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 thèse.

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) [24], *Propositional Satisfiability* (SAT) [5], *Satisfiability Modulo Theory* (SMT) [4].

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 respectively $\top$ or $\bot$). 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 clause is a *tautology* if it is always true, a clause that contains a positive and negative value of a clause for example. 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. In this case, unassigned variable are called *dont care*. A formula is said to be *satisfiable* (SAT) if there is at least one assignment that satisfies it; otherwise the formula is *unsatisfiable* (UNSAT).

An NP-complete problem

The SAT problems is the first NP-complete algorithm as proven by Stephen Cook in 1971 [6]. NP-completeness means that a SAT problem can be solved with a non deterministic machine in polynomial time (NP) and is also NP-hard. A problem is said NP-hard if everything in NP can be transformed into it in polynomial time. One of the most important unsolved problem in theoretical computer science is the P versus NP problem. This question is one of the seven millennium prize problems.

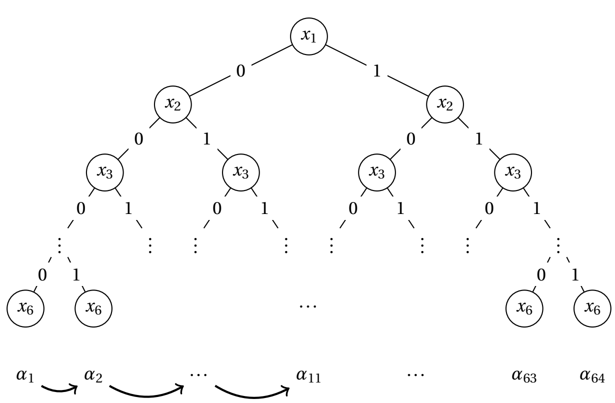
Some particular form of the SAT problems can be computed in linear time for 2-SAT [3] where each clause is in binary form i.e. size of two. Others particular form can be solved in polynomial time like Horn SAT [3] in which it suffice to apply *unit propagation* explained in section 2.1 until fix point. Xor-SATisfiability where each clause contains exclusive or belong also to the polynomial class.

Solving a SAT problem

Two kinds of algorithm exists to solve satisfiability problems. First, the *incomplete* algorithm which does not provide any guarantee that will eventually report either any satisfiable assignment or declare that formula is unsatisfiable. This kind of algorithm is out of scope of this thesis. Second, the *complete* algorithm, which provides a guarantee that if an assignment exists it will be reached or it will declare that formula is unsatisfiable. This section describes different *complete* algorithm to solve a propositional formula.

A naive algorithm

A naive approach to solve a SAT problem is to try all possible assignments. In total, for a propositional formula with n variables, it leads to 2^n assignments in the worth case. Figure 2.1 illustrate the search tree for a given problem with six variables. In this case α_{11} $(\neg x_1, \neg x_2, x_3, \neg x_4, x_5, \neg x_6)$ is found as a solution of the problem. In the general case, this algorithm is obviously intractable on real problems even for a formula with few variables.



$$\omega_1 = \{x_1, x_2, x_3\}$$
$$\omega_2 = \{x_4, x_5, x_6\}$$
$$\omega_3 = \{\neg x_1, \neg x_5\}$$
$$\omega_4 = \{\neg x_2, \neg x_4\}$$
$$\omega_5 = \{\neg x_3, \neg x_4\}$$
$$\omega_6 = \{\neg x_3, \neg x_6\}$$

Figure 2.1: All possible assignments for a problem with 6 variables

Davis Putnam Logemann Loveland (DPLL) algorithm

One of the first non memory intensive algorithm to solve the SAT problems is the Davis Putnam Logemann Loveland (DPLL) algorithm [8]. It explores a binary tree using depth first search as given in Algorithm 1. The construction of the tree is related to a *decision* literal (line 8) then, recursive call with each value are checked. When a leaf report UNSAT(line 5), other branches are explored. By recursive construction of the algorithm, when positive and negative value of a literal reach UNSAT, solver *backtracks* at most one level, this fact is called *chronological backtracking*. If all leaves report UNSAT the formula φ is unsatisfiable line 14. Finally, if any branch found a solution i.e. the problem is empty, formula is satisfiable corresponding assignment is returned (lines 10 and 12)

```
1  function DPLL(φ: CNF formula, α assignment)
2      returns an assignment if φ is SAT and UNSAT otherwise
3      φ, α ← unitPropagation(φ, α);
4      if {} ∈ φ then
5          return UNSAT;                          // This branch is UNSAT
6      if φ = {} then
7          return α;                              // φ is SAT
8      x ← assignDecisionLiteral();
9      if α ← DPLL(φ ∪ {x}, α) ≠ UNSAT then
10         return α
11     else if α ← DPLL(φ ∪ {¬x}, α) ≠ UNSAT then
12         return α
13     else
14         return UNSAT;                          // φ is UNSAT
```

Algorithm 1: The DPLL algorithm.

An important function in the DPLL algorithm is `unitPropagation` line 3 and it is detailed in Algorithm 2. It searches all unit clauses, then, to ensure satisfiability these literals must be true and added to the current assignment. Formula is then simplified as follow, clauses that contains this literal are already satisfied and can be deleted; negative literals are removed from the clause that belongs to him. This procedure ends when either no unit clause remains or an inconsistency was found (empty clause).

