T-79.194 Seminar on Theoretical Computer Science / Spring 2003: Exploiting Symmetries in SAT

Matti Järvisalo
Helsinki University of Technology
matti.jarvisalo@hut.fi

April 4, 2003

Abstract

We discuss a preprocessing-based approach to finding and exploiting symmetries in the structure of SAT instances in order to solve SAT instances more efficiently. The approach consists of mapping the problem of finding symmetries of a SAT instance to the graph automorphism problem, which is solved to determine the symmetries. The found symmetries are used in constructing additional constraints that are added to the original SAT instance to guide the search process.

1 Introduction

The propositional satisfiability problem (SAT) is a fundamental problem in computer science. SAT is one of the simplest "canonical" examples of a search problem, as all known approaches to determining satisfiability are clearly worst-case exponential; compare the best known approach [1] with $\approx 1.3^n$ worst-case to a complete search where all 2^n possible truth assignments are considered. The k-SAT problem was the first problem shown to be NP-complete[6] for $k \geq 3$ [2], which adds to the importance of SAT. The dominant complete SAT solvers today are mostly based on the DP/DLL procedure [13, 14] presented in the early 1960s. In recent years, research in the area has made significant progress due to algorithmic ideas and refinements and new implementational techniques[1] (see e.g. [10, 11, 12]), as the rise in computational power available has made it possible to solve industrial-size problems. Thus, interest in SAT is not only theoretical, as it has applications in numerous search-related industrial problem areas such as microprocessor verification [17], FPGA layout design [18], planning [15], etc. Industrial SAT instances are typically rather structured; they

[1] For a survey on approaches to satisfiability checking, see [16].

are typically derived from areas where there exists inherent structure, as e.g. circuits are based on logical flow.

In this paper we discuss the means of using inner structure of SAT instances to make the solving procedure more efficient. In particular, reduction from the problem of finding symmetries of SAT instances to the graph automorphism problem [5] is discussed. The graph automorphism problem is solved in order to find symmetries in the structure of the SAT instance. SAT symmetries are interesting, as they induce equivalence classes in the truth assignment space; given a representative of a particular equivalence class, if the representative is (not) a model for the instance, all truth assignments of the same class are also (not) models. Therefore, the solver should ideally reason only about one representative of each equivalence class. The found symmetries are used in constructing symmetry-breaking clauses (the term *symmetry-breaking predicates* is also used) that are added to the original instance in order to help the SAT solver to prune the search space more efficiently. This is done by giving an ordering on the truth assignments of the instance, and constructing the symmetry-breaking clauses so that they only allow the smallest representative of each equivalence class of truth assignments to satisfy the instance. As argued in [3], this makes sense as the graph automorphism problem is believed not to be NP-complete, and efficient (linear average-case) implementations for solving the problem exist [8, 9] for all but an exponentially small family of graphs.

The flow of solving SAT by first applying symmetry-breaking to the instance, as presented in Figure 1, goes as follows.

1. SAT symmetries are found by mapping the problem in a *graph automorphism problem* (see Section 3)

2. The automorphism problem is solved using a specialized program (see Section 4)

3. Found symmetries are used to construct *symmetry-breaking clauses* (see Section 3)

4. These clauses are added to the original SAT instance

5. The modified instance is solved using a SAT solver (see Section 4)

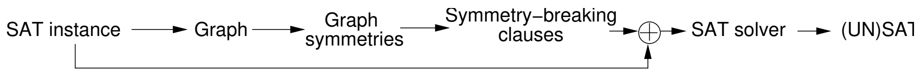

Figure 1: The flow of solving SAT by first applying symmetry-breaking to the instance.

Our discussion derives mostly from the work of Aloul et al. on symmetry-breaking [3]; we omit most of the explicit references to [3]. The idea of applying symmetry-breaking to search problems was first formulated by Crawford et al. [7].

2 Preliminaries

Let L be a set of propositional (Boolean) variables. A *literal* is a variable $v \in L$ or its negation, $\overline{v}$. A *clause* of length n is a disjunction $\bigvee_{i=1}^{n} l_i$, where l_i are literals. A *truth assignment* for L is a function $A : L \rightarrow \{\texttt{true}, \texttt{false}\}$. A is extended to literals so that $A(\overline{l}) = \overline{A(l)}$ holds $\forall l \in L$. A is called a *model* for a set of clauses $T = \{C_i\}_{1 \leq i \leq m}$, if for all C_i, there is a literal l in C_i for which $A(l) = \texttt{true}$. We denote the models of a set of clauses T by $\mathcal{M}(T)$. With these definitions, we define the propositional satisfiability problem (SAT) as follows.

