Solving Difficult SAT Instances in the Presence of Symmetry

Fadi A. Aloul, Arathi Ramani, Igor L. Markov and Karem A. Sakallah
Department of EECS, University of Michigan, Ann Arbor 48109-2122
{faloul,ramania,imarkov,karem}@eecs.umich.edu

ABSTRACT

Research in algorithms for Boolean satisfiability and their implementations [23, 6] has recently outpaced benchmarking efforts. Most of the classic DIMACS benchmarks [10] can be solved in seconds on commodity PCs. More recent benchmarks take longer to solve because of their large size, but are still solved in minutes [25]. Yet, small and difficult SAT instances must exist because Boolean satisfiability is NP-complete.

We propose an improved construction of symmetry-breaking clauses [9] and apply it to achieve significant speed-ups over current state-of-the-art in Boolean satisfiability. Our techniques are formulated as preprocessing and can be applied to any SAT solver without changing its source code. We also show that considerations of symmetry may lead to more efficient reductions to SAT in the routing domain.

Our work articulates SAT instances that are unusually difficult for their size, including satisfiable instances derived from routing problems. Using an efficient implementation to solve the graph automorphism problem [18, 20, 22], we show that in structured SAT instances difficulty may be associated with large numbers of symmetries.

Categories and Subject Descriptors

I.1.2 [**Algorithms**]: Algebraic algorithms.

General Terms

Algorithms, experimentation, verification.

Keywords

SAT, CNF, faster, search, symmetry, difficult, instances, speed-up.

1. INTRODUCTION

Boolean satisfiability (SAT) is a pivotal problem in Computer Science and has numerous applications in Design Automation that range from microprocessor verification [25] to FPGA layout [19]. A one-million-dollar prize is offered by the Clay Institute for Mathematical Sciences for a complete polynomial-time SAT solver or a proof that such an algorithm does not exist (the P-vs-NP problem). Neither is likely to be found. Nevertheless, industrial applications motivate intensive research in SAT algorithms that quickly solve real-life instances. The fundamental framework for state-of-the-art SAT algorithms was laid out in the 1960s, but a number of recent improvements in algorithms and implementation techniques [23, 6] have led to performance breakthroughs. Most DIMACS challenge benchmarks [10] from the early 1990s are now solved in seconds on commodity PCs. Recently posted SAT benchmarks [25] take somewhat longer to solve (minutes), but that is primarily due to their enormous size (50MB+ files, etc). With the exception of artificially constructed families of benchmarks, it appears that SAT can be solved in polynomial time "for practical purposes".

It is well known that the dominant back-track solvers, such as GRASP [23] and CHAFF [6] do not perform well on randomly-created 3-SAT instances with ≈ 4.3 clauses per variable. However, such instances do not arise in Design Automation because application-derived SAT instances are typically structured. Attempts to explain the ease of structured instances were successful for certain applications [21], and generic ways to exploit certain types of structure were proposed [1].

Our work addresses both benchmarking and algorithmic aspects of SAT research. Given the excellent performance of existing SAT solvers, there is no room for improvement on easy benchmarks, and we focus on difficult instances.[1] Since the works of Haken and Urquhart [24] on lower bounds for resolution and back-tracking algorithms for SAT, several instance families have been known to require exponential time for DP/DLL (Davis-Putnam and Davis-Logemann-Loveland) solvers. For example, a recent lower bound for the pigeon-hole problem is $\Omega(2^{n/20})$ [2] where n is the number of pigeons. Another such family was constructed by Urquhart in terms of expander graphs and with considerable use of randomization [24]. Indeed, state-of-the-art SAT solvers, such as CHAFF, take a long time to solve those instances (see Table 1), but the relevance of such pathological cases to Design Automation is questionable. While lower bounds for SAT are often proven for unsatisfiable instances, it remains to be seen whether satisfiable instances can be difficult for the best solvers. We demonstrate CAD-related SAT instances, both satisfiable and unsatisfiable, that are very difficult for their size. Moreover, an easy instance of any size can be made difficult by adding a small difficult instance to it and connecting the two by inconsequential clauses to defeat partitioning.

Over many years, empirical research in algorithms for Design Automation identified a number of fundamental problem formulations, such as Boolean satisfiability, and mustered significant efforts to solve them efficiently. State of the art is gauged by optimized solver implementations ("engines"). Performance break-throughs are often due to novel algorithmic ideas, leaner implementations or the ability to apply a highly optimized engine in a novel way. In this work, we suggest that graph automorphism engines can be applied to the satisfiability problem in certain cases. Given that the graph automorphism problem is thought to not be NP-complete (thus potentially easier than SAT) and that very little CAD research was done on high-performance engines for graph automorphism (one such work is [16]), there may be significant room for future improvement. To be precise, we will be dealing with the colored variant of the graph automorphism problem that can be easily extended to hypergraphs (see definitions in Section 2).

Several works suggested that "breaking symmetries" in CNF formulae can speed up SAT solvers [3, 4, 5, 9, 16]. Symmetries of a CNF formula include clause-preserving permutations of variables. Such permutations may involve arbitrarily many variables at once, e.g., a complete cyclic shift. In this work, we do not address permutations that change the CNF formula but leave unchanged the Boolean function it represents.[2] However, if such symmetries are detected by other techniques [14], our proposed methods can process them in the same way as

DAC 2002 June 10-14, 2002, New Orleans, Louisiana, USA

[1] In practice, the difficulty of domain-specific classes of SAT instances is often known, and adequate SAT algorithms can be chosen. Otherwise, one can run several SAT solvers in parallel until one of them finishes. On a single processor, this may buy exponential speed-ups at the cost of a constant-factor slow-down.

