets are implicitly determined from the many-sorted first order logic representation of the problem at hand. We extend the standard notions of conflicts and clause learning to the multiway branching framework, introducing *symmetric learning*. The idea is to efficiently compute a small set of variables which, when fixed to values in a certain manner, falsify *all* branches of a multiway branch-point. In practice, we found that invoking symmetric learning was essential to fully reap the benefits of multiway branching.

SymChaff integrates seamlessly with most of the other standard features of modern SAT solvers, extending them in the context of symmetry wherever necessary. These include fast unit propagation, good restart strategy, effective constraint database management, etc.

1.2 Related Work

One of the most successful techniques for handling symmetry in both SAT and general CSPs originates from the work of Puget [46], who showed that symmetries can be *broken* by adding one lexicographic ordering constraint per symmetry. Crawford et al. [10] showed how this can be done by adding a set of simple "lex-constraints" or *symmetry breaking predicates* (SBPs) to the input specification to weed out all but the lexically-first solutions. The idea is to identify the group of permutations of variables that keep the CNF formula unchanged. For each such permutation π, clauses are added so that for every satisfying assignment σ for the original problem whose permutation $\pi(\sigma)$ is also a satisfying assignment, only the lexically-first of σ and $\pi(\sigma)$ satisfies the added clauses. In the context of CSPs, there has been a lot of work in the area of SBPs. Petrie and Smith [44] extended the idea to

value symmetries, Puget [48] applied it to products of variable and value symmetries, and Walsh [57] generalized the concept to symmetries acting simultaneously on variables and values, on set variables, and other forms. Puget [47] has recently proposed a technique for creating dynamic lex-constraints, with the goal of minimizing adverse interaction with the variable ordering used in the search tree.

In the context of SAT, value symmetries naturally manifest themselves as variable symmetries,[2] and work on SBPs has taken a different path. Tools such as `Shatter` by Aloul et al. [1, 4] improve upon the basic SBP technique by using lex-constraints whose size is only linear in the number of variables rather than quadratic. Further, they use graph isomorphism detectors like `Saucy` by Darga et al. [11] to generate symmetry breaking predicates only for the generators of the algebraic groups of symmetry. While saving SBPs only for the generators of the symmetry groups is often relatively fast, the problem of computing graph isomorphism through a tool like `Saucy`, however, is not known to have any polynomial time algorithms, and is conjectured to be strictly between the complexity classes P and NP [cf. 32]. Hence, one must resort to heuristic or approximate solutions. Further, while there are formulas for which few SBPs suffice, the number of SBPs one needs to add in order to break *all* symmetries can be exponential. This is typically handled in practice by discarding "large" symmetries, i.e., those involving too many variables with respect to a fixed threshold. This may, however, result in a much slower SAT-based solution method as indicated by our experiments on clique coloring and logistics problems.

The work of Motter et al. [41, 42] employs a technique called compressed breadth-first search, using so-called zero-suppressed binary decision diagrams (ZDDs). This method, implemented as the solver `Cassatt`, is able to solve the pigeonhole formulas in polynomial time, specifically, in $\Theta(n^4)$ time for PHP^n_{n-1}. The solver also shows promising performance in a few other domains.

A very different and indirect approach for addressing symmetry is embodied in SAT solvers such as `PBS` by Aloul et al. [2], `pbChaff` by Dixon et al. [15], and `Galena` by Chai and Kuehlmann [8], which utilize non-CNF formulations known as pseudo-Boolean inequalities. Their logic reasoning is based on what is called the Cutting Planes proof system which, as shown by Cook et al. [9], is strictly stronger than resolution on which DPLL type CNF solvers are based. Since this more powerful proof system is difficult to implement in its full generality, pseudo-Boolean solvers often implement only a subset of it, typically learning only CNF clauses or restricted pseudo-Boolean constraints upon a conflict. Pseudo-Boolean solvers may lead to purely syntactic representational efficiency in cases where a single constraint such as $y_1 + y_2 + \ldots + y_k \leq 1$ is equivalent to $\binom{k}{2}$ binary clauses. More importantly, they are relevant to symmetry because they sometimes allow implicit encoding. For instance, the single constraint $x_1 + x_2 + \ldots + x_n \leq m$ over n variables captures the essence of the pigeonhole formula PHP^n_m over nm variables which is provably exponentially hard to solve using resolution-based methods without symmetry considerations. This implicit representation, however, is not suitable in certain applications such as clique coloring and planning that we discuss. In fact, for unsatisfiable clique coloring instances, even pseudo-Boolean solvers provably require exponential time.

One could conceivably keep the CNF input unchanged but modify the solver to detect and handle symmetries during the search phase as they occur. Although this approach is quite natural, its only implementation so far in a general purpose SAT solver appears to

[2] For example, a multi-valued CSP variable v with domain $\{1,2,\ldots,5\}$ is typically encoded in SAT using a set $\{x_{v,1}, x_{v,2}, \ldots, x_{v,5}\}$ of five Boolean variables; a value symmetry for v between values $\{2,3,5\}$ then translates to the variable symmetry between $x_{v,2}, x_{v,3}$, and $x_{v,5}$.

be sEqSatz by Li et al. [37], whose technique seems somewhat specific to the problems considered and shows limited improvement upon zChaff itself (without symmetry considerations). Symmetry handling during search has been explored with mixed results in the CSP domain using frameworks like SBDD and SBDS [e.g. 17, 19, 23, 24]. Related work in SAT has been done in the specific areas of automatic test pattern generation by Marques-Silva and Sakallah [39] and SAT-based model checking by Shtrichman [53]. In both cases, the solver utilizes global information obtained at a stage to make subsequent stages faster.

In other domain-specific work on symmetries in SAT, Fox and Long [20] presented a framework for planning problems that is very similar to ours in essence. However, their work has two limitations. The obvious one is that they provide a planner and not a general purpose reasoning engine. The second is that unlike typical SAT-based planners, their approach does not guarantee plans of optimal length when multiple (non-conflicting) actions are allowed to be performed at each time step. This issue does not arise in our approach.

Finally, Dixon et al. [14] give a generic method of representing and dynamically maintaining symmetry in SAT solvers using algebraic techniques that guarantee polynomial size unsatisfiability proofs of many difficult formulas. The strength of their work lies in a strong group theoretic foundation and comprehensiveness in handling all possible symmetries. The computations involving group operations that underlie their current implementation are, however, often quite expensive, scaling as a high degree polynomial.

2 Preliminaries

We begin with a brief review of the basic concepts we will need. A propositional formula in conjunctive normal form (CNF) is a conjunction (AND) of clauses, where each clause is a disjunction (OR) of literals and a literal is a propositional (Boolean) variable or its negation. A pseudo-Boolean formula is a conjunction of pseudo-Boolean constraints, where each pseudo-Boolean constraint is a weighted inequality over propositional variables with (typically) integer coefficients. A clause is called "unit" if all but one of its literals are set to FALSE; the remaining literal must be set to TRUE to satisfy the clause. Similarly, a pseudo-Boolean constraint is called "unit" if variables have been set in such a way that all its unset literals must be set to TRUE to satisfy the constraint. Finally, unit propagation is a technique common to SAT and pseudo-Boolean solvers that recursively simplifies the formula by appropriately setting unset variables in unit constraints so as to immediately satisfy them.

2.1 DPLL-based SAT Solvers

The approach we present in this article is applicable to all DPLL-based systematic SAT solvers designed for CNF as well as pseudo-Boolean constraints. At each step these solvers use some heuristic to select a literal to branch on (a "decision"). This literal is set to TRUE at the current decision level and the formula is simplified using unit propagation. If there is a conflict at this point, i.e., a variable is implied to be both TRUE and FALSE, the branch is declared as a failure and, typically, a conflict clause is learned which prevents the solver from unnecessarily exploring similar unsatisfiable branches in subsequent steps. At this point the solver backtracks and flips the assignment of the decision literal to FALSE. If on the other hand there is no conflict, the solver proceeds by branching on another literal. If all variables are set without a conflict, one has obtained a satisfying assignment and the search terminates

successfully. On the other hand, when all branches have been unsuccessfully explored, the formula is declared unsatisfiable.

This process is sound and complete, i.e., it finds a satisfying assignment if there exists one, and reports unsatisfiability otherwise. Various other features and optimizations, such as random restarts, watched literals, and conflict-directed backjumping, are added to this basic structure to increase efficiency.

