Shatter: Efficient Symmetry-Breaking for Boolean Satisfiability

Fadi A. Aloul, Igor L. Markov, Karem A. Sakallah
Department of EECS, University of Michigan
{faloul, imarkov, karem}@eecs.umich.edu

Abstract

Boolean satisfiability (SAT) solvers have experienced dramatic improvements in their performance and scalability over the last several years [5, 7] and are now routinely used in diverse EDA applications. Nevertheless, a number of practical SAT instances remain difficult to solve [9] and continue to defy even the best available SAT solvers [5, 7]. Recent work pointed out that symmetries in the Boolean search space are often to blame. A theoretical framework for detecting and breaking such symmetries was introduced in [2]. This framework was subsequently extended, refined, and empirically shown to yield significant speed-ups for a large number of benchmark classes in [1].

Symmetries in the search space are broken by adding appropriate symmetry-breaking predicates (SBPs) to a SAT instance in conjunctive normal form (CNF). The SBPs prune the search space by acting as a filter that confines the search to non-symmetric regions of the space without affecting the satisfiability of the CNF formula. For symmetry breaking to be effective in practice, the computational overhead of generating and manipulating the SBPs must be significantly less than the run time savings they yield due to search space pruning. In this paper we present several new constructions of SBPs that improve on previous work. Specifically, we give a linear-sized CNF formula that selects lex-leaders (among others) for single permutations. We also show how that formula can be simplified by taking advantage of the sparsity of permutations. We test these improvements against earlier constructions and show that they yield smaller SBPs and lead to run time reductions on many benchmarks.

Categories and Subject Descriptors

T0.1 Fundamental CAD Algorithms

General Terms

Algorithms, Experimentation, Verification, Routing

Keywords

SAT, CNF, backtrack search, symmetries, clause learning, logic simplification, graph automorphism

1. Introduction

Many search, synthesis, and optimization problems arising in algorithmic applications exhibit symmetries. The presence of multiple, symmetric solutions may 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 to eliminate parts of the search space and, thereby, simplify the solution process.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2003, June 2-6, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-688-9/03/0006...$5.00.

In this paper we study the Boolean satisfiability problem—one of the most important in Computer Science—in the presence of symmetry. Crawford et al. [2] laid the theoretical foundation for generating symmetry-breaking predicates for CNF formulas that possess permutational symmetries. The basic idea is to detect all such symmetries using a reduction to graph automorphism. For each such symmetry, an SBP—whose CNF representation is quadratic in the number of problem variables—is constructed. A SAT solver is then applied to the conjunction of the original formula and the SBPs created from each permutation in the group of symmetries. This approach breaks *all* permutational symmetries and will be referred to as *full* symmetry breaking. The downside is that the number of symmetries in the symmetry group is usually exponential in the number of problem variables making such full symmetry breaking impractical. They reduce the number of symmetries that need to be considered by building a *symmetry tree* and pruning it to remove unnecessary duplication. This, unfortunately, still does not preclude the need to consider an exponential number of permutations in order to break all symmetries.

This general framework was extended in [1] to handle phase-shift symmetries and their composition with permutational symmetries. Additionally, more efficient, albeit still quadratic, CNF constructions of the symmetry-breaking predicates were introduced. Finally, symmetry-breaking was only applied to the generators of the symmetry group as opposed to the entire set of symmetries. They argued, and presented strong empirical evidence, that full symmetry breaking is unnecessary and that such *partial* symmetry breaking is often quite effective.

In this paper we present a computational framework for generating optimized SBPs that improves on previous work [1, 2]. Specifically, we describe SBP constructions for single permutations that yield a CNF formula whose size is *linear*, rather than *quadratic*, in the number of variables in the SAT problem. We also show how that formula can be simplified by taking advantage of the sparsity of permutations. We test these improvements against earlier constructions and show that they yield much smaller SBPs and lead to significant run time reductions on many benchmark families.

The remainder of the paper is organized as follows. Section 2 presents the necessary definitions and notation. Section 3 covers previous work. The proposed efficient SBP constructions are described in Section 4. We show experimental results in Section 5, and the paper concludes in Section 6.

