SymChaff: A Structure-Aware Satisfiability Solver

Ashish Sabharwal
Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350, U.S.A.
ashish@cs.washington.edu

Abstract

We present a novel low-overhead framework for encoding and utilizing structural symmetry in propositional satisfiability algorithms (SAT solvers). We use the notion of complete multi-class symmetry and demonstrate the efficacy of our technique through a solver SymChaff that achieves exponential speedup by using simple tags in the specification of problems from both theory and practice.

Efficient implementations of DPLL-based SAT solvers are routinely used in areas as diverse as planning, scheduling, design automation, model checking, verification, testing, and algebra. A natural feature of many application domains is the presence of symmetry, such as that amongst all trucks at a certain location in logistics planning and all wires connecting two switch boxes in an FPGA circuit. Many of these problems turn out to have a concise description in many-sorted first order logic. This description can be easily specified by the problem designer and almost as easily inferred automatically. SymChaff, an extension of the popular SAT solver zChaff, uses information obtained from the "sorts" in the first order logic constraints to create symmetry sets that are used to partition variables into classes and to maintain and utilize symmetry information dynamically.

Current approaches designed to handle symmetry include: (A) symmetry breaking predicates (SBPs), (B) pseudo-Boolean solvers with implicit representation for counting, (C) modifications of DPLL that handle symmetry dynamically, and (D) techniques based on ZBDDs. SBPs are prohibitively many, often large, and expensive to compute for problems such as the ones we report experimental results for. Pseudo-Boolean solvers are provably exponentially slow in certain symmetric situations and their implicit counting representation is not always appropriate. Suggested modifications of DPLL either work on limited global symmetry and are difficult to extend, or involve expensive algebraic group computations. Finally, techniques based on ZBDDs often do not compare well even with ordinary DPLL-based solvers. SymChaff addresses and overcomes most of these limitations.

Introduction

In recent years, general purpose propositional satisfiability algorithms (SAT solvers) have been designed and shown to be very successful in handling and even outperforming specific solvers on problems from many real-world domains including hardware verification (Biere *et al.* 1999; Velev & Bryant 2001), automatic test pattern generation (Konuk & Larrabee 1993; Stephan, Brayton, & Sangiovanni-Vincentelli 1996), planning (Kautz & Selman 1992), and scheduling (Gomes *et al.* 1998). With a large community of researchers working towards a better understanding of SAT, it is not surprising that many competing general purpose systematic SAT solvers have come into light in the past decade such as Grasp (Marques-Silva & Sakallah 1996), Relsat (Bayardo Jr. & Schrag 1997), SATO (Zhang 1997), zChaff (Moskewicz *et al.* 2001), Berkmin (Goldberg & Novikov 2002), and March-eq by Marijn Heule and Hans van Maaren. All of these solvers fall into the category of systematic DPLL-based solvers, and build upon a basic branch and backtrack technique (Davis, Logemann, & Loveland 1962). With the addition of features such as smart branch selection heuristics, conflict clause learning, random restarts, conflict-directed backjumping, fast unit propagation using watched literals, etc., these have been quite effective in solving challenging problems from various domains.

Despite the success, one aspect of many theoretical as well as real-world problems that we believe has not been fully exploited is the presence of symmetry. Symmetry occurs naturally in many application areas. For example, in FPGA routing, all wires or channels connecting two switch boxes are equivalent; in circuit modeling, all inputs to a multiple fanin AND gate are equivalent; in planning, all boxes that need to be moved from city A to city B are equivalent; in multi-processor scheduling (or cache coherency protocols), all available processors (or caches, respectively) are typically equivalent. While there has been work on using this equivalence or symmetry in domain-specific algorithms and techniques, current general purpose complete SAT solvers are unable to fully capitalize on symmetry as suggested by our experimental results.