2.2 Constraint Satisfaction Problems and Symmetry

A constraint satisfaction problem (CSP) is a collection of constraints over a set $V = \{x_1, x_2, \ldots, x_n\}$ of n variables, where each variable x_i takes values in its domain D_i. A solution to a CSP is an assignment of values to variables such that all constraints are satisfied. Although the following notions are generic, our focus in this work will be on CNF and pseudo-Boolean constraints over propositional variables. In other words, our constraints will be either clauses or pseudo-Boolean inequalities, and our variables will all have domain $\{\text{TRUE}, \text{FALSE}\}$.

Symmetry may exist in various forms in a CSP. Our focus will be on *variable symmetry*, and we define it in terms of permutations of variables that preserve certain properties of the CSP.[3] For a positive integer q, we will use $[q]$ to denote the set $\{1, 2, \ldots, q\}$.

Let σ be a permutation of $[n]$. With abuse of notation, extend σ in a natural manner to elements and subsets of V (i.e., to variables and collections of variables) by defining the following: for $x_i \in V$, define $\sigma(x_i) = x_{\sigma(i)}$; for $V' \subseteq V$, define $\sigma(V') = \{\sigma(x) \mid x \in V'\}$. For a p-ary constraint $C(x_{i_1}, x_{i_2}, \ldots, x_{i_p})$ over V, let $\sigma(C)$ denote the constraint $C(x_{\sigma(i_1)}, x_{\sigma(i_2)}, \ldots, x_{\sigma(i_p)})$. For a CSP Γ, define $\sigma(\Gamma)$ to be the new CSP consisting of the constraints $\{\sigma(C) \mid C \in \Gamma\}$.

Definition 1 A permutation σ of the variables of a CSP Γ is a *global variable symmetry* of Γ if $\sigma(\Gamma) = \Gamma$.

Definition 2 Let Γ be a CSP on the variable set V. Then $V' \subseteq V, |V'| = k$, induces a *k-complete global variable symmetry* of Γ if *every* permutation σ of V satisfying $\sigma(V') = V'$ and $\sigma(x) = x$ for $x \notin V'$ is a global variable symmetry of Γ.

In other words, the k variables in V' can be arbitrarily interchanged without changing the original problem.[4] Such symmetries exist in simple problems such as the pigeonhole principle in Example 1, where all pigeons (and holes) are symmetric. As mentioned earlier, these symmetries can be detected and exploited efficiently using various known techniques such as cardinality constraints [2, 8, 15].

We will extend this basic notion of symmetry to a richer one in Section 3.1. For brevity, we will refer to a global variable symmetry as simply a *symmetry*.

[3] We do not consider *pure value symmetry* because all our domains are Boolean; if both values of a Boolean variable are symmetric, then it is a true "don't care" and can be removed from the problem. A *literal symmetry* or *variable-value symmetry* mapping literals to literals could in principle be more generic than variable symmetries mapping variables to variables. Our framework, however, works with high level "objects" in the problem rather than Boolean variables, and thus leads naturally to variable symmetry only.

[4] In group theoretic terms, this forms a full group of symmetry over V'.

2.3 Many-Sorted First Order Logic

It will be convenient (though not necessary) to view symmetry from the perspective of many-sorted first order logic, which we briefly review here. In first order logic, one can express universally and existentially quantified logical statements about variables and constants that range over a certain domain with some inherent structure. For instance, the domain could be the finite set $[n]$ with the successor relationship of the first n natural numbers as its structure, and a (false) universally quantified logical statement over it could be that every element in the domain has a successor.

In *many-sorted* logic, the domain of variables and constants may be divided up into various types or "sorts" of elements that are quantified over independently. In other words, many-sorted first order logic extends first order logic with type information. The reader is referred to standard texts such as by Gallier [22] for further details. We remark here that many-sorted first order logic is known to be exactly as expressive as first order logic itself. In this sense, sorts or types add convenience but not power to the logic.

As an example, consider again the pigeonhole principle where the domain consists of a set P of pigeons and a set H of holes. The problem can be stated as the succinct 2-sorted first order formula $[\forall(p \in P)\ \exists(h \in H)\ .\ X(p,h)] \wedge [\forall(h \in H,\ p_1 \in P,\ p_2 \in P)\ .\ (p_1 \neq p_2 \rightarrow (\neg X(p_1,h) \vee \neg X(p_2,h)))]$, where $X(p,h)$ is the predicate "pigeon p maps to hole h." We can alternatively write this 2-sorted first order logic formula even more concisely as $[\forall^P i\ \exists^H j\ .\ x_{i,j}] \wedge [\forall^H j\ \forall^P i,k\ .\ (i \neq k \rightarrow (\neg x_{i,j} \vee \neg x_{k,j}))]$

Recall on the other hand from Example 1 that the CNF formulation of same problem requires $|P| + |H|\binom{|P|}{2}$ clauses. As we will see shortly, the sort-based quantified representation of problems lies at the heart of our approach by providing us the base "symmetry sets" to start with.

3 Symmetry Framework and SymChaff

We now describe our new symmetry framework in a generic way, briefly referring to specific implementation aspects of SymChaff as appropriate.

The motivation and description of our techniques can be best understood with a few concrete examples in mind. We use three relatively simple logistics planning problems depicted in Figure 1. In all three of these problems, there are k trucks $T_1, T_2, \ldots, T_k$ initially at a location L_{TB} (truckbase). There are several locations as well as a number of packages. Each package is initially at a certain location and needs to be transported to a certain destination location. Actions that can be taken at any step include driving a truck from one location to another, and loading or unloading multiple boxes (in parallel) onto or from a truck. The task is to find a minimum length plan such that all boxes arrive at their destined locations and all trucks return to L_{TB}. Actions that do not conflict in their pre- or post-conditions can be taken in parallel.

Let $s(i) = (i \bmod n) + 1$ denote the cyclic successor of i in $[n]$.

Example 2 (PlanningA) Let $k = \lceil 3n/4 \rceil$. For $1 \leq i \leq n$, there is a location L_i that has two packages $P_{i,1}$ and $P_{i,2}$. The goal is to deliver package $P_{i,1}$ to location $L_{s(i)}$ and package $P_{i,2}$ to location $L_{s(s(i))}$.

The shortest plan for this problem is of length 7 for any n. The idea behind the plan is to use 3 trucks to handle 4 locations. E.g., truck T_1 transports $P_{1,1}$, $P_{1,2}$, and $P_{2,1}$, truck T_2 transports $P_{3,1}$, $P_{3,2}$, and $P_{4,1}$, and truck T_3 transports $P_{2,2}$ and $P_{4,2}$. The 7 steps for T_1 involve

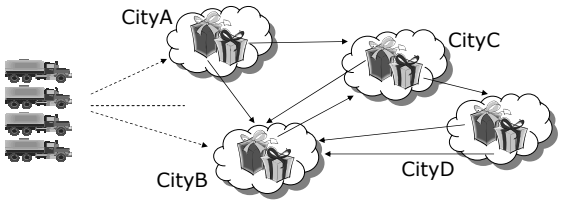

Fig. 1 The setup for logistic planning examples

(i) driving to L_1, (ii) loading the two boxes there, (iii) driving to L_2, (iv) unloading $P_{1,1}$ and loading $P_{2,1}$, (v) driving to L_3, (vi) unloading the two boxes it is carrying, and (vii) driving back to L_{TB}.

Example 3 (`PlanningB`) Let $k = \lceil n/2 \rceil$. For $1 \leq i \leq n$, there are 5 packages at location L_i that are all destined for location $L_{s(i)}$. This problem has more symmetries than `PlanningA` because all packages initially at the same location are symmetric.

The shortest plan for this problem is of length 7 and assigns one truck to two consecutive locations. E.g., the 7 steps for truck T_1 include (i) driving to L_1, (ii) loading all boxes there, (iii) driving to L_2, (iv) unloading the boxes it is carrying and loading all boxes originally present at L_2, (v) driving to L_2, (vi) unloading all boxes it is carrying, and (vii) driving back to L_{TB}.

Example 4 (`PlanningC`) Let $k = n$. For $1 \leq i \leq n$, there are locations $L_i^{\text{src}}, L_i^{\text{dest}}$ and packages $P_{i,1}, P_{i,2}$. Both these packages are initially at location L_i^{src} and must be delivered to location L_i^{dest}. Here not only the two packages at each source location are symmetric but all n tuples $(L_i^{\text{src}}, L_i^{\text{dest}}, P_{i,1}, P_{i,2})$ are symmetric as well.