When DPLL algorithm is executed on the formula in Figure 2.1, after the decision of literal $\neg x_1$ and $\neg x_2$ unit propagation detects that x_3 must be true. This propagation prevents to explore assignments from α_1 to α_8. Moreover, application to unit propagation provokes more unit clauses and leads directly to a solution. An important part of efficiency of DPLL is due to choose the variable that divide the search tree made by the procedure `assignDecisionLiteral` . The objective of this function is to find a literal that will generate a maximum of unit propagation. Intuitively, decision literals can be viewed as

function `unitPropagation(`*φ: CNF formula, α assignment*`)`
    returns *CNF formula and assignment α*
    while $\{l\} \in \varphi$ ***and*** $\{\} \notin \varphi$ **do**
        `// Remove all clauses containing ` l`, all literals ` $\neg l$
        $\varphi \leftarrow \varphi\,|_l$
        $\alpha \leftarrow \alpha \cup \{l\}$
    return φ, α

Algorithm 2: Unit propagation

"guesses" and propagated literals can be viewed as "deductions". Finding a optimal variable is NP-Hard. Different heuristics exists to choose the decision variable, some of them will be presented in section 16.

Conflict Driven Clause Learning (CDCL) algorithm

The principal weakness of DPLL algorithm is to make same inconsistencies several times (principally due to chronological backtracking), incurring unnecessary CPU usage.
Conflict Driven Clause Learning (CDCL) algorithm 3 is another sound and complete algorithm to resolve a SAT problem and overcome principal weakness of DPLL.

Algorithm 3 gives an overview of CDCL, Like DPLL, it walks on a binary search tree. Initially, the current assignment is empty and decision level that indicated the depth of the search tree noted as dl is set to zero. Algorithm first applies unit propagation to the formula φ for the current assignment α (line 6). Note that it is exactly the same procedure as the one used for DPLL. An inconsistency or a *conflict* at level zero indicates that the formula is unsatisfiable, and the algorithm reports it (lines 8 and 9). When the conflict is occurring at a higher level, it reason was analyzed and a clause called *conflict clause* is deduced (line 10). This clause is *learnt* (line 12) (added to the formula). This clause is redundant from the current formula and so as it does not change the satisfiability of φ. It also avoids encountering a conflict with the same causes in the future. Working of this function will be presented thereafter. The analysis is completed by the computation of a *backjump*, solver unassign some literals and decrease the decision level (line 11). As the level can be much lower than the current assignment this is called *non chronological backtracking*. Finally, if no conflict appears, the algorithm chooses a new decision literal (lines 14 and 15). The above steps are repeated until the satisfiability status of the formula is determined.

Conflict Analysis

A conflict is an inconsistency discovered by the solver, a situation that requires for a variable to be set simultaneously to the $\top$and $\bot$value. Figure 2.2 shows an assignments that leads to a conflict. First the solver chose $\neg x_1$ as decision then $\neg x_6$ and then $\neg x_5$. This last one propagates x_4 which in turn propagates x_2 and x_3. On clause ω_1, x_3 needs to be $\top$and $\bot$in ω_5 so a conflict appears.

```
function CDCL(φ: CNF formula)
    returns ⊤ if φ is SAT and ⊥ otherwise
  dl ← 0;                                          // Current decision level
  α ← ∅;
  while not all variables are assigned do
    φ, α ← unitPropagation(φ|α, α);
    if {} ∈ φ then                                 // A conflict occurs
      if dl = 0 then
        return ⊥;                                  // φ is UNSAT
      ω ← analyzeConflict();
      dl ← backjumpAndRestartPolicies();
      φ ← φ ∪ {ω};
    else
      α ← α ∪ assignDecisionLiteral();
      dl ← dl + 1;
  return ⊤;                                        // φ is SAT
```

Algorithm 3: The CDCL algorithm.

$$
\begin{array}{lllll}
\omega_1 = \{x_1, x_2, x_3\} & \omega_1 = \{x_1, x_2, x_3\} & \omega_1 = \{x_1, x_2, x_3\} & \omega_1 = \{x_1, x_2, x_3\} & \omega_1 = \{x_1, x_2, x_3\} \\
\omega_2 = \{x_4, x_5, x_6\} & \omega_2 = \{x_4, x_5, x_6\} & \omega_2 = \{x_4, x_5, x_6\} & \omega_2 = \{x_4, x_5, x_6\} & \omega_2 = \{x_4, x_5, x_6\} \\
\omega_3 = \{\neg x_1, \neg x_5\} & \omega_3 = \{\neg x_1, \neg x_5\} & \omega_3 = \{\neg x_1, \neg x_5\} & \omega_3 = \{\neg x_1, \neg x_5\} & \omega_3 = \{\neg x_1, \neg x_5\} \\
\omega_4 = \{\neg x_2, \neg x_4\} & \omega_4 = \{\neg x_2, \neg x_4\} & \omega_4 = \{\neg x_2, \neg x_4\} & \omega_4 = \{\neg x_2, \neg x_4\} & \omega_4 = \{\neg x_2, \neg x_4\} \\
\omega_5 = \{\neg x_3, \neg x_4\} & \omega_5 = \{\neg x_3, \neg x_4\} & \omega_5 = \{\neg x_3, \neg x_4\} & \omega_5 = \{\neg x_3, \neg x_4\} & \omega_5 = \{\neg x_3, \neg x_4\} \\
\omega_6 = \{\neg x_3, \neg x_6\} & \omega_6 = \{\neg x_3, \neg x_6\} & \omega_6 = \{\neg x_3, \neg x_6\} & \omega_6 = \{\neg x_3, \neg x_6\} & \omega_6 = \{\neg x_3, \neg x_6\}
\end{array}
$$

$$
\alpha = \quad \mathrm{D}\ \neg x_1 \quad \mathrm{D}\ \neg x_6 \quad \mathrm{D}\ \neg x_5 \quad \mathrm{P}\ x_4 \quad \mathrm{P}\ \neg x_2 \longrightarrow \mathrm{C}
$$

Figure 2.2: Decisions/Propagations that leads to a conflict

This series of decisions would provoke same propagation and leads to the same conflict. To avoid this situation, the solver needs to analyze the reason of the conflict with so called *implication graph*. Implication graph represents the current state of the solver proof system. It records every dependencies among variables and so it is updated either when a variable is assigned on decision/propagation or when a variable is unassigned. The implication graph is a directed acyclic graph (DAG) in which a vertex represents an assigned variable labeled as $l@dl(l)$ where l represents assigned literal and $dl(l)$ represents the decision level of the literal l. Root vertexes , that have no incoming edges, are literals chosen by decision heuristics and others are propagated literals. Incoming arcs labeled with a clause represents the *reason* of this propagation. This clause must be assertive i.e. all of its literals are false except one that are not yet assigned. Figure 2.3 shows implication graph of the previous example (fig. 2.2) until the conflict.