[2] Such permutations can be called "semantic" symmetries versus "syntactic" symmetries that leave the CNF formula unchanged.

symmetries of the CNF formula. Similarly, many of the works we cite do not deal with symmetry detection, but rather assume that symmetries of the Boolean function are given. Using this assumption, two main directions were explored: (a) preprocessing the original CNF formula by adding symmetry-breaking clauses that do not affect satisfiability but speed up search [9], (b) extending SAT solvers, particularly those based on back-tracking, to dynamically use symmetries during the search process [5]. In this paper we pursue the pre-processing approach due to its simplicity, but will outline how our techniques can be applied within a back-tracking solver for increased efficiency.

Prior works on symmetries in SAT predate recent breakthroughs in SAT solvers and typically use several carefully constructed instances to illustrate their approach. E.g., Crawford et al. [9] suggest that symmetry-based techniques allow the pigeon-hole instances to be solved in polynomial time, but their empirical data [9, Figure 3] do not support this suggestion. Also it remains unclear whether the performance of leading-edge SAT solvers can be improved via the use of symmetries. In principle, the overhead due to symmetry detection and usage may outweigh the benefits, and it remains to be seen that useful CNF formulae have many symmetries. Pólya (1937), Erdös and Rényi (1963) proved that a random graph on n vertices has *no symmetries* with probability $1 - \binom{n}{2}2^{-n-2}(1+o(1))$ [12, p. 1461]. This claim can be extended to CNF formulae using constructions in Section 2, but structured real-world instances may have richer symmetries. Indeed, Boolean functions from synthesis applications may have many symmetries [14]. If exponentially many symmetries exist, adding all possible symmetry-breaking clauses can be disastrous [9]. Despite these pitfalls, symmetry-based approaches have been useful in model checking [13, 7], verification [16], logic synthesis [15] and DSP algorithms [11].

In this work, we propose an automated flow that starts with a CNF formula in the DIMACS format and detects all of its symmetries (not just pairwise swaps). In this flow, all symmetries are captured implicitly, with exponential compression. The CNF formula is then preprocessed adding symmetry-breaking clauses that do not affect satisfiability. A black-box SAT solver is applied to the preprocessed CNF instance to produce the final answer; any satisfying assignment to this instance is (or corresponds to) a satisfying assignment of the original instance, and if the preprocessed instance is unsatisfiable then so is the original instance.

Our construction of symmetry-breaking clauses is novel. It is more economical and provides better coverage than that in [9]. Additionally, it directly applies to the compressed representation of all symmetries in the format produced by graph automorphism software [17, 18, 20, 22]. Our empirical results show significant improvements on CNF instances arising in Design Automation applications as well as highly randomized provably-difficult Urquhart benchmarks [24]. Two extensions are proposed to speed up symmetry detection. One is opportunistic symmetry detection, where only some symmetries are found. The other extension pursues domain-specific symmetries and leads to improvements of SAT formulations by adding domain-specific symmetry-breaking clauses. Thus generic symmetry detection is avoided by creating symmetry-less SAT instances that can be solved quickly.

The remaining material is organized as follows. Symmetry detection is described in Section 2 and symmetry-breaking in Section 3. Section 4 discusses constructions of SAT benchmarks. Our empirical results are presented in Section 5 and further extensions in Section 6.

2. FINDING SYMMETRIES

In general, a symmetry of a discrete object is a permutation of its components that leaves the object unchanged. Every discrete object has at least one symmetry — the "do-nothing" permutation. It is easy to see that a composition of two symmetries is a symmetry, and that composition with the do-nothing permutation does not change a symmetry. The composition of symmetries is associative, and every symmetry has an inverse. Composition is often *not* commutative. Abstract algebraic structures defined axiomatically in terms of such a composition operation (multiplication) are commonly called *groups*. In this work we will only deal with groups of symmetries whose elements can be thought of as permutations. A permutation can be represented by cycles, e.g., $(23)(567)$ represents a permutation on a set of at least 7 mark (elements). This permutation swaps marks 2 and 3, it cycles marks 5, 6 and 7 in that order. All other marks, e.g., 1 and 4, are left unchanged.

Computational group theory is approximately 25 years old, and great strides were made in the last decade with the development of the GAP package ("Groups, Algebra and Programming") [22]. A major efficiency in computational group theory comes from the notion of irredundant sets of generators of a group. A set of generators is made of group elements such that any other group element can be composed of generators and their inverses (no uniqueness required). Elementary group theory implies that any irredundant set of generators for any group with $N > 1$ elements contains *at most* $\log_2 N$ elements, e.g., the $k!$ permutations on k marks can be generated by (12) and $(12..k)$. Thus, representing groups by sets of generators *always ensures exponential compression*. Computational group theory provides efficient algorithms for manipulating groups represented by sets of generators, without decompression. Therefore, an intelligent algorithm for symmetry detection may return a small set of generators rather than list all symmetries.

COLORED AUTOMORPHISM PROBLEMS. Given a graph, a *symmetry* is a permutation of its vertices that maps edges to edges. In case of directed graphs, edge orientations must be preserved. In the Graph Automorphism problem one seeks all symmetries of a given graph, e.g., in terms of group generators. It is known that all graphs except for an exponentially small family have *no symmetries* [12, p. 1461]. No worst-case polynomial-time algorithms are known for this problem, but it is commonly believed not to be NP-complete unless P=NP. Polynomial-time algorithms are available in many special cases [12, p. 1511]. Generic algorithms [17, 16] are based on linear-time partition refinement passes; a simple version finishes in three passes for all but an exponentially small family of graphs [12, p. 1513].