It is easily seen that the shortest plan for this problem is of length 5 and assigns one truck to each source-destination pair. E.g., the 5 steps for T_1 involve (i) driving to L_1^{src}, (ii) loading the two boxes there, (iii) driving to L_1^{dest}, (iv) unloading the two boxes it is carrying, and (v) driving back to L_{TB}.

For a given plan length, such a planning problem can be converted into a CNF formula using tools such as `Blackbox` by Kautz and Selman [31] and then solved using standard SAT solvers. The variables in this formula are of the form *load-$P_{i,1}$-onto-T_j-at-L_k-time-t*, etc. We omit the details [cf. 30].

3.1 k-complete m-class Symmetries

Consider a CSP Γ over a set $V = \{x_1, x_2, \ldots, x_n\}$ of variables as before. We generalize the idea of complete symmetry for Γ to complete multi-class symmetry. Let $V_1, V_2, \ldots, V_m$ be disjoint subsets of V of cardinality k each. Let $V_0 = V \setminus \left(\bigcup_{i \in [m]} V_i\right)$. Order the variables in each $V_i, i \in [m]$, arbitrarily and let $y_i^j, j \in [k]$, denote the j^{th} variable of V_i.

Let σ be a permutation of the set $[k]$. Define $\bar{\sigma}$ to be the permutation of V induced by σ and $V_i, 0 \leq i \leq m$, as follows: $\bar{\sigma}(x) = x$ for $x \in V_0$ and $\bar{\sigma}(x) = y_i^{\sigma(j)}$ for $x = y_i^j \in V_i, i \in [m]$. In other words, $\bar{\sigma}$ maps variables in V_0 to themselves and applies σ in parallel to the indices of the variables in each class $V_i, i \in [m]$, simultaneously.

Definition 3 If $\bar{\sigma}$ is a global symmetry of Γ for *every* permutation σ of $[k]$ then the set $\{V_1, V_2, \ldots, V_m\}$ is a *k-complete m-class (global) symmetry* of Γ. The sets $V_i, i \in [m]$, are referred to as the *variable classes*. Variables in V_i are said to be *indexed by* the *symindex set* $[k]$.

Note that a k-complete 1-class symmetry is simply a k-complete symmetry. Complete multi-class symmetries correspond to the case where variables from multiple classes can be simultaneously and coherently changed in parallel without affecting the problem.[5] This happens naturally in many problem domains.

Example 5 Consider the logistics planning problem `PlanningA` (Example 2) for $n = 4$ converted into a unsatisfiable CNF formula corresponding to plan length 6. The problem has $k = 3$ trucks and is 3-complete m-class symmetric for appropriate m. The variable classes V_i of size 3 are indexed by the symindex set $\{1,2,3\}$ and correspond to sets of 3 variables that differ only in which truck they use. For example, variables *unload-*$P_{2,1}$*-from-*T_1*-at-*L_2*-time-*5, *unload-*$P_{2,1}$*-from-*T_2*-at-*L_2*-time-*5, and *unload-*$P_{2,1}$*-from-*T_3*-at-*L_2*-time-*5 comprise one variable class which is denoted by *unload-*$P_{2,1}$*-from-*$\mathbf{T_j}$*-at-*L_2*-time-*5. The many-sorted representation of the problem has one universally quantified sort for the trucks. The problem `PlanningA` remains unchanged, e.g., when T_1 and T_2 are swapped in all variable classes simultaneously.

In more complex scenarios, a variable class may be indexed by multiple symindex sets and be part of more than one complete multi-class symmetry. This will happen, for instance, in the `PlanningB` problem (Example 3) where variables *load-*$\mathbf{P_{2,a}}$*-onto-*$\mathbf{T_j}$*-at-*L_4*-time-*4 are indexed by two symindex sets, $a \in [5]$ and $j \in [3]$, each acting independent of the other. This problem has a universally quantified 2-sorted first order representation.

Alternatively, multiple object classes, even in the high level description, may be indexed by the same symindex set. This happens, for example, in the `PlanningC` problem (Example 4), where $L_i^{\text{src}}, L_i^{\text{dest}}, P_{i,1}$, and $P_{i,2}$ are all indexed by i. This results in symmetries involving an even higher number of variable classes indexed by the same symindex set than in the case of `PlanningA` type problems.

3.2 Symmetry Representation

`SymChaff` takes as input a CNF file in the standard DIMACS format [29] as well as a `.sym` symmetry file S that encodes the complete multi-class symmetries of the input formula. Lines in S that begin with `c` are treated as comments. S contains a header line `p sym nsi ncl nsv` declaring that it is a symmetry file with `nsi` symindex sets, `ncl` variable classes, and `nsv` symmetric variables.

Symmetry is represented in the input file S and maintained inside `SymChaff` in three phases. First, *symindex sets* are represented as consecutive, disjoint intervals of positive integers. In the `PlanningB` example for $n = 4$, the three trucks would be indexed by the set $[1 \,..\, 3]$ and the 5 packages at location $L_i, 1 \le i \le 4$, by symindex sets $[3+5(i-1)+1 \,..\, 3+5i]$, respectively. Here $[p \,..\, q]$ denotes the set $\{p, p+1, \ldots, q\}$. Second, one *variable class* is defined for each variable class V_i and associated with each symindex set that indexes variables in it. Finally, a *symindex map* is created that associates with each symmetric variable the variable class it belongs to and the indices in the symindex sets it is indexed by.

[5] In group theoretic terms, this corresponds to the product of several full symmetry groups.

```
c Symmetry file for php-004-003.cnf
c 4 pigeons, 3 holes, 12 symmetric variables
c 2 symindex sets, 1 varclass
c
p sym 12 2 1
c
c symindex sets
1 4 0
2 7 0
0
c varclasses
1 4 7 0
0
c
c symindex mappings
1 1 1 5 0
2 1 1 6 0
3 1 1 7 0
4 1 2 5 0
5 1 2 6 0
6 1 2 7 0
7 1 3 5 0
8 1 3 6 0
9 1 3 7 0
10 1 4 5 0
11 1 4 6 0
12 1 4 7 0
0
```

Fig. 2 A sample symmetry file, `php-004-003.sym`

For instance, variable *load*-$P_{2,4}$-*onto*-T_3-*at*-L_4-*time*-4 in problem `PlanningB` will be associated with the variable class *load*-$\mathbf{P_{2,a}}$-*onto*-$\mathbf{T_j}$-*at*-L_4-*time*-4 and with indices $j = 3$ and $a = 3+5(2-1)+4 = 12$. The symmetry input file S is a straightforward encoding of symindex sets, variable classes, and symindex map.

Example 6 As another example and as an illustration of the exact syntax of S, we give the actual symmetry input file for the pigeonhole problem PHP_3^4 in Figure 2. There are two symindex sets, one for the 4 pigeons and the other for the 3 holes. These correspond to the consecutive, disjoint intervals $[1 \,..\, 4]$ and $[5 \,..\, 7]$, respectively, and are associated with the right end-points of the intervals, 4 and 7. All 12 variables of the problem are symmetric to each other and thus belong to the only variable class for the problem (commented as "varclass" in the Figure). This variable class is indexed by the two symindex sets associated with the right end-points 4 and 7. Finally, the symindex map says, for example, that variable 5, which happens to correspond to the variable $x_{2,2}$ in PHP_3^4, belongs to the first (and only) variable class and is indexed by the index 2 from the first symindex set and the index 6 from the second symindex set associated with its variable class.

3.3 Multiway Index-based Branching

A distinctive feature of `SymChaff` is multiway symindex-based branching. Suppose at a certain stage the variable selection heuristic suggests that we branch by setting variable x to FALSE. `SymChaff` checks to see whether x has any complete multi-class symmetry left in

the current stage. (Note that symmetry in our framework reduces as variables are assigned truth values.) x, of course, may not be symmetric at all to start with. If x doesn't have any symmetry, SymChaff proceeds with the usual DPLL style 2-way branching by setting x now to FALSE and later to TRUE. If it does have symmetry, SymChaff arbitrarily chooses a symindex set $I, |I| = k \geq 2$, that indexes x and creates a $(k+1)$-way branch-point based on I. Let $x_1, x_2, \ldots, x_k$ be the variables indexed by I in the variable class V' to which x belongs ($x = x_{i'}$ for some i'). For $0 \leq i \leq k$, the i^{th} branch of this branch-point sets $x_1, \ldots, x_i$ to FALSE and $x_{i+1}, \ldots, x_k$ to TRUE. The idea behind this multiway branching is that it only matters *how many* of the x_i are set to FALSE and not which exact ones. This reduces the search for a satisfying assignment from up to 2^k different partial assignments of $x_1, \ldots, x_k$ to only $k+1$ different ones. This clearly maintains completeness of the search and is the key to the good performance of SymChaff.

