École doctorale EDITE de Paris (ED130)
Informatique, Télécommunication et Électronique

Thèse de doctorat de l'Université Sorbonne Université

Spécialité : **Ingénierie / Systèmes Informatiques**

Présentée par : **Hakan Metin**

Pour obtenir le grade de :

Docteur de l'Université Sorbonne Université

Sujet de la thèse :
Exploitation des symétries dynamiques pour la résolution des problèmes SAT

Soutenue le :

Devant le jury composé de :

Rapporteurs :	Y. XXX	XXX
	Y. XXX	XXX
Examinateurs :	Y. XXX	XXX
	Y. XXX	XXX
	Y. XXX	XXX
	Y. XXX	XXX

Ah la these

CONTENTS

SAT Theory NP complete Cite [COOK 71]

Used In
- Formal methods Hardware model checking;Software model checking; Termination analysis of term-rewrite systems ; Test pattern generation (testing of software hardware) ; etc
- AI planning, game n queen sudoku
- Bio info Haplotype inference ; Pedigree checking ; Analysis of Genetic; Regulatory Networks; etc.
- Design Automation: Equivalence checking; Delay computation; Fault diagnosis ; etc
- Security: Cryptanalysis ; Inversion attacks on hash functions; etc

Where can found
- Computationally hard problems: Graph coloring, Traveling salesperson
- Mathematical: van der Waerden numbers ; Quasigroup open problems
- Core engine for other solvers :0-1 ILP/Pseudo Boolean ;QBF ; #SAT ; SMT ; MAXSAT;
- Integrated into theorem provers :HOL ; Isabelle ;

- Integrated into widely used software: Eclipse provisioning system based on a Pseudo Boolean solver Suse 10.1 dependency manager based on a custom SAT solver

Interest in BMC [ ]Biere 99]

SAT solver Def

Input: Can encode any Boolean formula into Normal Form

Classical simplification: - Resolution - Subsumption

More complicated Hidden tautology BVA ... Loo Heule simplification slides

Algo CDCL

CHAPTER 1

INTRODUCTION

~~Nowadays, computers are powerfull and used in many applications in different domains. On of this domain is critical application, these applications are running in planes, cars and some software must be secure. Proving the correctness of these software leads to combinatorial explosion.~~

Over the years, computers scientist have developed many techniques to solve this kind of problems like *constraint programming* (CP) [10], *Propositional Satisfiability* (SAT) [3], *Solver Modulo Theory* (SMT) [2].

In this thesis, we focus on *propositional satisfiability* that is used in many applications in different domains: *formal methods*: hardware model checking, software model checking, etc; *artificial inteligence*: planning; *games resolution*: sudoku, n-queens, *Bioinformatics* : Haplotype inference, *design automation* : equivalence checking

At its most basic, symmetry is some transformations of an object that leaves this object unchanged. In the case of satisfiability problems it maps a solution of a problem to another solution of the problem.

CHAPTER 2

PRELIMINARIES

2.1 SAT basics

Satisfiability problem

A *Boolean variable*, or *propositional variable*, is a variable that has two possible values : true or false (noted $\top$ or $\bot$, respectively). A *literal* l is a propositional variable or its negation. For a given variable x, the positive literal is represented by x and the negative one by $\neg x$. A *clause* ω is a finite disjunction of literals represented equivalently by $\omega = \bigvee_{i=1}^{k} l_i$ or the set of its literals $\omega = \{l_i\}_{i \in [\![1,k]\!]}$. A clause with a single literal is called *unit clause*. A *conjunctive normal form (CNF) formula* φ is a finite conjunction of clauses. A CNF can be either noted $\varphi = \bigwedge_{i=1}^{k} \omega_i$ or $\varphi = \{\omega_i\}_{i \in [\![1,k]\!]}$. We denote $\mathcal{V}_\varphi$ ($\mathcal{L}_\varphi$) the set of variables (literals) used in φ (the index in $\mathcal{V}_\varphi$ and $\mathcal{L}_\varphi$ is usually omitted when clear from context).

For a given formula φ, an *assignment* of the variables of φ is a function $\alpha : \mathcal{V} \mapsto \{\top, \bot\}$. As usual, α is *total*, or *complete*, when all elements of $\mathcal{V}$ have an image by α, otherwise it is *partial*. By abuse of notation, an assignment is often represented by the set of its true literals. The set of all (possibly partial) assignments of $\mathcal{V}$ is noted $Ass(\mathcal{V})$.