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. Symmetries are studied in abstract algebra in terms of *groups* [4]. A *group* is a set with a binary associative operation defined on it such that there is a *unit* element and every element has a unique inverse. In general, a set of group elements such that any other group element can be expressed as their product is called a *generating set*. The size of any irredundant generating set is no greater than the binary logarithm of the group size and in general is much smaller. A group G

is *cyclic* if it consists of an element x and all elements of G are powers of x; x is known as the cyclic group generator.

The symmetric group $S(\Omega)$ on a finite set Ω is the group of all permutations of Ω. If $\Omega = \{1, \ldots, n\}$, the group is commonly denoted by S_n. 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. For example, (12)(456) can denote a permutation that swaps elements 1 and 2, and maps 4 to 5, 5 to 6 and 6 to 4. Cycle notation is preferable to tabular notation for *sparse* permutations that map most elements of Ω to themselves.

A symmetry (*automorphism*) of a graph is a permutation of its vertices that maps edges to edges. If vertices are labeled by integers (*colors*), we may additionally require symmetries to 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., $(x_3 + x_6' + x_7)$, and a *CNF formula* is a conjunction of clauses, e.g., $(x_3 + x_6' + x_7)(x_{10} + x_{11}')(x_{20})$. A *binary* clause has two literals and can be viewed as an implication between variables, e.g., $(x_{11} \rightarrow x_{10})$. The CNF-SAT decision problem seeks to find a truth assignment that satisfies a given CNF formula or to prove that the formula is unsatisfiable.

We will assume a total ordering on the 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 $a \mapsto b$ then $a' \mapsto b'$ 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*) [1]. 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 from 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 taken to be the number of literals in its CNF formula. 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 from 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 CNF Symmetries via Graph Automorphism

Given a CNF formula, a graph is constructed such that the group of CNF symmetries is isomorphic to the group of graph automorphisms. A simple construction [1] represents every clause by a vertex of color 2, and every variable by two vertices of color 1 (one for the positive and one for the negative literal) connected by *Boolean consistency edges*. Every literal in the CNF formula is then represented by a bi-partite edge. The construction in [2] treats binary clauses differently. It leaves out their clausal vertices and connects their literal vertices by double-edges. Since some graph automorphism programs (e.g. GAP/NAUTY - http://www.gap-system.org/) 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 [1]; in our experiments, we did not find any.

Note that the graph automorphism problem is believed to be outside P, yet not NP-complete. In general, finding CNF symmetries is often easier than solving SAT. Furthermore, excellent graph automorphism software is currently available [6] and typically returns a small set of irredundant generators, rather than the complete set of permutations, which ensures exponential compression.

3.2 The Lex-Leader Formulae

The entire construction of symmetry-breaking predicates rests on the notion of lex-leader formulae introduced by Crawford et al. [2]. Given a group of symmetries $\Pi = \{\pi_1, \ldots, \pi_m\}$ for a CNF formula defined over a set of totally-ordered variables $x_1 < x_2 < \ldots < x_n$, the LL-SBP is defined as follows:

$$\mathrm{PP}(\pi) = \bigwedge_{1 \le i \le n} \left[\bigwedge_{1 \le j \le i-1} \left(x_j = x_j^{\pi} \right) \right] \rightarrow \left(x_i \le x_i^{\pi} \right) \quad (1)$$

$$\text{LL-SBP}(\Pi) = \bigwedge_{1 \le i \le m} \mathrm{PP}(\pi_i) \quad (2)$$

In the sequel we will refer to $\mathrm{PP}(\pi)$ as the *permutation predicate* for permutation π. By introducing auxiliary variables $e_j = (x_j = x_j^{\pi})$, each PP can be translated to a CNF formula with $5n$ clauses and $0.5n^2 + 13.5n$ literals.

In general, the CNF representations of the PPs defined in (1) tend to have duplicate clauses and even tautologies. In addition, the PPs of different permutations may have identical clauses leading to duplication in the LL-SBP defined in (2). Recognizing the existence of such redundancies, Aloul el al. [1] introduced a more efficient version of (1) that eliminates duplicate clauses and tautologies. They also noted that the LL-SBP in (2) can be replaced by a much smaller *partial* LL-SBP that uses only an irredundant set of generators $G(\Pi)$ for the group Π, i.e.,

$$\text{PLL-SBP}(G(\Pi)) = \bigwedge_{\pi_i \in G(\Pi)} \mathrm{PP}(\pi_i) \quad (3)$$

4. Improved SBP Constructions

In this section we present several improvements to the constructions in [1, 2].

