ShatterPB: Symmetry-Breaking for Pseudo-Boolean Formulas

Fadi A. Aloul[b], Arathi Ramani[a], Igor L. Markov[a], Karem A. Sakallah[a]
{faloul, ramania, imarkov, karem}@umich.edu
[a]Department of EECS, University of Michigan, Ann Arbor, USA
[b]School of Computer Engineering, American University in Dubai, UAE

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) [15] enabled new applications in design automation [1, 10, 16]. 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, 5, 6]. 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 [3, 4, 7]. 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.

In this paper, we contribute 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 [3, 4, 7]. 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).

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

Figure 1. 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

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

$$a_1 x_1 + a_2 x_2 + \ldots + a_n x_n \leq b \quad (1)$$

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 CNF Symmetries

Leading-edge complete SAT solvers [15] implement the basic Davis-Logemann-Loveland (DLL) algorithm [9] 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. The work in [3, 4, 7] 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. [7] presented a theoretical framework for detecting and using permutational symmetries in CNF formulas. An extension of this framework in [3] 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 [4]. 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 $(a + b + c \geq 1)$.

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 [14] and SAUCY [8].

Structural symmetries in CNF formulas can be detected via a reduction to graph automorphism [13]. 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 and negative) with color 2.

As the runtime of graph automorphism tools 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. This optimization can, in some cases, result in spurious graph automorphisms [3]. Fortunately, this is uncommon in CNF applications, and spurious graph symmetries are easy to test for [3].

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 [7]. 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. [7] 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. [4] 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 [4] 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 the generators[3] of the group of symmetries rather than the entire set of symmetries is sufficient to yield significant runtime and memory reductions.

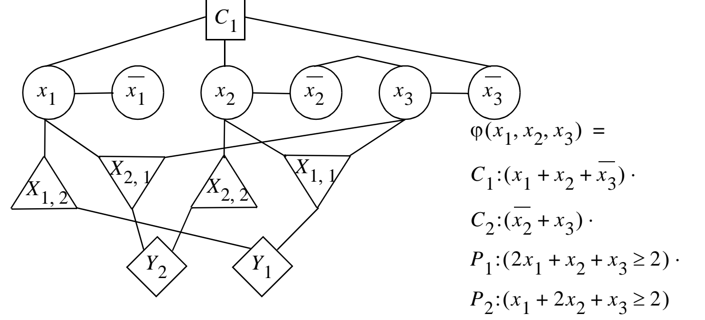

Figure 2. Example showing the graph representing formula φ. Different vertex shapes corresponds to different vertex colors.

4. Detecting and Using PB Symmetries

Similar to the techniques from [3] (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 [4] 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.
- Any non-PB (pure CNF) clauses are also treated just like in the CNF case.
- Clause vertices are painted in color 1; 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.

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].

Table 1. Search runtimes of PB formulas with and without SBPs (for generators only) using PBS.