The assignment α *satisfies* the clause ω, denoted $\alpha \models \omega$, if $\alpha \cap \omega \neq \emptyset$. Similarly, the assignment α satisfies the propositional formula φ, denoted $\alpha \models \varphi$, if α satisfies all the clauses of φ. Note that a formula may be satisfied by a partial assignment. A formula is said to be *satisfiable* (SAT) if there is at least one assignment that satisfies it; otherwise the formula is *unsatisfiable* (UNSAT).

Algorithm

A naive approach to solve a SAT problem is to try all possible assignments. In total, for a proposition formula with n variables, it leads to 2^n assignments and is intractable even for a formula with few variables. One the first non memory intensive algorithm to solve the SAT problems is the Davis Putnam Logemann Loveland (DPLL) algorithm. This algorithm introduce the *unit propagation*. This principle force the value of a literal when a clause is *assertive*, i.e. has all literals to false and one unassigned.

Hakan: Insert algorithm DPLL

The state of the art sound and complete algorithm to resolve a SAT problem is Conflict-Driven Clause learning (CDCL) algorithm 1. This algorithm is inspired by Davis Putnam Logemann Loveland (DPLL) algorithm [4].

The CDCL algorithm walks a binary search tree. It first applies unit propagation to the formula φ for the current assignment α (line 5). A conflict at level 0 indicates that the formula is not satisfiable, and the algorithm reports it (lines 7-8). If a conflict is detected, it is analyzed, which provides a *conflict clause* explaining the reason for the conflict (line 9). The analysis is completed by the computation of a backjump point to which the algorithm backtracks (line 10). This clause is learnt (line 11), as it does not change the satisfiability of φ, and avoids encountering a conflict with the same causes in the future. Finally, if no conflict appears, the algorithm chooses a new decision literal (line 13-14). The above steps are repeated until the satisfiability status of the formula is determined.

```
1  function CDCL(φ: CNF formula)
2      returns ⊤ if φ is SAT and ⊥ otherwise
3    dl ← 0 ;                                   // Current decision level
4    while not all variables are assigned do
5      isConflict ← unitPropagation();
6      if isConflict then
7        if dl = 0 then
8          return ⊥ ;                           // φ is UNSAT
9        ω ← analyzeConflict();
10       dl ← backjumpAndRestartPolicies();
11       φ ← φ ∪ {ω} ;
12     else
13       assignDecisionLiteral();
14       dl ← dl + 1;
15   return ⊤;                                  // φ is SAT
```

Algorithm 1: The CDCL algorithm.

2.2 Groups basics

Symmetries is related to a branch of mathematics called group theory. This section give us an overview of group theory.

Groups

A *group* is a structure $\langle G, * \rangle$, where G is a non empty set and $*$ a binary operation such the following axioms are satisfied:

- *associativity*: $\forall a, b, c \in G, (a * b) * c = a * (b * c)$
- *closure*: $\forall a, b \in G, a * b \in G.$
- *identity*: $\forall a \in G, \exists e$ such that $a * e = e * a = a$
- *inverse*: $\forall a \in G, \exists b \in G$, commonly denoted a^{-1} such that $a * a^{-1} = a^{-1} * a = e$

Note that *commutativity* is not required i.e $a * b = b * a$, for $a, b \in G$. The group is *abelian* if it satisfies the commutativity rule. Moreover, the last definition leads to important properties which are: i) uniqueness of the identity element. To prove this property, assume $\langle G, * \rangle$ a group with two identity elements e and f then $e = e * f = f$. ii) uniqueness of the inverse element. To prove this property, suppose that an element x_1 has two inverses, denoted b and c in group $\langle G, * \rangle$, then

$$\begin{array}{lll} b & = & b * e \\ & = & b * (a * c) \quad c \text{ is an inverse of } a, \text{so } e = a * c \\ & = & (b * a) * c \quad \textit{associativity} \text{ rule} \\ & = & e * c \quad b \text{ is an inverse of } a, \text{so } e = a * b \\ & = & c \quad \textit{identity} \text{ rule} \end{array}$$

The structure $\langle G, * \rangle$ is denoted as G when clear from context that G is a group with a binary operation. In this thesis, we interested only with the *finite* groups i.e with a finite number of elements.

Given a group G, a *subgroup* is a non empty subset of G which is also a group with the same binary operation. If H is a subgroup of G, we denote as $H \leq G$. A group has at least two subgroups: i) the subgroup composed by identity element $\{e\}$, denoted *trivial* subgroup. All other subgroups are *nontrivial*; ii) the subgroup composed by itself, denoted *improper* subgroup. All other subgroups are *proper*.

Generators of a group

If every elements in a group G can be expressed as a linear combination of a set of group of elements S = $\{g_1, g_2, ..., g_n\}$ then we say G is generated by the S. we denote this as G = $\langle S \rangle$ = $\langle \{g_1, g_2, ..., g_n\} \rangle$

Permutation groups

A *permutation* is a bijection from a set X to itself.