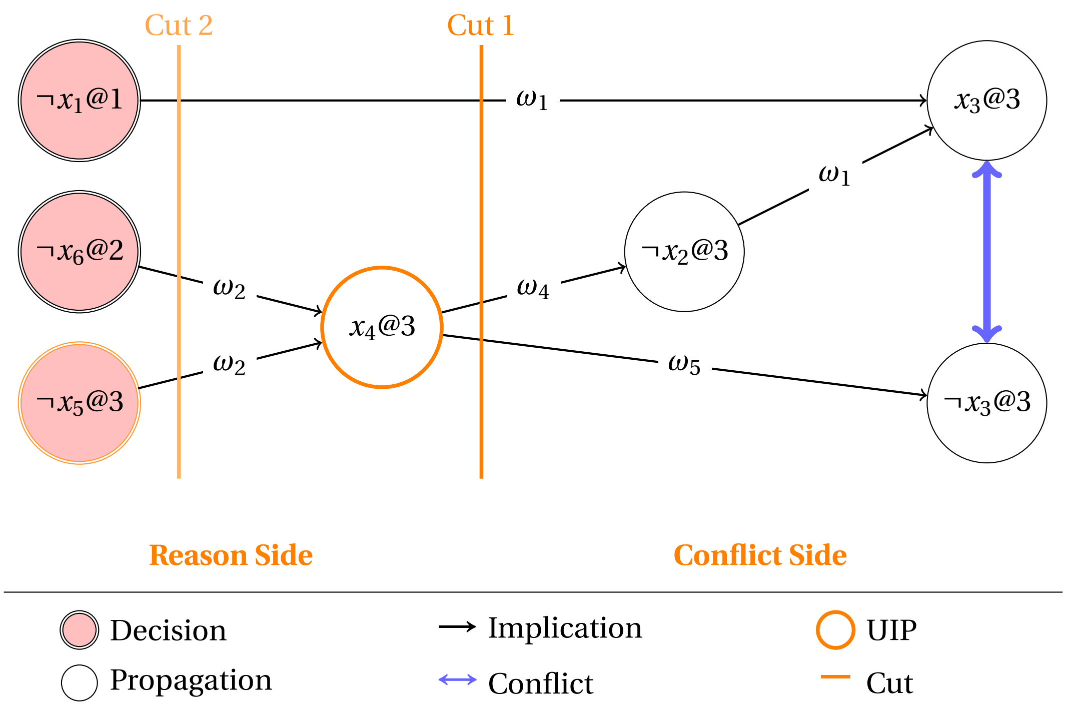


Figure 2.3: Implication graph

`analyzeConflict` procedure analyzes this graph to found the reason of the conflict. To do that, a search of *unique implication point* (UIP) is performed. UIP of a decision level in the implication graph is a variables which lies on every path from the decision to the conflict. Note that, there are many UIP for a given decision level. In such case, UIPs are ordered according to the distance with the contradiction. The first UIP is the closest to the conflict. It is well known that the first UIP provides the smallest set of assignment that is responsible for the contradiction [28]

An UIP divides the implication in two sides with a *cut*, the reason side contains decision variables that is responsible for the contradiction and the conflict side that contains the conflict. Note that, UIP is always is the reason side. Figure 2.3 depicts two cuts in the implication graph.

Once the reason side of a conflict is established, a conflict driven clause or conflict clause is produced. It purposes to avoid same contradiction. To build this clause, it suffices to negate literals that have an ongoing arc to the first cut that contains first UIP. In fig. 2.3, produced clause will be $\omega_l = \{x_1, \neg x_4\}$. Since the information of this clause is redundant with respect to the original formula. It can be added without any satisfiability restrictions. The conflict clause can be simplified using the implication graph [27].

`backjumpAndRestartPolicies` procedure is executed after producing the conflict clause. It will unassigned all decision until the first one that is responsible for the inconsistency. Add the conflict clause prune search space that contains no solution. This is the key point of the CDCL algorithm. The restart policy will be discussed in the next section. In our example Figure 2.3, the target decision level is one. The first UIP variable must be assertive, and will be propagated in the next loop of the solving algorithm. If a conflict implied only one level, the decision variable must be assigned to the opposite value at level zero. Roughly speaking, to ensure satisfiability of the formula, this literal must be true without any decision.

Heuristics

This sections gives an overview of different heuristics present in modern SAT solvers.

Decision heuristics. Variable used to divide problems have a huge impact on the overall solving time by the solver. Decision variable may impact the number of propagation and so the depth of the search tree.

Variable State Independent Decaying Sum (VSIDS) [23] is one of the decision heuristic and used nowadays in almost all solvers. Each variable has an activity and was increased by a multiplicative factor when it participate to the resolution of the conflict. A solver has thousands conflicts during the solving and so activity of variables are very volatile. Decision heuristics choose unassigned variable with the highest activity. The idea behind this heuristics is to solve "hard" part of problem at the top of the search tree. Hence, it is much more efficient when coupled with the restart heuristics.

Learning rate based branching (LRB [20]) is a most recent decision heuristics. It is a generalization of VSIDS and its goal is to optimize the *learning rate* (LR), defined as the ability to generate learnt clauses. The LRB of a variable is the weighted average (computed with *exponential recency weighted average* (ERWA)) value taken by its LR over the time. Unassigned variable with a highest LRB are chose as decision. The idea behind this heuristics is to keep variables that used to generate learnt clause in the search tree.