Previous Work

A technique that has worked quite well in handling symmetry is to add symmetry breaking predicates (SBPs) to the input specification to weed out all but the lexically-first solutions (Crawford *et al.* 1996). Tools such as Shatter (Aloul, Markov, & Sakallah 2003) use graph isomorphism detectors like Saucy to generate SBPs. This latter problem of computing graph isomorphism is not known to have any polynomial time solution, and is conjectured to be strictly between the complexity classes P and NP (see e.g. Köbler, Schöning, & Torán 1993). Further, the number of SBPs one needs to add in order to break all symmetries may be prohibitively large. This is typically handled by discarding "large" symmetries. This may, however, result in a much slower SAT solution as indicated by some of our experiments.

Solvers such as PBS (Aloul *et al.* 2002a), pbChaff (Dixon, Ginsberg, & Parkes 2004), and Galena (Chai & Kuehlmann 2003) utilize non-CNF formulations known as pseudo-Boolean (PB) inequalities. They are based on the Cutting Planes proof system which is known to be strictly stronger than the resolution proof system on which DPLL type CNF solvers are based (Cook, Coullard, & Turan 1987). Since this more powerful proof system is difficult to implement in its full generality, PB solvers often implement only a subset of it, typically learning only CNF clauses or restricted PB constraints upon a conflict. PB 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 over nm variables (described in detail later) 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.

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, we are unaware of its implementation in a general purpose SAT solver besides sEqSatz (Li, Jurkowiak, & Purdom 2002) whose technique appears to be somewhat specific and whose results are not too impressive compared to zChaff itself. Related work has been done in specific areas of automatic test pattern generation (Marques-Silva & Sakallah 1997) and SAT-based model checking (Shtrichman 2004), where the solver utilizes global information obtained at a stage to make subsequent stages faster.

Dixon *et al.* (2004) give a generic method of representing and dynamically maintaining symmetry using group theoretic techniques that guarantee polynomial size proofs of many difficult formulas. The underlying group computations, however, are often quite expensive.

Our Contribution

We propose a new technique for representing and dynamically maintaining symmetry information for DPLL-based satisfiability solvers. We present an evaluation of our ideas through our tool SymChaff and demonstrate empirical exponential speedup in a variety of problem domains from theory and practice. While our framework as presented applies to both CNF and PB formulations, the current implementation of SymChaff uses pure CNF representation.

A key difference between our approach and that based on SBPs is that we use a high level description of a problem rather than its CNF representation to obtain symmetry information. (We give concrete examples of this later in the paper.) This leads to several advantages. The high level description of a problem is typically very concise and reveals its structure much better than a relatively large set of 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." If one prefers to compute these symmetries automatically, off-the-shelf graph isomorphism tools can be used. Using these tools on the concise high level description will, of course, be much faster than using the same tools on a substantially larger CNF encoding.

While it is natural to pick 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. 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 and handle it using a $(k+1)$-*way branch* based on I that maintains completeness of the search and shrinks the search space by as much as $O(k!)$. 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 conflict and clause learning to the multiway branch setting, introducing *symmetric learning*. Our solver SymChaff integrates seamlessly with most of the 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.

Preliminaries

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 (PB) formula is a conjunction of PB constraints, where each PB 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 PB 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. Unit propagation is a technique common to SAT and PB solvers that recursively simplifies the formula by appropriately setting unset variables in unit constraints.

DPLL-based SAT Solvers

The technique we present in this paper can be applied to all DPLL based systematic SAT solvers designed for CNF as well as PB 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. Various other features and optimizations, such as random restarts, watched literals, conflict-directed backjumping, etc., are added to this basic structure to increase efficiency.

Constraint 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 variables. Although the following notions are generic, our focus in this paper will be on CNF and PB constraints over propositional variables. We will use the notation $[n]$ to denote the set $\{1, 2, \ldots, n\}$. Let σ be a permutation of $[n]$. Define $\sigma(x_i) = x_{\sigma(i)}$ and $\sigma(V' \subseteq V) = \{\sigma(x) \mid x \in V'\}$. For a constraint C over V, let $\sigma(C)$ be the constraint resulting from C by applying σ to each variable of C. For a CSP Γ, define $\sigma(\Gamma)$ to be the new CSP consisting of the constraints $\{\sigma(C) \mid C \in \Gamma\}$.