Example: given a set $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$, $g = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ x_2 & x_3 & x_1 & x_4 & x_6 & x_5 \end{pmatrix}$
g is a permutation that maps x_1 to x_2, x_2 to x_3, x_3 to x_1, x_4 to x_4, x_5 to x_6 and x_6 to x_5.

Permutations are generally written in *cycle notation* and the self mapped elements are omitted. So the permutation in cycle notation will be : $g = (x_1\ x_2\ x_3)\ (x_5\ x_6)$. We say *support* of the permutation g noted $supp(g)$ the elements that not mapped to themselves, $supp(g) = \{x \in X \mid g(x) \neq x\}$.

The set of permutations of a given set X form a group G, with the composition operation $(\circ)$ and called *permutation group*. The *symmetric group* id the set of all possible permutations of a set X and noted $\mathfrak{S}(X)$. So, a *permutation group* is a subgroup of $\mathfrak{S}(X)$.

A permutation group G induces a *equivalence relation* on the set of element X being permuted. Two elements $x_1, x_2 \in X$ are equivalent if there exists a permutation $g \in G$ such that $gx_1 = x_2$. Then equivalence relation partitions X into *equivalence classes* referred to as the *orbits* of X under G. The orbit of an element x under group G (or simply orbit of x when clear from the context) is the set $[x]_G = \{g.x \mid g \in G\}$

CHAPTER 3

SYMMETRY AND SAT

The group of permutations of $\mathcal{V}$ (i.e. bijections from $\mathcal{V}$ to $\mathcal{V}$) is noted $\mathfrak{S}(\mathcal{V})$. The group $\mathfrak{S}(\mathcal{V})$ naturally acts on the set of literals: for $g \in \mathfrak{S}(\mathcal{V})$ and a literal $\ell \in \mathcal{L}$, $g.\ell = g(\ell)$ if ℓ is a positive literal, $g.\ell = \neg g(\neg \ell)$ if ℓ is a negative literal. The group $\mathfrak{S}(\mathcal{V})$ also acts on (partial) assignments of $\mathcal{V}$ as follows: for $g \in \mathfrak{S}(\mathcal{V})$, $\alpha \in Ass(\mathcal{V})$, $g.\alpha = \{g.\ell \mid \ell \in \alpha\}$. Let φ be a formula, and $g \in \mathfrak{S}(\mathcal{V})$. We say that $g \in \mathfrak{S}(\mathcal{V})$ is a symmetry of φ if for every *complete* assignment α, $\alpha \models \varphi$ if and only if $g.\alpha \models \varphi$. The set of symmetries of φ is noted $S(\varphi) \subseteq \mathfrak{S}(\mathcal{V})$.

The previous mathematical definitions of group theory is applied to the CNF formula. So, the group of permutations of $\mathcal{V}$ (i.e. bijections from $\mathcal{V}$ to $\mathcal{V}$) is noted $\mathfrak{S}(\mathcal{V})$. We say that $g \in \mathfrak{S}(\mathcal{V})$ is a symmetry of φ if following conditions holds:

- permutation fixes the formula, $g(\varphi) = \varphi$
- g commutes with the negation: $g(\neg l) = \neg g(l)$

The set of symmetries of φ is noted $S(\varphi) \subseteq \mathfrak{S}(\mathcal{V})$. The sets of symmetries of a formula φ preserves the satisfaction, for every *complete* assignment α, $\alpha \models \varphi \leftrightarrow g(\alpha) \models \varphi$ for $g \in S(\varphi)$. The group $S(\varphi)$ also acts on (partial) assignments of $\mathcal{V}$ as follows: for $g \in S(\varphi)$, $\alpha \in Ass(\mathcal{V})$, $g.\alpha = \{g.\ell \mid \ell \in \alpha\}$. Hakan: Permutations acts also on clauses

The next section presents how to compute the set of *generators* of a given formula.

3.1 Symmetry detection in SAT

For the detection of symmetries in SAT, we fist introduce the graph automorphism notion. Given a colored graph $G = (V, E, \gamma)$, with vertex set $V \in [1, n]$, edge set E and γ a function that apply a mapping : $V \to C$ where C is a set of *colors*. An automorphism of G is a permutation from its vertices $g : V \to V$ such that:

- $\forall (u, v) \in E \implies (g(u), g(v)) \in E$
- $\forall v \in V, \gamma(v) = \gamma(g(v))$

The graph automorphism problem is to find if a given graph has a non trivial permutation group. The computational complexity of this algorithm is conjectured to be strictly between P and NP. Several tools exists to tackle this problem like `saucy3` [7], `bliss` [6], `nauty` [9], etc.