Restarts. Another important heuristic is *restart*. Basically the solver abandons it current assignment and so start from the top of the tree, while maintaining other information notably learnt clauses but also scores of variables in the decision heuristic. It prevents the solver to get stuck in "hard" (heavy tailing [14] part of the search space and can not escape due to backjump few levels after conflict resolution. Restart is best effort heuristics, hoping that, with more information, a better assignments was made. Hence, in practice, SAT solvers usually restarts after a certain number of conflict. Empirically a solver with restart has a better results [15] and is today used in almost all state of the art solvers.

Preprocessing / Inprocessing

In order to optimize resolution time by the solver, some transformation to simplify the original formula can be applied. This is done by *preprocessing* engine before the start of solving. When it is used at some point during the solving, usually after a restart, it is called *inprocessing*.

Simplification of the formula is made by removing clauses and/or variables.
Variable elimination simplification is based on *Resolution inference rule*. Suppose two clauses $\omega_1 = \{x_1, x_i, ..., x_j\}$ and $\omega_2 = \{\neg x_1, y_i, ..., y_j\}$. The resolution inference rule allows to derive clause $\omega_3 = \{x_i, ..., x_j, y_i, ..., y_j\}$ which is called the *resolvent* as it results from resolving two clauses on the literal x_1 and $\neg x_1$. Moreover, applying variable elimination until either an empty clause is derived (unsatisfiable formula) or no more application of the resolution are possible (satisfiable formula). This is a complete algorithm to solve a SAT problem. Its major issue is to explicitly generate all resolvent and can be exponential in CNF size. Hence, the memory of computer will be limiting factor.

Subsumption is a simple principle to remove clauses. Suppose two clauses ω_1 and ω_2 such that $\omega_1 \subset \omega_2$, then ω_2 can be safely removed from the original formula. *Self Subsuming resolution* is a principle that use resolution rule and subsumption. Resolvent clause subsumes the original one. Example $\omega_1 = \{x_1, \neg x_2, x_3\}$ and $\omega_2 = \{x_1, \neg x_2, x_3, x_4\}$, then resolvent clause will be $\omega_3 = \{x_1, x_3\}$ which subsumes ω_2. This principle is presents in `SatElite` [12] preprocessor engine and used in almost all modern SAT solvers.

Other simplification techniques exists such that *Gaussian elimination* which detect sub formula in a xor-SAT form and solve it in a polynomial complexity. Moreover, this technique can also be used as inprocessing [26].

Some techniques exploits the structure of the original formula and add relevant clauses to speed up the resolution time of the SAT solver. One of them use community structure of the formula to find good clauses to add into. A preprocessor engine doing that is `modprep` [2]. Usage of symmetries also add relevant clause in the formula and will be detailed in the next chapter.

Parallel SAT solving

With the emergence of multi core architectures and increasing power of computer, one way to optimize the solving of a SAT problem is the exploitation of these cores. Effectively, SAT problems are a good candidate for parallelism. *Portfolio* is a technique that launches several SAT solver in parallel with different heuristics (decisions, restarts, ...) that communicates or not between us. When one of them found a solution or found that none exists, the overall computation is finished. Another technique to make parallel SAT solver exists and called *divide and conquer* in which the search space was divided dynamically according to positive and negative value of the decision literal. Several solvers cooperate to found solution, each of them are assigned to sub formula induced by the division. Some specific techniques like load balancing and work stealing is applied to avoid a solver to be idle. A recent framework *PaInleSS* (a Framework for Parallel SAT Solving) can be used to easily create a new parallel SAT solver with different heuristics [18] [19]. Authors of this framework win the parallel tracks of SAT competition [1] in 2018.

[1] `http://www.satcompetition.org/`

CHAPTER 3

SYMMETRY AND SAT

Despite SAT solving is an NP complete algorithm it works well on many real industrial problems. This is principally due to capacity to cut off search space with learning clause. Another ways to cut off search space is the exploitation of symmetry. Some instances exhibit symmetries and not taking them into account forces solvers to needlessly explore isomorphic search space. Hakan: symmetrie leaves object invariant

3.1 Groups basics

As symmetries is a belongs to a branch of mathematics called theory group. 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

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

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*, 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:

$$supp_g = \{x \in X \mid g.x \neq x\}$$

A variable x is *stable* by a permutation g if $x \notin supp_g$. A clause ω is *stabilized* by a permutation g if $\omega \cap supp_g = \emptyset$.

A set of permutations of a given set X form a group G, with the composition operation $(\circ)$ and is called *permutation group*. The *symmetric group* is 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 an *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 $g.x_1 = x_2$. Equivalence relations partition 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 $[x]_G = \{g.x \mid g \in G\}$

3.2 Symmetries in SAT

The previous mathematical definitions of group theory is applied to the CNF formula. The symmetric 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 assignments possibly partial of $\mathcal{V}$ as follows:

$$\forall g \in \mathfrak{S}(\mathcal{V}),\ \alpha \in Ass(\mathcal{V}),\ g.\alpha = \{g.\ell \mid \ell \in \alpha\}.$$

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 $G_(\varphi)$ and is a subgroup of $\mathfrak{S}(\mathcal{V})$. Symmetries of a formula φ preserves the satisfaction, for every *complete* assignment α:

$$\alpha \models \varphi \Leftrightarrow g.\alpha \models \varphi$$

3.3 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` [17], `bliss` [16], `nauty` [22], 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 [5]:

- *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 a clause to its literals by linking the corresponding clause node and literal nodes;
- *boolean consistency edges*: connect each pair of literals that correspond to the same variables.