Symmetry may exist in various forms in Γ. Permutation σ will be called a *global symmetry* of Γ if $\sigma(\Gamma) = \Gamma$. Suppose there exists $V' \subseteq V, |V'| = k$, such that *every* permutation σ satisfying $\sigma(V') = V'$ and $\sigma(x) = x$ for $x \notin V'$ is a global symmetry of Γ, then V' is called a *k-complete (global) symmetry* of Γ. In other words, the k variables in V' can be arbitrarily interchanged without changing the original problem. Such symmetries exist in simple problems such as the pigeonhole principle where all pigeons (and holes) are symmetric. This can be detected and exploited using various known techniques such as cardinality constraints (Dixon, Ginsberg, & Parkes 2004; Chai & Kuehlmann 2003).

Many-Sorted First Order Logic

In first order logic[1], one can express universally and existentially quantified logical statements about variables and constants that range over a domain with some structure. The domain may be divided up into various types or "sorts" of elements that are quantified over independently. Consider the pigeonhole problem 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." The CNF formulation of same problem requires $|P| + |H|\binom{|P|}{2}$ clauses.

[1]A comprehensive introduction to many-sorted first order logic is beyond the scope of this paper. The reader is referred to standard texts (e.g. Gallier 1986) for details.

Symmetry Framework and SymChaff

We describe in this section 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. Consider the following relatively simple logistics planning problem. There are k trucks $T_1, T_2, \ldots, T_k$ at a location L_{TB} (truckbase). For $1 \leq i \leq n$, there is a location L_i that has two packages $P_{i,1}$ and $P_{i,2}$. Let $s(i) = (i \bmod n) + 1$ denote the cyclic successor of i in $[n]$. The goal is to deliver package $P_{i,1}$ to location $L_{s(i)}$ and package $P_{i,2}$ to location $L_{s(s(i))}$. 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.

Call this problem `PlanningA` and let $k = \lceil 3n/4 \rceil$. In this case it, the shortest plan 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 $P_{4,2}$. For a given plan length, such a planning problem can be converted into a CNF formula using tools such as Blackbox (Kautz & Selman 1998) 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 (see Kautz & Selman 1992).

We will also refer to the following two variants of the above problem. In `PlanningB`, 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. Let $k = \lceil n/2 \rceil$. It can again be verified that the shortest plan for this problem is of length 7 and assigns one truck to two consecutive locations. In `PlanningC`, 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}. Let $k = n$. It is easily seen that the shortest plan for this problem is of length 5 and assigns one truck to each location. 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.

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 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 variables in each class $V_i, i \in [m]$, simultaneously.

If $\bar{\sigma}$ is a global symmetry of Γ for *every* permutation σ of $[k]$ then the set $\{V_1, V_2, \ldots, V_m\}$ will be called a *k-complete m-class (global) symmetry* of Γ. Note that a k-complete 1-class symmetry is simply a k-complete symmetry. We refer to $V_i, i \in [m]$, as *variable classes* and say that variables in V_i are *indexed by* the *symindex set* $[k]$.

Such symmetries correspond to the case where variables from multiple classes can be simultaneously and coherently changed in parallel without affecting the problem. This happens naturally in many problem domains. For instance, consider the logistics planning example PlanningA described above 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 [3] and correspond to sets of 3 variables that differ only in which truck they use. E,g,, 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-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 described above where variables load-$P_{2,a}$-onto-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, 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.