The Graph Automorphism problem may be constrained by vertex labels — symmetries must map each vertex into a vertex with the same label. Label constraints are computationally easy and can be formally reduced to plain graph automorphism. Labels are often expressed by integers and called colors (no relation to *graph coloring*). Another extension is to colored *hypergraphs* — symmetries must map hyperedges to hyperedges (of the same cardinality because no two vertices can map to one). The colored hypergraph automorphism problem reduces to the colored graph automorphism via the bipartite graph of the hypergraph. This graph contains a vertex for each hypergraph vertex and hyperedge, and connects them with edges according to the hypergraph's incidence relation. Graph vertices in the hyper-edge part are painted with a new color, and other vertices retain their original colors.

Brendan McKay implemented a practical algorithm for Graph Automorphism [17] in a software package called NAUTY [18], which has been continually improved for the last 20 years (version 2.0 released in 2001). NAUTY has been integrated into the computational group theory system GAP [22] by means of the GRAPE package [20]. This integration enables efficient group-theoretic operations on the results returned by NAUTY and facilitates some of our proposed algorithms. In 1998, Manku et al. [16] claimed speed-ups over a pre-2.0 version of NAUTY in the context of hardware verification. However, their code is not generic (built into a larger system) and is no longer supported.

CNF SYMMETRIES VIA GRAPH AUTOMORPHISM. The problem of finding symmetries of a CNF formula is reduced to colored graph automorphism, similarly to the reduction from the hypergraph automorphism outlined above. Every variable is represented by two vertices that correspond to the positive and negative literals. Every clause is represented by a vertex, and bipartite edges connect those vertices to vertices of relevant literals. Clause vertices are painted with color 1

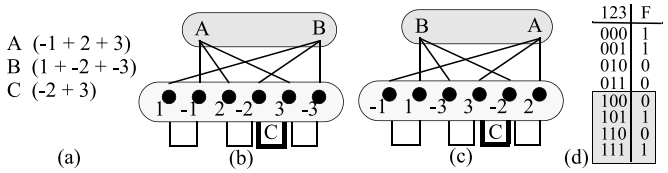

Figure 1: A CNF formula with three clauses A, B and C and three variables (a) is converted into a bipartite graph (b) for symmetry detection purposes. The two-literal clause C is represented by one edge (bold) while larger clauses A and B are represented by a vertex and three edges each. Any symmetry must map $C \rightarrow C$ therefore this instance has only one non-trivial symmetry (1 -1)(2 -3)(-2 3)(A B) shown in (c). The first cycle yields a symmetry-breaking clause $(\bar{1})$ which reduces the search space by half (d). Alternatively, the clause (2 + 3) corresponding to the third cycle can be added.

and literal vertices are painted with color 2. To ensure Boolean consistency, vertices of opposite literals are mated by direct edges. An additional simplification, motivated by [9, footnote 6], is to represent each two-literal clause by an edge directly connecting their two literals rather than by two edges and a vertex. In the final colored graph, $2 * Vars$ vertices represent the positive and negative literals and the remaining $Clauses - 2 * LitClauses$ vertices represent clauses. Figure 1 shows an example. This reduction has the advantage of detecting phase-shift symmetries ($a \rightarrow \bar{a}$) and their compositions with permutational symmetries.

3. SYMMETRY-BREAKING

Symmetries induce equivalence classes in the solution space (in group theory, they are called *orbits*). Given a satisfying truth assignment, all other truth assignments to which it can be mapped by symmetries, must also be satisfying. Similarly, symmetries always map unsatisfying assignments to unsatisfying assignments. Therefore, for a complete SAT solver it suffices to reason about one representative from each such class. This restriction can be implemented by selecting unique representatives from every equivalence class and adding clauses that are only satisfied on those representatives. An earlier construction of such symmetry-breaking clauses [9] is based on a given ordering of variables. Its main idea is (i) to order all elements from the solution space lexicographically, and (ii) to select the lexicographically smallest element from each equivalence class as its representative.

The construction described in [9] is applied to every given symmetry and generates many redundant clauses. To prune redundant clauses, the authors propose the concept of a symmetry tree, but it is not well supported by efficient algorithms for permutation groups, does not always prevent redundant clauses and is itself not always prunable to polynomial size [9]. Phase-shift symmetries were not addressed in that work.

We compute symmetry-breaking clauses on a per-symmetry basis [9], but consider only irredundant sets of symmetry generators, returned by graph automorphism programs. By breaking generator symmetries only, one does not necessarily break all symmetries, except for some cases [9]. So far, the power of such partial symmetry-breaking has not been evaluated, but we believe that a reasonable coverage is often achieved because an irredundant set of generators contains "maximally independent" symmetries — none of them can be expressed in terms of others.

Our construction is formulated in terms of cycles of a permutation (cf. [9]). For the variable swap (ab) the construction in [9] entails one additional variable and six symmetry-breaking clauses. Our construction below entails only one clause. First observe that if the cycle (ab) is a symmetry, whenever there is a satisfying assignment with $a = 0, b = 1$, there should be a symmetric (equivalent) satisfying assignment with $a = 1, b = 0$ and other variables unchanged. To allow only the first assignment, we add the symmetry-breaking clause $(\bar{a} + b)$, which can also be interpreted as $(a \leq b)$. Similarly, to "break" a cycle of length three (abc), we add $(\bar{a} + b)(\bar{b} + c)$, i.e., $(a \leq b)(b \leq c)$. To prevent transitivity violations, one has to choose an ordering of all variables at the beginning, and always use the $\leq$ sign consistently with that ordering.