$$\omega_1 = \{x_1, x_2, x_3\}$$
$$\omega_2 = \{x_4, x_5, x_6\}$$
$$\omega_3 = \{x_1, x_4\}$$
$$\omega_4 = \{x_2, x_5\}$$
$$\omega_5 = \{x_3, x_6\}$$
$$\omega_6 = \{\neg x_1, \neg x_2\}$$
$$\omega_7 = \{\neg x_1, \neg x_3\}$$
$$\omega_8 = \{\neg x_2, \neg x_3\}$$
$$\omega_9 = \{\neg x_4, \neg x_5\}$$
$$\omega_{10} = \{\neg x_4, \neg x_6\}$$
$$\omega_{11} = \{\neg x_5, \neg x_6\}$$

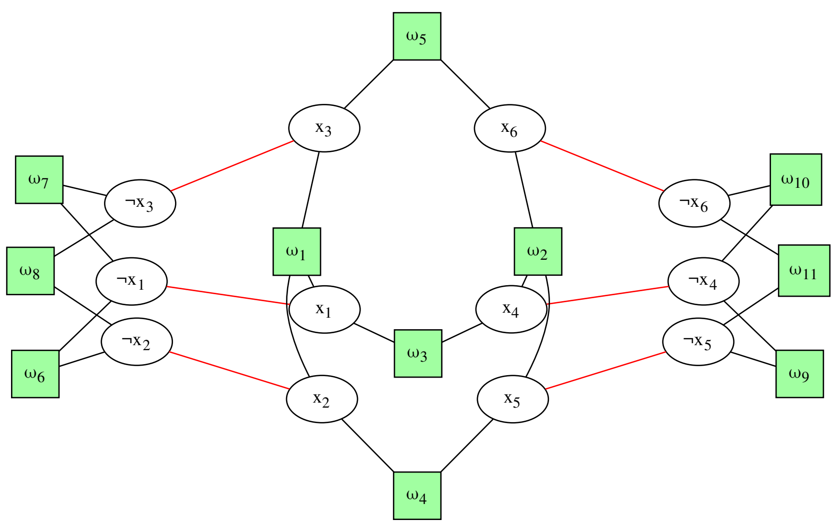


Figure 3.1: Example of constructed symmetry graph for a given CNF

Figure 3.1 shows the graph representation of a CNF. This problem have 6 variables and 11 clauses. So, the graph will have 12 + 11 = 33 vertexes where 12 represents literals vertexes (circle in the figure) and 11 represents number of clause vertexes (square in the figure). The graph will also have 6 + 24 = 30 edges, 6 for boolean consistency (Hakan: XXX color in the figure) and 24 edges that relies clauses vertexes to the literals.

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. Figure 3.2 represents the optimized version the graph.

$\omega_1 = \{x_1, x_2, x_3\}$
$\omega_2 = \{x_4, x_5, x_6\}$
$\omega_3 = \{x_1, x_4\}$
$\omega_4 = \{x_2, x_5\}$
$\omega_5 = \{x_3, x_6\}$
$\omega_6 = \{\neg x_1, \neg x_2\}$
$\omega_7 = \{\neg x_1, \neg x_3\}$
$\omega_8 = \{\neg x_2, \neg x_3\}$
$\omega_9 = \{\neg x_4, \neg x_5\}$
$\omega_{10} = \{\neg x_4, \neg x_6\}$
$\omega_{11} = \{\neg x_5, \neg x_6\}$

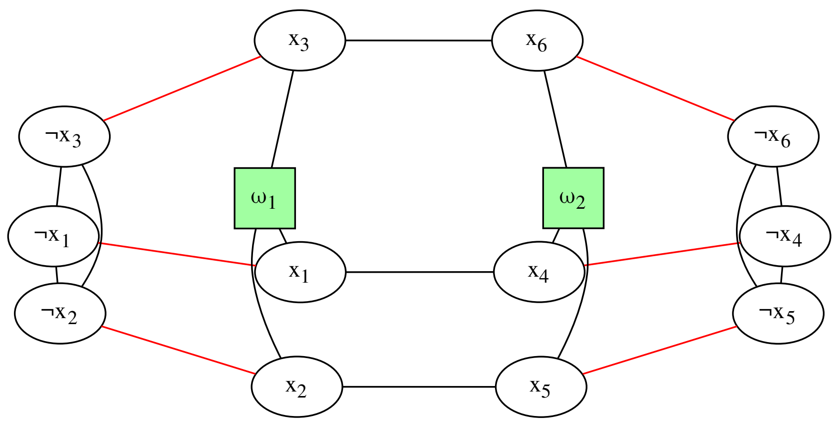


Figure 3.2: Example of constructed symmetry graph for a given CNF

After the construction of a such graph, a graph automorphism tools take it as input and gives the set of generators as output.With the previous graph, the following generators are obtained:

$$g_1 = (x_2\ \ x_3)(x_5\ \ x_6)(\neg x_2\ \ \neg x_3)(\neg x_5\ \ \neg x_6)$$
$$g_2 = (x_1\ \ x_2)(x_4\ \ x_5)(\neg x_1\ \ \neg x_2)(\neg x_4\ \ \neg x_5)$$
$$g_3 = (x_1\ \ x_4)(x_2\ \ x_5)(x_3\ \ x_6)(\neg x_1\ \ \neg x_4)(\neg x_2\ \ \neg x_5)(\neg x_3\ \ \neg x_6)$$

These permutations form a permutation group and so induce an equivalence relation. Figure 3.3 shows graphical representation of an orbit, where each node represents a literal. Two literals are linked with an arc if it exists a permutation that maps one to the other. An orbit must be a *strongly connected component* (SCC). Some permutations have a special form like two dimensional array as in this example. A further section (3.4) shows how to exploit this special form.

3.4 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.

