Symmetry Breaking for Boolean Satisfiability: The Mysteries of Logic Minimization

Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah
The University of Michigan, Ann Arbor
{faloul, imarkov, karem}@umich.edu

Abstract

Boolean Satisfiability solvers improved dramatically over the last seven years [14, 13] and are commonly used in applications such as bounded model checking, planning, and FPGA routing. However, a number of practical SAT instances remain difficult to solve. Recent work pointed out that symmetries in the search space are often to blame [1]. The framework of symmetry-breaking (SBPs) [5], together with further improvements [1], was then used to achieve empirical speed-ups.

For symmetry-breaking to be successful in practice, its overhead must be less than the complexity reduction it brings. In this work we show how logic minimization helps to improve this trade-off and achieve much better empirical results. We also contribute detailed new studies of SBPs and their efficiency as well as new general constructions of SBPs.

1 Introduction

Many search, synthesis and optimization problems arising in algorithmic applications exhibit symmetries. The presence of multiple, symmetric solutions may lead to degeneracy and slow down known algorithms for such problems. Symmetries can make it more difficult to conclude that a a given instance of a search problem has no solutions - because symmetric sub-instances may be independent. However, once the symmetries are identified, it is often easy for people to "mod out" by symmetry and simplify the problem at hand. Of course, when the number of symmetries is high, even simple bookkeeping requires a computer program.

In this work we study the Boolean SATisfiability problem (CNF-SAT) - one of the most important in Computer Science - in the presence of symmetry. Previous work contributed the framework of symmetry-breaking predicates that proceeds as follows.

1. The symmetries are identified using a reduction to the Graph Automorphism problem.
2. Symmetry-breaking predicates (SBP) are produced.
3. A SAT-solver is applied to the conjunction of the original formula and symmetry-breaking predicates.

A reduction to Graph Automorphism that detects all permutational symmetries was proposed in [5]. That work also contributed the first general construction of SBPs for single permutations in tabular form. The authors pointed out that constructing SBPs for every symmetry is often impractical, and developed the concept of *symmetry-tree* that reduces the number of clauses added to the original CNF formula. Yet it does not always avoid exponential number of added clauses.

The symmetry-breaking framework was further improved in [1] which contributed

- a new reduction to Graph Automorphism that detects all permutational symmetries, phase shifts and their compositions (we call them *mixed symmetries*),
- empirical evaluation of symmetry-breaking performed only for generators of the symmetry group,
- more efficient symmetry-breaking predicates for mixed symmetries in cycle notation,
- strong empirical evidence that symmetry-breaking is practically useful when best available SAT-solvers are used.

In this paper we further improve the symmetry-breaking framework for CNF-SAT using logic minimization to simplify SBPs and empirically improve runtime of SAT-solvers. We contribute new constructions of full and partial symmetry-breaking predicates of smaller size, produced in many cases by explicit logic optimization.

Analyses of symmetry-breaking include orbit counts and estimates of efficiency of symmetry-breaking predicates. Additionally, we provide some justification for breaking the symmetries of only the generators. A number of related questions are still open however.

The remaining part of the paper is organized as follows. Section 2 presents the necessary definitions and notation. Section 3 covers previous work. More efficient SBP constructions are given in Section 4, and a theoretical discussion of symmetry-breaking by generators in Section 5. We show experimental results in Section 6, and the paper concludes in Section 7.

2 Definitions and Notation

Intuitively, a symmetry of a discrete object is a transformation, e.g., permutation, of its components that leaves the object intact. For example, the six permutational symmetries of the equilateral triangle can be thought of as permutations of its vertices. Symmetries are studied in abstract algebra in terms of *groups* [6].

A *group* is a set with a binary associative operation (often thought of as multiplication) defined on it such that there is a *unit* element and every element has a unique inverse. A group is called Abelian if the operation is commutative, e.g., the *cyclic* group of order *k* consists of powers of *g* modulo *k*. In general, a set of group elements such that any other group element can be expressed as their product is called a *generating set*. Any irredundant generating set is no greater than the binary logarithm of the group size and can be much smaller. A *subgroup* of a group is a subset that is closed under the operation.

The symmetric group $S(\Omega)$ on a finite set Ω is the group of all permutations of Ω. If $|\Omega| = n$, the group is commonly denoted by S_n. A *homomorphism* from group *G* to group *H* is a mapping such that a product of two group elements is mapped to the product of their images, and such a mapping is called an *isomorphism* when its inverse exists and is a homomorphism. A group *G acts* on a set Ω when a homomorphism is given from *G* to $S(\Omega)$. For an element of Ω, its *G-orbit* is the set of elements of Ω to which it can be mapped by elements of *G*. Orbits define an equivalence relation on Ω.

Permutations of Ω, often denoted by lower-case Greek letters, can be written in *tabular form* where the elements of Ω are written in the first row and their images in the second row. For example, the image of element *i* under the permutation π will be denoted i^{π} and written below *i*. We also use *cycle notation*, which can be produced from the tabular notation by (i) constructing directed edges from elements of Ω to their images, and (ii) listing the disjoint cycles of this directed graph. Single-element cycles are implicit and never listed, and two-element cycles are called *transpositions*. For example, (12)(456) can denote a permutation that swaps elements 1 and 2, maps 4 to 5, 5 to 6 and 6 to 4. Cycle notation is preferable to tabular notation for *sparse* permutations that map most of elements of Ω to themselves. The set of elements that are not mapped to themselves is called the *support* of the permutation. The *cycle type* of a permutation is a sequence of integers n_i for $i \in \{2, \ldots, k\}$; n_i is the number of *i*-cycles in the permutation.

A symmetry (*automorphism*) of a graph is a permutation of its vertices that maps edges to edges. If vertices are labeled by integers (*colored*), one may additionally require that symmetries preserve labels.