There exists different ways to encode a SAT problems, which leads to different symmetries in these problem. When a symmetry depends on the structure of the problem, we say *syntactic* symmetries. In contrast, symmetries were *semantic*, when it is not inherent to the encoding. To find symmetries in SAT problem, the formula is transformed into colored graph and an automorphism tool is applied onto. Specifically, given a formula φ with m clauses over n variables, the graph is constructed as follows:

- *clause nodes*: represent each of the m clauses by a node with color 0;
- *literals nodes*: represent each of the l literals by a node with color 1;
- *clauses edges*: connect each clause node to the node of the literals that appear in clause;
- *boolean consistency edges*: connect each pair of literals that correspond to the same variables.

Hakan: Explication du graph + informations num nodes num edges. Probleme reel battleship

The battleship problems place *** and two * in grid 3x4
1 2 3
4 5 6
7 8 9
10 11 12

one ship per row.

Produced graph contains 12 * 2 = 24 + 21 = 45 nodes and 24 + 36 = 60 edges

```
p cnf 12 21
−1 −2 0
−1 −3 0
−2 −3 0
−4 −5 0
−4 −6 0
−5 −6 0
−7 −8 0
−7 −9 0
−8 −9 0
−10 −11 0
−10 −12 0
−11 −12 0
1 4 7 10 0
2 5 8 11 0
3 6 9 12 0
1 6 8 10 0
2 4 9 11 0
3 5 7 12 0
1 5 9 10 0
2 6 7 11 0
3 4 8 12 0
```

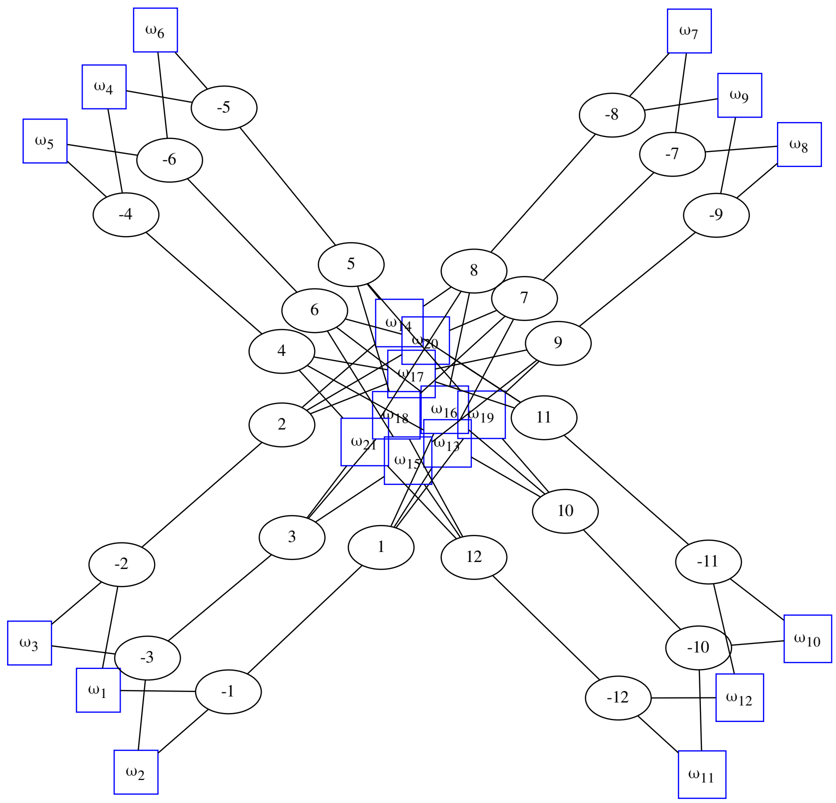

Figure 3.1: Example of constructed symmetry graph for a given CNF

An optimization of this graph is possible with the usage of binary clauses i.e. a clause with only two literals. The clause node can be omitted and we connect the two literals. As we cannot distinguish between the optimized edge and boolean consistency edges, we must check if the produced permutations are spurious. To do so, as we ensure the permutation commutes with the negation it suffice to check: $\forall x \in Supp(g), g.\neg x = \neg g.x$. Roughly speaking, we check if the image of the negation of x is equals to the negation of the image of x, for each element x in the support of the permutation. This optimization allows to compute symmetries of the problem more efficiently. In the previous example, the graph has deleted 12 nodes and 12 edges. More generally, the graph removes as many nodes and edges as binary clauses on the formula.

```
p cnf 12 21
−1 −2 0
−1 −3 0
−2 −3 0
−4 −5 0
−4 −6 0
−5 −6 0
−7 −8 0
−7 −9 0
−8 −9 0
−10 −11 0
−10 −12 0
−11 −12 0
1 4 7 10 0
2 5 8 11 0
3 6 9 12 0
1 6 8 10 0
2 4 9 11 0
3 5 7 12 0
1 5 9 10 0
2 6 7 11 0
3 4 8 12 0
```