When one branches and sets variables, the symindex sets must be updated to reflect this change. When proceeding along the i^{th} branch in the above setting, two kinds of *symindex splits* happen. First, if x also happens to be indexed by an index j in another symindex set $J = [a \,..\, b] \neq I$, we must split J into up to three symindex sets given by the intervals $[a \,..\, j-1]$, $[j \,..\, j]$, and $[j+1 \,..\, b]$ because j's symmetry has been destroyed by this value assignment to x. To reduce the number of splits, SymChaff replaces x with another variable in its variable class V' for which $j = a$ and thus the split divides J into two new symindex sets only, $[a \,..\, a]$ and $[a+1 \,..\, b]$. This first kind of split is done once for the multiway branch-point for x based on I, and is independent of the value of i. The second kind of split is, in fact, much simpler: it divides $I = [c \,..\, d]$ into up to two symindex sets given by $[c \,..\, i]$ and $[i+1 \,..\, d]$. This captures the fact that both the first i and the last $k-i$ indices of I remain symmetric in the i^{th} branch of the multiway branch-point.

Note that it is possible for x to be indexed multiple times by the same symindex set I, such as when $x_{i,i'}$ denotes an edge in a graph with nodes forming the symindex set I. In this case, $x_{i,i}$ must be treated differently than $x_{i,i'}, i \neq i'$. The easiest way to achieve this is to start by splitting the symindex set I because of the first index i of $x_{i,i'}$, and then further split I because of the second index i' of $x_{i,i'}$. An example domain where such splitting is needed is the clique coloring domain, to be discussed in experimental section.

Symindex sets that are split while branching must be restored when a backtrack happens. When a backtrack moves the search from the i^{th} branch of a multiway branching step to the $i+1^{\text{st}}$ branch, SymChaff deletes the symindex set split of the second type created for the i^{th} branch and creates a new one for the $i+1^{\text{st}}$ branch. When all $k+1$ branches are finished, SymChaff also deletes the split of the first type created for this multiway branch-point and backtracks.

In terms of implementation, note that while the variable classes and the symindex map remain static, the symindex sets change dynamically as SymChaff proceeds assigning values to variables. In fact, when sufficiently many variables have been assigned truth values, all complete multi-class symmetries will be destroyed. For efficient access and manipulation, SymChaff stores variable classes in a vector data structure from the Standard Template Library (STL) of C++, the symindex map as a hash table, and symindex sets together as a multiset containing only the right end-points of the consecutive, disjoint intervals corresponding to the symindex sets. A symindex set split is achieved by adding the corresponding new right end-point to the multiset, and symindex sets are combined when backtracking by deleting the end-point.

```
c Symmetry order file for (f)clqcolor-xx-xx-xx.cnf
c 3 varclasses, 3 order sets
c
p ord 3 3
c
c varclass index order
1 1 1 0
2 2 1 0
3 1 2 0
0
c varclass order
3 0
2 0
1 0
0
```

Fig. 3 A sample symmetry ordering file, `clqcolor.ord`

3.4 Symmetric Learning

We extend the notion of conflict-directed clause learning to our symmetry framework. When all branches of a $(k+1)$-way symmetric branch-point b have been explored, `SymChaff` learns a *symconflict clause* C such that when all literals of C are set to FALSE, unit propagation falsifies *every* branch of b. This process clearly maintains soundness of the search. The symconflict clause is learned even for 2-way branch-points and is computed as follows.

Suppose a k-way branch-point b starts at decision level d. If the i^{th} branch of b leads to a conflict without any further branching, two things happen. First, `SymChaff` learns a standard conflict clause following the FirstUIP strategy of `zChaff` [59]. Second, it stores in a set S_b associated with b the decision literals at levels higher than d that are involved in the conflict. On the other hand, if the i^{th} branch of b develops further into another branch-point b', `SymChaff` stores in S_b those literals of the symconflict clause recursively learned for b' that have decision level higher than d. When all branches at b have been explored, the symconflict clause learned for b is $\bigvee_{\ell \in S_b} \neg \ell$.

3.5 Static Ordering of Symmetry Classes and Indices

It is well known that the variable order chosen for branching in any DPLL-based SAT solver has a tremendous impact on its efficiency. A good order for selecting variable classes and symindex sets for multiway branching can similarly often boost the performance of `SymChaff` as well. While we leave dynamic strategies for selecting variable classes and symindex sets as future work, `SymChaff` does support static ordering through a very simple and optional `.ord` order file given as additional input. This file specifies an ordering of variable classes as an initial guide to the VSIDS (variable state independent decaying sum) variable selection heuristic of `zChaff` [40], treating asymmetric variables in a class of their own. Further, for each variable class indexed by multiple symindex sets, it allows one to specify an order of priority on symindex sets.

The exact `.ord` file structure is shown as an example in Figure 3. Here, as in many cases, a good ordering is an attribute of the problem domain rather than of specific problem instances — in this case, the clique coloring domain, to be discussed shortly (Section 4.1). The file specifies that the variables from variable class 3 should be used for symmetric branching before variables from class 2, which in turn should be used before variables from

class 1. Variable class 4 implicitly denotes the set of all asymmetric variables, and has the least priority because it does not appear in the `.ord` file. Further, when branching on a variable in, say, class 2, the specified ordering dictates that index 2 should be used for splitting before index 1 is tried.

3.6 Integration of Standard SAT Solver Features

The efficiency of state-of-the-art SAT and pseudo-Boolean solvers relies heavily on various features that have been developed, analyzed, and tested over the last decade. SymChaff integrates well with most of these features, either using them without any change or extending them in the context of multiway branching and symmetric learning. The only significant and relatively new feature that neither SymChaff nor the version of zChaff on which it is based currently support is assignment stack shrinking based on conflict clauses which was introduced by Nadel [43] in the solver Jerusat.

For completeness, we make a digression to give a flavor of how assignment stack shrinking works. When a conflict occurs because a clause C' is violated and the resulting conflict clause C to be learned exceeds a certain threshold length, the solver backtracks to almost the highest decision level of the literals in C. It then starts assigning to FALSE the unassigned literals of the violated clause C' until a new conflict is encountered, which is expected to result in a smaller and more pertinent conflict clause to be learned.

Returning to SymChaff, it supports fast unit propagation using watched literals, good restart strategies, effective constraint database management, and smart branching heuristics in a very natural way. In particular, it uses zChaff's watched literals scheme for unit propagation, deterministic and randomized restart strategies, and clause deletion mechanisms without any modification, and thus gains by their use as any other SAT solver would. While performing multiway branching for classes of variables that are known to be symmetric, SymChaff starts every new multiway branch-point based on the variable that would have been chosen by VSIDS variable and value selection heuristic of zChaff, thereby retaining many advantages that effective heuristics like VSIDS have to offer.

Conflict clause learning is extended to symmetric learning as described earlier. Conflict-directed backjumping in the traditional context allows a solver to backtrack directly to a decision level d if variables at levels d or higher are the only ones involved in the conflicts in both branches at a branch-point other than the branch variable itself. SymChaff extends this to multiway branching by computing this level d for all branches at a multiway branch-point by looking at the symconflict clause for that branch-point, discarding all intermediate branch-points and their respective partial symconflict clauses, backtracking to level d, and updating the symindex sets.

While conflict-directed backjumping is always beneficial, fast backjumping may not be so. This latter technique, relevant mostly to the FirstUIP learning scheme of zChaff, allows a solver to jump directly to a higher decision level d when even one branch leads to a conflict involving variables at levels d or higher only (in addition to the variable at the current branch-point). This discards intermediate decisions which may actually be relevant and in the worst case will be made again unchanged after fast backjumping. SymChaff provides this feature as an option which turns out to be helpful in certain domains and detrimental in others. To maintain consistency of symconflict clauses learned later, the level d' to backjump to is computed as the maximum of the level d as above and the maximum decision level $\bar{d}$ of any variable in the partial symconflict clause associated with the current multiway branch-point.

