Chapter 6

SYMMETRY IN SATISFIABILITY SOLVERS

As discussed earlier, we have seen tremendous improvement in the capabilities of general purpose SAT solvers in the past decade. The state-of-the-art techniques make them 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 argue has not been fully exploited is the presence of *symmetry* or *equivalence* amongst the underlying objects.

The concept of symmetry in the context of SAT solvers 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 wires or channels connecting two switch boxes are equivalent; in circuit modeling, all inputs to a multiple fanin AND or OR gate are equivalent; in planning, all boxes that need to be moved from city A to city B are equivalent; in multi-processor scheduling, all available processors are equivalent; in cache coherency protocols in distributed computing, all available caches are typically equivalent. When such problems are translated into CNF formulas to be fed to a SAT solver, the underlying equivalence or symmetry translates into a symmetry between the variables of the formula.

There has been work on using this symmetry in *domain-specific* algorithms and techniques. However, our experimental results suggest that current general purpose complete SAT solvers are unable to fully capitalize on symmetry. This chapter focuses on developing a new general purpose technique towards this end and on empirically evaluating its effectiveness in comparison with other known approaches.

Example 6.1. 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-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 $\{1, 2, \ldots, p\}$.

(a) 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

(b) 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 [60] used PHP^n_{n-1} to show the first ever exponential lower bound for resolution. 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 [78] testify that the pigeonhole formulas provide a class of hard instances for most of the complete SAT solvers which are based on the clause learning proof system, and hence on resolution (cf. Chapter 4).

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. 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 $x_{i,\ell}$.

Remark 6.1. While we use PHP^n_m as a motivation for the work presented in this chapter, we would like to remind the reader that the techniques we develop are much more general and capable of handling symmetry in more complex forms that we will describe in due course. There are known techniques to handle the pigeonhole problem in SAT solvers, such as the use of cardinality constraints by Chai and Kuehlmann [32] and Dixon et al. [49]. However, such approaches either do not generalize or do not perform as well in the presence of more complex forms of symmetry.

Previous Work

A technique due to Crawford et al. [41] that has worked quite well in handling symmetry is to add *symmetry breaking predicates* 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. Tools such as Shatter by Aloul et al. [4] improve upon this technique and use graph isomorphism detectors like Saucy by Darga et al. [42] to generate symmetry breaking predicates. 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 [cf. 73]. Hence, one must resort to heuristic or approximate solutions. Further, the number of symmetry breaking predicates 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 by Aloul et al. [5], pbChaff by Dixon et al. [49], and Galena by Chai and Kuehlmann [32] utilize non-CNF formulations known as pseudo-Boolean

inequalities. They are based on the cutting planes proof system which, as mentioned in Section 3.2, 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.

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 by Li et al. [81] whose technique appears to be somewhat specific and whose results are not too impressive compared to zChaff itself. Related work has been done in the specific areas of automatic test pattern generation by Marques-Silva and Sakallah [85] and SAT-based model checking by Shtrichman [102]. In both cases, the solver utilizes global information obtained at a stage to make subsequent stages faster.

In other domain-specific work, Fox and Long [52] presented a framework for planning problems that is very similar to ours in essence. However, their work has two disadvantages. The obvious one is that they provide a planner, not a general purpose reasoning engine. The second is that their approach does not guarantee plans of optimal length when multiple (non-conflicting) actions are allowed to be performed at each time step.

Dixon et al. [48] give a generic method of representing and dynamically maintaining symmetry using group theoretic techniques that guarantee polynomial size 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.

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 pseudo-Boolean formulations, the current implementation

of SymChaff uses pure CNF representation.

A key difference between our approach and that based on symmetry breaking predicates 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 this chapter.) 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 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. 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.

6.1 Preliminaries

The technique we present in this work can be applied to all DPLL based systematic SAT solvers designed for CNF as well as pseudo-Boolean formulas.

Definition 6.1. 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.

This generalizes the notion of a clause; $(a \vee b \vee c)$ is equivalent to the pseudo-Boolean inequality $a + b + c \geq 1$.