Consider the set of Boolean variables $x_1, \ldots, x_n$. A *literal* is either a variable or its negation. A *clause* is a disjunction of literals, e.g., $(x_3 + \overline{x_6} + x_7)$, and a *CNF formula* is a conjunction of clauses, e.g., $(x_3 + \overline{x_6} + x_7)(x_{10} + \overline{x_{11}})(x_{20})$. A *binary* clause has two literals and can be viewed as an implication between variables, e.g., $(x_{11} \Rightarrow x_{10})$. An instance of the CNF-SAT decision problem is given by a CNF formula, the question is whether the formula can be satisfied by an assignment of Boolean values to the variables.

We will assume a total ordering of variables $x_1, \ldots, x_n$ and consider the induced lexicographic ordering of the 2^n truth assignments, i.e., 0-1 strings of length *n*. We now assume that a group acts on the set of literals, subject to the *Boolean consistency* constraint, which requires that if $(a \to b)$ then $(\bar{a} \to \bar{b})$ for any literals *a* and *b*. Such an action unambiguously induces a corresponding action on the set of truth assignments. We focus on orbits of this action. The *lex-leader* of an orbit is defined as the lexicographically smallest element. A *lex-leader predicate* (LL-predicate) for the action is a Boolean function on $x_1, \ldots, x_n$ that evaluates to true only on lex-leaders of orbits.

Consider a permutation on the set of literals. Given a CNF formula, we can permute literals in it, potentially changing the formula. A permutation of literals is a symmetry of a given CNF formula if Boolean consistency is observed and the formula is preserved under the permutation (in other words, every clause must map into a clause with the same polarities of literals). In particular, we consider simultaneous negations of sets of variables (*phase-shifts*) and compositions of permutations and phase-shifts (*mixed symmetries*). Given a CNF formula, we consider its group of symmetries and its corresponding action on truth assignments. A symmetry-breaking predicate (SBP) is a Boolean function that evaluates to true on at least one element of each orbit of the group of symmetries. In this work, we will consider SBPs that are expressed by CNF formulae, and the *size* of an SBP is the number of literals its CNF involves. Observe that adding an SBP to the original CNF formula does not affect the satisfiability, but restricts the possible solutions to those selected by the SBP.

A *full* SBP is an SBP that selects exactly one element of each orbit, otherwise we call an SBP *partial*. A *lex-leader SBP* (LL-SBP) is an SBP that selects lex-leaders only. An LL-SBP is a full SBP. For SBPs that are not full, it is often important that they select lex-leaders, among other elements. We call such SBPs *partial lex-leader SBPs* (PLL-SBPs).

3 Previous Work

3.1 The Natural LL-SBP of Crawford et al.

Let $x = (x_1, x_2, \ldots, x_n)$ be a vector of Boolean variables, $\Pi = \{\pi_1, \ldots, \pi_m\}$ be a permutation group acting on x. A PLL-SBP $\mathrm{PLL}(\pi)$ is defined for every group element, and their conjunction is an LL-SBP for Π [5]. The construction relies on the ordering $x_1 < x_2 < \ldots < x_n$:

$$\mathrm{PLL}(\pi) = \mathrm{AND}_{1 \le i \le n} C(\pi, i) \tag{3.1}$$

$$C(\pi, i) = [\mathrm{AND}_{1 \le j \le i-1}(x_j = x_j^{\pi})] \rightarrow (x_i \le x_i^{\pi}) \tag{3.2}$$

Introducing auxiliary variables $e_j = (x_j = x_j^{\pi})$, this predicate yields a CNF formula with $5n$ clauses and $0.5n^2 + 13.5n$ literals. The SBP for the entire group is:

$$\mathrm{LL}(\Pi) = \mathrm{AND}_{1 \le i \le m}(\mathrm{PLL}\pi_i) \tag{3.3}$$

This predicate may contain redundant clauses, in particular some $C(\pi, i)$ may be tautologies and $\mathrm{PLL}(\pi)$ for different permutations may have identical clauses. Moreover, there are often exponentially many symmetries, and the natural LL-SBP for the entire group is infeasible. Crawford et al. construct a *symmetry tree* to prune "redundant" symmetries, but that may still yield exponential-sized LL-SBPs. Additionally, they have shown NP-completeness of identifying lex-leaders in certain circumstances. A more recent work by Luks and Roy [9] studies the complexity LL-SBPs for Abelian permutation groups and shows that, in general, LL-SBPs may have to be exponentially large. However, careful re-ordering of variables enables polynomial-sized formulas.

3.2 Further Improvements

3.2.1 CNF Symmetries via Graph Automorphism

While CNF symmetries are not limited to permutations of variables, we build, for a given CNF formula, a graph such that the group of CNF symmetries is isomorphic to the group of graph automorphisms. A simple construction represents every clause by a vertex of color 2, and every variable by two vertices of color 1 (one for positive literals and one for negative) connected by *Boolean consistency edges*. Every literal in the CNF formula then is represented by a bi-partite edge. An improved construction treats binary clauses differently. It leaves out their clausal vertices and connects their literal vertices by double-edges. Since some graph automorphism programs do not allow double-edges, the work in [1] uses a model with single edges which can result in spurious graph automorphisms (one-sided error) if the original CNF formula contains binary clauses forming circular chains of implications. Fortunately, this rarely happens in CNF applications and spurious graph symmetries can be easily tested for. In our experiments, no spurious symmetries were detected.

The graph automorphism problem is believed to be outside P, yet not NP-complete. Common algorithms finish in linear time if the only symmetry is trivial, and polynomial time is guaranteed in the bounded-degree case [2,3]. Finding CNF symmetries is often easier than solving SAT, and excellent software is available [11, 12].

3.2.2 Breaking Symmetries By Generators

Breaking *all* symmetries may not speed up search because there are often exponentially many of them and their PLL-SBPs may be redundant [5]. It is not necessary either. Breaking *enough* symmetries, whose SBPs are short CNF clauses, may provide a better trade-off.