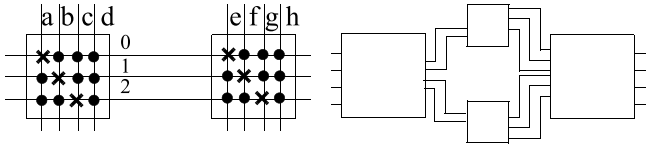

Figure 2: Construction of difficult SAT instances: (left) two switch-boxes in common FPGA architectures, (right) similar N-by-M switch-boxes are used to build hard satisfiable instances.

Longer cycles require more complex symmetry-breaking clauses, but one can always improve on the construction from [9]. We observed that symmetry generators produced for CNF formulae often consist of just 2-cycles and only rarely have 3-cycles. A cycle $(\bar{a}a)$ means that the value of a can be fixed arbitrarily, and this can be expressed by a one-literal symmetry-breaking clause. The construction in [9] does not address such phase-shift symmetries and never results in one-literal clauses.

If no cycles generate single-literal clauses (which achieve maximal pruning if all clauses have length ≤ 2), we can produce symmetry-breaking clauses from any one 2- or 3-cycle of a symmetry. That significantly speeds up SAT solvers in many cases. However, clauses of the form $(\bar{a} + b)$ achieve no pruning in areas of the solution space where the variables involved have identical values. A key idea in that case, similar to that in [9] is to process another cycle. Namely, for a symmetry $(ab)(cd)(ef)...$, we first add $(\bar{a} + b)$, then $(a = b) \Rightarrow (c \leq d)$, then $((a = b)(c = d)) \Rightarrow (e \leq f)$, etc. This construction can be efficiently implemented with one additional variable per cycle to indicate the equality of all variables in the cycle. A sample clause with new variables looks like $(\bar{x}_{a=b} + \bar{x}_{c=d} + \bar{e} + f)$. To ensure consistency, we sort marks within cycles and sort cycles by their first marks.

The construction of symmetry-breaking clauses is dwarfed by the symmetry-detection time. However, with every cycle processed, we add larger and larger symmetry-breaking clauses. Since large clauses typically do not have a great effect on the behavior of SAT solvers, we optionally limit symmetry-breaking clauses to the first 10 cycles of every symmetry. For the price of incomplete coverage, this technique considerably reduces the overhead of symmetry-breaking clauses. In our experiments it often performed better than the addition of symmetry-breaking clauses for all cycles. Moreover, extending back-track algorithms for SAT to dynamically check conditions of the form $((a = b)(c = d)...(u = v))$ may lead to improvements over pure pre-processing.

4. DIFFICULT SAT INSTANCES

FPGA ROUTING INSTANCES. A recent comparative study of two Boolean formulations of FPGA detailed routing constraints [19] showed that problem encoding affects the difficulty of SAT instances. Our work uses the better formulation, but still produces difficult instances. Two such constructions are shown in Figure 2 in terms of FPGA switch-boxes (see [19] for details on SAT formulations). The one on the left entails routing $N + k$ connections through N tracks and yields unsatisfiable instances that for $k = 1$ resemble the well-known pigeon-hole benchmarks. Empirical results in Table 1 are shown for six routing configurations (`chnl`) in which one tries to route (a) 11, 12 or 13 connections through 10 tracks, and (b) 12, 13 or 20 connections through 11 tracks. These instances are extremely difficult for the leading-edge SAT solver CHAFF [6] and also have many symmetries. They can appear as sub-instances in larger routing instances, and such sub-instances may be difficult to find.

From the benchmarking point of view, it is natural to expect *unsatisfiable* instances among the most difficult to solve. Indeed, randomized restarts used by CHAFF [6] typically allow it to avoid difficult regions of the search space and to quickly find satisfying solutions if they exist. However, our second construction is designed to create difficult *satisfiable* instances that trap even the best solvers in hopeless regions of their solution space for a long time before a satisfying solution can be found. The main idea is to create a satisfiable instance with a large number of hard-to-avoid unsatisfiable sub-instances. If the number of unsatisfiable

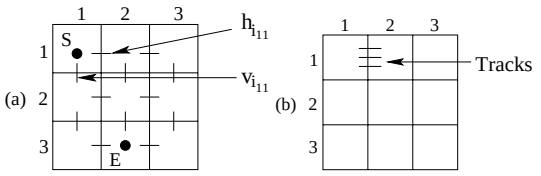

Figure 3: Construction of difficult SAT instances (global routing).

branches is much larger than the number of satisfiable branches, then random restart will keep on jumping from one unsatisfiable branch to another for a long time. Solvers without random restarts will, too, need to prove the unsatisfiability of many branches.

Our second construction entails routing a number of wires through four N-by-K FPGA switch-boxes of the type used in the first construction. The rightmost switch-box in the configuration in Figure 2 has several redundant outgoing tracks that are divided into two channels. Each channel is connected to a smaller switch-box with an insufficient number of outgoing tracks. The two groups of tracks that leave the smaller switch-boxes are connected to the left-most switch-box. When routing connections through tracks right-to-left, connections must be split between switch-boxes subject to the throughput constraints of switch-boxes. However, to a SAT solver, the throughput constraints are obscured by the pigeon-hole principle. SAT solvers first partition the connections between the two channels and back-track from every partition that does not lead to a satisfying assignment. If the capacities of the two channels leading to the smaller switchboxes are greater than the throughput of those switchboxes, an overwhelming majority of partitions will lead to unsatisfiable pigeon-hole instances. On average, at least several such instances must be solved before a good partition is found. SAT solvers without random restarts will also need to solve many pigeon-hole sub-instances before finding satisfying solutions. Empirical results for these satisfiable instances (`fpga`) in Table 1 show that they are very difficult for CHAFF. We observe that these instances become more difficult when the difference between the throughput of the small switchboxes and the capacities of the channels that lead to them is increased. This is consistent with our observations for the unsatisfiable `chnl` instances.

