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}@umich.edu

ABSTRACT

Research in algorithms for Boolean satisfiability and their efficient implementations [26, 8] has recently outpaced benchmarking efforts. Most of the classic DIMACS benchmarks from the early 1990s [12] can be solved in seconds on commodity PCs. More recent benchmarks take longer to solve primarily because of their large size, but are still solved in minutes [28]. However, small and difficult SAT instances must exist because Boolean satisfiability is NP-complete.

Our work articulates a number of SAT instances that are unusually difficult for their size, including satisfiable instances from global routing and detailed routing for FPGAs [22]. Using an efficient implementation to solve the graph automorphism problem [21, 23, 25], we show that in structured SAT instances difficulty is sometimes associated with large numbers of symmetries.

We propose a new, improved construction of symmetry-breaking clauses [11] and apply them to empirically demonstrate very significant speed-ups over current state of the art in Boolean satisfiability. Our techniques are formulated as pre-processing and can be applied to an arbitrary SAT solver without modifying its source code. We also show that considerations of symmetry may lead to more efficient reductions to SAT in the routing domain and potentially other applications.

1. INTRODUCTION

Boolean satisfiability (SAT) is a pivotal problem in Computer Science and has numerous applications in Design Automation that range from microprocessor verification [28] to FPGA layout [22]. A one-million-dollar prize is offered by the Clay Institute for Mathematical Sciences for a complete truly polynomial-time SAT solver or a proof that such an algorithm does not exist (the P-vs-NP problem). Hardly anyone expects that such an algorithm will ever be found. Nevertheless, industrial applications motivated intensive research in SAT algorithms that quickly solve instances arising in applications. The fundamental framework for state-of-the-art SAT algorithms was laid out in 1960s, but a number of recent improvements in algorithms and implementation techniques [26, 8] have lead to performance breakthroughs. A majority of the DIMACS benchmarks [12] published as a challenge in the early 1990s are now solved in seconds on commodity PCs. With the exception of several artificially constructed families of benchmarks, it looks like SAT can be solved in polynomial time "for practical purposes". Recently posted SAT benchmarks [28] take somewhat longer to solve (minutes), but that is primarily due to their enormous size (many of them require 50MB+ files). This only reinforces the apparent polynomial-time solvability of practical SAT instances.

It is well-known that the dominant back-track solvers, such as GRASP [26] and CHAFF [8] 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 easyness of structured instances were successful for certain applications [24], and generic ways to exploit certain types of structure were proposed [1].

Our work addresses both benchmarking aspects of SAT research and algorithmic aspects. Given the excellent performance of existing SAT solvers, there is little room for improvement on easy benchmarks, and we focus on difficult instances.[1] Since the works of Haken and Urquhart [27] on lower bounds for resolution and back-tracking algorithms for SAT in the 1980s, several instance families have been known to require exponential time for DP/DLL solvers. For example, a recently improved lower bound for the pigeon-hole problem is $\Omega(2^{n/20})$ [3] where n is the number of pigeons. The pigeon-hole problem can be quickly solved by induction, but the proof system behind back-track solvers is rather restrictive and does not allow polynomial-sized proofs for pigeon-hole instances. Another such family was constructed by Urquhart in terms of expander graphs and with considerable use of randomization [27]. Indeed, state-of-the-art SAT solvers, such as CHAFF, take a very long time to solve those instances (see Table 1), but the relevance of such pathological cases to Design Automation is questionable. In particular, lower bounds for SAT are typically proven for unsatisfiable instances, and it remains to be seen whether any satisfiable instances can be difficult for state-of-the-art solvers. In this paper, 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 very difficult by adding a small difficult instance to it and connecting the two by inconsequential clauses to defeat partitioning.[2]

SAT Symposium 2002 Cincinnati, Ohio

[1] In practice, the difficulty of domain-specific classes of SAT instances is often known, and the relevant SAT algorithms can be easily chosen. Alternatively, one can run several SAT solvers in parallel until the first one finishes. On a single-processor computer this may potentially bring exponential speed-ups for the cost of a constant-factor slow-down.

[2] Note that this argument easily defeats global statistical measures of instance complexity such as the clause-to-variable ratio.

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 breakthroughs 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 [19]), 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 3).