*Definition 2.1 (*SAT*). Given a set of clauses T, does $\mathcal{M}(T) \neq \emptyset$ hold?*

Term SAT *instance* is used for a set of clauses.

We now define basic group theoretical concepts[2] needed in our discussion.

Definition 2.2 (Group). An (algebraic) group $G = (S, \circ)$ is a set S with a binary operation $\circ$ which together satisfy the following properties:

1. *(Closure) If $a,\ b \in G$, then $a \circ b \in G$.*
2. *(Associativity) $\forall a, b, c \in G$, $(a \circ b) \circ c = a \circ (b \circ c)$.*
3. *(Identity) $\exists e \in G$ (unit element) such that $\forall a \in G$ $a \circ e = e \circ a = a$.*
4. *(Inverse) $\forall a \in G$ there exists a unique $a^{-1} \in G$ (the inverse of a) such that $a \circ a^{-1} = a^{-1} \circ a = e$.*

Definition 2.3 (Group generators). A set of generators $(g_1, \ldots, g_n)$ is a set of group elements such that (possibly repeated) application of the generators on themselves and each other is capable of producing all the elements in the group.

Definition 2.4 (Redundant and irredundant set of generators). A set of generators is said to be redundant if any of its elements can be expressed in terms of other elements in the set. A set of generators is said to be irredundant if it is not redundant.

Definition 2.5 (Homomorphism). A mapping $f : G_1 \rightarrow G_2$ between to groups $G_1 = (S_1, \circ)$ and $G_2 = (S_2, \bullet)$ is a homomorphism iff for any $a, b \in G_1$ we have $f(a \circ b) = f(a) \bullet f(b)$.

Definition 2.6 (Group isomorphism and automorphism). An isomorphism is a homomorphism for which an homomorphic inverse mapping exists. Two groups G_1, G_2 are said to be isomorphic if there exists an isomorphism between G_1 and G_2. An isomorphic mapping $f : G \rightarrow G$, acting on a group G, is called an automorphism of G.

Definition 2.7 (Permutation). A permutation on a finite set S is a bijective (one-to-one) mapping $\pi : S \rightarrow S$.

[2] For a more complete introduction to abstract algebra, see e.g. [4].

The group of all permutations of a set S is denoted by $\mathrm{Sym}(S)$.

Permutations can be compactly presented with *cycles.* As an example, consider a permutation π on the set $\{1,2,3\}$, with $\pi(1)=3$, $\pi(2)=2$, and $\pi(3)=1$. In cycles, this is presented as $(13)(2)$. Usually cycles of length one are left out, so π is actually denoted by (13).

The following two theorems are presented without proof from [3].

Theorem 2.8 *Any group isomorphism maps sets of generators to sets of generators, and irredundant sets of generators to irredundant sets of generators.*

Theorem 2.9 *For any group G with $N > 1$ elements, any irredundant set of generators of G contains at most $log_2 N$ elements.*

As an example, $\mathrm{Sym}(\{1,2,3\}) = \{e, (12), (13), (23), (123), (321)\}$ can be generated using generators (12) and (23) as follows: $(12)(12) = e$, $(12)(23) = (123)$, $(23)(12) = (321)$, $(12)(321) = (13)$.

Next, we define some basic graph theoretical concepts[3] needed in our discussion.

Definition 2.10 (Graph). A graph $G = (V, E)$ consists of a set of vertices V and a set of edges $E \subseteq V \times V$.

An extension to the definition of a graph is the notion of a *colored graph*, in which there is a label, or a color, associated with each vertex.

Definition 2.11 (Graph isomorphism and automorphism/symmetry). Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, an isomorphism is a bijective mapping $f : V_1 \to V_2$ for which it holds that $(v_1, v_2) \in E_1$ iff $(f(v_1), f(v_2)) \in E_2$. Given a graph $G = (V, E)$, a symmetry (an automorphism) is a permutation π for which it holds that $(v_1, v_2) \in E$ iff $(\pi(v_1), \pi(v_2)) \in E$.