GLOBAL ROUTING INSTANCES. We propose a new construction of difficult randomized *satisfiable* instances unrelated to pigeonholes. They express routing two-pin connections in a grid with edge capacity constraints. To ensure that an instance is satisfiable but difficult, we use *randomized flooding*. Namely, we create a routing configuration by adding shortest possible routes while unused routing resources (edge capacities) remain. Shortest routes are created by breadth-first-search between pairs of randomly chosen grid cells or, if that fails, by finding a maximal shortest route starting at a given grid cell with unused routing resources. After a routing configuration is created, routes are erased and their end-points are used to formulate a SAT instance.

Our SAT encoding of routing instances has two components. One deals with *route definition* and captures possible ways to route each connection. The other addresses *capacity constraints* and restricts the number of connections that can be routed across a grid cell boundary.

Route definition. Routes are specified in terms of edges across cell boundaries in a grid. For each connection, there are routing tracks across each cell boundary on the grid. In the SAT formulation, each track is treated as a variable. Figure 3 (a) illustrates routing tracks in a 3-by-3 grid. Horizontal tracks for connection i are labeled $h_{i_{r,c}}$, where r and c are the row and column indices of the cell whose boundary the track crosses. Vertical tracks are labeled $v_{i_{r,c}}$. In Figure 3 (a), let the points marked S and E be the terminals of some two-terminal connection i. The SAT formulation proceeds as follows. Consider the terminal marked S. A route for this connection must pass through $h_{i_{1,1}}$ or $v_{i_{1,1}}$. Therefore, we add the clause $(h_{i_{1,1}} + v_{i_{1,1}})$. Since these two track selections are incompatible, we add the mutual exclusion clause $(\overline{h}_{i_{1,1}} + \overline{v}_{i_{1,1}})$.

We now push the cells reachable from the possible tracks into a queue. The queue contains cells reachable from those already visited. A list of visited cells is also maintained so that a cell is not pushed on the queue twice. While the queue is not empty, cells are popped off it and new clauses are introduced for the route tracks across the cell boundaries. In our example, assume that the cell to the right of S is popped off the queue. Since this cell is not an endpoint of the connection, exactly two of its boundaries must be selected. The cell boundaries in this case are $h_{i_{1,2}}, h_{i_{1,1}}$ and $v_{i_{1,2}}$. We therefore introduce the clauses $(h_{i_{1,1}} + v_{i_{1,2}})$, $(h_{i_{1,1}} + h_{i_{1,2}})$, and $(h_{i_{1,2}} + v_{i_{1,2}})$. However, again it is not possible for more than two tracks to be selected. Therefore, we add clauses of the form: $(h_{i_{1,1}} \wedge v_{i_{1,2}}) \Rightarrow h_{i_{1,2}}$. This procedure is repeated for every cell popped off the queue until the queue is empty.

Capacity constraints. Each grid cell boundary has a capacity associated with it to restrict the number of connections that can be routed through it. The capacity limits are intended to prevent congestion. If C is the capacity limit for an edge of a grid cell, we include C variables per edge for each connection. In other words, each connection can be routed through one of C tracks across a cell boundary as shown in Figure 3 (b).

Consider two connections i and j. Consider horizontal route tracks for each connections, $h_{i_{r,c}}$, and $h_{j_{r,c}}$ for some row r and column c. Let $i_{r,c_1}, i_{r,c_2}, \ldots i_{r,c_C}$ and $j_{r,c_1}, j_{r,c_2}, \ldots j_{r,c_C}$ be the C extra variables introduced in the SAT formulation for the horizontal track in question. Then clearly, for any $i_{r,c_k}, 1 \leq k \leq C$, $i_{r,c_k} \Rightarrow h_{i_{r,c}}$, and also $h_{i_{r,c}} \Rightarrow (i_{r,c_1} + \ldots + i_{r,c_C})$. Clauses of this form are added to the SAT instance. Another restriction is that a route cannot pass through two tracks in the same channel (edge of a grid cell), i.e., if for some $k, 1 \leq k \leq C$, if i_{r,c_k} is true, then for all $l, 1 \leq l \leq C, l \neq k, (i_{r,c_k} \Rightarrow -i_{r,c_l})$. These clauses are also added. Finally, two connections cannot be routed through the same track, i.e. for all $k, 1 \leq k \leq C$, $(i_{r,c_k} \Rightarrow -j_{r,c_k})$ for all $j \neq i$, where j represents another connection. By combining the aforementioned techniques, we are able to express routing instances as SAT problems.

We created ten routing configurations by randomly flooding a 3-by-3 routing grid with connections subject to edge capacity constraints of 3. Then we applied the SAT encoding above. Table 1 shows empirical results for the five most difficult instances (`grout`).

5. THE EFFECT OF SYMMETRY-BREAKING

Our computational experiments were performed on PCs with AMD Athlon processors @1.2GHz and 1Gb of RAM. All codes were compiled with `g++ 2.95.4 -O3` and ran on Debian Linux. In addition to the instances described in Section 4 (`chnl` and `fpga`) and (`grout`), Table 1 lists six standard pigeon-hole instances (`hole`), five families of artificially constructed randomized Urquhart benchmarks (`Urq`) [24] and seven recent benchmarks from the micro-processor verification domain [25]. CHAFF runtimes in Table 1 are averages of (up to) 20 independent starts because CHAFF uses randomization internally and results of different runs may vary significantly. All runs that did not complete in 1000 seconds were aborted and did not contribute to averages. The percent of time-outs is shown for each instance.