4 Benchmarks and Empirical Evaluation

SymChaff is implemented on top of zChaff version 2003.11.04. The input to SymChaff is a .cnf formula file in the standard DIMACS format, a .sym symmetry file, and an optional .ord static symmetry order file. It uses the default parameters of zChaff. The program was compiled using g++ 3.3.3 for RedHat Linux 3.3.3-7. Experiments were conducted on a cluster of 36 machines running Linux 2.6.11 with four 2.8 GHz Intel Xeon processors on each machine, each with 1 GB memory and 512 KB cache.

Tables 1, 2, 3, and 4 report statistics of and results for several parameterizations of two problems from proof complexity theory, three planning problems, and a routing problem from design automation. These problems are discussed below. Satisfiable instances of some of these problems were easy for all solvers considered and are therefore omitted from the tables. Except for the planning problems for which automatic "tags" were used (described later), the .sym symmetry files were automatically generated by a straightforward modification to the scripts used to create the .cnf files from the problem descriptions. For all instances, the time required to generate the .sym file was negligible compared to the time required to generate the corresponding .cnf file, and is therefore not reported. The .sym files were in addition extremely small compared to the corresponding .cnf files.

The SAT solvers used in the comparative study were SymChaff, zChaff version 2003.11.04 [40], and March-eq-100 [28]. Symmetry breaking predicates (SBPs) were generated using Shatter version 0.3 [1], which uses the graph isomorphism tool Saucy [11]. Since SymChaff is implemented on top of zChaff version 2003.11.04, we used the same version of zChaff as well as the then state-of-the-art version of March-eq for a fair comparison. Note that zChaff won the best solver award for industrial benchmarks in the SAT 2004 competition [35] while March-eq-100 won the corresponding award for handmade benchmarks.

Tables 1 and 2 provide details of all unsatisfiable and satisfiable formulas, respectively, used for experimentation. Specifically, the data reported in these tables includes:

- the number of high level objects in the problem, typically ranging from around 10 to 200;
- the number of variables and clauses in the CNF formula, typically ranging from a few thousand to over two million clauses;
- various symmetry related numbers, including the number of symmetric objects considered by SymChaff, the number of symindex sets (between 1 and 3), the number of variable classes (from 1 up to around 1,500), and the number of variables mapped to these classes (often a large fraction of the CNF variables); and
- the number of clauses added by Shatter in order to break symmetries, which is typically quite large, ranging from a few thousand to nearly 200,000.

Tables 3 and 4 compare the performance of SymChaff with zChaff and March-eq without SBPs, and with zChaff after SBPs were added. For the last of these, we report the run-times separately for generating SBPs (denoted here as the time used by Shatter, which includes the time to run the graph isomorphism tool Saucy) and for solving the resulting formula after SBPs are added, as well as the sum of these two. We summarize the results here before individually describing the benchmarks used and the solver behavior observed. A domain-specific comparison with other approaches like ZBDDs, algebraic group theoretic techniques, and pseudo-Boolean solvers is also left for the detailed discussion below.

In summary, SymChaff outperformed both zChaff and March-eq without SBPs in all but excessively easy instances by several orders of magnitude. SymChaff solved each of

the formulas in a few seconds to a few minutes, while zChaff and March-eq often timed out after 6 hours. For the SBP approach, even generating SBPs from the input CNF formula was typically quite slow compared to a complete solution by SymChaff. For example, log-pair-11t5 required more than 1.5 hours to even generate SBPs but was solved by SymChaff in 16 seconds. The effect of adding SBPs before feeding the problem to zChaff was mixed, helping to various extents in some instances and hurting in others. For example, zChaff was able to solve most php and chnl instances in a few minutes after adding SBPs, but actually became slower on unsatisfiable log-rotate and log-pair formulas after adding SBPs, apparently because the overhead of extra variables and clauses encoding SBPs over-shadowed the search space reduction provided by them. In either case, the result was never any better than using SymChaff, and orders of magnitude worse when the cumulative time of Shatter and zChaff was considered.

4.1 Problems from Proof Complexity

Pigeonhole Principle. php-*n*-*m* is the classic pigeonhole problem described in Example 1 for n pigeons and m holes. The corresponding formulas are satisfiable if and only if $n \leq m$. They are known to be exponentially hard for resolution [27, 49] but easy when the symmetry rule is added [34]. Symmetry breaking predicates can therefore be used for fast CNF SAT solutions. The price to pay is symmetry detection in the CNF formula, i.e., generation of symmetry breaking predicates using graph isomorphism tools. We found this process to be significantly costly in terms of the overall runtime.

pbChaff and Galena, on the other hand, use an explicit pseudo-Boolean encoding and rely on learning good pseudo-Boolean conflict constraints. They do overcome the drawbacks of the symmetry breaking predicates technique but are nonetheless slower than SymChaff.

SymChaff uses two symindex sets corresponding to the pigeons and the holes, and one variable class containing all the variables. It solves this problem with exactly $m-1$ branch-points and $2m-3$ actively explored branches. These branch-points are all based on the symindex set for holes, and the two branches explored per hole (except for some savings when nearly all variables have been assigned values) correspond to assigning pigeon $i, 1 \leq i \leq m$, to either only hole i or to no hole at all. When the assignment of pigeon i to only hole i fails, symmetric learning immediately lets the solver deduce that assigning pigeon i to more than one hole will also fail, so that the remaining $m-1$ branches of this branch-point result in an immediate backtrack. The time spent at each of the $2m-3$ branches is linear in the size of the formula in the worst case, although significantly better in practice due to the efficient data structures implemented in SAT solvers which obviate the need to visit all clauses at each branch.

This simple analysis contrasts well with one of the fastest current techniques for this problem (other than the implicit pseudo-Boolean encoding) by Motter and Markov [41] which is based on ZBDDs and requires a fairly involved analysis to prove that it runs in time $\Theta(m^4)$ [42].

Clique Coloring Principle. The formula clqcolor-*n*-*m*-*k* encodes the clique coloring problem whose solution is a set of edges that form an undirected graph G over n nodes such that two conditions hold: G contains a clique of size m and G can be colored using k colors so that no two adjacent nodes get the same color. The formula is satisfiable if and only if $m \leq n$ and $m \leq k$.

As mentioned earlier, this is an example of a domain where some variables are indexed twice by the same symindex set. Specifically, the edge variables, $e_{i,j}$, are indexed twice by the symindex set corresponding to the nodes of G. While the nodes of G are indeed all interchangeable, one must be careful with the encoding, so as to allow a clear partition of the variables into n kinds. To achieve this, the encoding we use has variables $e_{i,j}$ only when $i \neq j$; we do not create variables $e_{i,i}$, which are unnecessary and would not be interchangeable with $e_{i,j}$ for $i \neq j$.

At first glance, this problem might appear to be a simple generalization of the pigeonhole problem. However, it evades fast solutions using SAT as well as pseudo-Boolean techniques even when the clique part is encoded implicitly using pseudo-Boolean methods. This is not surprising, given that Pudlák [45] has shown this problem to be exponentially hard for the cutting planes proof system on which pseudo-Boolean solvers are based.

Our experiments indicate that not only finding symmetries from the corresponding CNF formulas is time consuming, `zChaff` is extremely slow even after taking symmetry breaking predicates into account. `SymChaff`, on the other hand, uses three symindex sets corresponding to nodes, membership in clique, and colors, and three variable classes corresponding to edge variables, clique variables, and coloring variables. It solves the problem with exactly $2k-2$ branch-points and $4k-5$ active branches, in a manner similar to the pigeonhole principle instances.

We note that this problem can also be solved in polynomial time using the group theoretic technique of Dixon et al. [14]. However, the group operations that underlie their implementation are polynomials of degree as high as 6 or 7, making the approach significantly slower in practice.

4.2 Problems from Applications

All planning problems were encoded using the high level STRIPS formulation of Planning Domain Description Language (PDDL) introduced by Fikes and Nilsson [18]. These were then converted into CNF formulas using the tool `Blackbox` version 4.1 by Kautz and Selman [31]. A PDDL description of a planning problem is a straightforward Lisp-style specification that declares the objects involved, their initial state, and their goal state. In addition to this instance-specific description, it also uses a domain-specific file that describes the available actions in terms of their preconditions and effects.