Irredundant generators are good candidates for symmetries to be broken because they cannot be expressed in terms of each other, which minimizes redundancies. A potential concern is that breaking generators alone may lead to the selection of more than one element from some orbits, an example with $|\Omega| = 4$ was provided to us by Eugene Goldberg. Yet, such partial symmetry breaking may be the best option because its overhead is small. When building LL-SBPs for different generators, one must use the same variable order. An LL-SBP for a given symmetry generator can be constructed as the natural LL-SBP for the cyclic group generated by it, or a PLL can be used.

3.2.3 Breaking Symmetries by Cycles

For a given permutation, choosing an SBP with fewer literals is also important for empirical success. To this end, the construction of small SBPs from [1] is based on the cycles of a permutation (with subsequent chaining) instead of the entire permutation in tabular form. This eliminates redundancies in the SBPs of individual permutations and is convenient because generators returned by graph automorphism programs are often sparse, i.e., involve very few variables. Additionally, one can create a catalog of LL-SBPs or PLL-SBPs for small cycle lengths. This improves upon the natural LL-SBP construction which entails a conjunction over non-trivial powers of a given permutation.

A major problem with cycle-based constructions arises when they are applied to symmetry generators and may influence the order of variables. LL-SBPs for all generators must use the same order of variables, but not necessarily the original order. In some applications the original order of variables works, and in many an appropriate order can be found quickly.

4 New Constructions for Single Perms

The main idea behind our new constructions is to decompose a permutation into cycles, construct SBPs for individual cycles and then chain them together.

4.1 Extensions and Improvements of The Natural LL-SBP Construction

We describe in this section a) an extension to Crawford's construction of the LL-SBP for a single permutation to enable the handling of mixed symmetries, and b) a simplification of the LL-SBP that yields a linear-sized CNF formula.

Without loss of generality, every symmetry that is a composition of permutations and pure phase shifts can be decomposed into a product of a permutation and a phase-shift acting after the permutation. That is because the group of phase shifts is *normal* in the group of compositions, i.e., the composition of an arbitrary permutation, an arbitrary phase shift and the permutation's inverse is necessarily a phase shift (not necessarily the same as the original). Such mixed symmetries can be incorporated in the construction of the LL-SBP by allowing a variable x_i to be mapped into either its image under the permutation x_i^{π} or the negation of that image $\overline{x_i^{\pi}}$ induced by the phase shift action.

Crawford's construction in (3.1) and (3.2) for the LL-SBP of a single-permutation can be simplified to yield a CNF formula whose size is linear, instead of quadratic, in the number of variables. To facilitate the following derivation let $l_i = (x_i \le x_i^{\pi})$, $g_i = (x_i \ge x_i^{\pi})$, and $g_0 \equiv 1$. Noting that $e_i = l_i g_i$, Crawford's LL-SBP can now be expressed as:

$$(g_0 \to l_1)(g_0 l_1 g_1 \to l_2)\dots(g_0 l_1 \dots l_{n-1} g_{n-1} \to l_n) \quad (4.1)$$

Factoring out the common prefix g_0 yields:

$$g_0 \to [l_1 \cdot (l_1 g_1 \to l_2)\dots(l_1 g_1 \dots l_{n-1} g_{n-1} \to l_n)] \quad (4.2)$$

which simplifies further, through absorption, to:

$$g_0 \to [l_1 \cdot (g_1 \to l_2)\dots(g_1 l_2 \dots l_{n-1} g_{n-1} \to l_n)] \quad (4.3)$$

The recursive structure of the formula is now revealed by comparing (4.1) and (4.3). Let $p_1, \dots, p_n$ be a sequence of predicates defined by:

$$\begin{aligned} p_i = {} & (g_{i-1} \to l_i) \cdot (g_{i-1} l_i g_i \to l_{i+1}) \\ & \dots(g_{i-1} l_i g_i \dots l_{n-1} g_{n-1} \to l_n) \end{aligned} \quad (4.4)$$

and let $p_{n+1} = 1$. Then,

$$p_i = g_{i-1} \to l_i p_{i+1} \qquad i = 1, \dots, n \quad (4.5)$$

Note that predicate p_1 represents the entire formula (4.1). The satisfiability of (4.1) can, thus, be determined by checking the satisfiability of the following equivalent, but simpler formula:

$$(p_1)(p_1 = g_0 \to l_1 p_2)\dots(p_n = g_{n-1} \to l_n p_{n+1}) \quad (4.6)$$

One final simplification replaces the equalities in (4.6) with implications since we are only interested in satisfying each of the predicates. We thus obtain:

$$(p_1)(p_1 \to g_0 \to l_1 p_2)\dots(p_n \to g_{n-1} \to l_n p_{n+1}) (4.7)$$

The CNF representation of (4.7) consists of $2n$ 3-literal and $2n$ 4-literal clauses for a total size of $14n$ literals.

4.2 Orbits of Cyclic Symmetries

While the cyclic group of size n naturally acts on the set 1..n, this action can be extended to the Boolean cube of 2^n truth assignments. Orbits of this action are called *necklaces* in combinatorics, and their number can be found using the celebrated counting theorem by Polya [6, Theorem 14.5]:

$$B(n) = (1/n)\sum_{d|n} 2^d \phi(n/d) \quad (4.8)$$

where the sum is taken over all divisors of *n*. $\phi(m)$ is the *Euler's totient function,* i.e., the number of positive integers not exceeding *m* which are relatively prime to *m* (1 is counted as being relatively prime to all numbers). The first several values (for *n=2,3,4,5*) of $B(n)$ are 3,4,6,8 and can be verified by direct counting. For prime *p*, $\phi(p) = p-1$ and thus the number of orbits is $(2(p-1)+2^p))/p$. Indeed, every orbit contains *p* elements, except for the two one-element orbits *00...0* and *11...1*. In general, orbit sizes must divide cycle length, and the number of necklaces is lower-bounded by $(2(n-1)+2^n))/n$. Thus it grows exponentially. The asymptotic estimate $\Theta(2^n/\mathrm{n})$ for the number of necklaces can be produced using $\phi(n/d) \le n-1$. Therefore the efficiency of full SBPs for single cycles is not bounded, at least as far as reductions in search space are concerned.