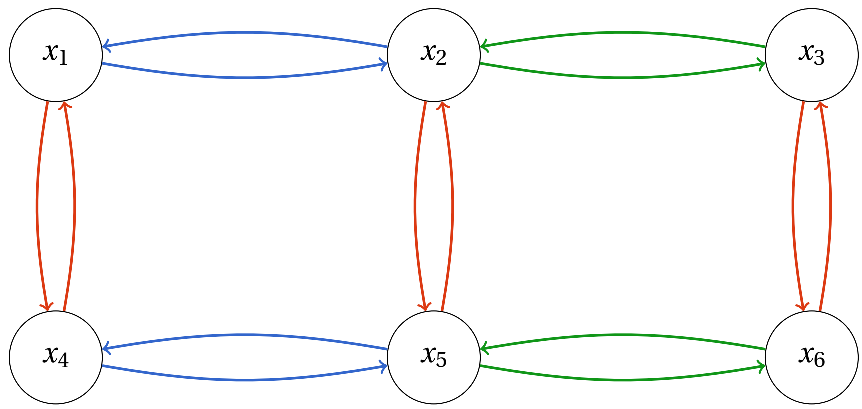


Figure 3.3: Graphical representation of an orbit

Static symmetry breaking

One way to exploit symmetry properties is to forbid equivalent assignments except one. Let 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].

In other word, the prefix of both assignment are equals according to the ordering relation $\prec$ and the next variable v has a different value, $\alpha(v) = \bot, \beta(v) = \top$, then $\alpha < \beta$. 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. □

Given a formula φ and its group of symmetry G, 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

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

symmetric SAT problem would be to explore only one assignment per orbit (for instance each lex-leader). Figure 3.4 shows different orbits, each dot in an orbit (ellipse in the figure) is an assignment, and the lex-leader is the empty red one. To avoid exploring symmetry search space, *symmetry breaking predicates* (SBP) also called *lex-leader constraints* are added to the formula. These constraints are only true for the *lex-leader* [7] and prevents other assignments from being explored.

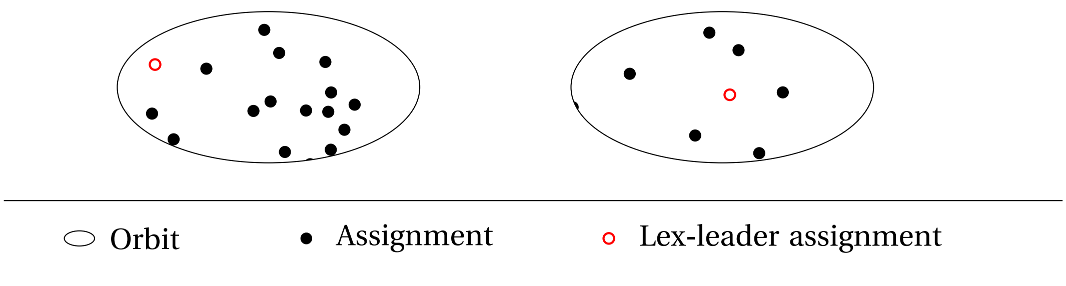


Figure 3.4: Show lex-leader per orbit

Lex-leader predicates for a permutation $g \in G_\varphi$ is defined as :

$$LL_g = \forall i : (\forall j < i : x_j = g.x_j) \Rightarrow x_i \preceq g.x_i$$

In other words, each assignment whose have a variable such that its image under g is smaller according to the ordering relation $\prec$, is pruned by LL_g. Conjunction of LL_g, for all permutation $g \in G_\varphi$ results a sound and complete symmetry breaking predicates also called *full symmetry breaking*. Only lex-leader assignment will be visited per orbit. Hence, finding the lex-leader of an orbit is computationally hard [21]. Conjunction of LL_g for some $G \subset G_\varphi$ results a symmetry breaking predicates that aims to visit at least one assignment per orbit and called is *partial symmetry breaking*. In both case, the set of symmetry breaking predicates generated is denoted as ψ.

Since a group may have a exponential number of permutations, all symmetry breaking predicates belongs to the group must be generated to ensure full symmetry breaking. These constraints will overload the solver and slow down its core (unit propagation). Hence, slow down overall time computation. Conversely, partial symmetry breaking add few constraints and bring often considerable reduction of the search space. Generally, the set of generators produced by automorphism tool is chose as subgroup. Partial symmetry breaking gives a good trade off between number of generated constraints and reduction of the search space.

Theorem 1 (Satisfiability preservation SBPs)**.** *Let φ be a formula and ψ the computed SBPs for the set of symmetries in G_φ:*

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

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 G_φ. Thus, g does not contradict β, which implies that $\beta \models \psi$. □

Generation of lex-leader constraints proposed by Crawford et al. [7] is defined as follow:

$$LL_g = \forall i : (\forall j < i : x_j = g.x_j) \Rightarrow \neg x_i \vee g.x_i$$

Figure 3.5 shows an example of generated clauses for the permutation g_3 of the previous example and an lexicographic order. Last constraint present in the figure produce tautological clause, effectively variable x_1 or x_4 are present in both polarity. The constraints of other variables produce also tautological clauses.

Order	:	$x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5 \prec x_6 \ \mid\ (\bot < \top)$
Permutation	:	$g_3 = (x_1\ \ x_4)(x_2\ \ x_5)(x_3\ \ x_6)(\neg x_1\ \ \neg x_4)(\neg x_2\ \ \neg x_5)(\neg x_3\ \ \neg x_6)$

Constraints	Generated SBP
$x_1 \preceq x_4$	$\neg x_1 \vee x_4$
$x_1 = x_4 \Rightarrow x_2 \preceq x_5$	$x_1 \vee x_4 \vee \neg x_2 \vee x_5$ $\neg x_1 \vee \neg x_4 \vee \neg x_2 \vee x_5$
$x_1 = x_4 \wedge x_2 = x_5 \Rightarrow x_3 \preceq x_6$	$x_1 \vee x_4 \vee x_2 \vee x_5 \vee \neg x_3 \vee x_6$ $\neg x_1 \vee \neg x_4 \vee x_2 \vee x_5 \vee \neg x_3 \vee x_6$ $x_1 \vee x_4 \vee \neg x_2 \vee \neg x_5 \vee \neg x_3 \vee x_6$ $\neg x_1 \vee \neg x_4 \vee \neg x_2 \vee \neg x_5 \vee \neg x_3 \vee x_6$
$x_1 = x_4 \wedge x_2 = x_5 \wedge x_3 = x_6 \Rightarrow x_4 \preceq x_1$	$x_1 \vee x_4 \vee x_2 \vee x_5 \vee x_3 \vee x_6 \vee \neg x_4 \vee x_1$ ... $\neg x_1 \vee \neg x_4 \vee x_2 \vee x_5 \vee x_3 \vee x_6 \neg x_4 \vee x_1$ ...

