Symmetry-Breaking for Pseudo-Boolean Formulas

Fadi A. Aloul[a,b], Arathi Ramani[a], Igor L. Markov[a], Karem A. Sakallah[a]

[a]Department of Electrical Engineering and Computer Science
University of Michigan - Ann Arbor
{faloul, ramania, imarkov, karem}@umich.edu

[b]Department of Computer Engineering
American University of Dubai - U.A.E.

Abstract. Many important tasks in circuit design and verification can be performed in practice via reductions to Boolean Satisfiability (SAT), making SAT a fundamental EDA problem. However such reductions often leave out application-specific structure, thus handicapping EDA tools in their competition with creative engineers. Successful attempts to represent and utilize additional structure on Boolean variables include recent work on 0-1 Integer Linear Programming (ILP) and on symmetries in SAT. Those extensions gracefully accommodate well-known advances in SAT-solving, but their combined use has not been attempted previously. Our work shows (i) how one can detect and use symmetries in instances of 0-1 ILP, and (ii) what benefits this may bring.

1 Introduction

Recent impressive speed-ups of solvers for Boolean satisfiability (SAT) [12, 17, 22] enabled new applications in design automation [1, 18, 23]. Reducing an application to SAT facilitates the reuse of existing efficient computational cores and leads to high-performance EDA tools with little development effort. However, major concerns about this approach are the loss and ignorance of high-level information and application-specific structure. With this in mind, researchers successfully extended leading algorithms for SAT-solving to handle more powerful constraint representations, e.g., 0-1 Integer Linear Programming (ILP) [1, 4, 5, 10, 24]. Another broad avenue of research leads to pre-processors for existing solvers and constraint representations, that extract high-level information and guide the solvers accordingly [2, 3, 6]. Our work extends existing techniques for detecting and using symmetries in SAT to the more general 0-1 ILP formulation that includes pseudo-Boolean (PB) constraints and an optional optimization objective.

Asymptotic improvements to SAT-solvers are often illustrated using the task of proving the pigeonhole principle without mathematical induction. Leading-edge complete SAT-solvers currently do not use induction and exhibit exponential runtime on respective instances, which are expressed in conjunctive normal form (CNF), because these instances do not allow polynomial-sized resolution proofs. However, when FPGA routing instances are modeled by SAT, every FPGA switchbox may generate a sub-problem equivalent to the pigeonhole principle, trapping SAT-solvers. Because this principle is a basic property of finite sets, it is likely to be intrinsic to other applications, such as the verification of multi-threaded programs, circuits with multiple copies of architectural components, and protocols for multi-party communication.

Recent research offers fully-automated tools that establish the pigeon-hole principle in polynomial time through the use of symmetry [2, 3, 6] or PB inequalities [5], particularly counting constraints. These tools have also been validated for important applications, such as FPGA and global routing.

Our work contributes a framework for detecting and using symmetries in instances of 0-1 ILP. When applied to SAT instances encoded as 0-1 ILPs, our framework works at least as well as those in [2, 3, 6]. In general, it detects all existing structural permutational symmetries, phase shift symmetries, and their compositions. We present experimental evidence showing that EDA problems expressed in PB form (i) sometimes have symmetries, and (ii) can be solved faster within our framework than previously.

The remainder of the paper is organized as follows. Section 2 presents a brief description of the CNF and PB representations. Section 3 presents the framework for detecting and using symmetries in CNF formulas. The framework is extended to handle PB formulas in Section 4. We show experimental results in Section 5, and the paper concludes in Section 6.

2 Preliminaries

A Boolean formula φ given in *conjunctive normal form* (CNF) consists of a conjunction of *clauses*, where each clause is a disjunction of *literals*. A literal is either a variable or its complement. A clause is *satisfied* if at least one of its literals has a value of 1, *unsatisfied* if all its literals are 0, and *unresolved* otherwise. Consequently, a formula is satisfied if all its clauses are satisfied, and unsatisfied if at least one clause is unsatisfied. The goal of the SAT solver is to identify an assignment to a set of binary variables that would satisfy the formula or prove that no such assignment exists (and that the formula is unsatisfiable).

In addition to CNF constraints, a Boolean formula can include PB constraints which are linear inequalities with integer coefficients[1] of the form:

$$a_1x_1 + a_2x_2 + \ldots + a_nx_n \leq b \tag{1}$$

where $a_i, b \in Z^+$ and x_i are literals of Boolean variables[2]. Using the relations $\overline{x_i} = (1 - x_i)$, $(Ax = b) \Leftrightarrow (Ax \leq b)(Ax \geq b)$, and $(Ax \geq b) \Leftrightarrow (-Ax \leq -b)$ any arbitrary PB constraint can be converted to the *normalized* form of (1) consisting of only positive coefficients. This normalization facilitates more efficient algorithms.

Figure 1(a) illustrates the difference between the CNF and PB encodings for the pigeon-hole (*hole-2*) instance. The instance can be represented by 6 variables, 9 clauses, and 18 literals when using the CNF encoding or by 6 variables, 5 PB constraints, and 12 literals when using the PB encoding. Clearly, PB constraints are more efficient than CNF clauses in representing counting constraints.