Symmetry Representation

SymChaff takes as input a CNF file in the standard DIMACS format as well as a `.sym` symmetry file S that encodes 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 $[1 \mathinner{..} 3]$ and the 5 packages at location $L_i, 1 \le i \le 4$, by symindex sets $[3 + 5(i-1) + 1 \mathinner{..} 3 + 5i]$, respectively. Here $[p \mathinner{..} q]$ denotes the set $\{p, p+1, \ldots, q\}$. Second, one *varclass* 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 varclass it belongs to and the indices in the symindex sets it is indexed by. For instance, variable load-$P_{2,4}$-onto-T_3-at-L_4-time-4 in problem PlanningB will be associated with the varclass load-$P_{2,a}$-onto-T_j-at-L_4-time-4 and with indices $j = 3$ and $a = 3 + 5(2-1) + 4 = 12$. We omit the exact syntax of the symmetry input file S in the interest of saving space. It is a straightforward encoding of symindex sets, varclasses, and symindex map.

Note that while the varclasses and the symindex map remain static, the symindex sets will dynamically change 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 varclasses in a vector data structure from the Standard Template Library (STL) of C++, the symindex map as a hash_map, 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.

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 branch by setting x now to FALSE and later to TRUE. If it does have symmetry, SymChaff arbitrarily chooses a symindex set $I, |I| = k \ge 2$, that indexes x and creates a $(k+1)$-way branch. Let $x_1, x_2, \ldots, x_k$ be the variables indexed by I in the varclass V' to which x belongs ($x \equiv x_j$ for some j). For $0 \le i \le k$, the i-th branch 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 is also indexed by an index j in a symindex set $J \equiv [a \mathinner{..} b] \ne I$, we must split J into up to three symindex sets given by the intervals $[a \mathrel{..} j-1]$, $[j \mathrel{..} j]$, and $[j+1 \mathrel{..} b]$ because j's symmetry has been destroyed by this assignment. To reduce the number of splits, SymChaff replaces x with another variable in its varclass for which $j = a$ and thus the split divides J into two new symindex sets only, $[a \mathrel{..} a]$ and $[a+1 \mathrel{..} b]$. This first kind of split is done once for the multiway branch for x and is independent of the value of i. The second kind of split divides $I \equiv [c \mathrel{..} d]$ into up to two symindex sets given by $[c \mathrel{..} i]$ and $[i+1 \mathrel{..} d]$. This, of course, 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 branching step.

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 $i+1$-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 $i+1$-st branch. When all $k+1$ branches are finished, SymChaff also deletes the split of the first type created for this multiway branch and backtracks.

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 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 branches and is computed as follows.

Suppose a k-way branch b starts at decision level d. If the i-th branch of b leads to a conflict without any further branches, two things happen. First, SymChaff learns the "firstUIP" clause following the conflict analysis strategy of zChaff (see Moskewicz *et al.* 2001 for details). 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 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 $\bigvee_{\ell \in S_b} \neg\ell$.

Static Ordering of Symmetry Classes and Indices

It is well known that the variable order chosen for branching in any DPLL-based solver has tremendous impact on efficiency. Along similar lines, the order in which variable classes and symindex sets are chosen for multiway branching can have significant impact on the speed of SymChaff.

While we leave dynamic strategies for selecting variable classes and symindex sets as ongoing and future work, SymChaff does support static ordering through a very simple and optional `.ord` order file given as input. This file specifies an ordering of variable classes as an initial guide to the VSIDS variable selection heuristic of zChaff, 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 file structure is omitted due to lack of space.

Integration of Standard Features

The efficiency of state of the art SAT and PB 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 in Jerusat (Nadel 2002).

SymChaff supports fast unit propagation, 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 based on the variable that would have been chosen by VSIDS branch selection heuristic of zChaff, thereby retaining many advantages that effective branch selection heuristics like VSIDS have to offer.

Conflict clause learning is extended into 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 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, discarding all intermediate branches 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 and the variable of the current branch. 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. 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.

Experimental Results

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.

Table 1 reports 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 thus omitted from the table. 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 `.cnf` file and is therefore not reported. The `.sym` files were in addition extremely small compared to the corresponding `.cnf` files.

The solvers used were SymChaff, zChaff version 2003.11.04, and March-eq-100. SBPs were generated using Shatter version 0.3 that uses the graph isomorphism tool Saucy. Note that zChaff won the best solver award for industrial benchmarks in the SAT '04 competition while March-eq-100 won the award for handmade benchmarks.