4.1 Linear-Sized Permutation Predicates

The first improvement we present is a CNF formula for PPs whose size is linear, rather than quadratic, in the number of variables. To facilitate the following derivation let $l_i = (x_i \leq x_i^{\pi})$, $g_i = (x_i \geq x_i^{\pi})$, and $g_0 \equiv 1$. Noting that $e_i = l_i g_i$, the PP in (1) can now be expressed as:

$$(g_0 \rightarrow l_1)(g_0 l_1 g_1 \rightarrow l_2)\ldots(g_0 l_1 \ldots l_{n-1} g_{n-1} \rightarrow l_n) \tag{4}$$

Factoring out the common prefix g_0 and simplifying yields:

$$g_0 \rightarrow [l_1 \cdot (g_1 \rightarrow l_2)\ldots(g_1 l_2 \ldots l_{n-1} g_{n-1} \rightarrow l_n)] \tag{5}$$

The recursive structure of the formula is now revealed by comparing (4) and (5). Let $p_1, \ldots, p_n$ be a sequence of *bit predicates* defined by:

$$p_i = g_{i-1} \rightarrow l_i p_{i+1} \qquad i = 1, \ldots, n; \qquad p_{n+1} = 1 \tag{6}$$

Note that predicate p_1 represents the entire formula (4). The satisfiability of (4) can, thus, be determined by checking the satisfiability of the following equivalent, but simpler formula:

$$(p_1)(p_1 = g_0 \rightarrow l_1 p_2)\ldots(p_n = g_{n-1} \rightarrow l_n p_{n+1}) \tag{7}$$

One final simplification replaces the equalities in (7) with implications since we are only interested in satisfying each of the predicates. We thus obtain:

$$(p_1)(p_1 \rightarrow g_0 \rightarrow l_1 p_2)\ldots(p_n \rightarrow g_{n-1} \rightarrow l_n p_{n+1}) \tag{8}$$

The CNF representation of (8) consists of $2n$ 3-literal and $2n$ 4-literal clauses for a total size of $14n$ literals. Schematically, (8) can be viewed as a multi-level logic circuit (see Figure 1). Hence, the truth assignments that satisfy (1) are those assignments that set the circuit output p_1 to 1. It is also interesting to note that bit predicates form a chain that is reminiscent of carry chains in ripple adders.

4.2 Elimination of Tautologies in PPs

For a given permutation, choosing an SBP with fewer literals is also important for empirical success. Since the permutations are typically sparse, the size of the CNF representation of (8) can be further reduced by eliminating all tautologies. We illustrate our approach with an example. Consider the permutation:

$$\pi = (y_1 y_4)(y_5 y_6)(y_7 y_8 y_9) \tag{9}$$

expressed in *cycle* notation. The permutation can be expressed in *tabular* form as follows:

$$\begin{pmatrix} y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9 \\ y_4 y_2 y_3 y_1 y_6 y_5 y_8 y_9 y_7 \end{pmatrix} \tag{10}$$

We proceed to create a sequence of bit predicates $p_1, p_2, p_3, \ldots$ using (7). The only difference is that we eliminate the predicates of the variables that do not move. For such variables note that $l_i = g_i = 1$. In this example, $l_2 = g_2 = l_3 = g_3 = 1$. This immediately simplifies the predicates of those variables to:

$$\begin{aligned} p_2 &= g_1 \rightarrow l_2 p_3 = g_1 \rightarrow p_3 \\ p_3 &= g_2 \rightarrow l_3 p_4 = p_4 \end{aligned} \tag{11}$$

which allows us to express p_1 directly in terms of p_4:

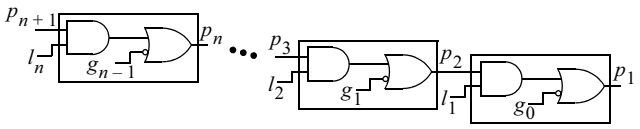

Figure 1. Circuit representation of (8)**.**

$$p_1 = g_0 \rightarrow l_1(g_1 \rightarrow p_4) \tag{12}$$

and p_4 reduces to:

$$p_4 = l_4 p_5 \tag{13}$$

The pattern should now be clear. Given a permutation, where variables y_i and y_j move and $y_{i+1}, \ldots, y_{j-1}$ do not move, the PP can be simplified to:

$$\begin{aligned} p_i &= g_{i-1} \rightarrow l_i(g_i \rightarrow p_j) \\ p_j &= l_j p_{j+1} \end{aligned} \tag{14}$$

All other bit predicates, representing variables that move, use the format described in (6).

4.3 Options for Partial Symmetry Breaking

Breaking *all* symmetries may not speed up search because there are often exponentially many of them and their PLL-SBPs may be redundant [2]. 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. Alternatively, we can use the powers or compositions of generators as candidates for symmetry breaking.

Additionally, we can reduce the size of SBPs by considering only the first k bits from each permutation. This is achieved by generating the SBPs up to the k-th predicate and setting the last used predicate to 1.

5. Experimental Results

In this section, we empirically show the advantage of using the proposed SBP constructions. The experiments were performed on an AMD Athlon 1.2 GHz machine with 1 GB of RAM running Linux. The run time limit for all experiments was set to 1000 seconds. The benchmarks included pigeon-hole [3], randomized Urquhart (URQ) [10], global routing (s3) [1], FPGA routing (fpga, chan) [8], and xor-chains [9]. We used the best available backtrack SAT solver Chaff [7]. Since Chaff is randomized, its run time varies, and we averaged all results over 200 independent runs.

Table 1 lists symmetry detection run times, number of symmetries, symmetry generators, and phase shifts. We use the reduction to graph automorphism from [1] which detects a wider range of symmetries than that from [2]. The table clearly shows the significant savings obtained when generators are used to represent the complete set of symmetries. Note that all generators consisted of cycles of size 2 only. The table also shows the percentage of bits that map to themselves (shown as "%RB"). Clearly, more than 80% of the bits in all instances are redundant and can lead to significant savings in run time and memory if removed from the SBPs.

Table 1 also compares SAT-solving run times for the original CNF instance and the instances augmented with generator SBPs using the construction in equation (3), reference [1], and the proposed constructions in Sec. 4.1 and 4.2. Clearly, the addition of SBPs sig-

Table 1. Search run times of CNF-SAT instances with and without PLL-SBPs (for generators only). Symmetry statistics including symmetry detection run time, number of symmetries, generators, and phase shifts are also provided.

Instance	V	C	Symmetry statistics					Chaff run time (sec)					Sec. 4.2 speedup over	
			Find (sec)	#Sym	#Gen	#P.S.	%RB	Orig	(3)	[1]	Sec. 4.1	Sec. 4.2	Orig	[1]
hole11	132	738	0.09	1.9E+16	21	0	83	219	5.43	0.03	3.52	0.02	11K	1.5
hole12	156	949	0.12	2.9E+18	23	0	84	1000	11.0	0.04	8.65	0.02	>50K	2
Urq3_4	36	220	0.10	5.2E+05	19	19	97	0.85	0.16	0.01	0.08	0.01	85	1
Urq3_9	37	236	0.04	1.1E+06	20	20	97	12.8	0.05	0.01	0.02	0.01	1280	1
s3-3-3-3	960	9156	8.92	6.9E+10	29	0	95	44.4	200	2.85	65.3	0.50	89	5.7
s3-3-3-8	912	8356	6.95	3.5E+10	28	0	95	21.3	141	2.61	138	0.93	23	2.8
fpga13_10	195	905	0.25	1.9E+17	28	0	88	1000	517	0.09	7.70	0.03	>33K	3
fpga13_12	234	1242	0.81	9.0E+20	32	0	89	1000	1000	0.08	25.8	0.05	>20K	1.6
chnl11_13	286	1742	0.96	1.2E+35	45	0	90	788	882	0.16	108	0.05	16K	3.2
chnl11_20	440	4220	4.48	1.9E+52	59	0	92	1000	1000	0.30	92.6	0.09	>11K	3.3
xor1_32	94	250	0.17	4.3E+09	32	32	98	830	12.6	1.7	1.73	1.7	488	1
xor1_36	106	282	0.34	6.9E+10	36	36	99	938	0.61	0.01	0.1	0.01	94K	1
Total	**3588**	**28296**	**23**	**1.9E+52**	**372**	**107**	**-**	**6854**	**3770**	**7.9**	**452**	**3.4**	**-**	**-**

Table 2. Total size of generator-only SBPs using various SBP constructions for the instances presented in Table 1.