3 Detecting and Using Symmetries in CNF Formulas

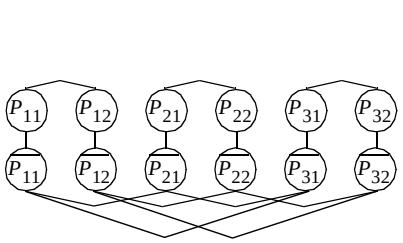
Leading-edge complete SAT solvers [12, 17, 22] implement the basic Davis-Logemann-Loveland (DLL) algorithm [8] for backtrack search with various improvements. This algorithm has exponential worst-case complexity and, despite dramatic improvements for practical inputs, the runtime of those SAT solvers grows exponentially with the size of the input on various instances including the pigeon-hole [9] benchmarks. The work in [2, 3, 6] empirically showed that the use of symmetry-breaking predicates (i) makes runtime on those instances polynomial, and (ii) speeds up the solution of some application-derived instances. Crawford et al. [6] presented a theoretical framework for detecting and using permutational symmetries in CNF formulas. An extension of this framework in [2] showed how to detect phase-shift symmetries (i.e. symmetries that map variables to their complements) and their compositions with permutational symmetries. Asymptotic efficiency of these techniques was improved in [3]. The general framework is described next.

1 Floating-point coefficients are also easily handled [1].

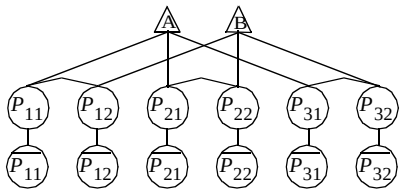
2 Any CNF clause can be viewed as a PB constraint, e.g. clause $(a \vee b \vee c)$ is equivalent to $(a + b + c \geq 1)$.

Constraint	CNF-only Encoding	Alternative PB Encoding
Each pigeon must be in at least one hole	$(P_{11} \vee P_{12})$ $(P_{21} \vee P_{22})$ $(P_{31} \vee P_{32})$	$(P_{11} + P_{12} \geq 1)$ $(P_{21} + P_{22} \geq 1)$ $(P_{31} + P_{32} \geq 1)$
Each hole can hold at most one pigeon	$(\overline{P_{11}} \vee \overline{P_{21}})$ $(\overline{P_{11}} \vee \overline{P_{31}})$ $(\overline{P_{12}} \vee \overline{P_{22}})$ $(\overline{P_{12}} \vee \overline{P_{32}})$ $(\overline{P_{22}} \vee \overline{P_{32}})$ $(\overline{P_{21}} \vee \overline{P_{31}})$	$(P_{11} + P_{21} + P_{31} \leq 1)$ $(P_{12} + P_{22} + P_{32} \leq 1)$

(a)



(b)



(c)

#	CNF-only Encoding	Alternative PB Encoding
1	$(P_{11}, P_{21})(P_{12}, P_{22})$ $(\overline{P_{11}}, \overline{P_{21}})(\overline{P_{12}}, \overline{P_{22}})$	$(P_{11}, P_{21})(P_{12}, P_{22})$ $(\overline{P_{11}}, \overline{P_{21}})(\overline{P_{12}}, \overline{P_{22}})$
2	$(P_{21}, P_{31})(P_{22}, P_{32})$ $(\overline{P_{21}}, \overline{P_{31}})(\overline{P_{22}}, \overline{P_{32}})$	$(P_{21}, P_{31})(P_{22}, P_{32})$ $(\overline{P_{21}}, \overline{P_{31}})(\overline{P_{22}}, \overline{P_{32}})$
3	$(P_{11}, P_{12})(P_{21}, P_{22})(P_{31}, P_{32})$ $(\overline{P_{11}}, \overline{P_{12}})(\overline{P_{21}}, \overline{P_{22}})(\overline{P_{31}}, \overline{P_{32}})$	$(P_{11}, P_{12})(P_{21}, P_{22})(P_{31}, P_{32})$ $(\overline{P_{11}}, \overline{P_{12}})(\overline{P_{21}}, \overline{P_{22}})(\overline{P_{31}}, \overline{P_{32}})$ (A, B)

(x, y) indicates that variable x maps to variable y under the given permutation.

(d)

Fig. 1. (a) Two possible encodings of the unsatisfiable pigeon-hole instance consisting of 2 holes and 3 pigeons using CNF and PB constraints. P_{ij} denotes pigeon i in hole j; (b) graph representing the CNF formula; (c) graph representing the PB formula. Different vertex shapes correspond to different vertex colors; (d) generators of the graph automorphism group of (b) and (c).

3.1 Detecting symmetries via graph automorphism

Given a graph, a symmetry (also called an *automorphism*) is a permutation of its vertices that maps edges to edges. For a directed graph, edge orientations must be maintained. The collection of symmetries of a graph is closed under composition and is known as the *automorphism group* of the graph. The problem of finding all symmetries of the graph is known as the *graph automorphism problem*. Efficient tools for detecting graph automorphism have been developed, such as NAUTY [16] and SAUCY [7].

Structural symmetries in CNF formulas can be detected via a reduction to graph automorphism [15]. A CNF formula is represented as an undirected graph with colored vertices such that the automorphism group of the graph is isomorphic to the symmetry group of the CNF formula. The two groups must share a one-to-one correspondence and also be isomorphic to enable the use of group generators as explained in the Section 3.2.