Figure 3.5: Example of generated SBPs for one permutation

Moreover, the number of clauses generated per constraint increase exponentially with the number of variable present in the permutation. Hence, Aloul et al [1] propose a more compact representation of symmetry breaking predicates.

Let g a permutation, let $supp_g = \{x_1, \cdots, x_n\}$ the support of the permutation g be ordered such that $x_i \preceq x_j$ iff $i \leq j$ and let $\{y_0, \cdots, y_n\}$ be a set of auxiliary variables disjoint from $supp_g$. These auxiliary variables encode equality of literals in such y_0 is set as an unit clause and encode the first equality. Following clauses encode a compact lex-leader for a permutation:

$\neg y_i \vee \neg x_{i-1} \vee \neg x_i \vee g.x_i$	$1 \le i \le n$	$\neg y_i \vee \neg x_{i-1} \vee \neg y_{i+1}$	$1 \le i \le n$
$\neg y_i \vee g.x_{i-1} \vee \neg x_i \vee g.x_i$	$1 \le i \le n$	$\neg y_i \vee g.x_{i-1} \vee \neg y_{i+1}$	$1 \le i \le n$

Figure 3.6 shows the compact encoding of generated clauses. This form grow linearly with the number of variables. Auxiliary variable encodes the equality of two literals allows to achieve this reduction. Three auxiliary variables are introduced in this example x_7, x_8, x_9 such that x_7 encode the equality of x_1 and x_4, x_8 equality of x_2 and x_5, and x_9 equality of x_3 and x_6.

Order : $x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5 \prec x_6 \mid (\bot < \top)$

Permutation : $g_3 = (x_1 \;\; x_4)(x_2 \;\; x_5)(x_3 \;\; x_6)(\neg x_1 \;\; \neg x_4)(\neg x_2 \;\; \neg x_5)(\neg x_3 \;\; \neg x_6)$

Constraints	Generated SBP
$x_1 \preceq x_4$	$\neg x_1 \vee x_4$
	x_7
$x_1 = x_4 \Rightarrow x_2 \preceq x_5$	$\neg x_7 \vee \neg x_1 \vee \neg x_2 \vee x_5$
	$\neg x_7 \vee \neg x_1 \vee x_8$
	$\neg x_7 \vee x_4 \vee \neg x_2 \vee x_5$
	$\neg x_7 \vee x_4 \vee x_8$
$x_1 = x_4 \wedge x_2 = x_5 \Rightarrow x_3 \preceq x_6$	$\neg x_8 \vee \neg x_2 \vee \neg x_3 \vee x_6$
	$\neg x_8 \vee \neg x_2 \vee x_9$
	$\neg x_8 \vee x_5 \vee \neg x_3 \vee x_6$
	$\neg x_8 \vee x_5 \vee \neg x_9$

Figure 3.6: Example of compact generated SBPs for one permutation

`Shatter` [1] is a tool fot partial symmetry breaking that computes symmetry with `saucy3` automorphism tool and generate a new formula with compact lex-leader encoding. It uses only generators given by the automorphism tool. Following table shows the number of symmetry breaking predicates clauses and the number of auxiliary variables added to the original formula.

battleship-12-12-unsat	936	144	1498	378
battleship-12-23-sat	1662	276	5464	1375
battleship-14-26-sat	2562	364	3688	929
battleship-14-27-sat	2653	378	7222	1814
battleship-16-16-unsat	2176	256	4388	1102
battleship-16-31-sat	3976	496	12094	3035
battleship-24-57-sat	16308	1368	40372	10113
chnl10_11	1122	220	2416	615
chnl10_12	1344	240	2736	696
chnl10_13	1586	260	3252	826
chnl11_12	1476	264	3204	813
chnl11_13	1742	286	3636	922
chnl11_20	4220	440	6760	1710
fpga10_15_uns_rcr	2130	300	4580	1160
fpga10_20_uns_rcr	3840	400	6768	1712
fpga11_12_uns_rcr	1476	264	3704	938
fpga11_13_uns_rcr	1742	286	4076	1032
fpga11_14_uns_rcr	2030	308	4740	1199
fpga11_15_uns_rcr	2340	330	5196	1314
fpga11_20_uns_rcr	4220	440	7864	1986
hole010	561	110	1054	269
hole015	1816	240	3280	828
hole020	4221	420	6478	1630
hole030	13981	930	21322	5346
hole040	32841	1640	44934	11254
hole050	63801	2550	81682	20446
Urq6_5	1756	180	109	0
Urq7_5	2194	240	143	0
Urq8_5	3252	327	200	0
x1_40	314	118	42	1
x1_80	634	238	80	0

An improvement of static symmetry breaking was made by Devriendt et al [10] with a tool called `BreakID`. It exploits some properties from the structure of generators. On some circumstance a linear number of constraints can break all group. The other tries to add a maximum of binary clauses that is useful because it can participate often to unit propagation and so to the conflict analysis.