4.3 Full Symmetry-Breaking for Single Cycles

If for every literal in a given cycle, its complementary literal is not in the cycle, the natural LL-SBP construction applies and can be improved as described above. Otherwise, we must apply the extended natural LL-SBP construction.

Lemma 4.1 If a symmetry of a CNF formula has a cycle of length n that contains literals a and $\bar{a}$, then

1. Every variable participating in the cycle, has both positive and negative literal in the cycle.
2. n must be even.
3. The distance between every pair of complementary literals is *n/2*.
4. Any contiguous sub-word of length *n/2* determines the remaining part of the cycle.

Proof: 1. The literal a can map to any literal b in the cycle, therefore by Boolean consistency $\bar{a}$ must map to $\bar{b}$, and $\bar{b}$ must be in the cycle. 2. Follows from 1. To prove 3. notice that the hop-distance from a to $\bar{a}$ in the cycle must equal that from $\bar{a}$ to a (by Boolean consistency), and the two add up to n. 4. Any contiguous sub-word of length *n/2* contains exactly one literal of every variable participating in the cycle and unambiguously determines where it maps (the last literal in the word must map to the complement of the first literal); the images of literals not in the sub-word are unambiguously determined by Boolean consistency.

Minimal LL-SBPs can be created for single-cycle permutations by constructing the Boolean function that selects the lex-leader of each orbit and minimizing this function using standard logic minimization procedure ESPRESSO[10]. Alternatively, a natural PLL-SBP can be created according to (3.1) for each permutation in the cyclic group generated by the given cycle; the minimal LL-SBP for the cycle is now obtained by minimizing the conjunction of these formulas. A listing of minimal LL-SBPs for *k*-cycles of various length is give in table Table 1. It is interesting to note that the LL-SBP for cycles whose length is less than 6 is composed entirely of binary clauses. We produced minimal LL-SBPs for cycles of length up to 20. Such LL-SBPs contain clauses of increasing lengths, but also k binary clauses. Since longer clauses are less effective during search, one may prefer to use a PLL-SBP that consists only of binary clauses.

4.4 Partial SBPs For Single Cycles

This is motivated by the desire to create strong predicates that can speed up search.

Theorem 4.2 A partial LL-SBP for the n-cycle is possible with $2n-2$ binary clauses.

Proof: Suppose we have a symmetry which is an n-cycle on variables $x_1, \ldots, x_n$. A set of binary clauses is defined by a partial order on variables, and we choose the partial order in which $x_1 \le \text{other vars}$ and $x_n \ge \text{other vars}$. We need to show that such a set of clauses picks at least one representative from each necklace class, in particular that lex-leaders satisfy every clause. The former is accomplished by giving an algorithm that for an arbitrary truth assignment finds a rotationally symmetric assignment that satisfies every clause. We then modify the algorithm to yield lex-leaders only, and show that they satisfy all clauses as well.

For an arbitrary truth assignment, consider two cases: (1) two variables are assigned different values, (2) otherwise. In case (2) all values are the same, i.e., all 0s or all 1s. These two truth assignments satisfy all clauses. In case 1, we can find two "neighboring" variables $x_i$and x_{i+1} that are assigned different values. Moreover, we can even find two "cyclically-neighboring" variables $x_i$and $x_{(i \bmod n)+1}$ such that the left variable =1 and the right variable =0, e.g., $x_1 = 1, x_2 = 0$ and $x_n = 1, x_1 = 0$ are valid examples. Any such assignment can be further rotated (by applying the cyclic symmetry) to an assignment with $x_n = 1, x_1 = 0$, which satisfies all clauses regardless of the values assigned to other variables.

We now need to show that every lex-leader satisfies the constructed SBP. Indeed, assume a lex-leader distinct from 000...0 such that $x_1 = 0, x_n = 0$. Then rotate the truth assignment in the direction from x_n to x_1 until $x_n = 1$. Each such rotation must lead to a *lexicographically smaller* representative. Contradiction.

This proof suggests a technique of identifying partial symmetry-breaking predicates. The key is finding a "canonical form" to which any truth assignment can be

Table 1: Minimal LL-SBPs for k-cycles

Perm	LL-SBP
$(x_1 x_2)$	$(\bar{x}_1 + x_2)$
$(x_1 x_2 x_3)$	$(\bar{x}_1 + x_2)(\bar{x}_2 + x_3)$
$(x_1 x_2 x_3 x_4)$	$(\bar{x}_1 + x_2)(\bar{x}_1 + x_3)(\bar{x}_2 + x_4)(\bar{x}_3 + x_4)$
$(x_1 x_2 x_3 x_4 x_5)$	$(\bar{x}_1 + x_2)(\bar{x}_1 + x_3)(\bar{x}_2 + x_4)$ $(\bar{x}_2 + x_5)(\bar{x}_3 + x_5)$
$(x_1 x_2 x_3 x_4 x_5 x_6)$	$(\bar{x}_1 + x_2)(\bar{x}_1 + x_3)(\bar{x}_1 + x_4)$ $(\bar{x}_3 + x_6)(\bar{x}_4 + x_6)(\bar{x}_5 + x_6)$ $(\bar{x}_2 + x_3 + x_4)(\bar{x}_2 + \bar{x}_3 + x_5)$

reduced (not necessarily uniquely) by rotations. In the example above, the canonical form was defined by particular values of only two variables. More refined canonical forms can be defined by inequalities, more involved arithmetic predicates, etc.

4.5 k-Cycle Chains

In this section we deal with collections of cycles of the same length. We first assume that no cycles in this collection include complementary literals.

Step one. Break the first cycle.