Assuming a CNF formula with V vertices and C clauses (single-literal clauses are removed by preprocessing the CNF formula), a graph is constructed as follows:

- A single vertex represents each clause (clause vertices).
- Each variable is represented by two vertices: positive and negative literals (literal vertices).
- Edges are added connecting a clause vertex to its respective literal vertices (incidence edges).
- Edges are added between opposite literals (Boolean consistency edges).
- Clause vertices are painted with color 1 and all literal vertices (positive & negative) with color 2.

As the runtime of graph automorphism tools, e.g. NAUTY [16] or SAUCY [7], usually increases with growing number of vertices, each binary clause can be represented with a single edge between the two literal vertices rather than a vertex and two edges (see Figure 1(b)). This optimization can, in some cases, result in spurious graph automorphisms [2]. Fortunately, this is uncommon in CNF applications, and spurious graph symmetries are easy to test for [2].

3.2 Using symmetries

Symmetries induce an equivalence relation on the set of truth assignments of the CNF formula, and every equivalence class (orbit) contains either satisfying assignments only or unsatisfying assignments only [6]. Therefore SAT-solving can be sped up, without affecting correctness, by considering only a few representatives (at least one) from each equivalence class. This constraint can be conveniently represented by conjoining additional clauses (symmetry-breaking predicates - SBPs) to the original CNF formula. One particular family of representatives are lexicographically smallest assignments in each equivalence class (lex-leaders). Crawford et al. [6] introduced an SBP construction whose CNF representation is quadratic in the number of problem variables. Their construction assumes a given variable ordering $x_1 < x_2 < \ldots < x_n$ and produces a permutation predicate (PP) for each permutational symmetry in the group of symmetries as follows:

$$PP(\pi) = \bigcap_{1 \le i \le n} \left[\bigcap_{1 \le j \le i-1} (x_j = x_j^{\pi}) \right] \rightarrow (x_i \le x_i^{\pi}) \qquad (2)$$

where x_i^{π} is the image of variable x_i under permutation π.

Aloul et al. [3] described a logically equivalent, but more efficient tautology-free SBP construction, whose size is *linear*, rather than *quadratic*, in the number of problem variables. In practice smaller SBPs may decrease search time. Strong empirical evidence in [3] shows that *full* symmetry breaking is unnecessary and that *partial* symmetry breaking is often more effective, because the number of symmetries can be very large. In particular, the authors showed that applying symmetry-breaking to a set of generators[3] of the group of symmetries rather than the entire set of symmetries leads to significant runtime and memory reductions. It is not necessary for the set of generators to be irredundant, but, according to empirical data, adding generators expressible in terms of other generators does not improve

3 Generators represent a set of symmetries whose product generates the complete set of symmetries. An irredundant set of generators for a group with $N > 1$ symmetries consists of at most $\log_2 N$ symmetries [11]. While their number can be as small as two, it typically grows with the size of the group. The graph shown in Figure 1(b) has 12 symmetries that can be captured using the 3 generators shown in Figure 1(d).

overall runtime. Moreover, standard algorithms for the graph automorphism problem always produce irredundant sets of generators. Those sets are typically not minimal and are, surprisingly, better for symmetry-breaking purposes than minimal sets that can be produced using group-theoretic algorithms in the GAP system at the cost of longer computation. If a given set of generators does not generate the full group, it may still be used for symmetry-breaking, but can also be improved. All in all, irredundant sets of generators produced by graph automorphism programs such as NAUTY and SAUCY tend to lead to the best overall runtimes in our symmetry-breaking flows.

4 Detecting and Using Symmetries in PB Formulas

Similar to the techniques from [2] (summarized in Section 3), we build a graph whose automorphism group is isomorphic to the group of PB symmetries. A graph automorphism program would produce generators of the automorphism group, which we reapply to the original PB instance. The isomorphism of the two symmetry groups is required to implicitly manipulate these groups in terms of generators. While our graph construction is novel, detected symmetries can be used by means of the known symmetry-breaking predicates (SBP) for SAT [3] because those are also applicable to 0-1 ILPs.

4.1 Graph construction for PB formulas

Given a formula with V variables, C clauses, and P PB constraints, we build a graph as follows:

- Variables are treated exactly the same as in the CNF case: two vertices per variable represent positive and negative literals. Each such pair is connected by a Boolean consistency edge.
- Any non-PB (pure CNF) clauses are also treated just like in the CNF case: two-literal clauses are represented by an edge connecting both literals. For other clauses, a vertex is created to represent the clause, and incidence edges connect the clause vertex to its literal vertices.
- Clause vertices are painted in color 1, and literal vertices in color 2.
- Literals in a PB constraint P_i are organized as follows:
  - The literals in P_i are sorted by coefficient value, and literals with the same coefficient are grouped together. Thus, if there are M different coefficients in P_i, we have M disjoint groups of literals, $L_1, \ldots, L_m$.
  - For each group of literals, L_j, with the same coefficient, a single vertex $X_{i,j}$ (coefficient vertex) is created to represent the coefficient value. Edges are then added to connect this vertex to each literal vertex in the group.
  - A different color is used for each distinct coefficient value encountered in the formula. This means that coefficient vertices that represent the same coefficient value in different constraints are colored the same.
- Each PB constraint P_i is itself represented by a single vertex Y_i (PB constraint vertex). Edges are added to connect Y_i to each of the coefficient vertices, $X_{i,1}, \ldots, X_{i,M}$ that represent its M distinct coefficients.
- The vertices $Y_1, \ldots, Y_P$ are colored according to the constraint's right-hand side (RHS) value b. Every unique value b implies a new color, and vertices representing different constraints with the same RHS value are colored the same.