SymChaff outperformed the other two solvers without SBPs in all but excessively easy instances. Generating SBPs from the input CNF formula was typically quite slow compared to a complete solution by SymChaff. The effect of adding SBPs before feeding the problem to zChaff was mixed, helping to various extents in some instances and hurting in others. In either case, it was never any better than using SymChaff without SBPs.

Problems from Proof Complexity

Pigeonhole Principle: php-n-m is the classic pigeonhole problem where the task is to map n pigeons into m holes without any overlaps. These formulas are satisfiable iff $n \leq m$. They are known to be exponentially hard for resolution (Haken 1985; Raz 2004) but easy when the symmetry rule is added (Krishnamurthy 1985). SBPs can therefore be used for fast CNF SAT solutions. The price to pay is symmetry detection which we found to be significant. pbChaff and Galena use an explicit PB encoding and rely on learning good PB conflict constraints. They are slower than SymChaff (execution times are not reported here for lack of space). SymChaff uses two symindex sets corresponding to pigeons and holes, and one variable class to solve this problem in time $\Theta(m^2)$. This contrasts well with one of the fastest current techniques for this problem (other than the implicit PB encoding) which is based on ZBDDs (Motter & Markov 2002) and requires fairly involved analysis to prove that it runs in time $\Theta(m^4)$ (Motter, Roy, & Markov 2005).

Clique Coloring Principle: clqcolor-n-m-k encodes the clique coloring problem where the task is to find a graph over n nodes that contains a clique of size m and can be colored using k colors so that no two adjacent nodes get the same color. These formulas are satisfiable iff $m \leq n$ and $m \leq k$. 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 PB techniques even when the clique part is encoded implicitly using PB methods. Indeed, it has been shown to be exponentially hard for the Cutting Planes proof system (Pudlák 1997). Our experiments indicate that not only finding symmetries from the corresponding CNF formulas is time consuming, zChaff is extremely slow even after taking SBPs into account. SymChaff uses three symindex sets corresponding to nodes, membership in clique, and colors, and three variable classes to solve the problem in time $\Theta(k^2)$. We note that this problem can also be solved in polynomial time (albeit with high polynomial degree) using the group theoreic technique of Dixon *et al.* (2004).

Problems from Applications

All planning problems were encoded using the high level STRIPS formulation of Planning Domain Description Language (PDDL) (Fikes & Nilsson 1971) and converted into CNF formulas using the tool Blackbox version 4.1. We modified Blackbox to generate symmetry information as well by using a very simple "tagged" PDDL description where an original PDDL declaration such as

$$
\begin{array}{ll}
\texttt{(:OBJECTS} & T_1\ T_2\ T_3 \\
 & L_1^{\text{src}}\ L_2^{\text{src}}\ L_1^{\text{dest}}\ L_2^{\text{dest}} \\
 & P_{1,1}\ P_{2,1}\ P_{1,2}\ P_{2,2}\texttt{)}
\end{array}
$$

in the `PlanningC` example is replaced with

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

The rest of the PDDL description remains unchanged and a `.sym` file is automatically generated using these tags.

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 iff $t \geq 4n - 1$. SymChaff uses two symindex sets corresponding to the balls and the grippers. The number of varclasses 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 poorly. Detecting symmetries from CNF using Shatter is not too difficult but does not speed up the solution process by any significant amount.

Logistics Planning A: This is the example `PlanningA` denoted now by log-rotate-n-t where n is the number of locations and t is the maximum plan length. The formula is satisfiable iff $t \geq 7$. SymChaff uses one symindex set corresponding to the trucks, and several varclasses. Here again SBPs, although not too hard to compute, provide less than a factor of two improvement. March-eq and zChaff were much slower than SymChaff on large instances, both unsatisfiable and satisfiable.

Table 1: Experimental results. ‡ indicates > 6 hours. The second column shows problem parameters. The last two problem sets are satisfiable while the rest are not.