Besides complexity-theoretic connections between variants of Boolean satisfiability, symmetries and the hypergraph automorphism problem [2, 18], a number of works suggested that "breaking symmetries" in CNF formulae can speed up SAT solvers [4, 5, 6, 7, 11, 19]. By a symmetry of a CNF formula we mean a permutation of its variables that does not change the formula, i.e., maps clauses to clauses. Such a permutation can, in principle, affect arbitrarily many variables at once, e.g., as in the case of 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.[3] However, if such symmetries are detected by other techniques (e.g., [16]), our proposed methods can process them in the same way as 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 [11], (b) extending SAT solvers, particularly those based on back-tracking, to dynamically use symmetries during the search process [7]. 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. For example, the work in [11] suggests that symmetry-based techniques allow the pigeon-hole instances to be solved in polynomial time,[4] but 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. Moreover, it remains to be seen that useful CNF formulae have sufficiently many symmetries. It was proved by Pólya (1937), Erdös and Rényi (1963) that a random graph on n vertices has *no symmetries* with probability $1 - \binom{n}{2}2^{-n-2}(1+o(1))$ [14, p. 1461]. A similar claim can be extended to CNF formulae using constructions in Section 3, but structured instances that arise in applications may have richer symmetries.[5] On the other hand, if exponentially many symmetries exist, adding exponentially many symmetry-breaking clauses can be disastrous, as pointed out in [11]. Nevertheless, symmetry-based approaches have been successful in model checking [15, 9], verification [19], synthesis of logic circuits [17] and DSP algorithms [13].[6]

In this work, we propose a completely automated flow that:

- starts with a CNF formula in the DIMACS format,
- detects all of its symmetries (not just pairwise swaps),
- represents all symmetries implicitly and *always* with exponential compression,
- preprocesses the CNF formula by adding symmetry-breaking clauses that do not affect satisfiability, and
- applies a black-box SAT solver 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 [11]. Additionally, it directly applies to the compressed representation of all symmetries in the exact format produced by modern software for the group automorphism problem [20, 21, 23, 25]. Most importantly, our empirical results show significant improvements on CNF instances arising in Design Automation applications as well as highly randomized provably-difficult Urquhart benchmarks [27].

Two extensions are developed to reduce the runtime of symmetry detection. One targets opportunistic symmetry detection, where only some symmetries are found (the main automated flow in no way relies on having all symmetries). The other extension attempts to point out domain-specific symmetries to users and suggest improvements of domain-specific SAT formulations by adding domain-specific symmetry-breaking clauses. The goal is to create symmetry-less SAT instances that can be solved much faster and entirely avoid generic symmetry detection.

The remaining part of the paper is organized as follows. The necessary algebraic background is covered in Section 2, Techniques for detecting symmetries are described in Section 3 and a simple example is given. Symmetry-breaking clauses are introduced in Section 4. Section 5 discusses constructions of SAT benchmarks and our empirical results. Further extensions are described in Section 6, and Section 7 concludes our work.

2. ALGEBRAIC BACKGROUND

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 the composition with the do-nothing permutation never changes symmetries. 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 elements (marks). 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.

The computational group theory is approximately 25 years old, and made great strides in the last decade with the development of

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

[4] The empirical data in [11, Figure 3] does not appear consistent with this suggestion.

[5] To this end, [16] demonstrated large numbers of symmetries in Boolean functions from synthesis applications.

[6] Some researchers limited the notion of symmetry to swaps of variables or groups of variables to achieve efficiency.

the GAP package ("Groups, Algebra and Programming") [25]. A major efficiency in the 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.[7] 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, it is reasonable to expect that an intelligent algorithm for symmetry detection will return a small set of generators rather than list all symmetries.

3. FINDING AND USING SYMMETRIES

3.1 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. The Graph Automorphism problem must find all symmetries of a given graph, e.g., in terms of group generators.[8] It is known that all graphs except for an exponentially small family have *no symmetries* [14, 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 [14, p. 1511]. Generic algorithms [20, 19] are based on linear-time partition refinement passes; a simple version finishes in three passes for all but an exponentially small family of graphs [14, p. 1513].

The Graph Automorphism problem is often constrained by vertex labels — symmetries must map each vertex into a vertex with the same label. In practice, label constraints do not introduce any computational difficulties and can be formally reduced to plain graph automorphism. They can be thought of as integers and are often called colors (this has nothing to do with the graph coloring problem). Another extension is to consider general colored hypergraphs rather than colored graphs. Symmetries must map hyperedges to hyperedges (of the same cardinality because no two vertices can map to one).