Instance name	S/U	Alternative PB encoding										CNF-only encoding								
		Instance size					Symmetry statistics			PBS time		Instance size				Symmetry statistics			PBS time	
		Orig			SBP		SAUCY	#	#	Orig	w/	Orig		SBP		SAUCY	#	#	Orig	w/
		V	C	PB	V	C	time	Sym	Gen		SBP	V	C	V	C	time	Sym	Gen		SBP
hole7	U	56	8	7	97	362	0.01	2.0E+08	13	0.11	0	56	204	97	362	0.01	2.0E+08	13	0.2	0
hole8	U	72	9	8	127	478	0.01	1.5E+10	15	0.64	0	72	297	127	478	0.01	1.5E+10	15	4.2	0
hole9	U	90	10	9	161	610	0.01	1.3E+12	17	7.35	0	90	415	161	610	0.01	1.3E+12	17	111	0
hole10	U	110	11	10	199	758	0.01	1.5E+14	19	66.3	0	110	561	199	758	0.01	1.5E+14	19	850	0
hole11	U	132	12	11	241	922	0.01	1.9E+16	21	431	0	132	738	241	922	0.02	1.9E+16	21	>1000	0.01
fpga10_8	S	120	88	18	256	980	0.02	6.7E+11	22	349	0	120	448	256	980	0.01	6.7E+11	22	13.2	0
fpga10_9	S	135	99	19	223	846	0.02	1.5E+13	23	>1000	0	135	549	223	846	0.02	1.5E+13	23	475	0
fpga13_10	S	195	140	23	334	1280	0.06	1.9E+17	28	>1000	0.01	195	905	334	1280	0.04	1.9E+17	28	>1000	0.02
fpga13_11	S	215	154	24	371	1424	0.06	1.3E+19	30	>1000	0.03	215	1070	371	1424	0.05	1.3E+19	30	>1000	0.02
fpga13_12	S	234	168	25	406	1560	0.08	9.0E+20	32	>1000	0.05	234	1242	406	1560	0.07	9.0E+20	32	>1000	0.02
chnl10_11	U	220	22	20	508	1954	0.05	4.2E+28	39	65	0	220	1122	508	1954	0.04	4.2E+28	39	628	0
chnl10_12	U	240	24	20	556	2142	0.06	6.0E+30	41	93	0	240	1344	556	2142	0.05	6.0E+30	41	>1000	0
chnl10_13	U	260	26	20	604	2330	0.07	1.0E+33	43	112	0	260	1586	604	2330	0.05	1.0E+33	43	>1000	0
chnl11_12	U	264	24	22	614	2370	0.07	7.3E+32	43	719	0	264	1476	614	2370	0.06	7.3E+32	43	>1000	0
chnl11_13	U	286	26	22	667	2578	0.09	1.2E+35	45	743	0	286	1742	667	2578	0.07	1.2E+35	45	>1000	0
chnl11_14	U	308	28	22	720	2786	0.10	2.4E+37	47	>1000	0	308	2030	720	2786	0.08	2.4E+37	47	>1000	0
grout-3.3-1	S	216	572	12	24	92	0.01	4	2	0.04	0	216	37292	24	92	2.11	4	2	0.07	0.05
grout-3.3-2	S	264	700	12	60	230	0.01	48	5	0.12	0	264	88480	60	230	18.15	48	5	0.21	0.11
grout-3.3-3	S	240	636	12	60	230	0.01	32	5	0.05	0	240	58776	60	230	10.34	32	5	0.11	0.05
grout-3.3-4	S	228	604	12	36	138	0.01	12	3	0.04	0	228	47116	36	138	3.04	12	3	0.28	0.05
grout-3.3-5	S	240	634	12	48	184	0.02	16	4	0.01	0	240	58774	48	184	7.8	16	4	0.09	0.1
grout-3.3u-1	U	624	1850	24	72	282	0.07	8	3	102	0.58	624	360650	72	282	224	8	3	>1000	103
grout-3.3u-2	U	672	1988	24	144	564	0.11	96	6	353	2.14	672	493388	144	564	686	96	6	30.2	11.2
grout-3.3u-3	U	624	1844	24	96	376	0.07	16	4	420	3.00	624	360644	96	376	291	16	4	5.00	1.1
grout-3.3u-4	U	672	1994	24	216	846	0.17	1152	9	9.88	0.33	672	493394	n/a	n/a	>1000	n/a	n/a	2.03	n/a
grout-3.3u-5	U	648	1924	24	264	1034	0.20	6912	11	14.7	0.05	648	423124	n/a	n/a	>1000	n/a	n/a	4.03	n/a
Total	-	**7365**	**13595**	**460**	**7104**	**27356**	**1.41**	**2.4E37**	**530**	**>8487**	**6.19**	**7365**	**2.4M**	**>6K**	**>25K**	**>3243**	**>2.4E37**	**>510**	**>12K**	**>116**

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. Additional information, including the proof of correctness, can be found in [2].

4.2 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 (Note that whether we are dealing with a maximization or a minimization objective does not affect symmetries, hence this information is ignored) 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. All satisfying assignments in an orbit 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.

5. Experimental Results

Below we empirically evaluate symmetry-breaking in 0-1 ILP. We use an Intel Pentium IV 2.8 GHz machine with 1 GB of RAM running Linux. All time-outs are 1000 seconds. Our benchmarks include instances from the pigeon-hole [10] (*hole*), global routing (*grout*) [1], and FPGA routing (*fpga*, *chnl*) [18] sets. We use the PB SAT solver PBS [1] (with settings "-D 1 -z") which incorporates modern techniques for CNF-SAT implemented in Chaff [15] and also handles PB constraints. We use the new graph automorphism tool SAUCY [8] which is empirically faster than NAUTY [14] on all our benchmarks. Symmetry-breaking predicates from [4] are applied to generators of the symmetry groups found by SAUCY.

Table 1 lists 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 table also compares runtimes for solving original instances and instances augmented with SBPs. We also report 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:

- All our instances have structural symmetries, but none of those are phase-shift symmetries.
- The *hole* and FPGA routing instances contain large numbers of symmetries, which 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. Formulas expressed in CNF-only form yield larger graphs on which SAUCY runs much slower.
- The addition of SBPs using the construction defined in [4] 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 <1s with 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.
- Typically SAT search runtimes for CNF-only instances exceed those for PB instances. An exception is the instance *grout-3.3u-3* which is solved in 1.1 sec with SBPs added to the CNF-only formula, compared to 3 seconds for the PB formula. We found that this is a side-effect of the VSIDS decision heuristic [15] used in PBS which prefers frequently-occurring variables. Indeed, the conversion to CNF replaces a single PB constraint with multiple CNF clauses, making some variables more frequent. In any case, the symmetry detection runtime in the CNF-only case is 291 seconds versus 0.07 seconds in the PB case.
- Runtimes of SAT-search and symmetry-finding do not correlate.