For a colored graph $G = (V, E)$, for a permutation π to be a symmetry it must in addition hold that $C(a) = C(\pi(a))\ \forall a \in V$, where $C(x)$ is the color of vertex x (i.e. π maps vertices to vertices of the same color).[4]

The set of symmetries of an object G form a group, referred to as the symmetry group of the object, which is denoted by $\mathrm{Aut}(G)$.

These concepts give us means of defining the graph automorphism problem (GraphAutomorphism).

Definition 2.12 (GraphAutomorphism). Given a graph G, find $Aut(G)$.

[3] For a more complete introduction to graph theory, see e.g. [5].

[4] The definition of an automorphism of a colored graph seems to vary; the definitions used here is sometimes referred to as an *isometric graph.* This definition is stricter than the following definition of an automorphism: For a colored graph $G = (V, E)$, for a permutation π to be a symmetry it must hold that $C(a) = C(b)$ iff $C(\pi(a)) = C(\pi(b))$.

Still, we need to define symmetries of a set of Boolean variables L. For $l \in L$ and $\pi \in \mathrm{Sym}(L)$, π is extended to literals so that $\pi(\bar{l}) = \overline{\pi(l)}$ holds $\forall l \in L$. For a clause $C = \bigvee_{i=1}^{n} l_i$, $\pi(C) = \bigvee_{i=1}^{n} \pi(l_i)$. For a set of clauses $T = \{C_i\}_{1 \le i \le m}$, $\pi(T) = \{\pi(C_i)\}_{1 \le i \le m}$. If $\pi(T) = T$, then π is a symmetry, or automorphism, of L, the set of variables in T.

We wish to obtain an irredundant set of generators of $\mathrm{Aut}(T)$, the group of symmetries in T, (for which it holds $\mathrm{Aut}(T) \subseteq \mathrm{Sym}(T)$) for a SAT instance T. The group $\mathrm{Sym}(T)$ consists of all permutations of the set of variables V in T. The size of $\mathrm{Sym}(T)$ is $|V|!$. Fortunately, Theorem 2.9 guarantees the existence of a compact presentation in the form of the irredundant generators.

3 Finding and Breaking SAT Symmetries

3.1 Types of Symmetries

Two types of SAT symmetries are considered: permutations of variables that map clauses to clauses, and so called *phase-shifts*, or *autosymmetries*, of the form $(a\bar{a})$, and compositions of these two.

Permutations of variables that map clauses to clauses. As an example, consider the set of clauses $T = \{a \vee b \vee c, d \vee e \vee f\}$. One permutation in the symmetry group of T is $(ad)(be)(cf)$.

Phase-shifts. Phase-shifts are permutations of the form $(a\bar{a})$, i.e., they interchange the literals a and $\bar{a}$ in a SAT instance. If these literals are interchangeable, they cannot affect the satisfiability of the instance. Thus such variables a can be assigned a truth value arbitrarily.

Both types of symmetries considered are so called *syntactic symmetries*, which preserve the presentation of a SAT instance, in contrast to so called *semantic symmetries* of Boolean functions, which could be though of as permutations that preserve the value of the function for all inputs. Syntactic symmetries are thus always semantic symmetries, but not vice versa; as an counterexample, consider $\{a, \bar{a}, a \vee b\}$ (which always evaluates to `false`) and a semantic symmetry (ab).

3.2 Finding Symmetries: Reduction from SAT symmetries to Graph Automorphism

There are many symmetry-preserving mappings from a SAT instance I with V variables and C clauses to a GraphAutomorphism instance. The mapping presented and used in [3] is now given. The result will be a bi-colored graph with up to $2 \cdot V + C$ vertices. Clauses of length one in the SAT instance are not considered; we assume that such clauses are removed from the instance before the mapping to a graph by repeated use of unit resolution.

The mapping derives from a hypergraph-presentation[5] of the SAT instance,

[5]In a hypergraph, edges can have more than two end-point vertices; such edges are called *hyperedges*.

where each literal is presented as a vertex, and each clause of length ≥ 3 as an hyperedge connecting the literals of that particular clause. Clauses of length two are presented with a (normal) edge between the two literal in a particular clause. Furthermore, there is an edge between each literal and its negation.