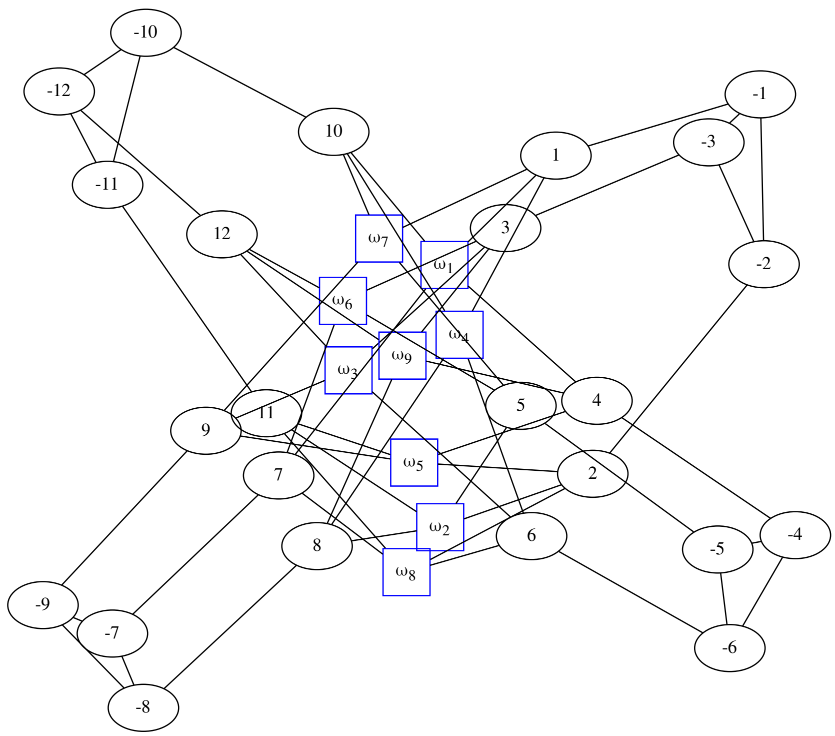

Figure 3.2: Example of constructed symmetry graph for a given CNF

When these graph is given to an automorphism tool like `bliss`, the following *generators* are obtained:

- g_0: (2 3)(5 6)(8 9)(11 12)(-2 -3)(-5 -6)(-8 -9)(-11 -12)
- g_1: (4 5 6)(7 9 8)(-4 -5 -6)(-7 -9 -8)
- g_2: (4 7)(5 8)(6 9)(-4 -7)(-5 -8)(-6 -9)
- g_3: (1 2)(5 6)(7 9)(10 11)(-1 -2)(-5 -6)(-7 -9)(-10 -11)
- g_4: (1 10)(2 11)(3 12)(-1 -10)(-2 -11)(-3 -12)

The visualization of the orbits of literals on the problem could be seen in figure 3.3, where each node represents a literal. Two literals are linked with an arc if it exists a permutation that maps the literal to the second one. By definition of the orbits, each literal belongs to a strongly connected components (SCC).