To detect symmetries in CNF formulae, we converted them into colored graphs (see Section 2). We then used the NAUTY program [17, 18]. At each run, the result was a list of permutation generators of the group of symmetries. Permutation generators are specified by cycles. For each SAT instance, Table 1 lists NAUTY runtime in seconds excluding I/O. the total number of symmetries and the number of permutation generators. Those symmetry detection implementations are deterministic and not affected by re-ordering of vertices in the input graph. For some benchmarks we built symmetry-breaking clauses only for ten cycles per symmetry. The first ten cycles typically capture most of the speed-up provided by "breaking" a given symmetry. After new clauses were added, the preprocessed CNF instance was solved with CHAFF. Table 1 lists average runtimes of 20 independent runs of CHAFF for each instance. Pre-processed CNFs never timed out in our experiments.

The last column in Table 1 shows relative speed-up ratios due to the

Table 1: CHAFF runtime on original SAT instances is compared to the combined runtime of symmetry detection and CHAFF on instances with symmetry-breaking clauses (the right-most column). The full name of benchmark `2dlx_ca_mc` is `2dlx_ca_mc_ex_bp_f` . The numbers of symmetry generators and max cycles used per generator are shown (10 or all). Pure search speed-up (that does not take symmetry detection into account) is also given. Results for opportunistic window-based symmetry-finding are also given and in most cases discover all or a large fraction of all symmetries. All benchmarks that we generated for these experiments are available at `http://www.eecs.umich.edu/~faloul/benchmarks.html`

Instance	Satis-fiable?	#variables and #clauses	Plain Chaff sec	Time -out %	Symmetries Finding sec	Number of	#generators cycles		Chaff sec	Speed-up ratios: total ; search only
hole07	UNS	56;204	0.37	0%	0.1	2.03e8	all	13	0.01	3.32; 36.50
hole08	UNS	72;297	1.27	0%	0.07	1.46e10	all	15	0.01	15.22; 94.15
hole09	UNS	90;415	3.79	0%	0.1	1.32e12	all	17	0.02	32.00; 204.97
hole10	UNS	110;561	22.44	0%	0.15	1.45e14	all	19	0.02	132; 1122
hole11	UNS	132;738	212.73	0%	0.18	1.91e16	all	21	0.03	1229.6; 7090.9
hole12	UNS	156;949	>1000	100%	0.24	2.98e18	all	23	0.04	— ; —
Urq3_5	UNS	46;470	232.44	10%	0.48	2.32e6	all	29	0.0	484.16; —
Urq4_5	UNS	74;694	250.01	25%	1.35	2.50e6	all	43	0.0	185.18; —
Urq5_5	UNS	121;1210	>1000	100%	13.15	>1e7	all	72	0.0	—; —
Urq6_5	UNS	180;1756	>1000	100%	62.93	>1e7	all	109	0.0	—; —
Urq7_5	UNS	240;2194	>1000	100%	176.62	>1e7	all	143	0.0	—; —
grout3.3-01	SAT	864;7592	19.01	0%	4.79	8.71e9	10	26	0.67	3.48; 28.37
grout3.3-03	SAT	960;9156	44.35	0%	8.94	6.97e10	10	29	0.40	4.75; 110.89
grout3.3-04	SAT	912;8356	19.36	0%	6.81	2.61e10	10	27	0.36	2.70; 53.79
grout3.3-08	SAT	912;8356	21.30	0%	7.14	3.48e10	10	28	0.67	2.73; 31.80
grout3.3-10	SAT	1056;10862	28.18	0%	10.65	3.48e10	10	28	0.85	2.45; 33.15
chnl10x11	UNS	220;1122	22.17	0%	0.45	4.20e28	all	39	0.11	39.91; 210.13
chnl10x12	UNS	240;1344	81.88	0%	0.61	6.04e30	all	41	0.12	111.63; 663.00
chnl10x13	UNS	300;2130	657.61	25%	1.28	4.50e37	all	47	0.17	454.78; 3961.4
chnl11x12	UNS	264;1476	207.37	0%	0.75	7.31e32	all	43	0.15	231.31; 1415.5
chnl11x13	UNS	286;1742	788.32	20%	1.08	1.24e35	all	45	0.16	633.45; 4792.2
chnl11x20	UNS	440;4220	>1000	100%	4.4	1.89e52	all	59	0.31	—; —
fpga10.08	SAT	120;448	7.56	0%	0.63	6.00e71	all	62	0.05	11.15; 157.56
fpga10.09	SAT	135;549	3.80	0%	0.88	6.33e77	all	68	0.03	4.16; 113.39
fpga12.11	SAT	198;968	694.00	50%	3.76	7.18e77	all	95	0.06	181.63; 11377.0
fpga12.12	SAT	216;1128	80.20	0%	5.31	7.44e77	all	104	0.13	14.74; 616.92
fpga12.08	SAT	144;560	246.70	10%	1.23	8.41e77	all	72	0.08	188.39; 3103.14
fpga12.09	SAT	162;684	885.00	80%	1.7	2.25e77	all	79	0.05	504.56; 16388.8
fpga13.09	SAT	176;759	550.00	85%	2.57	2.56e77	all	84	0.06	208.81; 8593.75
fpga13.10	SAT	195;905	>1000	100%	4.04	5.76e77	all	93	0.08	— ; —
fpga13.12	SAT	234;1242	>1000	100%	6.9	8.85e77	all	110	0.08	— ; —
2dlx_ca_mc*	UNS	3250;24640	6.54	0%	38.36	9.36e77	10	66	6.30	0.15; 1.04
2pipe.cnf	UNS	892; 6695	2.08	0%	10.74	2.26e45	10	38	1.56	0.17; 1.33
2pipe_1_ooo	UNS	834; 7026	2.55	0%	9.37	8	10	3	1.80	0.23; 1.41
2pipe_2_ooo	UNS	925; 8213	3.43	0%	11.14	32	10	5	2.82	0.25; 1.22
3pipe	UNS	2468;27533	36.44	0%	463.57	7.29e77	10	85	19.65	0.08; 1.85
4pipe	UNS	5237;80213	337.61	0%	>1000	—	—	—	—	—; —
5pipe	UNS	9471;195K	325.92	0%	>1000	—	—	—	—	—; —
WINDOW-BASED SYMMETRY FINDING (1000 variables per window)										
2dlx_ca_mc*	UNS	3250;24640	6.54	0%	3.17	2.34e77	10	64	5.42	0.76; 1.21
2pipe	UNS	892; 6695	2.08	0%	10.47	2.26e45	10	38	1.30	0.18; 1.63
2pipe_1_ooo	UNS	834; 7026	2.55	0%	9.02	8	10	3	1.80	0.24; 1.41
2pipe_2_ooo	UNS	925; 8213	3.43	0%	11.09	32	10	5	2.80	0.25; 1.23
3pipe	UNS	2468;27533	36.44	0%	3.63	1.42e77	10	78	36.20	0.91; 1.01
4pipe	UNS	5237;80213	337.61	0%	9.32	1.03e78	10	142	334.00	0.98 ; 1.01
5pipe	UNS	9471;195K	325.92	0%	29.42	3.64e78	10	227	290.50	1.02; 1.12