As an example, consider the SAT instance $T = \{A, B, C\}$, where the clauses are $A = \overline{a} \vee b \vee c$, $B = a \vee \overline{b} \vee \overline{c}$, and $C = \overline{b} \vee c$. T presented as a hypergraph is shown in Figure 2.

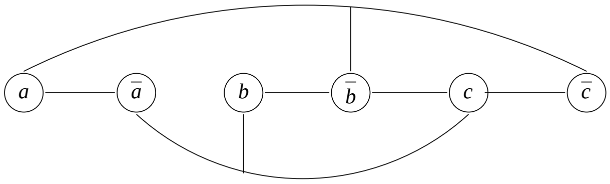

Figure 2: The hypergraph constructed from the SAT instance $\{\overline{a} \vee b \vee c, a \vee \overline{b} \vee \overline{c}, \overline{b} \vee c\}$.

The hypergraph can be transformed into a bi-colored bipartite graph by coloring the vertices of the hypergraph in color 1 (◯), and adding for each hyperedge a new vertex of color 2 (□). The added vertices are each connected with edges to the vertices that are end-points of the corresponding hyperedge. The hypergraph of Figure 2, transformed into a colored graph, is shown in Figure 3. The colored graph has $2 \cdot V + C - C_2$ vertices, where C_2 is the number of clauses of length two in the corresponding SAT instance.

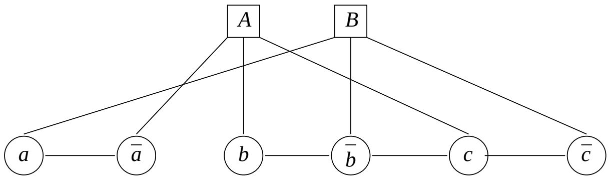

Figure 3: The bi-colored bipartite graph corresponding to the hypergraph of Figure 2.

Such a colored graph has the following property.

Theorem 3.1 *The set of symmetries of the bi-colored graph correspond one-to-one to the set of symmetries of the* SAT *instance I, assuming that I does not contain circular chains of implications*[6] *[3].*

[6]A *circular chain of implications* over the variables $v_1, v_2, \ldots, v_n$ is a set of n clauses of length two of the form $\{l_i \vee l_{i+1}\}_{i=1}^{n-1} \cup \{l_n \vee l_1\}$, where each $l_i = v_i$ or $l_i = \overline{v_i}$. Notice, that for such a set there exists only two models.

The presence of circular chains of implications can be the cause of so called *spurious symmetries* that do not correspond to any SAT symmetries. This problem must be dealt with, either after solving the graph automorphism problem by reasoning which symmetries are spurious, or before finding the symmetries by identifying the circular chains in original SAT instance and preprocessing the instance accordingly.

3.3 Breaking Symmetries

In symmetry-breaking, we add clauses to the original SAT instance that act as additional constraints on the models of the instance. As mentioned in Section 1, symmetries induce equivalence classes in the truth assignment space; given a representative of a particular equivalence class, if the representative is (not) a model of the instance, all truth assignments of the same class are also (not) models. Therefore, it suffices for the solver to reason about one representative from each equivalence class. We now discuss how this could be achieved.

We assume that by solving the graph automorphism we get an irredundant set of generators of the symmetry group of the graph. As Theorem 2.8 states, the obtained generator set is also an irredundant set of generators of the symmetry group of the original SAT instance. While the generators give us means of applying all the symmetries of the instance, applying them in practice can cause unwanted blowup in the instance size, as we have to add clauses to the original instance in order to break the symmetries that the generators represent. Thus in [3], the authors perform only *partial* symmetry-breaking, by breaking generator symmetries only. The argument is that the set of irredundant generators is the most valuable, as none of these generators can be expressed in terms of the others, all having thus an individual effect in pruning the search space.

To construct the symmetry-breaking clauses in a consistent way, we must have a total ordering $\leq$ on the literals. In the following we use $\overline{a} \leq a \leq \overline{b} \leq b \leq \overline{c} \leq c \leq \ldots$ This literal ordering directly gives an ordering on the truth assignments over the variables. The smallest representative from each equivalent class of truth assignments is chosen, and (ideally) all other assignments belonging to the same class are made unsatisfiable with the symmetry-breaking clauses.