We modified `Blackbox` to generate symmetry information as well by using a very simple "tagged" or annotated PDDL description, where an original PDDL declaration such as

$$\begin{array}{l}\texttt{(:OBJECTS } T_1\ T_2\ T_3 \\ \qquad\quad L_1^{\mathrm{src}}\ L_2^{\mathrm{src}}\ L_1^{\mathrm{dest}}\ L_2^{\mathrm{dest}} \\ \qquad\quad P_{1,1}\ P_{2,1}\ P_{1,2}\ P_{2,2}\texttt{)}\end{array}$$

in the `PlanningC` example is replaced with

$$\begin{array}{lll}\texttt{(:OBJECTS } T_1\ T_2\ T_3 & \texttt{-} & \texttt{SYMTRUCKS} \\ \qquad L_1^{\mathrm{src}}\ L_2^{\mathrm{src}} & \texttt{-} & \texttt{SYMLOCS} \\ \qquad L_1^{\mathrm{dest}}\ L_2^{\mathrm{dest}} & \texttt{-} & \texttt{SYMLOCS} \\ \qquad P_{1,1}\ P_{2,1} & \texttt{-} & \texttt{SYMLOCS} \\ \qquad P_{1,2}\ P_{2,2} & \texttt{-} & \texttt{SYMLOCS)}\end{array}$$

Here `SYMTRUCKS` and `SYMLOCS` are the tags used to pass on the symmetry or equivalence information that there are two symmetry sets (for trucks and locations), that $\{T_1, T_2, T_3\}$ are interchangeable because of the first set, and that

```
(define (problem PlanningA-03)
  (:domain logistics-strips-sym)
  (:objects
    truck1
    truck2
    truck3 - SYMTRUCKS

    package1
    package2
    package3
    package4
    package5
    package6
    truckbase
    location1
    location2
    location3
    city1
  )
  (:init
    (TRUCK truck1)
    (TRUCK truck2)
    (TRUCK truck3)
    (OBJ package1)
    (OBJ package2)
    (OBJ package3)
    (OBJ package4)
    (OBJ package5)
    (OBJ package6)
continued ...
```

```
...continued
    (LOCATION truckbase)
    (LOCATION location1)
    (LOCATION location2)
    (LOCATION location3)
    (CITY city1)

    (at package1 location1)
    (at package2 location1)
    (at package3 location2)
    (at package4 location2)
    (at package5 location3)
    (at package6 location3)
    (at truck1 truckbase)
    (at truck2 truckbase)
    (at truck3 truckbase)
    (in-city truckbase city1)
    (in-city location1 city1)
    (in-city location2 city1)
    (in-city location3 city1)
  )
  (:goal (and
    (at package1 location2)
    (at package2 location3)
    (at package3 location3)
    (at package4 location1)
    (at package5 location1)
    (at package6 location2)
  ))
)
```

Fig. 4 The annotated PDDL file for `PlanningA` with $n = 3$

$\left\{(L_1^{\text{src}}, L_1^{\text{dest}}, P_{1,1}, P_{1,2}), (L_2^{\text{src}}, L_2^{\text{dest}}, P_{2,1}, P_{2,2})\right\}$ are interchangeable simultaneously as 4-tuples because of the second set. Other than this symmetry annotation, the PDDL description remains unchanged. An appropriate `.sym` file is automatically generated from this annotated PDDL using our modified PDDL-to-CNF converter within `Blackbox`.

Example 7 For concreteness, we give the actual PDDL specification for our `PlanningA` example with $n = 3$ locations and $k = \lceil 3n/4 \rceil = 3$ trucks in Figure 4. The underlined tag "`SYMTRUCKS`" is the only change to the usual specification of the problem needed to process symmetry information automatically.

We are now ready to present four application-oriented problems and discuss experimental results. Three of these are planning problems.

Gripper Planning. The problem `gripper-`n`-`t is our simplest planning example. It consists of $2n$ balls in a room that need to be moved to another room in t steps using a robot that has two grippers that it can use to pick up balls. The corresponding formulas are satisfiable if and only if $t \geq 4n-1$.

`SymChaff` uses two symindex sets corresponding to the balls and the grippers. The number of variable classes is relatively large and corresponds to each action that can be performed without taking into account the specific ball or gripper used. While `SymChaff` solves this problem easily in both unsatisfiable and satisfiable cases, the other two solvers perform

Table 1 Statistics of unsatisfiable formulas used in the experiments. Number of SBP clauses is unknown in the cases where `Shatter` timed out after 6 hours.

Problem & parameters		high level **objects**	`.cnf` file		`.sym` file				SBP clauses
			vars	clauses	sym objs	sets	classes	sym vars	
php	009-008	17	72	297	17	2	1	72	478
	013-012	25	156	949	25	2	1	156	1,102
	051-050	101	2,550	63,801	101	2	1	2,550	19,798
	091-090	181	8,190	368,641	181	2	1	8,190	64,438
	101-100	201	10,100	505,101	201	2	1	10,100	79,598
clqcolor	05-03-04	12	60	194	12	3	3	60	260
	12-07-08	27	324	4,526	27	3	3	324	1,764
	20-15-16	51	1,020	50,906	51	3	3	1,020	6,176
	30-18-21	69	2,070	196,911	69	3	3	2,070	12,573
	50-40-45	135	6,750	2,524,145	135	3	3	6,750	—
gripper	02t6	8	378	3,022	6	2	70	332	1,806
	04t14	12	1,966	33,602	10	2	198	1,840	10,846
	06t22	16	4,706	116,982	14	2	326	4,500	31,526
	10t38	24	13,642	543,518	22	2	582	13,276	94,750
log-rotate	06t6	25	3,435	167,822	5	1	285	2,910	24,526
	08t6	32	6,097	517,319	6	1	896	5,376	55,770
	09t6	36	8,364	910,423	7	1	1,077	7,539	102,688
	11t6	44	14,428	2,391,053	9	1	1,487	13,383	196,712
chnl	010-011	21	220	1,122	20	2	22	220	1,954
	011-020	31	440	4,220	22	2	40	440	4,034
	020-030	50	1,200	17,460	40	2	60	1,200	11,406
	050-100	150	10,000	495,200	100	2	200	10,000	98,206

poorly. Further, detecting symmetries from CNF using `Shatter` is not too difficult but does not speed up the solution process by any significant amount.

Table 2 Statistics of satisfiable formulas used in the experiments.

Problem & parameters		high level **objects**	`.cnf` file		`.sym` file				SBP clauses
			vars	clauses	sym objs	sets	classes	sym vars	
gripper	02t7	8	468	4,062	6	2	86	412	2,254
	04t15	12	2,128	36,710	10	2	214	1,992	11,750
	06t23	16	4,940	123,214	14	2	342	4,724	33,102
	10t39	24	14,020	559,166	22	2	598	13,644	97,382
log-rotate	06t7	25	4,907	315,406	5	1	832	4,160	35,254
	07t7	29	7,249	648,665	6	1	1,057	6,342	65,810
	08t7	32	8,927	1,014,741	6	1	1,308	7,848	81,978
	09t7	36	12,358	1,818,395	7	1	1,585	11,095	152,256
log-pair	05t5	27	3,147	166,562	10	2	158	3,120	30,728
	07t5	37	7,055	717,184	14	2	158	7,028	79,380
	09t5	47	13,347	2,220,462	18	2	158	13,320	162,164
	11t5	57	22,599	5,583,308	22	2	158	22,572	271,772

Table 3 Experimental results on unsatisfiable formulas with a 6 hour timeout.

Problem & parameters		SymChaff	zChaff	March-eq	Symmetry Breaking Predicates		
					Shatter	zChaff	SUM
php	009-008	**0.01**	0.22	1.55	0.07	0.10	0.17
	013-012	**0.01**	1017	—	0.09	0.01	0.10
	051-050	**0.24**	—	—	13.71	0.50	14.21
	091-090	**0.84**	—	—	245.51	3.47	248.98
	101-100	**1.20**	—	—	466.54	6.48	473.02
clqcolor	05-03-04	**0.02**	0.01	0.21	0.09	0.01	0.10
	12-07-08	**0.03**	—	—	5.09	4929.95	4935.04
	20-15-16	**0.26**	—	—	748.14	—	—
	30-18-21	**0.60**	—	—	20801.49	—	—
	50-40-45	**8.76**	—	—	—	—	—
gripper	02t6	**0.02**	0.03	0.07	0.20	0.04	0.24
	04t14	**0.84**	2820.37	—	3.23	983.40	986.63
	06t22	**3.37**	—	—	23.12	—	—
	10t38	**47.00**	—	—	193.85	—	—
log-rotate	06t6	**0.74**	1.47	21.55	8.21	0.93	9.14
	08t6	**2.03**	4.29	295.23	31.4	4.21	35.61
	09t6	**8.64**	15.67	3835.42	74.06	28.94	103.00
	11t6	**51.00**	12827.39	—	324.86	17968.33	18293.19
chnl	010-011	**0.04**	8.61	—	0.20	0.02	0.22
	011-020	**0.06**	135.26	—	0.28	0.03	0.31
	020-030	**0.05**	—	—	4.60	0.10	4.70
	050-100	**1.75**	—	—	810.82	1.81	812.63

Table 4 Experimental results on satisfiable formulas with a 6 hour timeout.