use of symmetry-breaking clauses. For a given CNF instance, the first number is the ratio of (i) CHAFF runtime on original instance, and (ii) the total runtime of symmetry detection and CHAFF on preprocessed instances. The second number is produced similarly, except that symmetry detection runtime is ignored. This is the maximal possible speed-up if symmetries are detected instantaneously or provided as domain-specific knowledge. We make several observations: **(1)** the proposed SAT instances are only a fraction of the size of recent micro-processor verification benchmarks [25], but are more difficult to solve; **(2)** some difficult SAT instances have astronomical numbers of symmetries; this includes the randomized `Urq` and `grout` benchmarks; **(3)** symmetry-breaking clauses often speed up the best available SAT solver CHAFF [6]; **(4)** symmetry-breaking clauses typically do not slow down CHAFF and often speed it up, even when few symmetries are present; **(5)** CHAFF runtime and symmetry detection runtime are not correlated; either step may be a bottleneck. **(6)** among the `chnl` instances, the hardest to solve was routing of 20 connections through 11 tracks. Adding extra unrouted connections consistently increased difficulty.

6. OPPORTUNISTIC SYMMETRY-FINDING

The use of symmetry-breaking clauses does not require finding *all* symmetries; symmetry detection can be performed *opportunistically*. An algorithm that does not guarantee to find all symmetries may finish sooner. Some symmetries may be found using domain-specific knowledge, and new clauses can be added during the creation of SAT instances. This may speed up the detection of other symmetries.

Window-based Symmetry Finding. We observed that a variable would sometimes be symmetric to another variable connected by a clause (one hop) or through a chain of two clauses (two hops). When this is not true for all symmetries of a CNF formula, many symmetries may be composable from permutation generators of that kind. We therefore focus on "local" symmetries that permute small subsets of variables and fix all other variables. We define the subsets by sliding a window of fixed size along a given linear ordering of the variables — either original variable ordering of the CNF formula or the connectivity-based MINCE ordering [1]. For a window, we consider the left and right cuts, as in Figure 4 (b). To find symmetries local to a given window, the standard

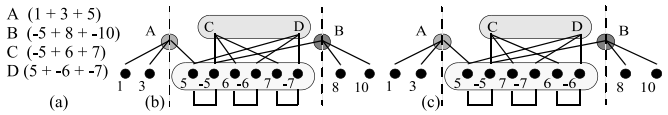

Figure 4: (a) window-based opportunistic symmetry detection for a CNF instance with ten variables and four clauses; (b) a colored graph for detecting only symmetries local to a window; (c) the "local" symmetry (6 7) (-6 -7).

construction of colored graph is applied to clauses and literals that are entirely inside the window. Each cut clause is represented by a vertex of unique color that is connected to literals inside the window. Since the size of this graph increases with cut size, a min-cut ordering improves runtime. We concatenate lists of permutation generators produced for different windows, consider the group generated by all those and use GAP [22] to produce an irredundant list of generators of this "global" group. Symmetry-breaking clauses are constructed from those generators. Producing symmetry-breaking clauses independently from each window and concatenating them may cause considerable redundancy. The trade-off between runtime, coverage and redundancy among windows depends on their overlap. Similarly, the window size affects the trade-off between runtime and coverage. We observe good empirical performance with windows of size 1000. Results in Table 1 show that our window-based technique found all or a significant portion of all symmetries for the micro-processor verification benchmarks [25] in a fraction of the runtime spent by complete symmetry-finding. If a randomized variable ordering is used, one could combine local permutation generators found for different orderings.