Using $\leq$, we sort cycles and generators as follows. Within a particular cycle, the literals in the cycle are ordered using $\leq$. In generators consisting of multiple cycles, the cycles are ordered by the literals in the first position of the cycles using $\leq$. The latter orders the cycles within a generator consistently, as there can be only a single cycle of the form $(a \ldots)$ for each literal a within a generator.

Cycles of length $n \geq 4$, $(a_1 a_2 a_3 a_4 \ldots a_n)$, do not generally have an equivalent cycle $(a_i a_{i+1} \ldots a_n a_1 \ldots a_{i-1})$ for which $a_i \leq a_{i+1} \leq \ldots \leq a_n \leq a_1 \ldots \leq a_{i-1}$ or $a_i \geq a_{i+1} \geq \ldots \geq a_n \geq a_1 \ldots \geq a_{i-1}$ holds, while cycles of length ≤ 3 always have such an ordering; with the ordering $1 \leq 2 \leq 3$, both cycles of length 2, (12) and (21) are ordered, while cycles of length three of the form (132) and (213) are equivalent to (321), and (231) and (312) are equivalent to (123). This is the reason why only cycles of length two and three are considered.

We first consider using generators with single cycles and then generators with multiple cycles in constructing symmetry-breaking clauses. We assume that the cycles and the generators are sorted as mentioned above. Without any symmetries, the 2^n truth assignments are each thought to form their own equivalence class. Now we construct bigger equivalence classes of truth assignments by comparing the smallest representatives of the classes. The generators give us the information on which classes can be combined as follows.

Single cycles of length two. Consider a cycle of the form (ab).

This implies that the truth values assigned to the variables corresponding to the literals a and b are interchangeable. Thus assigning $a \leftarrow \texttt{true}$, $b \leftarrow \texttt{false}$ is equivalent to assigning $a \leftarrow \texttt{false}$, $b \leftarrow \texttt{true}$. Assuming we have $(a \leq b)$, to break this symmetry we add the clause $\overline{a} \vee b$ to the instance, which disallows assigning $a \leftarrow \texttt{true}$, $b \leftarrow \texttt{false}$. This prunes the search space by $\frac{1}{4}$. By doing so we combine such two equivalence classes into one where in the representative truth assignments the variables a and b are assigned opposite truth values and the other variables the same values. The smallest representative of such class will have $a \leftarrow \texttt{false}$, $b \leftarrow \texttt{true}$.

If $b = \overline{a}$ (i.e. a phase-shift $(a\overline{a})$), the variable a can be assigned an arbitrary truth value without affecting the satisfiability of the SAT instance. We add $\overline{a}$ as a unit clause to the instance, disallowing assigning $a \leftarrow \texttt{true}$. This prunes the search space by $\frac{1}{2}$. By doing so we combine such equivalence classes into one where in the representative truth assignments the variable a is assigned opposite truth values and the other variables the same values. The smallest representative of such class will have the assignment $a \leftarrow \texttt{false}$.

Single cycles of length three. Consider a cycles of the form (abc), where $a \leq b \leq c$. As all cycles can be split into a composition of cycles of lengths two, we add the clause set $\{\overline{a} \vee b, \overline{b} \vee c\}$ to the instance, corresponding to the generator $(ab)(bc)$. This prunes the search space by $\frac{1}{4}$. By doing so we combine such two equivalence classes into one where in the representative truth assignments (i) the variables a and b are assigned the same and c the opposite truth, and (ii) the variables b and c are assigned the same and a the opposite truth, and where in both cases the other variables are assigned the same values. The smallest representative of the class induced by (i) will have $a \leftarrow \texttt{false}$, $b \leftarrow \texttt{false}$, $c \leftarrow \texttt{true}$, and (ii) $a \leftarrow \texttt{false}$, $b \leftarrow \texttt{true}$, $c \leftarrow \texttt{true}$.

While symmetry-breaking clauses derived from cycles of the form $(a\overline{a})$ prune the search space in half, the truth assignment on such variables a do not affect the satisfiability as they can be arbitrarily chosen. Moreover, for other cycles of the form (ab), no pruning effect is achieved with the above construction in those regions of the search space were the variables a and b are assigned identical values. With symmetries with multiple cycles, this is dealt with by processing more than the first cycle as follows.