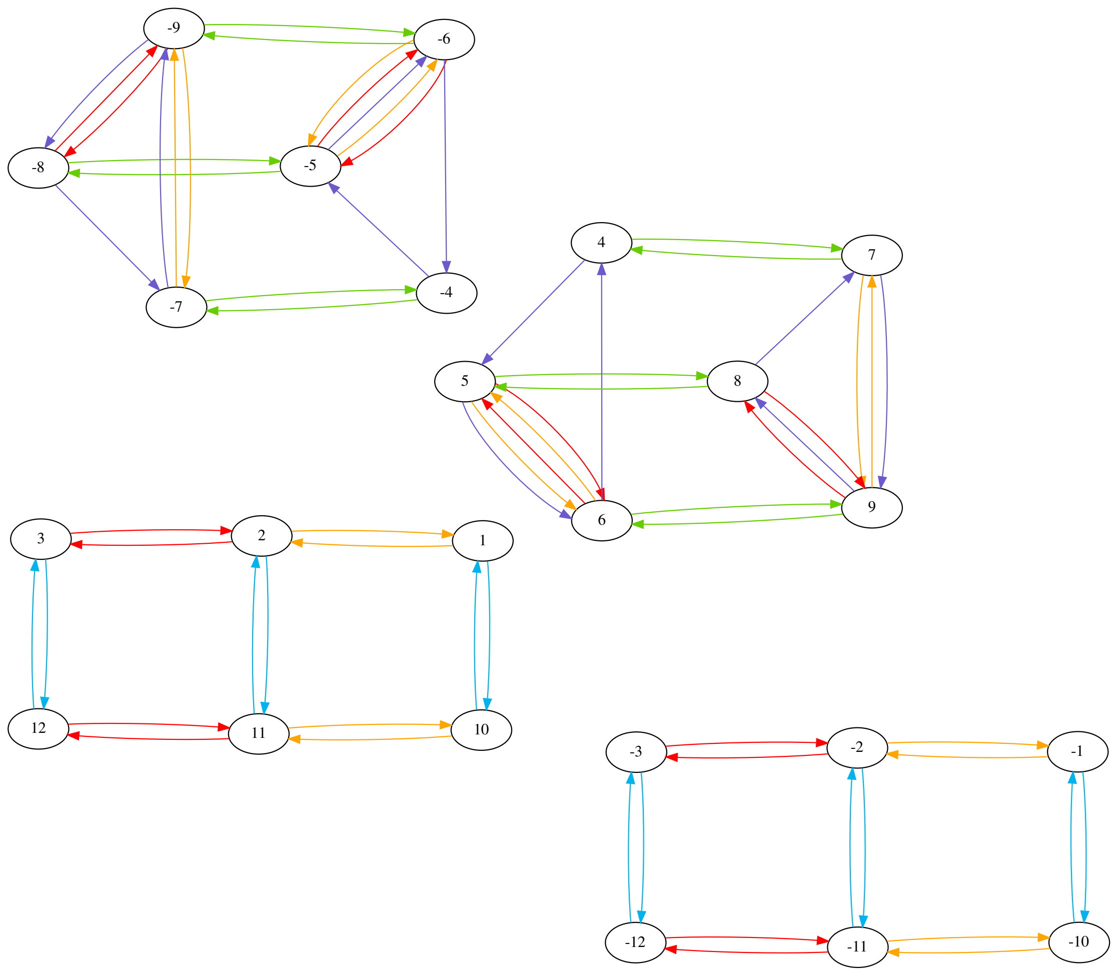

Figure 3.3: Orbits

3.2 Usage of symmetries

Symmetry breaking aims at eliminating symmetry, either by *statically* posting symmetry breaking constraints that invalidate symmetric assignments, or by altering the search space *dynamically* to avoid symmetric search paths.

In order to exploit the symmetry properties of a SAT problem, we need to introduce an ordering relation between the assignments.

Definition 1 (Assignments ordering)**.** *We assume a total order, $\prec$, on $\mathcal{V}$. Given two assignments $(\alpha, \beta) \in Ass(\mathcal{V})^2$, we say that α is strictly smaller than β, noted $\alpha < \beta$, if there exists a variable $v \in \mathcal{V}$ such that:*

- *for all $v' \prec v$, either $v' \in \alpha \cap \beta$ or $\neg v' \in \alpha \cap \beta$.*
- *$\neg v \in \alpha$ and $v \in \beta$.*[1]

Note that $<$ coincides with the lexicographical order on *complete* assignments. Furthermore, the $<$ relation is monotonic as expressed in the following proposition.

Proposition 1 (Monotonicity of assignments ordering)**.** *Let $(\alpha, \alpha', \beta, \beta') \in Ass(\mathcal{V})^4$ be four assignments.*

$$\text{If } \alpha \subseteq \alpha' \text{ and } \beta \subseteq \beta',\ \text{then } \alpha < \beta \implies \alpha' < \beta'$$

Proof. The proposition follows on directly from Definition 1. □

Let φ a formula, and G the group from the formula. The *orbit of α under G* (or simply the *orbit of α* when G is clear from the context) is the set $[\alpha]_G = \{g.\alpha \mid g \in G\}$. The lexicographic leader (*lex-leader* for short) of an orbit $[\alpha]_G$ is defined by $min_<([\alpha]_G)$. This *lex-leader* is unique because the lexicographic order is a total order. The optimal approach to solve a symmetric SAT problem would be to explore only one assignment per orbit (for instance each lex-leader). However, finding the lex-leader of an orbit is computationally hard [8].