The colored hypergraph automorphism problem easily reduces to the colored graph automorphism via the bipartite graph of the hypergraph (which represents each vertex and each hyperedge by a vertex, and connects them with edges according to the incidence relation of the hypergraph). Graph vertices in the hyper-edge part are painted with a new color, and other vertices preserve their colors.

Brendan McKay implemented a practical algorithm for Graph Automorphism [20] in a software package called NAUTY [21], 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 [25] by means of the GRAPE package [23]. This integration enables efficient group-theoretic operations on the results returned by NAUTY and facilitates some of our proposed algorithms. In 1998, the work in [19] claimed speed improvements over a pre-2.0 version of NAUTY in the context of hardware verification. However, their code is not generic (is built into an application-specific system) and is no longer supported.

[7] For example, the group of *all* permutations on k marks has $k!$ elements, but can be generated by only two generators: (12) and $(12..k)$.

[8] *Isomorphism testing* for a pair of graphs, i.e., testing the existence of a 1:1 mapping of one graph onto another reduces to a special case of the Graph Automorphism problem.

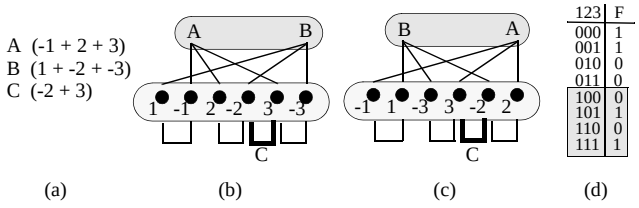


Figure 1: A CNF formula with three clauses A, B and C and three variables (a) is converted into a bipartite (2-colored) graph (b) for symmetry detection purposes. Note that the two-literal clause C is represented by one edge (bold) while larger clauses A and B are represented by a vertex and three edges each. Any symmetry must map the clause C must onto itself, 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.

3.2 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 and literal vertices are painted with color 2. To ensure Boolean consistency, vertices of opposite literals are mated by direct edges. Among several competing reductions, the reduction above worked best in our experiments; it has an added advantage of detecting symmetries of the type $a \to \bar{a}$ (phase shifts) and their compositions with permutational symmetries. An additional simplification, originally suggested in [11, footnote 6] in the context of the pigeon-hole instances, allows for the representation of arbitrary two-literal clauses by edges directly connecting their two literals rather than by two edges and a vertex. This simplification lead to very significant improvements in our experiments. In the final colored graph, $2Vars$ vertices represent the positive and negative literals and the remaining $Clauses - 2LitClauses$ vertices represent clauses. An example is given in Figure 1.

4. SYMMETRY BREAKING

Symmetries induce equivalence classes in the solution space (in group theory, they are called *orbits*). Given a satisfying truth assignment, all 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. Such a restriction can be achieved by selecting unique representatives from every equivalence class and adding clauses that are only satisfied on those representatives. A construction for such symmetry-breaking clauses was proposed in [11], based on a given ordering of variables. The main idea is (i) to order all elements from the solution space lexicographically, and (ii) to select the lexicographically smallest representative from each equivalence class as its representative.

The construction described in [11] is applied to every symmetry given 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 [11]. Phase shift symmetries were not addressed in that work.

As in [11], we compute compute symmetry-breaking clauses on a per-symmetry basis, but will achieve pruning by only processing symmetries from an irredundant set of generators, which can be returned by graph automorphism programs. [11] mentions that but by breaking generator symmetries only, one does not necessarily breaks all symmetries except for some cases. While no evaluation (empirical or theoretical) is available for the power of such partial symmetry-breaking, we expect it to achieve a reasonable coverage for only a small fraction of the cost entailed by breaking all symmetries. This is because an irredundant set of generators contains "maximally independent" symmetries (none of them can be expressed in terms of others).