Figure 2 shows a graph that represents a formula with both CNF clauses and PB constraints. CNF clauses are represented as in Section 3, but PB constraints have different coefficients and require special treatment as explained above. Vertices $X_{1,1}$ and $X_{2,1}$ represent the coefficient value of 1 and are shown as upward triangles (for color), while $X_{1,2}$ and $X_{2,2}$ represent the coefficient value of 2 and are shown as downward triangles (a different color). The two PB constraint vertices, Y_1 and Y_2, have the same color/shape since the two PB constraints have equal RHS values.

$$\varphi(x_1, x_2, x_3) = C_1{:}(x_1 + x_2 + \overline{x_3}) \cdot$$
$$C_2{:}(\overline{x_2} + x_3) \cdot$$
$$P_1{:}(2x_1 + x_2 + x_3 \geq 2) \cdot$$
$$P_2{:}(x_1 + 2x_2 + x_3 \geq 2)$$

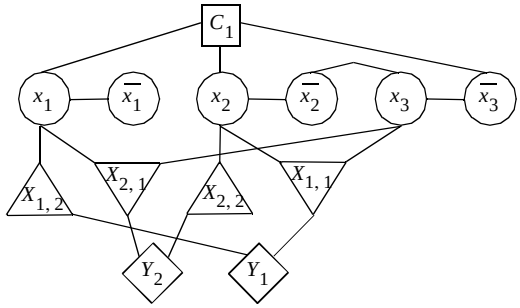


Fig. 2. Example showing the graph representing formula φ.
Different vertex shapes corresponds to different vertex colors.

4.2 Proof of correctness

We will rely on the correctness proof of the graph construction for CNFs proposed in [2]. To restate their key result, we first review the necessary terminology. A circular chain of implications over the variables $x_1, x_2, \ldots, x_n$ is defined in [2] as a set of N binary clauses equivalent to $(y_1 \Rightarrow y_2)(y_2 \Rightarrow y_3)\ldots(y_{N-1} \Rightarrow y_N)(y_N \Rightarrow y_1)$, where for each $k \in 1\ldots N$, $y_k = x_k$ or $y_k = \overline{x_k}$. Assigning a value to any y_k triggers an implication sequence that determines the values of all literals involved. Thus, such a chain allows only two satisfying assignments. The key theorem follows.

Theorem 4.1 *Assume that a given CNF formula does not contain a circular chain of implications over any subset of its variables. Then, with respect to the proposed construction of the colored graph from a CNF formula, the symmetries of the formula correspond one-to-one to the symmetries of the graph* [2].

The caveat with circular chains is due to an optimization where binary clauses, unlike larger clauses, are represented by single edges.This reduces the number of vertices, but now binary-clause edges and Boolean consistency edges are indistinguishable. A graph symmetry mapping a binary-clause edge to a Boolean consistency edge (or vice versa) would not correspond to a SAT symmetry. Using a graph-theoretical lemma, the work in [2] shows that such spurious symmetries require circular chains of implications. Moreover, such chains are trivial to test for and do not appear in practice. In the graph, a chain of implications corresponds to a cycle with alternating positive and negative literal vertices.

To establish an analogous result for our graph construction for PB formulas, we first observe that the addition of PB constraints to a CNF formula cannot create new alternating cycles in the graph. That is because the colors of PB constraint and coefficient vertices are different from the colors of literal and clause vertices. It is thus impossible for an edge connecting literal vertices to coefficient vertices (or coefficients to PB constraints) to be mapped into a Boolean consistency edge. Therefore, the only prohibited case for PB formulas is the presence of implication chains in the CNF component.

Theorem 4.2 *Assume that a given formula, with CNF and PB constraints, does not contain a circular chain of implications over any subset of its variables in its CNF component. Then, with respect to the proposed construction of the colored graph from a PB formula, the symmetries of the formula correspond one-to-one to the symmetries of the graph.*

Proof. We begin by showing that any symmetry in the original formula corresponds to a colored symmetry in the constructed graph. A permutational symmetry that maps a to b in the formula will map vertex a to vertex b, vertex $\overline{a}$ to vertex $\overline{b}$, and the edge $a\overline{a}$ to $b\overline{b}$ in the graph. Since $a, \overline{a}, b, \overline{b}$ all have the same color, the symmetry is preserved. For a phase shift symmetry, vertices a and $\overline{a}$ are interchanged, leaving the edge $a\overline{a}$ in place, and any binary clausal edges are swapped at the correspond-

ing clause vertex. For example, edges ac and $\bar{a}c$ are swapped through vertex c. For a PB formula, a and $\bar{a}$ might also be connected to one or more coefficient vertices. These connections would also be swapped at the respective vertices. Again, only vertices of the same color are mapped one to another. Thus a consistent mapping of literals or variables in the formula, when carried over to the graph, must preserve the colors of graph vertices.