#	(3)	[1]	Sec. 4.1	Sec. 4.2
Var	120K	8.6K	120K	**8.6K**
Cl	599K	44K	478K	**34K**
Lit	36M	404K	1.7M	**119K**

nificantly reduces the search run time, and the approach in Sec. 4.2 leads to the greatest savings in run time.

Table 2 compares the size of SBP predicates produced by the above four constructions. The construction proposed in Sec. 4.2 entails the smallest number of variables, clauses, and literals for all analyzed instances.

In order to further reduce the size of SBPs, we tested all presented instances after considering only the first k bits from each permutation. Figure 2 plots the total search run time for all instances as a function of k. The construction proposed in Sec. 4.2 was used in the experiment. Interestingly, the total search run time decreases as additional bits are considered. Nevertheless, the performance improvements fade after considering 10 bits per permutation for almost all instances. This confirms the intuition that breaking all symmetries does not necessarily speed up the search process. In fact, breaking the symmetries for only a subset of permutations, i.e. irredundant sets of generators, and considering a limited number of bits from each permutation seems to be sufficient to significantly reduce the search run times in all cases.

Our final experiment compares the search run time and memory requirements of different choices of permutation sets to break. We considered three such sets: generators, powers of generators, and compositions of generators. Since all generators produced by the graph automorphism program were expressed as products of length-2 cycles, the set of their powers did not yield additional permutations. However, the set of permutations obtained by composing the 372 generators yielded a total of 5402 permutations. Using the construction proposed in Sec. 4.2, the SBPs for these permutations consisted of 326K variables, 1.3M clauses, and 4.5M literals. This significant increase in the size of the SBP was reflected in a much larger search run time: 29 seconds as opposed to 3.4 seconds for the generator-only SBP. This provides further empirical evidence for the use of generator-only SBPs to speed up satisfiability search.

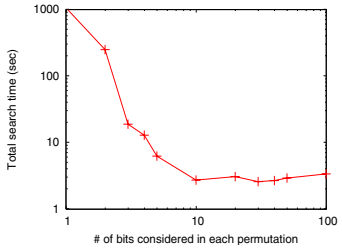

Figure 2. Total search run times of instances shown in Table 1 when only k bits are considered from each generator.

6. Conclusions

In this work we extended and improved the framework of symmetry-breaking predicates for solving Boolean Satisfiability by constructing more efficient CNF representations of symmetry-breaking predicates. The proposed techniques lead to empirical speed-ups in backtrack search and smaller memory requirements for the best available SAT solvers. Additionally, we gave new justifications of partial symmetry-breaking by generators.

Acknowledgments

This work was funded by the DARPA/MARCO Gigascale Silicon Research Center and an Agere Systems/SRC Research fellowship.

7. References

[1] F. Aloul, A. Ramani, I. Markov, and K. Sakallah, "Solving Difficult SAT Instances in the Presence of Symmetries," in *Proc. of the Design Automation Conference* (DAC), 731-736, 2002.

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

[3] DIMACS Challenge benchmarks in *ftp://Dimacs.rutgers.EDU/pub/challenge/sat/benchmarks/cnf*.

[4] I. Gessel and R. Stanley, "Algebraic Enumeration," in *R. Graham, M. Grötschel and L. Lovász, eds, Handbook of Combinatorics*, vol. 2, MIT Press, 1021-1061, 1995.

[5] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT-solver," in *Proc. of DATE*, 142-149, 2002.

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

[7] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *DAC,* 530-535, 2001.

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

[9] SAT 2002 Competition, *http://www.satlive.org/SATCompetition/submittedbenchs.html*

[10] A. Urquhart, "Hard Examples for Resolution," in *Journal of the ACM*, 34(1), 209-219, 1987.