The exploitation of symmetries statically is called *static symmetry breaking*. It acts like a preprocessor which add *symmetry breaking predicates* (SBP) at the original formula and solve the augmented problem,

Proposition 2 (Satisfiability preservation)**.** *Let φ a CNF problem and ψ the computed symmetry breaking predicates. Solve $\varphi \cup \psi$ is equi-satisfiable to solve φ.*

Proof. On the first case, if the initial formula φ is UNSAT, then the augmented problem $\varphi \cup \psi$ is trivially UNSAT. On the second case, if φ is SATthen $\varphi \cup \psi$ is also SATbecause ψ forbids only non lex-leader assignment so the formula still be SAT. □

[1] We could have chosen as well $v \in \alpha$ and $\neg v \in \beta$ without loss of generality.

CHAPTER 4

SYMMSAT

Drawbacks of the static-based approaches.

In the general case, the size of the *sbp* can be exponential in the number of variables of the problem so that they cannot be totally computed. Even in more favorable situations, the size of the generated *sbp* is often too large to be effectively handled by a SAT solver [8]. On the other hand, if only a subset of the symmetries is considered then the resulting search pruning will not be that interesting and its effectiveness depends heavily on the heuristically chosen symmetries [3]. Besides, these approaches are preprocessors, so their combination with other techniques, such as *symmetry propagation* [5], can be very hard. Also, tuning their parameters during the solving turns out to be very difficult. For all these reasons, some classes of SAT problems cannot be solved yet despite exhibiting symmetries.

Proposed solution.

To handle these issues, we propose a new approach that reuses the principles of the static approaches, but operates dynamically: the symmetries are broken during the search process without any pre-generation of the *sbp*. To do so, we elaborate the notions of *symmetry status tracking* and *effective symmetric breaking predicates* (*esbp*).

The approach is implemented using a couple of components: (1) a *Conflict Driven Clauses Learning (CDCL) search engine*; (2) *a symmetry controller*. Roughly speaking, the first component performs the classical search activity on the SAT problem, while the second observes the engine and maintains the status of the symmetries. When the controller detects a situation where the engine is starting to explore a redundant part[1], it orders the engine to operate a backjump. The detection is performed thanks to *symmetry status tracking* and the backjump order is given by a simple injection of an *esbp* computed on the fly.

[1] Isomorphic to a part that has been/will be explored.

The main advantage of such an approach is to cope with the heavy (and potentially blocking) pre-generation phase of the static-based approaches, but also offers opportunities to combine with other dynamic-based approaches, like the *symmetry propagation* technique [5]. It also gives more flexibility for adjusting some parameters on the fly. Moreover, the overhead for non symmetric formulas is reduced to the computation time of the graph automorphism.

The extensive evaluation of our approach on the symmetric formulas of the last six SAT contests shows that it outperforms the state-of-the-art techniques, in particular on unsatisfiable instances, which are the hardest class of the problem.

What we propose here is a best effort approach that tries to eliminate, *dynamically*, the *non lex-leading* assignments with a minimal computation effort. To do so, we first introduce the notions of *reducer*, *inactive* and *active* permutation with respect to an assignment α.

Definition 2 (Reducer, inactive and active permutation)**.** *A permutation g is a* reducer *of an assignment α if $g.\alpha < \alpha$ (hence α cannot be the lex-leader of its orbit. g reduces it and all its extensions). g is* inactive *on α when $\alpha < g.\alpha$ (so, g cannot reduce α and all the extensions). A symmetry is said to be* active *with respect to α when it is neither inactive nor a reducer of α.*

Proposition 3 restates this definition in terms of variables and is the basis of an efficient algorithm to keep track of the status of a permutation during the solving. Let us, first, recall that the *support*, $\mathcal{V}_g$, of a permutation g is the set $\{v \in \mathcal{V} \mid g(v) \neq v\}$.

Proposition 3. *Let $\alpha \in Ass(\mathcal{V})$ be an assignment, $g \in \mathfrak{S}\mathcal{V}$ a permutation and $\mathcal{V}_g \subseteq \mathcal{V}$ the support of g. We say that g is:*

1. a reducer of α *if there exists a variable $v \in \mathcal{V}_g$ such that:*
   - $\forall\ v' \in \mathcal{V}_g$, *s. t.* $v' \prec v$, *either* $\{v', g^{-1}(v')\} \subseteq \alpha$ *or* $\{\neg v', \neg g^{-1}(v')\} \subseteq \alpha$,
   - $\{v, \neg g^{-1}(v)\} \subseteq \alpha$;
2. inactive *on α if there exists a variable $v \in \mathcal{V}_g$ such that:*
   - $\forall\ v' \in \mathcal{V}_g$, *s. t.* $v' \prec v$, *either* $\{v', g^{-1}(v')\} \subseteq \alpha$ *or* $\{\neg v', \neg g^{-1}(v')\} \subseteq \alpha$,
   - $\{\neg v, g^{-1}(v)\} \subseteq \alpha$;
3. active *on α, otherwise.*

When g is a *reducer* of α we can define a predicate that contradicts α yet preserves the satisfiability of the formula. Such a predicate will be used to discard α, and all its extensions, from a further visit and hence pruning the search tree.