We now show that every colored symmetry in the graph corresponds to a symmetry in the original formula. This is easily seen in the PB case because we use one color for literals, one for non-binary clauses, one set of colors for coefficient values, and one set for coloring constraints according to RHS value. Different groups above use different colors. Therefore, if $a \rightarrow b$ then $\bar{a} \rightarrow \bar{b}$ since $\bar{b}$ is the only vertex connected to b that is the same color as $\bar{a}$. A similar statement is more difficult to prove in the presence of CNF clauses, but it is proven in [2] for CNF clauses under the assumption that no circular chains of implications exist and is extended to mixed CNF-PB formulas as explained above.

Theorem 4.3 *Under the assumption of Theorem 4.2, the symmetry groups of the PB formula and the multicolored graph are isomorphic.*

Proof. It can be easily verified that the one-to-one mapping of symmetries described above is a homomorphism. Furthermore, a one-to-one homomorphism is an isomorphism.

Given a colored graph symmetry, we can uniquely reconstruct the PB symmetry to which it corresponds, provided we maintain the correspondence between variables and their positive and negative literal vertices. Symmetries in the graph are detected using SAUCY [7], and used to reconstruct symmetries in the PB formula. SBPs are added to the formula as CNF clauses using the efficient construction in [3]. The use of SBPs results in significant pruning of the search space and can speed up PB solvers as demonstrated in Section 5.

4.3 Handling an optimization function

To accommodate an optimization objective in 0-1 ILP instances, one has to intersect the symmetries of the PB constraints (which we already can detect) with the symmetries of the objective. Rather than find those two groups separately and compute the intersection explicitly, we modify our original graph construction to produce the intersection instantly.

The objective function is represented by a new vertex of a unique color[4] and coefficient vertices in the same way as PB constraints are represented. The function vertex connects to its coefficient vertices, which connect to literals appearing in the objective function with respective coefficients. This construction prohibits all PB symmetries that modify the objective function.

When symmetries are detected for PB constraints, their use through known SBPs for SAT symmetries is justified by the fact that we are still dealing with a constraint satisfaction problem on Boolean variables. However, additional reasoning is required to substantiate the use of the same SBPs in an optimization problem. The intuition here is that by breaking symmetries, one can speed up search without affecting the optimal cost in the optimization problem. We now show that adding SBPs preserves at least one optimal solution, and thus the optimal cost.

SBPs must pick at least one representative from every equivalence class under symmetry. If one truth assignment in such an orbit satisfies all PB constraints, then so do all assignments in the orbit. Furthermore, if an orbit contains such satisfying assignments, they all must have the same cost because they are symmetric. Given an optimization problem, there must be at least one solution with the optimal cost. By the arguments above, SBPs will preserve at least one solution from the same orbit, and that solution must have the same cost. Thus, the optimal cost is preserved.

4 Note that whether we are dealing with a maximization or a minimization objective does not affect symmetries, hence this information is ignored.

Table 1. Search runtimes of PB formulas with and without SBPs (for generators only) using PBS. Size of original instances and SBPs is shown. Symmetry statistics including symmetry detection runtime, number of symmetries, and generators are also provided. All runtimes are reported in seconds.

		Alternative PB encoding									
		Instance size					Symmetry statistics			PBS time	
Instance name	S/U	Orig			SBP		SAUCY	#	#	Orig	w/
		V	C	PB	V	C	time	Sym	Gen		SBP
hole-7	U	56	8	7	97	362	0.01	2.0E+08	13	0.11	0
hole-8	U	72	9	8	127	478	0.01	1.5E+10	15	0.64	0
hole-9	U	90	10	9	161	610	0.01	1.3E+12	17	7.35	0
hole-10	U	110	11	10	199	758	0.01	1.5E+14	19	66.3	0
hole-11	U	132	12	11	241	922	0.01	1.9E+16	21	431	0
fpga10_8	S	120	88	18	256	980	0.02	6.7E+11	22	349	0
fpga10_9	S	135	99	19	223	846	0.02	1.5E+13	23	>1000	0
fpga13_10	S	195	140	23	334	1280	0.06	1.9E+17	28	>1000	0.01
fpga13_11	S	215	154	24	371	1424	0.06	1.3E+19	30	>1000	0.03
fpga13_12	S	234	168	25	406	1560	0.08	9.0E+20	32	>1000	0.05
chnl10_11	U	220	22	20	508	1954	0.05	4.2E+28	39	65	0
chnl10_12	U	240	24	20	556	2142	0.06	6.0E+30	41	93	0
chnl10_13	U	260	26	20	604	2330	0.07	1.0E+33	43	112	0
chnl11_12	U	264	24	22	614	2370	0.07	7.3E+32	43	719	0
chnl11_13	U	286	26	22	667	2578	0.09	1.2E+35	45	743	0
chnl11_14	U	308	28	22	720	2786	0.10	2.4E+37	47	>1000	0
grout-3.3-1	S	216	572	12	24	92	0.01	4	2	0.04	0
grout-3.3-2	S	264	700	12	60	230	0.01	48	5	0.12	0
grout-3.3-3	S	240	636	12	60	230	0.01	32	5	0.05	0
grout-3.3-4	S	228	604	12	36	138	0.01	12	3	0.04	0
grout-3.3-5	S	240	634	12	48	184	0.02	16	4	0.01	0
grout-3.3u-1	U	624	1850	24	72	282	0.07	8	3	102	0.58
grout-3.3u-2	U	672	1988	24	144	564	0.11	96	6	353	2.14
grout-3.3u-3	U	624	1844	24	96	376	0.07	16	4	420	3.00
grout-3.3u-4	U	672	1994	24	216	846	0.17	1152	9	9.88	0.33
grout-3.3u-5	U	648	1924	24	264	1034	0.20	6912	11	14.7	0.05
Total	-	**7365**	**13595**	**460**	**7104**	**27356**	**1.41**	**2.4E37**	**530**	**>8487**	**6.19**