PB constraints can be expressed as pure CNF constraints (and vice versa), but symmetries are not necessarily preserved during re-expression. One such conversion does not add variables, but adds exponentially many clauses [1]. While it preserves all symmetries, symmetry detection runtimes significantly increase, as seen from Table 1. An alternate linear-overhead conversion used in [1] for global routing uses *additional* variables to simulate "counting" constraints. It avoids exponential overhead, but obscures original symmetries because it uses *adder* and *comparator* circuits to enforce counting constraints. The directional nature of the comparator is incompatible with symmetry.

In alternate experiments we replace PBS by the best commercial ILP solver CPLEX [12] (version 7.0) and found that symmetry-breaking slows down CPLEX. We cannot currently explain this because the specific algorithms used by CPLEX are not described publicly. It is known that symmetry-breaking slows down stochastic search for Boolean Satisfiability [17], e.g., the heuristic solver WalkSAT [19]. Yet, all major complete SAT solvers are sped-up by symmetry-breaking [3].

To evaluate symmetry-breaking in Boolean optimization problems, we tested Max-SAT instances from FPGA routing and the optimization version of the pigeon-hole problem in addition to Max-ONEs instances from the FPGA routing and n-queens set. Max-SAT problems seek a variable assignment to maximizes the number of satisfied CNF clauses, and Max-ONE instances seek to maximize the number of variables set to 1 in a satisfiable instance. The Max-SAT and Max-ONEs instances were constructed following [1]. The results of relevant experiments are given in Table 2 and Table 3, respectively. The tables show symmetry detection runtimes, number of symmetries, and symmetry generators. Runtimes for solving original instances versus instances augmented with SBPs are also shown. "*Unsat*" in Table 2 indicates the minimum (i.e. optimal) number of original unsatisfiable clauses. "*MaxOnes*" in Table 3 gives the optimal number of 1s in a satisfying assignment. Our instances contain a large number of symmetries, and are solved much faster when symmetry-breaking is used.

6. Conclusions

Our work seeks to capture and exploit structure in Boolean problems. We describe how to pre-process 0-1 ILP instances to detect symmetries and use them to speed up search and optimization. Empirically, we obtain a speedup of several orders of magnitude on some application-derived instances, e.g., FPGA routing. We show that re-expressing PB constraints in terms of CNF may lead to the loss of symmetry information or cause a substantial increase in problem size. Ongoing work deals with (i) improved graph constructions, and (ii) EDA applications.

Table 2. 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
hole7	56	204	1	0.04	(7!)(8!)	13	0.32	0.01
hole8	72	297	1	0.09	(8!)(9!)	15	7.51	0.01
hole9	90	415	1	0.19	(9!)(10!)	17	76	0.03
hole10	110	561	1	0.36	(10!)(11!)	19	>1000	0.02
hole11	132	738	1	0.66	(11!)(12!)	21	>1000	0.06

Table 3. 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

Acknowledgments. This work was funded in part by NSF ITR Grant #0205288.

7. References

[1] F. Aloul, A. Ramani, I. L. Markov, and K. Sakallah, "Generic ILP versus Specialized 0-1 ILP," in *Proc. ICCAD*, 450-457, 2002.

[2] F. Aloul, A. Ramani, I. L. Markov, and K. Sakallah, "Symmetry-Breaking for Pseudo-Boolean Formulas," in *SymCon*, 1-12, 2003.

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

[4] F. Aloul, I. L. Markov, and K. Sakallah, "Shatter: Efficient Symmetry-Breaking for Boolean Satisfiability," in *Proc. DAC,* 836-839, 2003.

[5] 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.

[6] D. Chai and A. Kuehlmann, "A Fast Pseudo-Boolean Constraint Solver," in *Proc. DAC,* 830-835, 2003.

[7] 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.

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

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

[10] DIMACS Challenge benchmarks, ftp://Dimacs.rutgers.EDU/pub/challenge/sat/benchmarks/cnf

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

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

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

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

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

[16] 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* (ISPD), 222-227, 2001.

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

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

[19] B. Selman, H. Kautz, and B. Cohen. "Noise strategies for local search," in *Proc. AAAI*, 337-343, 1994.