Unlike that in [11], our construction is formulated in terms of cycles of a permutation, i.e., the format in which permutations are commonly represented. First consider the variable swap (ab). The construction in [11] entails one additional variable and the total of six symmetry-breaking clauses. Our construction below achieves the same effect with only one clause. First observe that if the cycle (ab) is a symmetry, whenever there is a satisfying assignment with $a = 1, b = 0$, there should be a symmetric (equivalent) satisfying assignment with $a = 0, b = 1$ and other variables unchanged. In order 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, in order to "break" a cycle of length three (abc), we add $(\bar{a} + b)(b + c)$, i.e., $(a \leq b)(b \leq c)$. In order 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. Longer cycles require more complex symmetry-breaking clauses, but one can always improve on the construction from [11]. Nevertheless, we observed that symmetry generators produced for CNF formulae typically have 2-cycles only, and only in rare cases have 3-cycles. A cycle of the form $(\bar{a}a)$ means that the value of variable a does not matter, and can be fixed arbitrarily. This can be expressed as one-literal symmetry-breaking clause. The construction in [11] does not address "phase-shift" symmetries and never results in one-literal clauses.

It can be seen that the above techniques handle all possible cycles of lengths two and three, and can be used with any one cycle of a symmetry. That significantly speeds up SAT solvers in many cases. However, symmetry-breaking clauses of the form $(\bar{a} + b)$ achieve no pruning on those areas of the solution space where the variables involved have identical values.[9] A key idea in that case, similar to that in [11] 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 additional variables, one per cycle, that indicate the equality of all variables in the cycle. A sample clause with new variables can look 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. An example is given in Figure 1.

The construction of symmetry-breaking clauses is dwarfed by the time required for symmetry detection. 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 the conditions of the form $((a = b)(c = d)...(u = v))$ may lead to improvements over pure pre-processing.

[9] In practice, we first look for cycles that can generate single-literal clauses. One such clause achieves maximal pruning possible for a given symmetry if all cycles have length ≤ 2.

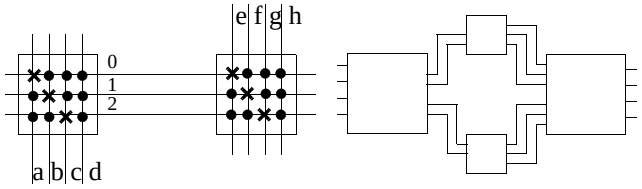


Figure 2: Construction of difficult SAT instances based on switch-boxes. The left picture shows two switch-boxes in the style of common FPGA architectures. In this example, one is trying to route four connections through three tracks (0,1,2), which is impossible by the pigeon-hole principle. On the right, similar *N*-by-*M* switch-boxes are used to construct difficult satisfiable instances.

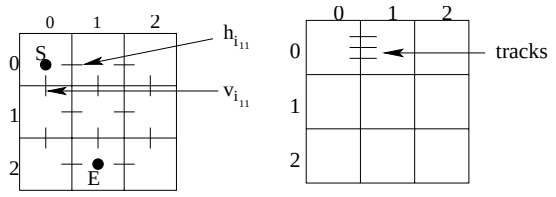
5. DIFFICULT SAT INSTANCES AND EMPIRICAL RESULTS

The pigeon-hole instances are provably difficult for back-track SAT solvers in general [3] and empirically difficult for leading-edge implementations CHAFF and GRASP as shown in Table 1. However, they are typically treated as artificial. Below, we derive equivalent instances from the domain of detailed routing for FPGAs and generalize them in several ways. We also give randomized constructions of difficult global routing instances.

5.1 Difficult FPGA Routing Instances

A recent comparative study of two Boolean formulations of FPGA detailed routing constraints [22] demonstrated that problem encoding can affect the difficulty of the resulting SAT instances. In this work, we use the better formulation from [22] and still produce difficult instances. Two such constructions are shown in Figure 2 in terms of FPGA switch-boxes (see [22] 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 instances. 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 results confirm that such unsatisfiable instances are extremely difficult for the leading-edge SAT solver CHAFF [8]. We point out that such instances can routinely appear as subproblems in larger FPGA routing problems and may not be easy to identify. Another important observation is that they possess a large number of symmetries (see Table 1).

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 [8] typically allow it to avoid difficult regions of search space and to quickly find satisfying solutions if they exists. 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 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) 3 X 3 grid with tracks marked (b) 3 x 3 grid with capacity 3

Figure 3: Construction of difficult SAT instances from the global routing domain.

a long time. Solvers without random restarts will fare no better as they 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 smaller switch-boxes are connected to the leftmost 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 connections between 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 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 observed that these satisfiable instances become more difficult with the increase of the difference between the throughput of the small switchboxes and the capacities of the channels that lead to them. This is consistent with our observations for the unsatisfiable `chnl` instances.