5 Experimental Results

In this section, we empirically show the advantage of breaking symmetries in formulas consisting of CNF and PB constraints. The experiments are performed on an Intel Pentium IV 2.8 GHz machine with 1 GB of RAM running Linux. The time-out limit for all experiments is set to 1000 seconds. The benchmarks include instances from the pigeon-hole [9], global routing (*grout*) [1], and FPGA routing (*fpga*, *chnl*) [20] set. We use the recent PB SAT solver PBS [1] with the settings "-D 1 -z". PBS incorporates modern CNF-SAT solver features implemented in Chaff [17] and can handle both CNF and PB constraints. We use the new graph automorphism tool SAUCY [7] which is empirically faster than NAUTY [16]. Symmetry-breaking predicates [3] are applied to generators of the symmetry group found by SAUCY.

Table 1 and Table 2 list symmetry detection runtimes, the number of symmetries, and symmetry generators. The size of the original formula and the SBP, in terms of the number of variables, clauses, and PB constraints, are shown too. The tables also compare runtimes for solving original instances and instances augmented with SBPs. Table 1 reports on the PB formulation and Table 2 reports on a CNF-only formulation derived by converting the PB constraints using the exponential transformation described in [1]. S/U indicates if the formula is satisfiable or unsatisfiable. We observe the following:

Table 2. Search runtimes of CNF formulas with and without SBPs (for generators only) using PBS. Size of original instances and SBPs is shown. Symmetry statistics including symmetry detection runtime, number of symmetries, and generators are also provided. All runtimes are reported in seconds. The CNF-only formulation is derived by converting the PB constraints using the exponential transformation described in [1].

		CNF-only encoding								
Instance name	S/U	Instance size				Symmetry statistics			PBS time	
		Orig		SBP		SAUCY time	# Sym	# Gen	Orig	w/ SBP
		V	C	V	C					
hole-7	U	56	204	97	362	0.01	2.0E+08	13	0.2	0
hole-8	U	72	297	127	478	0.01	1.5E+10	15	4.2	0
hole-9	U	90	415	161	610	0.01	1.3E+12	17	111	0
hole-10	U	110	561	199	758	0.01	1.5E+14	19	850	0
hole-11	U	132	738	241	922	0.02	1.9E+16	21	>1000	0.01
fpga10_8	S	120	448	256	980	0.01	6.7E+11	22	13.2	0
fpga10_9	S	135	549	223	846	0.02	1.5E+13	23	475	0
fpga13_10	S	195	905	334	1280	0.04	1.9E+17	28	>1000	0.02
fpga13_11	S	215	1070	371	1424	0.05	1.3E+19	30	>1000	0.02
fpga13_12	S	234	1242	406	1560	0.07	9.0E+20	32	>1000	0.02
chnl10_11	U	220	1122	508	1954	0.04	4.2E+28	39	628	0
chnl10_12	U	240	1344	556	2142	0.05	6.0E+30	41	>1000	0
chnl10_13	U	260	1586	604	2330	0.05	1.0E+33	43	>1000	0
chnl11_12	U	264	1476	614	2370	0.06	7.3E+32	43	>1000	0
chnl11_13	U	286	1742	667	2578	0.07	1.2E+35	45	>1000	0
chnl11_14	U	308	2030	720	2786	0.08	2.4E+37	47	>1000	0
grout-3.3-1	S	216	37292	24	92	2.11	4	2	0.07	0.05
grout-3.3-2	S	264	88480	60	230	18.15	48	5	0.21	0.11
grout-3.3-3	S	240	58776	60	230	10.34	32	5	0.11	0.05
grout-3.3-4	S	228	47116	36	138	3.04	12	3	0.28	0.05
grout-3.3-5	S	240	58774	48	184	7.8	16	4	0.09	0.1
grout-3.3u-1	U	624	360650	72	282	224	8	3	>1000	103
grout-3.3u-2	U	672	493388	144	564	686	96	6	30.2	11.2
grout-3.3u-3	U	624	360644	96	376	291	16	4	5.00	1.1
grout-3.3u-4	U	672	493394	n/a	n/a	>1000	n/a	n/a	2.03	n/a
grout-3.3u-5	U	648	423124	n/a	n/a	>1000	n/a	n/a	4.03	n/a
Total	-	**7365**	**2.4M**	**>6K**	**>25K**	**>3243**	**>2.4E37**	**>510**	**>12K**	**>116**