Multiple cycles. Consider a generator of the form $(ab)(cd)(ef)\ldots$, where $a \leq c$, $c \leq e, \ldots$. From such a generator, we construct multiple symmetry-breaking clauses as follows. First of all, we get $\overline{a} \vee b$ (a implies b), as discussed above. To add constraints to the part of the search space were a and b are assign identical truth values ($a \Leftrightarrow b$ holds), we do the following. For $(ab)(cd)$, we take a new variable $x_{a \equiv b}$ that is $\texttt{true}$ exactly when $a \Leftrightarrow b$ holds; $x_{a \equiv b} \Leftrightarrow (a \Leftrightarrow b)$, equivalent to $(a \vee b \vee x_{a\equiv b}) \wedge (a \vee \overline{b} \vee \overline{x}_{a\equiv b}) \wedge (\overline{a} \vee b \vee \overline{x}_{a\equiv b}) \wedge (\overline{a} \vee \overline{b} \vee x_{a\equiv b})$, or in conjunctive normal form, written as a set of clauses, $C = \{a \vee b \vee x_{a\equiv b}, a \vee \overline{b} \vee \overline{x}_{a\equiv b}, \overline{a} \vee b \vee \overline{x}_{a\equiv b}, \overline{a} \vee \overline{b} \vee x_{a\equiv b}\}$. We then require that either $a \Leftrightarrow b$ does not hold or $\overline{c} \vee d$ holds, the latter constraint coming from the cycle (cd). Thus we get the symmetry-breaking clause $\overline{x}_{a\equiv b} \vee \overline{c} \vee d$ in addition to the clause set C. Continuing with the processing of the generator, $(ab)(cd)(ef)$ is considered, from which, using the same idea, we get an additional set of symmetry-breaking clauses $\{\overline{x}_{a\equiv b} \vee \overline{x}_{c\equiv d} \vee \overline{e} \vee f\} \cup \{c \vee d \vee x_{c\equiv d}, c \vee \overline{d} \vee \overline{x}_{c\equiv d}, \overline{c} \vee d \vee \overline{x}_{c\equiv d}, \overline{c} \vee \overline{d} \vee x_{c\equiv d}\}$, and so on. From a generator with n cycles of length two, we get $5n$ symmetry-breaking clauses using this construction.

The construction of symmetry-breaking clauses above has the following property [3].

Theorem 3.2 *Take an arbitrary permutation consisting of cycles of length two only, which itself is sorted and in which each cycle is sorted using the ordering on the literals. Applying the symmetry-breaking clauses constructed from such a permutation to the original* SAT *instance I, for each equivalence class of models of I it holds that only the smallest representative of that class is a model of the modified instance.*

Furthermore, for a permutation having cycles only of length two and of odd lengths, applying the symmetry-breaking clauses constructed from only the cycles of lengths two in the permutation it holds that for each equivalence class of models of the original instance, smallest representative of that class is a model of the modified instance (but not necessarily the only model from the class) [3].

4 Practical Issues

4.1 From SAT symmetries to GraphAutomorphism

As mentioned in the previous section, there are numerous ways of mapping SAT into GraphAutomorphism in such a way that the symmetries of the instances are isomorphic. Indeed, in [3], some of these mappings are described and compared in the means of vertex colors, the sums of vertices and edges, spurious symmetry creation, phase-shift detection, and usability with the graph automorphism engine (see below) the authors use in a black-box fashion. The authors conclude that the reduction described also in this paper should be the most effective one, and directly usable with the automorphism engine, while the problem with circular chains of implications can (and must) be dealt with before or after the reduction in linear time. Empiric evidence on the size of the graph produced by the considered mappings on some structured instances is also presented. The results seem to show that the mappings are rather equivalent in the size sense.

4.2 Solving the Graph Automorphism Problem

The symmetry groups of the SAT instance can be found using the graph automorphism software NAUTY[7] [8], which is integrated in the GAP math software package [9] for group and algebraic operations. NAUTY, given a colored graph, gives as an answer a set of irredundant generators for the symmetry group of the problem. Although NAUTY is used in a black-box fashion, the authors report that the software seems to report only rarely spurious symmetries, while no empiric evidence or reason for this is given. Furthermore, the author mention observing that the running time of NAUTY typically grows with the number of vertices and symmetries, but may decrease with growing numbers of vertex colors and graph edges, with no justification. We believe that the number of symmetries should typically be inversely proportional to the number of vertex colors, as vertices must be mapped to vertices of the same color, making these two numbers related.