Definition 3 (Effective Symmetry Breaking Predicate)**.** *Let $\alpha \in Ass(\mathcal{V})$, and $g \in \mathfrak{S}\mathcal{V}$. We say that the formula ψ is an effective symmetry breaking predicate (esbp for short) for α under g if:*

$$\alpha \not\models \psi \textit{ and for all } \beta \in Ass(\mathcal{V}), \beta \not\models \psi \Rightarrow g.\beta < \beta$$

The next definition gives a way to obtain such an effective symmetry-breaking predicate from an assignment and a reducer.

Definition 4 (A construction of an *esbp*)**.** *Let φ be a formula. Let g be a symmetry of φ that reduces an assignment α. Let v be the variable whose existence is given by item 1. in Proposition 3. Let $U = \{v', \neg v' \mid v' \in \mathcal{V}_g$ and $v' \preceq v\}$. We define $\eta(\alpha, g)$ as $(U \cup g^{-1}.U) \setminus \alpha$.*

Example. Let us consider $\mathcal{V} = \{x_1, x_2, x_3, x_4, x_5\}$, $g = (x_1\, x_3)(x_2\, x_4)$, and a partial assignment $\alpha = \{x_1, x_2, x_3, \neg x_4\}$. Then, $g.\alpha = \{x_1, \neg x_2, x_3, x_4\}$ and $v = x_2$. So, $U = \{x_1, \neg x_1, x_2, \neg x_2\}$ and $g^{-1}.U = \{x_3, \neg x_3, x_4, \neg x_4\}$ and we can deduce than $\eta(\alpha, g) = (U \cup g^{-1}.U) \setminus \alpha = \{\neg x_1, \neg x_2, \neg x_3, x_4\}$.

Proposition 4. *$\eta(\alpha, g)$ is an effective symmetry-breaking predicate.*

Proof. It is immediate that $\alpha \not\models \eta(\alpha, g)$.

Let $\beta \in Ass(\mathcal{V})$ such that $\beta \wedge \eta(\alpha, g)$ is UNSAT. We denote a α' and β' as the restrictions of α and β to the variables in $\{v' \in \mathcal{V}_g \mid v' \preceq v\}$. Since $\beta \wedge \eta(\alpha, g)$ is UNSAT, $\alpha' = \beta'$. But $g.\alpha' < \alpha'$, and $g.\beta' < \beta'$. By monotonicity of $<$, we thus also have $g.\beta < \beta$. □

It is important to observe that the notion of *ebsp* is a refinement of the classical concept of *sbp* defined in [1]. In particular, like *sbp*, *esbp* preserve satisfiability.

Theorem 1 (Satisfiability preservation)**.** *Let φ be a formula and ψ an ebsp for some assignment α under $g \in S(\varphi)$. Then,*

$$\varphi \textit{ and } \varphi \wedge \psi \textit{ are equi-satisfiable.}$$

Proof. If $\varphi \wedge \psi$ is SAT then φ is trivially SAT. If φ is SAT, then there is some assignment β that satisfies φ. Without loss of generality, β can be chosen to be the lex-leader of its orbit under $S(\varphi)$. Thus, g does not reduce β, which implies that $\beta \models \psi$.

□

CHAPTER 5

CONCLUSION

This is conclusion.

Bibliography

[1] F. Aloul, K. Sakallah, and I. Markov. Efficient symmetry breaking for boolean satisfiability. *IEEE Trans. Computers*, 55(5):549–558, 2006.

[2] C. Barrett and C. Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[3] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[5] J. Devriendt, B. Bogaerts, B. de Cat, M. Denecker, and C. Mears. Symmetry propagation: Improved dynamic symmetry breaking in SAT. In *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*, pages 49–56, 2012.

[6] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In D. Applegate, G. S. Brodal, D. Panario, and R. Sedgewick, editors, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, pages 135–149. SIAM, 2007.

[7] H. Katebi, K. Sakallah, and I. Markov. Symmetry and satisfiability: An update. *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 113–127, 2010.

[8] E. M. Luks and A. Roy. The complexity of symmetry-breaking formulas. *Ann. Math. Artif. Intell.*, 41(1):19–45, 2004.

[9] B. D. McKay. nauty user's guide (version 2.2). Technical report, Technical Report TR-CS-9002, Australian National University, 2003.

[10] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.