Step two. Add a chaining predicate. A chaining predicate is an AND of sub-predicates. Each sub-predicate is of the form $A_i(\dots) \Rightarrow B_i(\dots)$. The variable *i* traverses all divisors of the cycle length, excluding itself. The A_i predicate depends on the variables of the broken cycle, and B_i depends on all variables of the remaining cycles. A_i is true when the truth assignment is in an orbit of length *i*. For example A_1 checks that all variables have identical values. A_2 is only needed for even cycles, it checks that all even-numbered variables have the same values and that all odd-numbered variables do. In general, A_i checks that any two variables with indices having the same remainder modulo *i* must have equal values. Thus the number of clauses is linear in cycle length. B_i is a symmetry-breaking predicate for power-*i* of the remaining cycles. It must be produced recursively. Observe that recursion is guaranteed to terminate because at every call either (i) the size of cycles decreases, OR (ii) the number of cycles decreases,... OR both. Note that the size of cycles cannot increase. It will stay the same if and only if the cycle length is prime, in which case the number of cycles decreases. The number of cycles can increase, but this can only happen when cycle length is composite, in which case the size of cycles must decrease.

Special cases may be simplified further simplified. For example, for cycles of prime length, we only have *i=1*, and chaining is particularly simple. In particular, cycles of length two appear very often, and an example is given in the next Section. In the general case, one can construct a PLL-SBPs by only including $A_1(\dots) \Rightarrow B_1(\dots)$, or using any subset of *i*'s.

As mentioned above, the number of orbits of an *n*-cycle is $o(2^n)$, thus the efficiency of symmetry-breaking is not limited in this case. This is not true, however, for *n* 2-cycles. The efficiency of symmetry-breaking is limited by 50% because the efficiency of every new cycle is 2x smaller than the efficiency of the previous cycle. For example, for one 2-cycle, there are 3 lex-leaders out of 4 assignments. For two 2-cycles, there are 4+2*3=10 lex-leaders out of 16 assignments, for three 2-cycles, there are 16+2*10=36 lex-leaders out of 64 assignments. For *k* 2-cycles, the number of lex-leaders is $P(k) = 4^{k-1} + 2P(k-1)$, and therefore, by induction. $0.5 \times 4^k \le P(k) \le 0.75 \times 4^k$. Moreover, as k increases $P(k) \to 0.5 \times 4^k$. In other words, *k* 2-cycles, asymptotically remove 50% of the solution space.

We now generalize our techniques to handle complementary literals. Our method of handling chains of cycles of length two must be extended to cycles of the form $(a\bar{a})$. Since a cycle of this form implies an SBP that breaks exactly 50% of the solution space, we should stop right there and ignore all other 2-cycles. In other words, when we have a chain of 2-cycles, we must first scan it for same-literal cycles. To chain longer cycles with complemented literals, we conjoin implications as described earlier.

4.6 Arbitrary Cycle Types

Constructions of SBPs for multiple cycles of the same length (described above) can be used as subroutines in SBP constructions for arbitrary permutations as follows.

Algorithm: Start with an empty SBP.

Step 1. Produce an SBP for all cycles of the shortest length *k*. Conjoin it to the previously accumulated SBP.

Step 2. Compute the *k*-th power of the permutation.

Step 3. If non-trivial cycles are left, start over (at Step 1) with the k-th power.

Example. Consider *(ab)(cd)(efg)(hijklm)*.

An LL-SBP for the permutation *(ab)(cd),* e.g., $(a+\bar{b})(a+b+x)(\bar{a}+\bar{b}+x)(\bar{x}+c+\bar{d})$ must be conjoined with n LL-SBP for the square of the original permutation *-(efg)(hjl)(ikm)*.

Theorem 4.3 Suppose that Step 1 in the algorithm above produces partial LL-SBPs. Then the algorithm, too, yields partial LL-SBPs.

Proof: First, we describe the order of variables for which the constructed SBP is a partial LL-SBP.

1. Variables from earlier cycles must appear earlier.
2. The relative order of variables from cycles of the same length must be that for which SBP from Step 1 are LL-SBPs.

The remaining part of the proof describes lex-leaders of orbits of the given permutation (with respect to the given ordering) and verifies that the constructed SBP evaluate to true on them.

In order to produce a lex-leader of a given truth assignment, we need to apply the permutation as many times as it takes. Now, suppose that the shortest cycles have length *k*. Since variables from those cycles go first, we first need to apply the permutation (less than *k* times) to choose lex-leaders for that part of the truth assignment. From now on, the values of those variables must be

fixed. Conveniently, applying the permutation *Nk* times (*N* is an integer) does not change the values of those variables. In other words, we need to find out how many times the *k*-th power must be applied to find a lex-leader in terms of the next batch of variables.
Because we are considering partial LL-SBPs for the first batch of variables independently of other variables, partial LL-SBPs can be used without modifications. The rest is by induction.
Observe that even when SBPs produced at step 1are LL-SBPs, the SBP constructed by the algorithm may not be full. In particular, it may select more than lex-leaders from orbits where the first batch of variables have equal values. That is because on such orbits the second batch of variables can be subject to arbitrary powers of the permutation. The same applies to further batches of variables and can be remedied by adding more symmetry-breaking clauses similar to those used in Section 4.5. In particular, for cycles of length *k* we need to distinguish orbits whose lengths are less than *k*, which leads to complications for non-prime *k*. However, we believe that this extended construction yields LL-SBPs.

5 Symmetry-breaking by Generators

The LL-SBP construction of Crawford et al. entails conjoining symmetry-breaking predicates built for each element of the symmetry group. While this maximally reduces search space, the set of added clauses may be overwhelmingly large. Therefore, the total search time may be increased. This section discusses the idea of conjoining only symmetry-breaking clauses of symmetry-generators - a potentially more practical alternative.

5.1 Generators and Subgroups

Lemma 5.1 Given a permutation, a symmetry-breaking predicate that distinguishes exactly one element in every orbit (i.e., fully breaks a given symmetry) also fully breaks every symmetry generated by this permutation, i.e., every power of this permutation.