Problem		SymChaff	zChaff	March-eq	Shatter	zChaff on Shatter
php	009-008	0.01	0.22	1.55	0.07	0.10
	013-012	0.01	1017	‡	0.09	0.01
	051-050	0.24	‡	‡	13.71	0.50
	091-090	0.84	‡	‡	245	3.47
	101-100	1.20	‡	‡	466	6.48
clqcolor	05-03-04	0.02	0.01	0.21	0.09	0.01
	12-07-08	0.03	‡	‡	5.09	4929
	20-15-16	0.26	‡	‡	748	‡
	30-18-21	0.60	‡	‡	20801	‡
	50-40-45	8.76	‡	‡	‡	‡
gripper	02t6	0.02	0.03	0.07	0.20	0.04
	04t14	0.84	2820	‡	3.23	983
	06t22	3.37	‡	‡	23.12	‡
	10t38	47	‡	‡	193	‡
log-rotate	06t6	0.74	1.47	21.55	8.21	0.93
	08t6	2.03	4.29	375	31.4	4.21
	09t6	8.64	15.67	3835	74	28.9
	11t6	51	12827	‡	324	17968
log-pair	05t5	0.46	0.38	3.65	25.19	0.65
	07t5	1.83	1.87	80	243	3.05
	09t5	6.29	6.23	582	1373	14.57
	11t5	15.65	18.05	1807	6070	34.4
chnl	010-011	0.04	8.61	‡	0.20	0.02
	011-020	0.06	135	‡	0.28	0.03
	020-030	0.05	‡	‡	4.60	0.10
	050-100	1.75	‡	‡	810	1.81
gripper	02t7	0.02	0.03	0.34	0.17	0.03
	04t15	2.03	1061	‡	0.23	1411
	06t23	7.27	‡	‡	19.03	‡
	10t39	92	‡	‡	193	‡
log-rotate	06t7	2.87	2.09	11	16.92	3.03
	07t7	7.64	6.85	27	55	47
	08t7	9.13	182	14805	62	358
	09t7	139	1284	814	186	1356

Logistics Planning C: This is the example `PlanningC` denoted now by `log-pairs-`n`-`t where n is the number of location pairs and t is the maximum plan length. The formula is satisfiable iff $t \geq 5$. SymChaff uses $n + 1$ symindex sets corresponding to the trucks and the location pairs, and several varclasses. This problem provides an interesting scenario where zChaff normally compares well with SymChaff but performs worse by a factor of two when SBPs are added. We also note that computing SBPs 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 (Aloul, Markov, & Sakallah 2003; Aloul *et al.* 2002b). It consists of two blocks of circuits with t tracks connecting them. The task is to route n nets from one block to the other using these tracks. The underlying problem is a disguised pigeonhole principle. The formula is solvable iff $t \geq n$. SymChaff uses two symindex sets corresponding to the end-points of the tracks in the two blocks, and $2n$ varclasses corresponding to the two endpoints for each net. While March-eq was unable to solve any instance of this problem considered, zChaff performed as well as SymChaff after SBPs were added. The generation of SBPs was, however, orders of magnitude slower.

Discussion and Future Work

SymChaff sheds new light into ways that high level symmetry, which is 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 on several problems from theory and practice, both unsatisfiable and satisfiable. The time and memory overhead it needs for maintaining data structures related to symmetry is fairly low and on problems with very few or no symmetries, it works as well as zChaff.

Our framework for symmetry is, of course, not tied to SymChaff. It can extend any state of the art DPLL-based SAT or PB solver. Two key places where we differ from earlier approaches are in using high level problem description to obtain symmetry information and in maintaining this information dynamically without using complicated group theoretic machinery. This allows us to overcome many drawbacks of previously proposed solutions.