5.2 Difficult Global Routing Instances

We propose another construction of difficult *satisfiable* instances. These are randomized, entirely unrelated to pigeon-hole instances and come from the domain of global grid-based routing. The goal is to route a number of two-pin connections in a grid with edge capacity constraints. To ensure that the instance is satisfiable but difficult to solve, we propose construction by *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 two 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 $(h_{i_{1,1}} + v_{i_{1,1}})$. Clearly, these two tracks cannot be selected at the same time, therefore we add the mutual exclusion clause $(\bar{h}_{i_{1,1}} + \bar{v}_{i_{1,1}})$.

We now push the cells reachable from the possible tracks into 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: $(h_{i_{1,1}} \wedge v_{i_{1,2}}) \Rightarrow \bar{h}_{i_{1,2}}$. 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`).

5.3 The Effect of Symmetry-breaking Clauses

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 Sections 5.1 (`chnl` and `fpga`) and 5.2 (`grout`), Table 1 lists six standard pigeon-hole instances (`hole`), five families of artificially constructed randomized Urquhart benchmarks (`Urq`) [27] and seven recent benchmarks from the micro-processor verification domain [28].

CHAFF runtimes in Table 1 are averages of (up to) 20 independent starts. Such experimental protocol is required because CHAFF uses randomization internally and results of different runs often

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.

Instance	Satis-	#varables	Plain	Time	Symmetries					Speed-up ratios:
	fiable?	and	Chaff	-out	Finding	Number	#generators		Chaff	total ;
		#clauses	sec	%	sec	of	#cycles		sec	search only
hole06	UNS	42;133	0.03	0%	0.03	3.63e6	all	11	0.01	0.77; 4.07
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	1229.56; 7090.88
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	— ; —
fpga13.13	SAT	254;1433	531.00	80%	9.8	6.16e77	all	118	0.08	53.75; 6721.52
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

vary significantly. All runs that did not complete in 1000 seconds were aborted and considered failures. The percent of time-outs is shown for each instance. Failed runs are not reflected in the averages when at least one run was successful.

In order to detect symmetries in CNF formulae, we converted them into colored graphs as described in Section 3. We then used the NAUTY program [20, 21] which is integrated with GAP [25] — a system for computational group theory, — by means of the GRAPE package [23]. 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/GRAPE/GAP runtime in seconds excluding I/O. the total number of symmetries and the number of permutation generators. Those symmetry detection implementations are entirely deterministic and, moreover, are not affected by re-ordering of vertices in the input graph. For some benchmark families we used a limit of ten cycles when constructing symmetry-breaking clauses. In general, the first ten cycles typically capture most of the speed-up provided by "breaking" a given symmetry. After symmetry-breaking clauses were added to the original CNF instance, the resulting preprocessed instance was solved with CHAFF. Table 1 lists average runtimes of 20 independent runs of CHAFF for each instance. There were no time-outs for any of pre-processed CNFs.

The last column in Table 1 shows relative speed-up ratios due to the use of symmetry-breaking clauses. For a given CNF instance, the first number in that column 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 with the only difference that symmetry detection runtime is ignored. This represents the maximal possible speed-up if symmetry detection is performed instantaneously (e.g., provided as domain-specific knowledge). Several observations are in order

- the SAT instances proposed in this paper are only a fraction of the size of recent micro-processor verification benchmarks [28], yet tend to be more difficult to solve;
- in some cases difficult SAT instances have astronomical numbers of symmetries; especially remarkable is the abundance of symmetries in randomized `Urq` and `grout` benchmarks;
- in many cases symmetry-breaking clauses enormously speed up the best available SAT solver CHAFF [8];
- symmetry-breaking clauses do not slow down CHAFF and often speed it up, even when few symmetries are present;
- CHAFF runtime and symmetry detection runtime are not correlated, Either step may be a bottleneck.
- among the `chnl` instances, the hardest to solve was routing of 20 connections through 11 tracks. In general, adding extra unrouted connections to `chnl` instances made them consistently more difficult. That is counter-intuitive.

Given that symmetry-breaking clauses clearly speed-up SAT search, we seek to further speed up the detection of symmetries.