Recall that a CNF 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. 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.

A DPLL-based systematic SAT or pseudo-Boolean solver implements the basic branch-and-backtrack procedure described in Section 2.3. Various features and optimizations, such as conflict clause learning, random restarts, watched literals, conflict-directed backjumping, etc., are added to this simple DPLL process in order to increase efficiency.

6.1.1 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 variables. Although the following notions are generic, our focus in this work will be on CNF and pseudo-Boolean constraints over propositional variables.

Symmetry may exist in various forms in a CSP. We define it in terms of permutations of variables that preserve certain properties. Let σ be a permutation of $[n]$. Extend σ by defining $\sigma(x_i) = x_{\sigma(i)}$ for $x_i \in V$ and $\sigma(V') = \{\sigma(x) \mid x \in V'\}$ for $V' \subseteq V$. 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 $\{\sigma(C) \mid C \in \Gamma\}$.

Definition 6.2. A permutation σ of the variables of a CSP Γ is a *global symmetry* of Γ if $\sigma(\Gamma) = \Gamma$.

Definition 6.3. Let V be a the set of variables of a CSP Γ. $V' \subseteq V, |V'| = k$, is a *k-complete (global) symmetry* of Γ if *every* permutation σ of V satisfying $\sigma(V') = V'$ and $\sigma(x) = x$ for $x \notin V'$ is a 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 by Chai and Kuehlmann [32] and Dixon et al. [49].

6.1.2 Many-Sorted First Order Logic

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 $\{1, 2, \ldots, n\}$ 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 [54] 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 6.1 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.

6.2 *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. We use three relatively simple logistics planning problems depicted in Figure 6.1. 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 6.2 (`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 (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}.

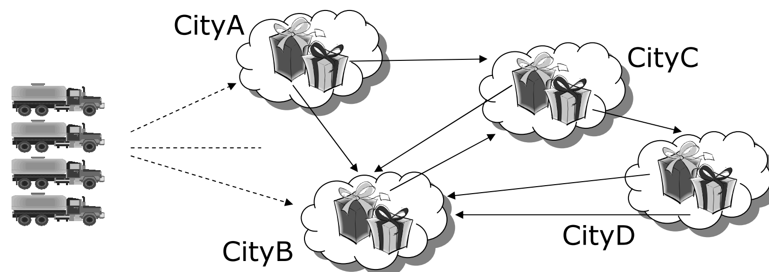

Figure 6.1: The setup for logistic planning examples

Example 6.3 (`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 6.4 (`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 [72] 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 70].

6.2.1 *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 jth 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 6.4. 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. This happens naturally in many problem domains.

Example 6.5. Consider the logistics planning problem `PlanningA` (Example 6.2) 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 6.3) 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 6.4), 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.

6.2.2 Symmetry Representation

SymChaff takes as input a CNF file in the standard DIMACS format [68] 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 $L_i, 1 \leq i \leq 4$, 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. 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 6.6. 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 6.2. 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 "vartype" 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.