Problem & parameters		SymChaff	zChaff	March-eq	Symmetry Breaking Predicates		
					Shatter	zChaff	SUM
gripper	02t7	**0.02**	0.03	0.34	0.17	0.03	0.20
	04t15	**2.03**	1061.75	—	0.23	1411.32	1411.55
	06t23	**7.27**	—	—	19.03	—	—
	10t39	**92.07**	—	—	193.58	—	—
log-rotate	06t7	2.87	**2.09**	10.99	16.92	3.03	19.95
	07t7	7.64	**6.85**	27.05	55.66	46.96	102.62
	08t7	**9.13**	182.59	14805.30	62.78	358.89	421.67
	09t7	**139.65**	1284.33	814.86	186.55	1356.44	1542.99
log-pair	05t5	0.46	**0.38**	3.65	25.19	0.65	25.84
	07t5	**1.83**	1.87	80.30	243.57	3.05	246.62
	09t5	6.29	**6.23**	582.70	1373.50	14.57	1388.07
	11t5	**15.65**	18.05	1807.72	6070.47	34.36	6104.83

Logistics Planning log-rotate. The problem log-rotate-n-t is the logistics planning example PlanningA with n as the number of locations and t as the maximum plan length. As described earlier, it involves moving boxes in a cyclic rotation fashion between the locations. The formula is satisfiable if and only if $t \geq 7$.

SymChaff uses one symindex set corresponding to the trucks, and several variable classes. Here again symmetry breaking predicates, although not too hard to compute, pro-

vide less than a factor of two improvement. `March-eq` and `zChaff` were much slower than `SymChaff` on large instances, both unsatisfiable and satisfiable.

Logistics Planning `log-pairs`. The problem `log-pairs-`n`-`t is the logistics planning example `PlanningC` with n as the number of location pairs and t as the maximum plan length. As described earlier, it involves moving boxes between n disjoint location pairs. The corresponding formula is satisfiable if and only if $t \geq 5$.

`SymChaff` uses $n+1$ symindex sets corresponding to the trucks and the location pairs, and several variable classes. This problem provides an interesting scenario where `zChaff` normally compares well with `SymChaff` but performs worse by a factor of two when symmetry breaking predicates are added. We also note that computing symmetry breaking predicates for this problem is quite expensive by itself.

Channel Routing. The problem `chnl-`t`-`n is from design automation and has been considered in previous works on symmetry and pseudo-Boolean solvers [1, 3]. It consists of two blocks of circuits with t tracks connecting them. Each track can hold one wire (or "net" as it is sometimes called). The task is to route n wires from one block to the other using these tracks. The underlying problem is a disguised pigeonhole principle. The formula is solvable if and only if $t \geq n$.

`SymChaff` uses two symindex sets corresponding to the end-points of the tracks in the two blocks, and $2n$ variable classes corresponding to the two end-points for each net. While `March-eq` was unable to solve any instance of this problem considered, `zChaff` performed as well as `SymChaff` after symmetry breaking predicates were added. The generation of symmetry breaking predicates was, however, orders of magnitude slower.

5 Discussion and Future Directions

`SymChaff` sheds new light into ways in which high level symmetry, typically obvious to the problem designer, can be used to solve problems more efficiently. It handles frequently occurring complete multi-class symmetries and is empirically exponentially faster than traditional SAT solvers on several problems from theory and practice, both unsatisfiable and satisfiable. It incurs low time and memory overheads for maintaining data structures related to symmetry, and on problems with very few or no symmetries, it works as well as `zChaff`. In particular, the time needed to generate the `.sym` files used by `SymChaff` is typically negligible, in stark contrast with the often high cost of generating symmetry breaking predicates using isomorphism detection.

As a structure-aware solver, `SymChaff` incorporates several new ideas, including simple but effective symmetry representation, multiway branching based on variable classes and symmetry sets, and symmetric learning as an extension of clause learning to multiway branching. Our framework for symmetry is, of course, not tied to `SymChaff`. It can be implemented on top of almost any state-of-the-art DPLL-based CNF or pseudo-Boolean solver. Two key places where we differ from earlier approaches are in using high level problem description to obtain symmetry information (instead of trying to recover it from the CNF formula) and in maintaining this information dynamically but without using a complex group theoretic machinery. This allows us to overcome many drawbacks of previously proposed solutions. We show, in particular, that straightforward annotation in the specification of planning problems is enough to automatically and quickly generate relevant symmetry

information, which in turn makes the search for an optimal plan several orders of magnitude faster. Similar performance gains are seen in other domains as well.

We observe that while complete multi-class symmetries, as introduced in this work, are prevalent in many SAT instances, our framework does not support, for example, symmetries that are initially absent but arise *after* some literals are set. Our symmetry sets only get refined from their initial value as decisions are made. Consider a planning problem where two packages P_1 and P_2 are initially at locations L_1 and L_2, respectively, (and hence asymmetric) but are both destined for location L^{dest}. If at some point they both reach a common location, they should ideally be treated as equivalent with respect to the remaining portion of the plan. The `airlock` domain introduced by Fox and Long [21] is a creative example where such dynamically created symmetries are the norm rather than the exception. While they do describe a planner that is able to exploit these symmetries, it is unclear how to incorporate such reasoning in a general purpose SAT solver besides resorting to on-the-fly computations involving the algebraic group of symmetries, which, as observed in the work of Dixon et al. [14], can sometimes be quite expensive in practice.

We conclude with some directions for further exploration of the ideas presented in this article. The symmetry representation and maintenance techniques we discussed may be exploited in several other ways. The variable selection heuristic of the DPLL process is the most notable example. This framework can be used in local search satisfiability solvers such as `Walksat` by Selman et al. [52] to make better, structure-aware variable flip choices and reduce the search space. It is also suitable, in an appropriately extended form, to general constraint satisfaction problems (CSPs). Finally, it can be applied also to problems containing *k-ring* multi-class symmetries, where the k underlying indices can be rotated cyclically without changing the problem (e.g., as in the `PlanningB` problem, Example 3). However, the best-case gain of a factor of k may not result in a significant speed-up on real-world instances.

Although `SymChaff` itself performs very well in practice, it is the first cut at implementing our generic framework and can be extended in several directions. Learning strategies for symconflict clauses other than the "decision variable scheme" that it currently uses may lead to better performance, and so may dynamic strategies for selecting the order in which various branches of a multiway branch-point are traversed, as well as a dynamic equivalent of the static `.ord` file that `SymChaff` supports. Extending it to handle pseudo-Boolean constraints is a relatively straightforward but promising direction. Creating a PDDL preprocessor for planning problems that uses graph isomorphism tools to annotate symmetries in the PDDL description would fully automate the planning-through-satisfiability process in the context of symmetry.

On the theoretical side, how does the technique of `SymChaff` compare in strength to proof systems such as resolution with symmetry [34, 55]? It is unclear whether it is as powerful as the latter or can even efficiently simulate all of resolution without symmetry. Answering this in the presence of symmetry may also help resolve an open question [6] of whether the clause learning proof system (without symmetry) underlying most of the modern SAT solvers can efficiently simulate all of resolution. Finally, a formal theoretical characterization of the proof system implemented by `SymChaff` will help us better understand its inherent strengths and weaknesses, especially compared to solvers based on proof systems beyond resolution, such as pseudo-Boolean and BDD-based solvers.

Acknowledgements The author would like to thank Paul Beame and Henry Kautz for insightful discussions and support through their NSF Award ITR-0219468, the creators of `zChaff` for making its source code publicly available, Dan Suciu for pointing to the concept of sorts in logic, the anonymous referees whose

comments and suggestions helped improve the article, and the editors of this special issue of Constraints for their patience and gentle prodding.