The symmetry representation and maintenance techniques of SymChaff may be exploited in several other ways. The variable selection heuristic of the DPLL process is the most noticeable example. This framework can perhaps be applied even to local search-based satisfiability tools such as Walksat (McAllester, Selman, & Kautz 1997) to make better choices and reduce the search space. As for the framework itself, it can be easily extended to handle k*-ring* multiclass symmetries, where the k underlying indices can be rotated cyclically without changing the problem (e.g. as in `PlanningB`). However, the best-case gain of a factor of k may not offset the overhead involved.

On the theoretical side, how the technique of SymChaff compares in strength to proof systems such as resolution with symmetry? 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 (Beame, Kautz, & Sabharwal 2004) of whether DPLL-based solvers (without symmetry) can efficiently simulate all of resolution.

SymChaff 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 are traversed, as well as a dynamic equivalent of the static `.ord` file that SymChaff supports. Extending it to handle PB constraints is a relatively straightforward but promising direction. Creating a PDDL preprocessor for planning problems that uses graph isomorphism tools to tag symmetries in the PDDL description would fully automate the planning-through-satisfiability process in the context of symmetry.

One limitation of our framework is that it does not support 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. Detecting such dynamically created symmetries, however, appears to require on-the-fly computations involving the symmetry group which are generally quite expensive (Dixon *et al.* 2004).

Acknowledgments

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, and Dan Suciu for pointing to sorts in logic.

References

[Aloul *et al.* 2002a] Aloul, F. A.; Ramani, A.; Markov, I. L.; and Sakallah, K. A. 2002a. PBS: A backtrack-search pseudo-boolean solver and optimizer. In *5th SAT*, 346–353.

[Aloul *et al.* 2002b] Aloul, F. A.; Ramani, A.; Markov, I. L.; and Sakallah, K. A. 2002b. Solving difficult SAT instances in the presence of symmetry. In *39th DAC*, 731–736.

[Aloul, Markov, & Sakallah 2003] Aloul, F. A.; Markov, I. L.; and Sakallah, K. A. 2003. Shatter: Efficient symmetry-breaking for boolean satisfiability. In *40th DAC*, 836–839.

[Bayardo Jr. & Schrag 1997] Bayardo Jr., R. J., and Schrag, R. C. 1997. Using CST look-back techniques to solve real-world SAT instances. In *14th AAAI*, 203–208.

[Beame, Kautz, & Sabharwal 2004] Beame, P.; Kautz, H.; and Sabharwal, A. 2004. Understanding and harnessing the potential of clause learning. *JAIR* 22:319–351.

[Biere *et al.* 1999] Biere, A.; Cimatti, A.; Clarke, E. M.; Fujita, M.; and Zhu, Y. 1999. Symbolic model checking using SAT procedures instead of BDDs. In *36th DAC*, 317–320.

[Chai & Kuehlmann 2003] Chai, D., and Kuehlmann, A. 2003. A fast pseudo-boolean constraint solver. In *40th DAC*, 830–835.

[Cook, Coullard, & Turan 1987] Cook, W.; Coullard, C. R.; and Turan, G. 1987. On the complexity of cutting plane proofs. *Discr. Applied Mathematics* 18:25–38.

[Crawford *et al.* 1996] Crawford, J. M.; Ginsberg, M. L.; Luks, E. M.; and Roy, A. 1996. Symmetry-breaking predicates for search problems. In *5th KR*, 148–159.

[Davis, Logemann, & Loveland 1962] Davis, M.; Logemann, G.; and Loveland, D. 1962. A machine program for theorem proving. *CACM* 5:394–397.

[Dixon *et al.* 2004] Dixon, H. E.; Ginsberg, M. L.; Luks, E. M.; and Parkes, A. J. 2004. Generalizing boolean satisfiability II: Theory. *JAIR* 22:481–534.

[Dixon, Ginsberg, & Parkes 2004] Dixon, H. E.; Ginsberg, M. L.; and Parkes, A. J. 2004. Generalizing boolean satisfiability I: Background and survey of existing work. *JAIR* 21:193–243.

[Fikes & Nilsson 1971] Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *J. AI* 2(3/4):198–208.