Proof: The orbits of one permutation and the group it generates are identical.

We now consider groups with more than one generator. Each generator is modeled by a cyclic sub-group.

Lemma 5.2 Consider a group *G* and its subgroups $\{H_i\}$ For an element of a *G*-orbit to be a lex-leader, it is necessary that it be the lex-leader in each of its H_i-orbits.

Proof: For every subgroup *H* of group *G*, *G*-orbits can be decomposed into a disjoint union of *H*-orbits. Therefore, for an element to be a lex-leader in its *G*-orbit, it must be a lex-leader in its *H*-orbit.

Corollary 5.3 A conjunction of (not necessarily full) lex-leader SBPs for sub-groups is a valid lex-leader SBP. However, it may not be a full SBP, and therefore not an LL-SBP.
Consider a group *G* and its subgroups $\{H_i\}$. Suppose that we are given SBPs for all subgroups. A conjunction of those SBPs may not be a valid SBP because there may be no representative in a given *G*-orbit that is picked as a representative in all of its H_i-orbits.

Lemma 5.4 A special case of Lemma 4.2 happens when all subgroups are cyclic and generated by generators of G. Lemmas 4.1 and 4.2 then imply that on every *G*-lex-leader, full SBPs for individual generators must evaluate to true. Therefore, a conjunction of SBPs for generators is a valid SBP.

Theorem 5.5 Consider a group *G* and its subgroups *H* and *K*, such that *G=HK* and *H* normal. Observe that *K* acts on *H*-orbits by multiplication on the left. We require that this action map *H*-lex-leaders to *H*-lex-leaders. In this case, the conjunction of full lex-leader SBPs for *H*-orbits and *K*-orbits is a full lex-leader SBP for *G*-orbits.

Proof: Take an arbitrary *G*-orbit and its element *p* that is the lex-leader of its *H*-orbit and its *K*-orbit. Let *q* be the lex-leader in *p*'s *G*-orbit. Find a $g \in G$, $h \in H$, $k \in K$ such that *g(p)=q* and *g=hk*. Then *q=h(k(p))* and *k(p)* is in the same *H*-orbit as *q*. Since *q* is the lex-leader of its *H*-orbit, either *h=e* or *k(p)* is not an *H*-lex-leader. The latter is impossible because *p* is an *H*-lex-leader and the action of *K* by multiplication on the left preserves *H*-lex-leaders. On the other hand, if *h=e*, then *q=k(p)* and *p* is in the same *K*-orbit as *q*. However, both *p* and *q* are lex-leaders of their *K*-orbits. Therefore *k=e* and *p=q*.

Corollary 5.6 Consider two permutations π_1 and π_2 with *disjoint support*. The conjunction of full lex-leader SBPs for π_1 and π_2 is a full lex-leader SBP for the Abelian group generated by those permutations.

5.2 Requirements for Variable Ordering

As mentioned in Section 3.2.3, SBPs built for individual permutations in terms of cycle notation require, in general, different variable orderings. Therefore such SBPs for symmetry generators may be incompatible. Instead, the natural LL-SBP construction can be used for cyclic groups generated by symmetry generators or, if the order of a generator is large, the natural PLL-SBP construction can be used.

For some structured CNF formulas, such as hole-*n* instances defined below, the original order of variables is compatible with cycle-based SBPs for all generators. Otherwise, careful analyses of supports of symmetry generators from a given set may still allow using more efficient LL-SBPs or PLL-SBPs for some generators.

For example, if all supports are disjoint, then it is easy to construct an overall variable ordering consistent with cycle-based SBPs. More generally, one can build the *support intersection graph* of a generating set where each generator is represented by a vertex and edges connect vertices when the supports of generators have non-empty intersections. A maximal independent set (MIS) in this graph flags generators for which cycle-based SBPs can be used. This introduces a partial variable order, which can be arbitrarily extended to an order of all variables so that the natural LL-SBPs or PLL-SBPs can be used for remaining generators.

5.3 Symmetric Groups and The Pigeonhole Principle

Lemma 5.7 [9] A full lex-leader SBP for the symmetric group S_n can be constructed by conjoining SBPs for *n-1* generating transpositions *(12), (23),...,(n-1 n)*. This SBP contains *n-1* binary clauses.

Proof: Because S_n can map any linear order on *n* variables to any order, its orbits over truth assignments are distinguished only by the number of zeros. Thus, there are *n+1* orbits, and lex-leaders can be chosen by requiring that the values of the variables form a non-decreasing sequence. This entails *n-1* binary clauses $(x_1 \le x_2)(x_2 \le x_3)\ldots(x_{n-1} \le x_n)$.

Lemma 5.8 When, in addition to permutational symmetries, we allow all phase-shift symmetries and their compositions, a full lex-leader SBP exists with n one-literal clauses.

Proof: It can be seen that the number of zeros is not an orbit invariant anymore because the group acts transitively on the Boolean cube, which becomes one orbit. The lex-leader is therefore the truth assignment *000...0*, and the LL-SBP consists of *n* one-literal clauses.

Recall that the pigeon-hole principle states that *n+1* objects (pigeons) cannot be assigned to *n* slots (holes). This is easily proven by induction. However, the pigeon-hole principle can also be phrased as an unsatisfiable instance of Boolean satisfiability (hole-*n* instances) and proven by SAT-solving algorithms for specific values of *n*, in which case induction is unavailable. The *n(n+1)* indicator-variables encode assignments of pigeons to holes. $n^2(n+1)/2$ binary mutual-exclusion clauses form *n* families - one per hole, and ensure that no two pigeons are in the same hole. Another family consists of *n+1 n*-literal clauses, one per pigeon, ensuring that every pigeon is assigned to least one hole. It has been proven [4] that no polynomial-length resolution proofs of the pigeon-hole principle exist. Because of this, the dominant SAT-solving frameworks due to Davis-Putnam (DP) and Davis-Longeman-Loveland (DLL) cannot solve the hole-*n* CNF instances, regardless of implementation. This agrees well with empirical results for best available SAT-solvers, such as Chaff [13]. However, short induction-less proofs of the pigeon-hole principle can be constructed using symmetry [8]. Observe that the group of symmetries of the hole-*n* instance is a Cartesian product of S_n and S_{n+1} because “all holes are symmetric” and, independently, “all pigeons are symmetric”.