4.3 Symmetry-Breaking and Instance Blowup

In [3], only partial symmetry-breaking is considered; only the generators of the symmetry group is used in symmetry breaking. When using a black-box backtracking solver (as the author use in [3]), all of the symmetries should indeed not be added as constraint clauses to the instance, as they can be closely related, and thus might give notable rise to the solving time, as the number of the clauses rises rather rapidly with the symmetry-breaking clauses. Furthermore, justified by empiric evidence on the processing time compared to the benefit gained via decrease in solver running time (not reported) the authors propose exploiting generators with multiple cycles only up to 10 cycles. Interestingly, it is also reported, that cycles of length two seem to dominate in the set of generators (this brings importance to Theorem 3.2).

4.4 Effect of Symmetry-Breaking on Some Structured Instances

In [3], empiric evidence on the effect of symmetry-breaking on the total time (symmetry extraction + solving the modified instance with added symmetry-breaking clauses) needed to solve structured instances[8] is given. The authors use as a black-box SAT solver the state-of-the-art solver Chaff [10], which is known for its good performance on structured instances[9]. Interestingly, after applying the symmetry-breaking clauses, all but few instances were solved by Chaff in a fraction of a second, while in some cases the time of solving the graph automorphism problem exceeded the time of solving the original SAT instance. For more more detailed results, please consult [3].

[7] NAUTY is developed by Brendan McKay.

[8] For descriptions of the instances, see [3].

[9] For instance, Chaff is the winner of the 2002 SAT Competition in industrial and handmade benchmarks categories; see `http://www.ee.princeton.edu/ chaff/index.php`.

4.5 On Which Instance Domains Exhibit Rich Symmetries and Which Don't

As an example of a problem the SAT mapping of which can exhibit rich symmetries is the pigeon-hole problem. In the pigeon-hole problem we are to find a way to assign n pigeons into m pigeon-holes so that (i) two pigeons are not assign to the same hole, and (ii) every pigeon must be assigned to one hole. Clearly this problem doesn't have a solution when $n > m$. The problem is highly symmetric, as each pigeon can be assigned to each hole. The symmetry can be broken by assigning the pigeons in some specific order to the holes.

Although the pigeon-hole problem is a good example of a symmetry-rich problem, it is quite artificial, as the symmetries in the domain are well-known and can thus be taken into account directly in the SAT mapping. When such domain-specific knowledge is not available, symmetry-breaking is of high value in solving such instances. This also gives the idea of applying the symmetry-extraction to a particular SAT mapping in order to determine how loose the mapping is, i.e., how many symmetries are conserved by the mapping. This approach could be used e.g. in evaluating SAT mappings of encryption methods such as DES [20].

Instance domains, where few symmetries are present, are intuitively such domains where the structure of the instances is generated uniformly at random. The archetype of such domain is the Random k-SAT problems, where the instances consist of a set of clauses of length k, and each clause if generated by picking three variables at random from a particular set of variables and negating each variable with probability 0.5 [19]. Depending on the randomness of the random source used, such instances could still exhibit structural symmetries.

5 Discussion

The task of exploiting symmetries of SAT instances is certainly an interesting and possibly, as the few results given in [3] imply, a fruitful research topic in the quest for more competent techniques for solving (especially structured) SAT instances. In [3], it is mentioned that domain-specific knowledge of the problems modelled in SAT could be used already in the construction phase of the SAT instances. So called *opportunistic symmetry extraction*, where only symmetries within a subset of the variables are determined, with the intention of making symmetry extraction more efficient, is also discussed. Still, we believe that a more comprehensive and precise treatment of the topic than that given in [3] would be in order. Many questions are left unanswered: why the affect of symmetry-breaking is very solver-dependent, as reported in [3]; how do different sets of irredundant generators behave in symmetry-breaking (NAUTY is used in [3] as black-box, and no study of why such and such generators are reported by the system); could semantic but not syntactic symmetries be efficiently exploited; are there more efficient ways of constructing symmetry-breaking clauses, as the construction (badly) presented in [3], relying on ideas from [7], easily experiences blowup in the number of symmetry-breaking clauses; to what degree are symmetries domain-specific, and do only the "inevitably symmetry-rich" instance domains exhibit exploitable amounts of symmetries; etc. Furthermore, could a more dynamic approach be taken in exploiting symmetries than the preprocessing-based approach discussed here, perhaps by integrating symmetry extraction into a SAT solver. This might give a more SAT specific way of exploiting symmetries, and the blowup in the instance size brought on by the symmetry-breaking clauses could possibly be avoided. On the other hand, the best graph automorphism engines are highly tuned, running in linear average-case time, so beating such techniques would be a big challenge indeed.