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 file for php-004-003.cnf
c 4 pigeons, 3 holes, 12 symmetric variables
c 2 symindex sets, 1 vartype
c
p sym 12 2 1
c
c symindex sets
1 4 0
2 7 0
0
c vartypes
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
```

Figure 6.2: A sample symmetry file, `php-004-003.sym`

6.2.3 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 branch by setting x now to FALSE and later to TRUE. If it does have symmetry, SymChaff arbitrarily chooses a symindex set $I, |I| = k \geq 2$, 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 variable class V' to which x belongs ($x = x_j$ for some j). For $0 \leq i \leq k$, the ith 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 ith 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 $J = [a \mathrel{..} b] \neq I$, 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 variable class 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 $I = [c \mathrel{..} d]$ 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 ith 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 ith 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 ith 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.

6.2.4 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 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 ith 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 Section 4.2.5). 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 ith 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

$\bigvee_{\ell \in S_b} \neg \ell$.

6.2.5 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 (cf. Section 2.3.2), 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.

6.2.6 Integration of Standard 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 [91] 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 (cf. Sections 2.3.2 and 4.2). 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 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 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 (in addition to the variable at 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 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.

6.3 Benchmark Problems and 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.

Tables 6.1 and 6.2 report 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 by

Huele et al. [64]. Symmetry breaking predicates 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 2004 competition [77] while March-eq-100 won the award for handmade benchmarks.

SymChaff outperformed the other two solvers without symmetry breaking predicates in all but excessively easy instances. Generating symmetry breaking predicates from the input CNF formula was typically quite slow compared to a complete solution by SymChaff. The effect of adding symmetry breaking predicates 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 symmetry breaking predicates.

6.3.1 Problems from Proof Complexity

Pigeonhole Principle: php-n-m is the classic pigeonhole problem described in Example 6.1 for n pigeons and m holes. The corresponding formulas are satisfiable iff $n \leq m$. They are known to be exponentially hard for resolution [60, 94] but easy when the symmetry rule is added [76]. 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 in time $\Theta(m^2)$. Note that although it must read the entire input file containing $\Theta(nm^2)$ clauses, it does not need to process all of these clauses given the symmetry information. Although reading the input file is quite fast in practice, we do not include the time spent on it when claiming the $\Theta(m^2)$ bound.

This contrasts well with one of the fastest current techniques for this problem (other than the implicit pseudo-Boolean encoding) by Motter and Markov [89] which is based on ZBDDs and requires a fairly involved analysis to prove that it runs in time $\Theta(m^4)$ [90].

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 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 pseudo-Boolean techniques even when the clique part is encoded implicitly using pseudo-Boolean methods. Indeed, Pudlák [93] has shown it 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 in time $\Theta(k^2)$, again ignoring the time spent on reading the input file.

We note that this problem can also be solved in polynomial time using the group theoretic technique of Dixon et al. [48]. 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.

6.3.2 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 [51]. These were then converted into CNF formulas using the tool Blackbox version 4.1 by Kautz and Selman [72]. 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" PDDL description where an original PDDL declaration such as

(:OBJECTS $T_1\ \ T_2\ \ T_3$
$L_1^{\text{src}}\ \ L_2^{\text{src}}\ \ L_1^{\text{dest}}\ \ L_2^{\text{dest}}$
$P_{1,1}\ \ P_{2,1}\ \ P_{1,2}\ \ P_{2,2}$)

in the `PlanningC` example is replaced with

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

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

Example 6.7. 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 6.3. The "tag" in bold is the only change to the usual specification of the problem needed to process symmetry information automatically.

```
(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)
 ))
)
```

Figure 6.3: A sample PDDL file for `PlanningA` with $n = 3$

We are now ready to present the four application-oriented problems for which we have 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 iff $t \geq 4n - 1$.

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

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 iff $t \geq 7$.

SymChaff uses one symindex set corresponding to the trucks, and several variable classes. Here again symmetry breaking predicates, although not too hard to compute, provide 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 iff $t \geq 5$.

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 [4, 6]. 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 iff $t \geq n$.

Table 6.1: Experimental results on UNSAT formulas. ‡ indicates > 6 hours.

Problem + parameters		SymChaff	zChaff	March-eq	Shatter	zChaff after 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

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.

6.4 Discussion

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

Table 6.2: Experimental results on SAT formulas. ‡ indicates > 6 hours.

	Problem + parameters	SymChaff	zChaff	March-eq	Shatter	zChaff after Shatter
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

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 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 without using complicated group theoretic machinery. This allows us to overcome many drawbacks of previously proposed solutions.

We show, in particular, that straightforward tagging in the specification of planning problems is enough to automatically generate relevant symmetry information which in turn makes the search for an optimal plan much faster. SymChaff incorporates several new ideas that allow this to happen. These include 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 branches.

One limitation of our approach 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. Consider, for instance, 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 [53] 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 group of symmetries which, as observed in the work of Dixon et al. [48], can sometimes be quite expensive.