6. OPPORTUNISTIC SYMMETRY FINDING

Since the use of symmetry-breaking clauses does not necessitate finding *all* symmetries (even in terms of generators), symmetry detection can be performed *opportunistically*. A symmetry detection algorithm that provides no guarantee that all symmetries were found may need to perform less work and would finish sooner. Another direction for speed-up is to detect at least some symmetries in domain-specific terms and add relevant symmetry-breaking clauses during the creation of SAT instance. This way, fewer symmetries will need to be detected by generic means (graph automorphism algorithms tend to finish sooner if fewer symmetries are present).

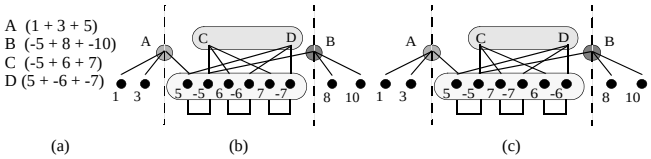


Figure 4: A window-based algorithm for opportunistic symmetry detection is illustrated on a CNF instance with ten variables and four clauses A, B, C and D (a). A colored graph for detecting only symmetries local to a window is shown in (b). The symmetry (6 7) (-6 -7) local to the window is shown in (c).

6.1 Window-based Symmetry Finding

When analyzing symmetries detected in many CNF instances, we observed that a variable would in many cases be mapped onto another variable connected by a clause (one hop) or through a chain of two clauses (two hops). When this is not the case for all symmetries of a CNF formula, many symmetries may be composable from permutation generators of that kind. We therefore focus on "localized" symmetries that are allowed to permute a small subset of variables and must fix all other variables. We determine such small subsets by sliding a window of fixed size along a given linear ordering of variables — either the original variable ordering of the CNF formula or the randomized connectivity-sensitive MINCE ordering [1]. For every window, we consider the left and the right cuts, as in Figure 4 (b). To find symmetries local to a given window, we apply our standard construction of a colored graph to clauses and literals that are entirely inside the window. Each cut clause is represented by a vertex of a unique color that is connected to those of its literals that are inside the window. Since the size of this graph increases as the cuts increase, a min-cut ordering would improve runtime. We concatenate lists of permutation generators produced for different windows, consider the group generated by all those and use GAP [25] to efficiently produce an irredundant list of generators for this "global" group. Symmetry-breaking clauses are constructed from those generators. We observe that producing symmetry-breaking clauses independently from each window and concatenating them could be a bad idea because of potentially considerable redundancy. The trade-off between runtime, coverage and the amount of redundancy between windows depends on the overlap between windows. Similarly, the window size affects the trade-off between runtime and coverage. We observe good empirical performance with windows of size of 1000. Results in Table 1 show that, when applied to micro-processor verification benchmarks [28], our window-based opportunistic symmetry finding method found all or a significant portion of all symmetries in a fraction of runtime spent by complete symmetry finding. If a randomized variable ordering is used, one could combine local permutation generators found for different orderings. While most of the generators are likely to be redundant, this may improve coverage.

6.2 Improving SAT formulations

One way to reduce the runtime of symmetry detection (possibly to zero) is to analyze symmetries discovered for several similar benchmarks and learn how to detect (or predict) symmetries in domain-specific terms. Given the well-understood structure and symmetries of the `hole`, `chnl` and `fpga` benchmarks, we evaluated this approach on the randomized `grout` benchmarks. Our analysis has shown that permuted variables in many cases correspond to neighboring tracks. For example, 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 domain-specific 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. As evidenced by further symmetry detection experiments, these constraints break *all* symmetries and considerably speed-up CHAFF: each `grout` instance is now solved by CHAFF in **0.50-0.80 seconds versus 19-45 seconds originally**. Even more dramatic speed-ups are achieved for `grout` instances built using larger routing grids. Additionally, it often takes little time to establish the absence of non-trivial symmetries.

While we have not studied source files for SAT instances from [28], we hypothesize that it may be possible to add domain-specific symmetry-breaking clauses to them and somewhat speed-up CHAFF.