- All presented instances exhibit structural symmetries, but none of the instances have phase-shift symmetries.
- The pigeon-hole and FPGA routing instances contain large numbers of symmetries. These symmetries, however, are compactly represented using irredundant sets of no more than 50 generators.
- SAUCY detects all symmetries in each instance in a fraction of a second for PB formulas. However, formulas expressed in CNF-only form yield larger graphs on which SAUCY runs much slower.
- The addition of SBPs using the construction defined in [3] significantly reduces the SAT search runtime.
- Except for the *grout-3.3u-2* and *grout-3.3u-3* instances, all PB formulas are solved in less than a second after adding their SBPs. Note that the number of symmetries and generators is small in the *grout-3.3u-2* and *grout-3.3u-3* instances and so results in smaller speed-ups.
- In most cases, the SAT search runtimes are greater for CNF-only instances than for PB instances. An exception is the instance *grout-3.3u-3* which is solved in 1.2 seconds after adding the SBPs

Table 3. Results of the Max-SAT experiment.

Unsat instance			#UnSat	Symmetry statistics			PBS time	
Name	V	C		SAUCY time	# Sym	# Gen	Orig	w/ SBP
chnl7_9	126	522	4	0.47	6.7E+18	29	>1000	0.37
chnl8_9	144	594	2	0.56	4.3E+20	31	35	0.43
chnl8_10	160	740	4	1.03	4.3E+22	33	>1000	0.95
chnl9_10	180	830	2	1.10	3.5E+24	35	438	0.37
chnl9_11	198	1012	4	2.01	4.2E+26	37	>1000	10.8
hole-7	56	204	1	0.04	(7!)(8!)	13	0.32	0.01
hole-8	72	297	1	0.09	(8!)(9!)	15	7.51	0.01
hole-9	90	415	1	0.19	(9!)(10!)	17	76	0.03
hole-10	110	561	1	0.36	(10!)(11!)	19	>1000	0.02
hole-11	132	738	1	0.66	(11!)(12!)	21	>1000	0.06

Table 4. Results of the Max-ONE experiment.

Satisfiable instance			*MaxONEs*	Symmetry statistics			PBS time	
Name	V	C		SAUCY time	# Sym	# Gen	Orig	w/ SBP
fpga8_7	84	273	14	0.01	4.2E+08	17	>1000	0.01
fpga9_7	95	317	14	0.01	2.1E+09	18	>1000	0.01
fpga9_8	108	396	16	0.01	6.7E+10	20	>1000	0.01
fpga10_8	120	448	16	0.01	6.7E+11	22	>1000	0.01
5-queens	125	6460	5	0.02	8(5!)	6	18.1	0.04
6-queens	216	16320	6	0.03	8(6!)	7	>1000	0.64
7-queens	343	35588	7	0.09	8(7!)	8	>1000	9.87
8-queens	512	69776	8	0.27	8(8!)	9	>1000	214

to the CNF-only formula, compared to 3.4 seconds for the PB formula. Further investigation showed that this is a side effect of the VSIDS decision heuristic [17] implemented in PBS which tends to first choose variables that occur frequently in the formula. The exponential conversion replaces a single PB constraint with multiple CNF clauses, which naturally increases the number of variable occurrences. In any case, the symmetry detection runtime in the CNF-only case is 291 seconds versus 0.07 seconds in the PB case.

- SAT search runtime and symmetry detection runtime are not correlated.

Since we know that PB constraints can be expressed as pure CNF constraints (and vice versa), it is of interest to know whether symmetries are preserved in the CNF version of a PB formula. It is possible to convert a PB formula to CNF using exactly the same variables, but adding exponentially many clauses [1]. Such a conversion is itself symmetric and preserves all symmetries. However, symmetry detection runtimes are likely to be longer, since larger graphs are constructed. This is clearly shown in Tables 1 and 2. On the other hand, the linear-overhead conversion used in [1] for global routing uses *additional* variables to simulate "counting" constraints. This conversion avoids exponential overhead, but destroys symmetries from the PB formula because it uses *adder* and *comparator* circuits to enforce counting constraints. The comparator circuit has an inherent direction, and that is incompatible with symmetry.

In alternative experiments we replace PBS by the best commercial ILP solver CPLEX [13] (we use version 7.0). Surprisingly, symmetry-breaking slows down CPLEX. The specific algorithms used by CPLEX are not described publicly, and without such knowledge it is difficult to explain our empirical observations. However, it is known that some algorithms for Boolean Satisfiability, e.g., the heuristic solver WalkSAT [21], are also slowed down when symmetries are broken [19]. Yet, all major complete solvers are sped-up by symmetry-breaking [2].

In order to evaluate the advantage of breaking symmetries in Boolean optimization problems, we tested Max-SAT instances from the FPGA routing and pigeon-hole set, in addition to Max-ONEs instances from the FPGA routing and n-queens set. The goal in the Max-SAT experiment is to identify a variable assignment that maximizes the number of satisfied CNF clauses in an unsatisfiable instance. On the other hand, the goal in the Max-ONE experiment is to identify the variable assignment that maximizes the number of variables set to 1 in a satisfiable instance. The Max-SAT and Max-ONEs instances were constructed as shown in [1]. The results for the Max-SAT and Max-ONEs experiments are listed in Table 3 and Table 4, respectively. Both tables indicate the size of the original CNF formula. The tables also list the symmetry detection runtimes, number of symmetries, and symmetry generators. The runtimes for solving original instances and instances augmented with SBPs are also shown. "*Unsat*" in Table 3 indicates the minimum (i.e. optimal) number of original unsatisfiable clauses. "*MaxOnes*" in Table 4 indicates the maximum (i.e. optimal) number of 1s in a satisfying assignment.