Theorem 5.9 For a hole-*n* instance, the conjunction of LL-SBPs for symmetry sub-groups S_n and S_{n+1} is <u>not</u> an LL-SBP. As a consequence, it is <u>not true</u> in general that a conjunction of LL-SBPs of groups *G* and *H* is an LL-SBP of $G \times H$.

Proof: We order the variables in the “hole-major” order, i.e., in *n* groups of *n+1*. Observe that hole-symmetries permute those groups, while pigeon-symmetries simultaneously permute variables inside each group. By applying pigeon-symmetries to an arbitrary truth assignment, we can permute the values of any two variables within the first group of n+1, and if their values are equal, we can permute the two variables having the same offsets in the second group, etc. Thus one can sort variables in the first group (zeros first), but this will bipartition variables in other groups. One can then sort each partition, which will create further partitions. Independently, hole-symmetries allow lexicographically sorting the *n* groups. An LL-SBP for pigeon-symmetries selects lex-leaders with respect to the above “pigeon-sort”. An LL-SBP for hole-symmetries selects lex-leaders with respect to the above “hole-sort”. The conjunction of such LL-SBPs will pick truth assignments that are both. However, some of those are not lex-leaders with respect to the Cartesian-product group. We give an example for hole-2, which has six variables, divided into two groups of three. The truth assignment (011,100) is a lex-leader with respect to both pigeon-sort and hole-sort. Yet, it can be mapped to a lex-smaller assignment - (001,110) - by applying the pigeon-symmetry that sorts variables 3-6 and the only non-trivial hole-symmetry.

Table 2: Search runtimes of CNF-SAT instances with and without PLL-SBPs. The symmetry detection runtime and the number of symmetry generators are also included.

	Symmetries		Chaff [13] Runtime (sec)				Speedup of NEW		
	Finding (sec)	#Generators	Orig	Craw	DAC	NEW	Orig	Craw	DAC
Urq3_5	0.27	29	1000	1000	0.01	0.01	>100K	>100K	1
Urq4_5	1.64	43	1000	1000	0.01	0.01	>100K	>100K	1
Urq5_5	14.6	72	1000	1000	0.01	0.01	>100K	>100K	1
Urq6_5	70	109	1000	1000	0.02	0.02	>50K	>50K	1
Urq7_5	186	143	1000	1000	0.02	0.02	>50K	>50K	1
hole7	0.01	13	0.33	0.05	0.01	0.01	33.10	5	1
hole8	0.01	15	1.05	0.08	0.01	0.01	151	11.48	1.43
hole9	0.23	17	3.94	0.13	0.01	0.01	393.70	12.87	1
hole10	0.35	19	21.0	0.22	0.01	0.01	2104	22.08	1
hole11	0.33	21	207	0.32	0.02	0.01	19463	30.12	1.54
hole12	0.42	23	1000	0.50	0.02	0.01	>100K	50.14	2
hole15	1.32	29	1000	1.39	0.04	0.02	44262	61.52	1.56
hole20	10.3	39	1000	5.98	0.09	0.04	25000	149.5	2.23
hole30	189	59	1000	52.23	0.30	0.16	6250	326.4	1.88

Table 3: Size of SPBs using the natural PLL-SBP, the construction from [1], and the proposed approach

	Natural PLL-SBP[5]			Construction from [1]			NEW		
	#V	#C	#Lits	#V	#C	#Lits	#V	#C	#Lits
Urq3_5	0	0	0	0	29	29	0	29	29
Urq4_5	0	0	0	0	43	43	0	43	43
Urq5_5	0	0	0	0	72	72	0	72	72
Urq6_5	0	0	0	0	109	109	0	109	109
Urq7_5	0	0	0	0	143	143	0	143	143
hole7	728	3640	30212	84	433	1517	143	366	808
hole8	1080	5400	53460	112	575	2074	179	278	1068
hole9	1530	7650	89505	144	737	2734	1302	466	1364
hole10	2090	10450	143165	180	919	3503	199	578	1696
hole11	2772	13860	220374	220	1121	4387	241	702	2064
hole12	3588	17940	328302	264	1343	5392	287	838	2468
hole15	6960	34800	929160	420	2129	9193	449	1318	3896
hole20	16380	81900	3660930	760	3839	18508	799	2358	6996
hole30	54870	274350	26255295	1740	8759	51013	1799	5338	15896
Total	89998	449990	31710403	3924	20251	98717	5398	12638	36652

6 Experimental Results

In this section, we empirically show the advantage of using smaller SBPs. The experiments were performed on an AMD Athlon 1.2 GHz machine with 1 GB of RAM running Linux. The runtime limit for all experiments was set to 1000 seconds. The benchmarks included hole-*n* instances and artificially constructed randomized Urquhart (Urq) instances [17]. We used the best available back-track SAT solver Chaff [13]. Since Chaff is randomized, its runtime varies, and we averaged all results over ten independent runs. Cursory analysis of distributions of results reveals normal-looking distributions and suggests that the averages we report are, indeed, representative.

Table 2 lists symmetry detection runtimes and the number of symmetry generators. We use the reduction to graph automorphism from [1] which detects a wider range of symmetries than that from [5] (the latter construction does not find any symmetries in the Urq instances, as seen from Table 3). The graphs are then passed to GAP/GRAPE/NAUTY [16,15,12]. Table 2 also compares SAT-solving runtimes for the original CNF instance and the instances augmented with symmetry-breaking predicates using the natural PLL-SBPs [5], the cycle-based SBPs [1], and the proposed approach. Clearly, the addition of SPBs significantly reduces the search runtime, and the proposed approach leads to the greatest savings in runtime.