References

[1] Hofmeister T., Schning U., Schuler R., Watanabe O. A Probabilistic 3-SAT Algorithm Further Improved". In Proceedings of the 19th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 2285, pp. 192-202. Springer-Verlag 2002.

[2] Cook S.A. The Complexity of Theorem-Proving Procedures. In Proceedings of the 3rd ACM Symposium on Theory of Computing, ppp 119–133. ACM 1971.

[3] Aloul F.A., Ramani A., Markov I.L., Sakallah K.A. Solving Difficult Instances of Boolean Satisfiability in the Presence of Symmetry. CSE-TR-463-02, University of Michigan 2002.

[4] Nicholson W.K. Introduction to Abstract Algebra, 2nd edition. John Wiley 1999.

[5] West D.B.. Introduction to Graph Theory, 2nd edition. Prentice Hall 2000.

[6] Papadimitriou C.H. Computational Complexity. Addison-Wesley 1994.

[7] Crawford J., Ginsberg M., Luks E., Roy A. Symmetry-Breaking Predicates for Search Problems. In Proceedings of Principles of Knowledge Representation and Reasoning '96, pp. 148–159. Morgan Kaufman 1996.

[8] McKay B.D. NAUTY User's Guide. Technical Report TR-CS-90-02, Computer Science Department, Australian National University, 1990. `http://cs.anu.edu.au/~bdm/nauty/`.

[9] Soicher L.H. Review of Mathematical Software GAP. Notices of American Mathematical Society, 41, 7, 1994. `http://www.gap-systems.org/`.

[10] Moskewicz M., Madigan C., Zhao Y., Zhang L., Malik S. Chaff: Engineering an Efficient SAT Solver. In Proceedings of the 39th Design Automation Conference, pp. 530–535. ACM 2001.

[11] Li C.M., Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In Proceedings of the 15th International Joint Conference on Artificial Intelligence, pp. 366–371. AAAI 1997.

[12] Goldberg E., Novikov Y. BerkMin: a Fast and Robust SAT Solver. In Proceedings of the Design, Automation and Test in Europe Conference, pp. 142–149. ACM 2002.

[13] Davis M., Putnam H. A Computing Procedure for Quantification Theory. Journal of the Association for Computing Machinery 7, 3, pp. 201–215, 1960.

[14] Davis M., Logemann G., Loveland D. A Machine Program for Theorem-proving. Communications of the Association for Computing Machinery 5, 7, pp. 394–397, 1962.

[15] Kautz H., Selman B. Planning as Satisfiability. In Proceedings of the 10th European Conference on Artificial Intelligence, pp 359–363. Wiley 1992.

[16] Gu J., Purdom P.W., Franco J., Wah B.W. 1996. Algorithms for the Satisfiability (SAT) Problem: a Survey. In Satisfiability Problem: Theory and Applications, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 35, AMS/ACM 1997.

[17] Velev M.N., Bryant R.E. Effective Use of Boolean Satisfiability Procedures on the Formal Verification of Superscalar and VLIW Microprocessors. In Proceedings of the Design Automation Conference, pp. 226–231. ACM 2001.

[18] Nam G., Aloul F., Sakallah K., Rutenbar R. A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints. In Proceedings of the International Symposium on Physical Design, pp. 222–227. ACM 2001.

[19] Crawford J.M., Auton L.D. Experimental Results on the Crossover Point in Random 3-SAT. Artificial Intelligence, 81, pp. 31–57, 1996.

[20] Massacci F., Marraro L. Logical Cryptanalysis as a SAT-problem: Encoding and Analysis of the U.S. Data Encryption Standard. Journal of Automated Reasoning, 24(1-2), pp. 165-203, 2000.