We constructed *n*-queens instances in terms of CNF constraints (at most one queen per row, per column and per diagonal). Each of those instances has 8 geometric symmetries (of the board), and also *n!* permutational symmetries. All of those symmetries were detected in our experiments, and respective symmetry-breaking predicates significantly reduce overall runtime.

6 Conclusions

Our work is motivated by the desire to capture and exploit structural information in Boolean problems. We build upon previous work that (i) extends leading SAT-solvers to handle more expressive constraints, such as pseudo-Boolean constraints, as well as optimization objectives [1, 4, 5, 10], and (ii) offers pre-processing techniques for SAT that detect and use symmetries in CNF instances [2, 3, 6]. Our main contribution is a similarly efficient pre-processing for 0-1 ILP instances that detects and uses symmetries to speed up search and optimization. Our empirical validation shows that on some application-derived instances, such as FPGA routing, we obtain a speedup of several orders of magnitude after breaking symmetries. We also study the possibility of re-expressing PB constraints in terms of CNF and conclude that this may lead to the loss of symmetry information or cause a substantial increase in problem size.

Our on-going work includes studying possible optimizations of our graph construction. In addition, we hope to facilitate the use of more expressive constraints, e.g. PB, in new EDA applications.

Acknowledgments

This work is funded by the DARPA/MARCO Gigascale Silicon Research Center and by the National Science Foundation under Grant No. 0205288.

7 References

[1] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Generic ILP versus Specialized 0-1 ILP," in *Proc. of International Conference on Computer-Aided Design* (ICCAD), 450-457, 2002.

[2] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah, "Solving Difficult Instances of Boolean Satisfiability in the Presence of Symmetries," to appear in *IEEE Trans. on Computer Aided Design*, September 2003.

[3] F. A. Aloul, I. L. Markov, and K. A. Sakallah, "Shatter: Efficient Symmetry-Breaking for Boolean Satisfiability," in *Proc. of the Design Automation Conference* (DAC), 836-839, 2003.

[4] P. Barth, "A Davis-Putnam based Enumeration Algorithm for Linear Pseudo-Boolean Optimization," *Technical Report MPI-I-95-2-003, Max-Planck-Institut Für Informatik*, 1995.

[5] D. Chai and A. Kuehlmann, "A Fast Pseudo-Boolean Constraint Solver," in *Proc. of the Design Automation Conference* (DAC), 830-835, 2003.

[6] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, "Symmetry-breaking predicates for search prob-

lems," in *Proc. of the Intl. Conference Principles of Knowledge Representation and Reasoning*, 148-159, 1996.

[7] P. Darga, "SAUCY: Graph Automorphism Tool," *http://www.eecs.umich.edu/~pdarga/pub/auto/saucy.html*

[8] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," in *Communications of the ACM*, 5(7), 394-397, 1962.

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

[10] H. Dixon and M. Ginsberg, "Inference methods for a pseudo-Boolean satisfiability solver," in *Proc. of the National Conference on Artificial Intelligence* (AAAI), 635-640, 2002.

[11] M. Hall Jr., "The Theory of Groups", *McMillan*, 1959.

[12] E. Goldberg and Y. Novikov, "BerkMin: A fast and robust SAT-solver," in *Proc. of the Design, Automation, and Test in Europe Conference* (DATE), 142-149, 2002.

[13] ILOG CPLEX, *http://www.ilog.com/products/cplex.*

[14] V. Manquinho and J. Marques-Silva, "On Using Satisfiability-Based Pruning Techniques in Covering Algorithms," in *Proc. of the Design, Automation, and Test in Europe Conference* (DATE), 356-363, 2000.

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

[16] B. McKay, "NAUTY User's Guide, Version 1.5," Technical Report TR-CS-90-02, *Dep. of Computer Science, Australian Nat. Univ.*, 1990.

[17] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. of the Design Automation Conference* (DAC), 530-535, 2001.

[18] 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. Symposium on Physical Design* (ISPD), 222-227, 2001.

[19] S. Preswitch, "Supersymmetric Modelling for Local Search", in *Intl. Workshop on Symmetry on Constraint Satisfaction Problems*, 2002.

[20] SAT Competition 2002, *http://www.satcomp.org*

[21] B. Selman, H. Kautz, and B. Cohen. "Noise strategies for local search," in *Proc. of the National Conference on Artificial Intelligence* (AAAI), 337-343, 1994.

[22] J. Silva and K. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," in *IEEE Trans. on Computers.*, 48(5), 506-521, 1999.

[23] M. Velev and R. Bryant, "Boolean Satisfiability with Transitivity Constraints," in *Proc. of the Conference on Computer-Aided Verification* (CAV), 86-98, 2000.

[24] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," in *Proc. of the Design Automation Conference* (DAC), 542-545, 2001.