Improving SAT Formulations One way to reduce the runtime of symmetry finding is to learn how to detect (or predict) symmetries from domain-specific knowledge. Given the well-understood structure and symmetries of the `hole`, `chnl` and `fpga` benchmarks, we evaluated this approach on (randomized) `grout` benchmarks. We noticed that permuted variables in many cases correspond to neighboring tracks, e.g., if two connections are routed in parallel through several grid cells, there is considerable freedom (symmetry) in track assignment. To break this symmetry, we added clauses that preserve the relative order of tracks taken by every pair of connections routed through the same two edges of a grid cell. In other words, if one connection is routed through track 2 when entering the cell, and another connection is routed through track 3 when entering the cell, then the connections are allowed to leave the cell through tracks 2 and 3 resp., 1 and 2 resp. or 1 and 3 resp. Such constraints speed-up CHAFF: each `grout` instance is now solved in **0.50-0.80 seconds versus 19-45 seconds**. More dramatic speed-ups are achieved for `grout` instances built with larger routing grids.

7. CONCLUSIONS

We describe an automated flow that finds symmetries in CNF instances and uses them to speed up SAT search. This flow dramatically speeds up the solution of two well-known provably-difficult benchmark families — pigeon-hole problems and Urquhart benchmarks. We propose constructions of realistic satisfiable and unsatisfiable SAT instances, arising in routing applications, that are unusually difficult for their size. Unlike most existing SAT benchmarks, our benchmark families enable studies of the asymptotic performance of SAT solvers.

Since symmetry-finding is a bottleneck, we speed it up using opportunistic approaches. In one, we only look for symmetries that permute small groups of variables. Those groups are determined by sliding a fixed-sized window along a given variable ordering. The second approach attempts to improve the construction of SAT instances by detecting symmetries in domain-specific terms so that new clauses can be added during construction. We find astronomically many symmetries in randomized Urquhart and `grout` benchmarks, showing that randomization is compatible with symmetries. We explain symmetries in `grout` benchmarks and break them using domain-specific knowledge.

Our proposed flow does not require source code modifications and should work with all complete SAT solvers. In experiments described in Table 1 but performed with GRASP [23] instead of CHAFF [6], our flow demonstrated speed-ups 1.5-5 times even for the micro-processor verification benchmarks. The proposed flow may not give improvement on arbitrary SAT benchmarks — many difficult SAT instances do not have symmetries [8]. While many DIMACS benchmarks [10] have large numbers of symmetries, they are easy and can be solved faster than their symmetries can be detected.

Acknowledgements. This work is funded by an Agere Systems/SRC Research fellowship, a DAC fellowship and DARPA/MARCO GSRC.

8. REFERENCES

[1] F. Aloul, I. Markov and K. Sakallah, "Faster SAT and Smaller BDDs via Common Structure", *ICCAD 2001*, pp. 443-448.

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

[3] B. Benhamou and L. Sais, "Tractability through symmetries in propositional calculus". *Journal of Autom. Reasoning*, vol. 12, (no.1), Feb. 1994. pp. 89-102.

[4] L. Brisoux, E. Gregoire, L. Sais, "Improving backtrack search for SAT by means of redundancy", *Foundations of Intelligent Systems. 11th Intl. Symp., ISMIS'99*. Warsaw, Poland, June '99. Berlin, Germany: Springer 1999. p.301-9.

[5] C. A. Brown, L. Finkelstein, and P. W. Purdom. "Backtrack searching in the presence of symmetry". (T. Mora, editor), *Applied algebra, algebraic algorithms and error correcting codes, 6th intl. conf.*, pages 99– 110. Springer-Verlag, 1988.

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

[7] E.M. Clarke et al., (Edited by: Hu, A.J.; Vardi, M.Y.) "Symmetry Reductions in Model Checking", CAV'98, pp.159-71.

[8] S. A. Cook, D. G. Mitchell, "Finding Hard Instances of the Satisfiability Problem: A Survey", *DIMACS Ser. Discr. Math. & Theor. Comp. Sci.*, '97.

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

[10] DIMACS Boolean Satisfiability Challenge Benchmarks: `ftp://dimacs.rutgers.edu/pub/challenge/sat/benchmarks/cnf`

[11] C.A.J. van Eijk, E.T.A.F. Jacobs, B. Mesman and A.H.Timmer, Identification and Exploration of Symmetries in DSP Algorithms, *DATE '99*, March 1999, pp. 602-608.

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

[13] C. Norris Ip and D. L. Dill, "Better verification through symmetry" , *Formal Methods in System Design*, 9(1/2):41-75, 1996.

[14] V. Kravets and K. Sakallah, "Generalized Symmetries of Boolean Functions", *ICCAD 2000*, pp. 526-532.

[15] V. Kravets and K. Sakallah, "Constructive Library-Aware Synthesis Using Symmetries", *DATE 2001*, pp. 208-213.

[16] G.S. Manku, R. Hojati and R. Brayton, "Structural symmetry and model checking", *Proc. Intl. Conf. Comp.-Aided Verific. (CAV '98)*, pp. 159-171.

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

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

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

[20] L. H. Soicher, "GRAPE: A System For Computing With Graphs and Groups", *in "Groups and Computation"* (L. Finkelstein and W.M. Kantor, eds), *DIMACS Ser. in Discr. Math. & Theor. Comp. Sci.* 11, pp. 287-291. `www-groups.dcs.st-andrews.ac.uk/~gap/Share/grape.html`

[21] M. Prasad, P. Chong, K. Keutzer, "Why is ATPG easy?", *DAC '99*.

[22] E. L. Spitznagel, "Review of Mathematical Software, GAP", *Notices Amer. Math. Soc.*, 41 (7), (1994), pp. 780–782. `http://www-groups.dcs.st-andrews.ac.uk/~gap/gap.html`

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

[24] A. Urquhart, "Hard Examples for Resolution", *JACM*, vol. 34, 1987.

[25] M.N. Velev, and R.E. Bryant, "Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors", *DAC 2001*, pp. 226-231. `http://www.ece.cmu.edu/~mvelev/#BENCHMARKS`