References

1. F. A. Aloul, I. L. Markov, and K. A. Sakallah. Shatter: Efficient symmetry-breaking for Boolean satisfiability. In *Proceedings of DAC-03: 40th Design Automation Conference*, pages 836–839, Anahein, CA, June 2003.
2. F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. PBS: A backtrack-search pseudo-Boolean solver and optimizer. In *Proceedings of SAT-02: 5th International Conference on Theory and Applications of Satisfiability Testing*, pages 346–353, Cincinnati, OH, May 2002.
3. F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult instances of Boolean satisfiability in the presence of symmetry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(9):1117–1137, 2003.
4. F. A. Aloul, K. A. Sakallah, and I. L. Markov. Efficient symmetry breaking for Boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, 2006.
5. R. J. Bayardo Jr. and R. C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of AAAI-97: 14th Conference on Artificial Intelligence*, pages 203–208, Providence, RI, July 1997.
6. P. Beame, H. Kautz, and A. Sabharwal. Understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, Dec. 2004.
7. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of DAC-99: 36th Design Automation Conference*, pages 317–320, New Orleans, LA, June 1999.
8. D. Chai and A. Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(3):305–317, 2005.
9. W. Cook, C. R. Coullard, and G. Turan. On the complexity of cutting plane proofs. *Discrete Applied Mathematics*, 18:25–38, 1987.
10. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings of KR-96: 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159, Cambridge, MA, Nov. 1996.
11. P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for CNF. In *Proceedings of DAC-04: 41st Design Automation Conference*, pages 518–522, San Diego, CA, June 2004.
12. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
13. M. Davis and H. Putnam. A computing procedure for quantification theory. *Communications of the ACM*, 7:201–215, 1960.
14. H. E. Dixon, M. L. Ginsberg, E. M. Luks, and A. J. Parkes. Generalizing Boolean satisfiability II: Theory. *Journal of Artificial Intelligence Research*, 22:481–534, 2004.
15. H. E. Dixon, M. L. Ginsberg, and A. J. Parkes. Generalizing Boolean satisfiability I: Background and survey of existing work. *Journal of Artificial Intelligence Research*, 21:193–243, 2004.
16. N. Eén and N. Sörensson. MiniSat: A SAT solver with conflict-clause minimization. In *Proceedings of SAT-05: 8th International Conference on Theory and Applications of Satisfiability Testing*, St. Andrews, U.K., June 2005.
17. T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *CP-01: 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 93–107, Paphos, Cyprus, Nov. 2001.
18. R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):198–208, 1971.
19. F. Focacci and M. Milano. Global cut framework for removing symmetries. In *CP-01: 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 77–92, Paphos, Cyprus, Nov. 2001.
20. M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of IJCAI-99: 16th International Joint Conference on Artificial Intelligence*, pages 956–961, July 1999.
21. M. Fox and D. Long. Extending the exploitation of symmetries in planning. In *Proceedings of AIPS-02: 6th International Conference on Artificial Intelligence Planning Systems*, pages 83–91, Apr. 2002.
22. J. H. Gallier. *Logic for Computer Science*. Harper & Row, 1986.

23. I. P. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic SBDD using computational group theory. In *CP-03: 9th International Conference on Principles and Practice of Constraint Programming*, volume 2833 of *Lecture Notes in Computer Science*, pages 333–347, Kinsale, Ireland, Sept. 2003.
24. I. P. Gent and B. M. Smith. Symmetry breaking in constraint programming. In *Proceedings of ECAI-00: 14th European Conference on Artificial Intelligence*, pages 599–603, Berlin, Germany, Aug. 2000.
25. E. Goldberg and Y. Novikov. BerkMin: A fast and robust sat-solver. In *Design, Automation and Test in Europe Conference and Exposition (DATE)*, pages 142–149, Paris, France, Mar. 2002.
26. C. P. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *Proceedings of AIPS-98: 4th International Conference on Artificial Intelligence Planning Systems*, pages 208–213, Pittsburgh, PA, June 1998.
27. A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–305, 1985.
28. M. Heule, J. van Zwieten, M. Dufour, and H. van Maaren. March_eq: Implementing additional reasoning into an efficient lookahead SAT solver. In *Proceedings of SAT-04: 7th International Conference on Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 345–359, Vancouver, BC, May 2004.
29. D. S. Johnson and M. A. Trick, editors. *Cliques, Coloring and Satisfiability: the Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.
30. H. A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of ECAI-92: 10th European Conference on Artificial Intelligence*, pages 359–363, Vienna, Austria, Aug. 1992.
31. H. A. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98*, Pittsburgh, PA, 1998.
32. J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: its Structural Complexity*. Birkhauser Verlag, 1993.
33. H. Konuk and T. Larrabee. Explorations of sequential ATPG using Boolean satisfiability. In *11th VLSI Test Symposium*, pages 85–90, 1993.
34. B. Krishnamurthy. Short proofs for tricky formulas. *Acta Informatica*, 22:253–274, 1985.
35. D. Le Berre and L. Simon (Organizers). SAT 2004 competition, May 2004. URL `http://www.satcompetition.org/2004`.
36. D. Le Berre and L. Simon (Organizers). SAT 2005 competition, June 2005. URL `http://www.satcompetition.org/2005`.
37. C. M. Li, B. Jurkowiak, and P. W. Purdom. Integrating symmetry breaking into a DLL procedure. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 149–155, Cincinnati, OH, May 2002.
38. J. P. Marques-Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Proceedings of ICCAD-96: International Conference on Computer Aided Design*, pages 220–227, San Jose, CA, Nov. 1996.
39. J. P. Marques-Silva and K. A. Sakallah. Robust search algorithms for test pattern generation. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, pages 152–161, Seattle, WA, June 1997.
40. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of DAC-01: 38th Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.
41. D. B. Motter and I. Markov. A compressed breadth-first search for satisfiability. In *ALENEX*, volume 2409 of *Lecture Notes in Computer Science*, pages 29–42, San Francisco, CA, Jan. 2002. Springer.
42. D. B. Motter, J. A. Roy, and I. Markov. Resolution cannot polynomially simulate compressed-BFS. *Annals of Mathematics and Artificial Intelligence*, 44(1-2):121–156, 2005.
43. A. Nadel. The Jerusat SAT solver. Master's thesis, Hebrew University of Jerusalem, 2002.
44. K. E. Petrie and B. M. Smith. Symmetry breaking in graceful graphs. In *CP-03: 9th International Conference on Principles and Practice of Constraint Programming*, volume 2833 of *Lecture Notes in Computer Science*, pages 930–934, Kinsale, Ireland, Sept. 2003.
45. P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, Sept. 1997.
46. J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *International Symposium on Methodologies for Intelligent Systems*, volume 689 of *Lecture Notes in Computer Science*, pages 350–361, Trondheim, Norway, June 1993.
47. J.-F. Puget. Dynamic lex constraints. In *CP-06: 12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 453–467, Nantes, France, Sept. 2006.

48. J.-F. Puget. An efficient way of breaking value symmetries. In *Proceedings of AAAI-06: 21st Conference on Artificial Intelligence*, Boston, MA, July 2006.
49. R. Raz. Resolution lower bounds for the weak pigeonhole principle. *Journal of the ACM*, 51(2):115–138, 2004.
50. A. Sabharwal. *Algorithmic Applications of Propositional Proof Complexity*. PhD thesis, University of Washington, Seattle, 2005.
51. A. Sabharwal. SymChaff: A structure-aware satisfiability solver. In *Proceedings of AAAI-05: 20th National Conference on Artificial Intelligence*, pages 467–474, Pittsburgh, PA, July 2005.
52. B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In Johnson and Trick [29], pages 521–532.
53. O. Shtrichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 1:5–24, 2004.
54. P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinatorial test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9): 1167–1176, 1996.
55. A. Urquhart. The symmetry rule in propositional logic. *Discrete Applied Mathematics*, 96-97:177–193, 1999.
56. M. N. Velev and R. E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003.
57. T. Walsh. General symmetry breaking constraints. In *CP-06: 12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 650–664, Sept. 2006.
58. H. Zhang. SATO: An efficient propositional prover. In *Proceedings of CADE-97: 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275, Townsville, Australia, July 1997.
59. L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of ICCAD-01: International Conference on Computer Aided Design*, pages 279–285, San Jose, CA, Nov. 2001.