Table 3 compares the size of predicates produced by constructions from [5], [1] and this work. Our construction entails the smallest number of variables, clauses, and literals for all instances analyzed.

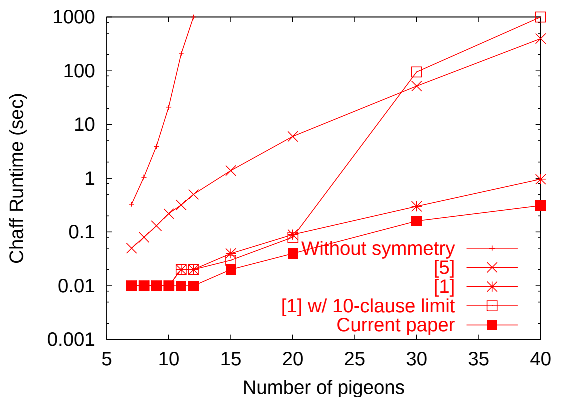

Figure 1: Comparison of runtimes for different SBP constructions for instances of the Pigeonhole principle.

7 Conclusions and Future Work

In this work we extended and improved the framework of symmetry-breaking predicates for solving Boolean Satisfiability by using logic minimization to construct more efficient CNF representations of symmetry-breaking predicates. Our constructions of symmetry-breaking predicates are in terms of cycles of permutations, which is particularly convenient when existing software for the graph automorphism problem is used. The proposed techniques lead to empirical speed-ups in back-track search for the best available SAT solvers, e.g., as shown in Figure 1.

Additionally, we gave new analyses of symmetry-breaking, including (i) estimates of efficiency of symmetry-breaking predicates, (ii) justification of symmetry-breaking by generators, and (iii) counterexamples to intuitive conjectures. Future work includes further efficiency improvement of symmetry-breaking predicates, more systematic studies of proposed constructions and empirical evaluation on applications arising the field of Electronic Design Automation.

We are pursuing improved algorithms for symmetry detection, e.g., along the lines of [1], where new techniques for opportunistic symmetry detection were proposed.

Acknowledgements. This work is funded by an Agere Systems/Semiconductor Research Corporation graduate fellowship and the DARPA/MARCO Gigascale Silicon Research Center (GSRC).

8 References

[1] F. Aloul, A. Ramani, I. Markov, K. Sakallah, “Solving Difficult SAT Instances in the Presence of Symmetries,” *DAC 2002*. pp. 731-736.

[2] L. Babai and E. M. Luks, ``Canonical Labeling of Graphs", *ACM Symp. on the Theory of Computing*, April 1983, pp. 171-183.

[3] L. Babai, “Automorphism Groups, Isomorphism, Reconstruction”, Chapter 27, pp. 1447-1541, In (R. L. Graham, M Grötschel and L. Lovász, eds, Handbook of Combinatorics, vol. 2, MIT Press, 1995).

[4] P. Beame, R. Karp, T. Pitassi and M. Saks, “The efficiency of Resolution and Davis-Putnam Procedure”, to appear in *SIAM Journal on Computing*. http://www.cs.washington.edu/homes/beame/papers/resj.ps..

[5] J. Crawford, M. Ginsberg, E. Luks and A. Roy, “Symmetry-breaking predicates for search problems”, *5th Intl Conf. Principles of Knowledge Representation and Reasoning (KR’96)*, Cambridge, MA, pp. 148-159.

[6] I. M. Gessel and R.P. Stanley, “Algebraic Enumeration”, Capter 21, p. 1059, In (R. L. Graham, M Grötschel and L. Lovász, eds, Handbook of Combinatorics, vol. 2, MIT Press, 1995).

[7] M. Hall Jr., ``The Theory of Groups", McMillan, 1959.

[8] B. Krishnamurthy, ``Short proofs for tricky formulas", *Acta Informatica*, 22:327-- 337, 1985.

[9] E. Luks and A. Roy, ``Symmetry Breaking in Constraint Satisfaction", *Intl. Conf. of Artificial Intelligence and Mathematics*, Ft. Lauderdale, FL, Jan 2-4, 2002.

[10] P. McGeer, J. Sanghavi, R. K. Brayton and A. Sangiovanni-Vincentelli, “ESPRESSO-Sgnature: A New Exact Minimizer for Logic Functions”, *IEEE Trans. on VLSI*, vol. 1, no. 4, pp. 432-440, December 1993.

[11] B. D. McKay, “Practical Graph Isomorphism”, *Congressus Numerantium*, 30 (1981), pp. 45-87.

[12] B. D. McKay, “Nauty user’s guide” (version 1.5), Technical report TR-CS-90-02, Australian National University, Computer Science Department, ANU, 1990. http://cs.anu.edu.au/~bdm/nauty/

[13] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik, “Chaff: Engineering an Efficient SAT Solver”, *DAC 2001*.

[14] J. P. M. Silva and K. A. Sakallah, “GRASP: A New Search Algorithm for Satisfiability”, *IEEE Trans. On Computers*, vol. 48, no. 5, May 1999.

[15] L. H. Soicher, “GRAPE: A System For Computing With Graphs and Groups”, *in "Groups and Computation"* (L. Finkelstein and W.M. Kantor, eds), *DIMACS Ser. in Discr. Math. & Theor. Comp. Sci.* 11, pp. 287-291.http://www.gap-system.org/~gap/Share/grape.html

[16] E. L. Spitznagel, “Review of Mathematical Software, GAP”, *Notices Amer. Math. Soc.*, 41 (7), (1994), pp. 780-782. http://www.gap-system.org

[17] A. Urquhart, “Hard Examples for Resolution”, *JACM*, vol. 34, 1987.

[18] A. Urquhart, ``The Symmetry Rule in Propositional Logic", 1996.