[Gallier 1986] Gallier, J. H. 1986. *Logic for Computer Science*. Harper & Row.

[Goldberg & Novikov 2002] Goldberg, E., and Novikov, Y. 2002. BerkMin: A fast and robust sat-solver. In *DATE*, 142–149.

[Gomes *et al.* 1998] Gomes, C. P.; Selman, B.; McAloon, K.; and Tretkoff, C. 1998. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *4th Int. Conf. Art. Intel. Planning Syst.*

[Haken 1985] Haken, A. 1985. The intractability of resolution. *Theoretical Comput. Sci.* 39:297–305.

[Kautz & Selman 1992] Kautz, H. A., and Selman, B. 1992. Planning as satisfiability. In *Proc., 10th Euro. Conf. on AI*, 359–363.

[Kautz & Selman 1998] Kautz, H. A., and Selman, B. 1998. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98*.

[Köbler, Schöning, & Torán 1993] Köbler, J.; Schöning, U.; and Torán, J. 1993. *The Graph Isomorphism Problem: its Structural Complexity*. Birkhauser Verlag.

[Konuk & Larrabee 1993] Konuk, H., and Larrabee, T. 1993. Explorations of sequential ATPG using boolean satisfiability. In *11th VLSI Test Symposium*, 85–90.

[Krishnamurthy 1985] Krishnamurthy, B. 1985. Short proofs for tricky formulas. *Acta Inf.* 22:253–274.

[Li, Jurkowiak, & Purdom 2002] Li, C. M.; Jurkowiak, B.; and Purdom, P. W. 2002. Integrating symmetry breaking into a DLL procedure. In *SAT*, 149–155.

[Marques-Silva & Sakallah 1996] Marques-Silva, J. P., and Sakallah, K. A. 1996. GRASP – a new search algorithm for satisfiability. In *ICCAD*, 220–227.

[Marques-Silva & Sakallah 1997] Marques-Silva, J. P., and Sakallah, K. A. 1997. Robust search algorithms for test pattern generation. In *27th FTCS*, 152–161.

[McAllester, Selman, & Kautz 1997] McAllester, D. A.; Selman, B.; and Kautz, H. 1997. Evidence for invariants in local search. In *AAAI/IAAI*, 321–326.

[Moskewicz *et al.* 2001] Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient SAT solver. In *38th DAC*, 530–535.

[Motter & Markov 2002] Motter, D. B., and Markov, I. 2002. A compressed breadth-first search for satisfiability. In *ALENEX*, volume 2409 of *LNCS*, 29–42. San Francisco, CA: Springer.

[Motter, Roy, & Markov 2005] Motter, D. B.; Roy, J. A.; and Markov, I. 2005. Resolution cannot polynomially simulate compressed-BFS. *Ann. of Math. and A.I.* 44(1-2):121–156.

[Nadel 2002] Nadel, A. 2002. The Jerusat SAT solver. Master's thesis, Hebrew University of Jerusalem.

[Pudlák 1997] Pudlák, P. 1997. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Logic* 62(3):981–998.

[Raz 2004] Raz, R. 2004. Resolution lower bounds for the weak pigeonhole principle. *J. Assoc. Comput. Mach.* 51(2):115–138.

[Shtrichman 2004] Shtrichman, O. 2004. Accelerating bounded model checking of safety properties. *Form. Meth. in Sys. Des.* 1:5–24.

[Stephan, Brayton, & Sangiovanni-Vincentelli 1996] Stephan, P. R.; Brayton, R. K.; and Sangiovanni-Vincentelli, A. L. 1996. Combinatorial test generation using satisfiability. *IEEE Trans. Comput.-Aided Design Integr. Circ.* 15(9):1167–1176.

[Velev & Bryant 2001] Velev, M., and Bryant, R. 2001. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. In *38th DAC*, 226–231.

[Zhang 1997] Zhang, H. 1997. SATO: An efficient propositional prover. In *14th CADE*, volume 1249 of *LNCS*, 272–275.