7. CONCLUSIONS

In this work we proposed, for the first time, a completely automated flow that is able to find symmetries in CNF instances and use them to speed up SAT search. This flow dramatically improves overall CPU time required to solve two well-known provably difficult SAT benchmark families — pigeon-hole problems and Urquhart benchmarks. Additionally, we propose constructions of realistic satisfiable and unsatisfiable SAT instances derived from applications in detailed routing of FPGAs [22] as well as satisfiable randomized instances derived from global routing. Those instances are unusually difficult for their size, e.g., when compared to recent microprocessor verification benchmarks [28]. Unlike the majority of existing SAT benchmarks, our benchmark families enable studies of the asymptotic performance (scalability) of SAT solvers. All of benchmarks used in this work will be posted on the Web.

Since symmetry finding is often a bottleneck in our flow, we propose two opportunistic approaches that speed up symmetry finding. 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 domain-specific construction of SAT instances by detecting symmetries in domain-specific terms so that symmetry-breaking clauses can be added during the SAT instance construction. Improved SAT encodings lead to reduced solver times. We also point out that design symmetries may be available in the process of IP reuse, if the reused IP was adequately characterized.

Our empirical results dispel the common belief that even a slightest randomization destroys symmetries. Astronomical numbers of symmetries were found in Urquhart benchmarks that are generated by a fundamentally randomized procedure, and similarly in the `grout` benchmarks. In the latter case, we were able to explain the symmetries and break them in domain-specific terms.

Our proposed flow can be expected to work at least as well with other complete SAT solvers and does not require source code modifications. For any solver slower than CHAFF, one can expect our flow to provide *more significant speed-ups*. For example, when we performed experiments described in Table 1 with GRASP [26] instead of CHAFF [8], our flow demonstrated speed-up for *all* microprocessor verification benchmarks we considered. The speed-up ranged from 1.5 times to 5 times and higher.

One should not expect the proposed flow to give improvement on arbitrary SAT benchmarks. Many DIMACS benchmarks [12] have large numbers of symmetries, but can be solved so quickly that the symmetry detection overhead is not justified. On the other hand, many difficult SAT instances do not have symmetries [10].

Acknowledgements

This work is funded by the DARPA/MARCO Gigascale Silicon Research Center, an Agere Systems/SRC Research fellowship and a DAC fellowship.

8. REFERENCES

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

[2] L. Babai, R. Beals and P. Takácsi-Nagy, "Symmetry and Complexity", *ACM Symp. Theory of Comp. (STOC) '92*, pp. 438-449.

[3] P. Beame and R. Karp, "The efficiency of Resolution and Davis-Putnam Procedure", submitted for publication. `http://www.cs.washington.edu/homes/beame/papers/resj.ps`

[4] B. Benhamou, "Theoretical Study of Dominance in Constraint Solving Satisfaction Problems", *Proc. 6th Intl Conf. AI: Methodology, Systems, Applications (AIMSA '94)* Sofia, Bulgaria, 21-24 Sept. 1994, pp. 91-99.

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

[6] L. Brisoux, E. Gregoire, L. Sais, "Improving backtrack search for SAT by means of redundancy", *Foundations of Intelligent Systems. 11th International Symposium, ISMIS'99*. Warsaw, Poland, 8-11 June '99. Berlin, Germany: Springer 1999. p.301-9. xii+676 pp.

[7] 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.

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

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

[10] S. A. Cook, D. G. Mitchell, "Finding Hard Instances of the Satisfiability Problem: A Survey", *DIMACS Ser. Discr. Math. and Theor. Comp. Sci.*, 1997.

[11] J. Crawford, M. Ginsberg, E. Luks and A. Roy, "Symmetry-breaking predicates for search problems", *Fifth Intl Conference on Principles of Knowledge Representation and Reasoning*, KR'96, Cambridge, MA, USA, 5-8 Nov. 1996 page/s: 148-59. Morgan Kaufman, 1996

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

[13] 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 1999. Munich, Germany, 9-12 March 1999, pp. 602-608.

[14] 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).

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

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

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

[18] E. Luks, "Hypergraph isomorphism and structural equivalence of boolean functions", ACM Symp. on Theory of Computing (STOC), 1999, 652-658.

[19] G.S. Manku, R. Hojati and R. Brayton, (Edited by: Hu, A.J.; Vardi, M.Y.) Computer Aided Verification. 10th International Conference, CAV'98. Structural symmetry and model checking. p.159-71. 1998

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

[21] Brendan 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/`

[22] G. Nam, F. Aloul, K. Sakallah, and R. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints", *ISPD*, 2001.

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

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

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

[26] 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.

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

[28] 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`