Special form of the group

Some formula presents specific type of symmetry called *row (column) interchangeability*, when a subset of variables are structured as a two dimensional matrix. Each row (column) are interchangeable with the symmetries. This form of symmetry is common in different kind of problem like pigeon hole problem in which pigeons and holes are interchangeable or in delivery system in which trucks of a fleet are interchangeable. Usage of row (column) interchangeability can significantly improved SAT performance. Effectively symmetries can be eliminated by the addition of only a linear number of symmetry-breaking constraints [13]. One condition must be satisfied to ensure this linear number of constraints: lexicographic order needs to respect the structure of the matrix. In practice, automorphism tools gives only the set generators which contains no information on the structure of the group. Authors of `BreakID` [10] develop an algorithm to detect this specific structure and exploit it.

Binary leax-leader constraints

`BreakID` has another approach that aims to post many lex-leader constraints. The first constraint of symmetry breaking predicates must produce a binary clause. Building many binary clauses is possible without enumerating the whole symmetry group. It suffices to compute the orbit of the smallest variable according to the ordering relation. As the orbit can be seen as a strong connected component, it must exist a permutation that permutes the smallest variable with all other variables in the same orbit. Then, as many binary clauses as variables (without the smallest variable) in the orbit can be added to the formula. Constructing a sequence of subgroups that stabilize the smallest variable (i.e. not have the smallest variable in its support) results to new binary clauses. This sequence ends when trivial subgroup is reached and is called a *stabilizer chain*.

Figure 3.7 shows application of the generation of binary clauses. In the example, considered group has three permutations and its graphical representation is show. Given the lexicographic order, the smallest variable is x_1 and all other variables are in its orbits. According to the ordering relation, five symmetry breaking predicates are generated with the formula $\neg x_1 \vee g.x_1$. Then, subgroup that stabilize x_1 is computed and it remains only one permutation g_2. As it smallest variable is x_2, the constraint $\neg x_2 \vee x_3$ is generated. Stabilizer chain leads to trivial group and no more binary clauses are generated.

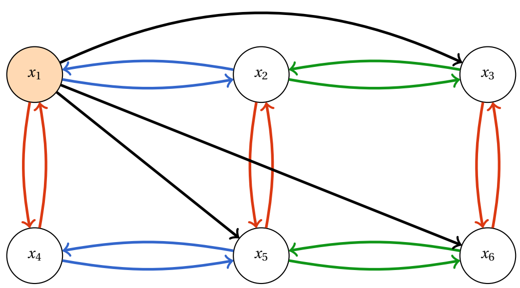
In total, six binary clauses is generated without adding any auxiliary variables. Moreover, a property can be observed, when the smallest variable have the greatest value ($\top$in this case), all variables in the orbits must have the same value.

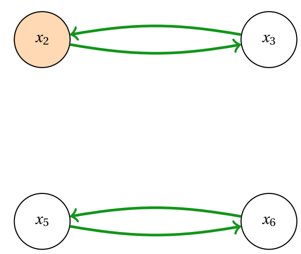
The size of the stabilizer chain is heavily dependent of the chose lexicographic order. More stabilizer discard permutations and more trivial subgroup is reached quickly and less binary clauses are generated. An incremental order is proposed to optimize number of generates binary clauses. First, orbit of all variables is computed and the variable with fewest number of occurrence is chose among the biggest orbit. The idea is that biggest orbit produce more clauses and the variable appearing in few permutations reduce the number of stabilized permutation. This procedure is applied until trivial group is reached. At the end, remaining variables are added to the order.

$$\text{Order} \quad : \quad x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5 \prec x_6 \quad | \quad (\bot < \top)$$

$g_1 = (x_2 \; x_3)(x_5 \; x_6)(\neg x_2 \; \neg x_3)(\neg x_5 \; \neg x_6)$
$g_2 = (x_1 \; x_2)(x_4 \; x_5)(\neg x_1 \; \neg x_2)(\neg x_4 \; \neg x_5)$
$g_3 = (x_1 \; x_4)(x_2 \; x_5)(x_3 \; x_6)(\neg x_1 \; \neg x_4)(\neg x_2 \; \neg x_5)(\neg x_3 \; \neg x_6)$

$g_1 = (x_2 \; x_3)(x_5 \; x_6)(\neg x_2 \; \neg x_3)(\neg x_5 \; \neg x_6)$





$\omega_1 = \{\neg x_1, x_2\}$ $\omega_4 = \{\neg x_1, x_5\}$

$\omega_2 = \{\neg x_1, x_3\}$ $\omega_5 = \{\neg x_1, x_6\}$ $\omega_6 = \{\neg x_2, x_3\}$

$\omega_3 = \{\neg x_1, x_4\}$

Figure 3.7: Caption

Evaluation of performance

Hakan: Faire une section pour ça An extremely important point is the chosen lexicographic order. Variable ordering may impact the number of generated constraint and so the performance of the underlying SAT solver. Different orders are studied in the literature. One of the simplest order was the sorted variables according to their numbers. Some others orders exists and exploit structural properties of the problem. In particular, the orbits of the variables in different ways. For example, the variables are chosen with their number of occurrences in the initial problem. This order, is equivalent to put largest orbit first and so on, because each variable on the same orbit must have the same number of occurrences. Another example of exploiting structural property of the orbit is the usage of *stabilizer*. The order choose a variable which maximize the number of stabilized permutations, removes the not stabilized ones and loop over until the empty set. The remaining variables are added to the order to get a complete order.

Conclusion

Static symmetry breaking acts as a preprocessor that augment the initial formula with symmetry breaking predicates. These constraints avoid exploration of symmetric search space. In the general case, the number of these clauses are often too large to be effectively handled by a SAT solver [21]. On the other hand, if only a subset of the symmetries is considered then the resulting search pruning will not be perfect and its effectiveness depends heavily on the heuristically chosen symmetries [5]. Hakan: Parler des optim de BReakID Despite the good reduction of the search space, some formula still intractable for a state of the art SAT solver.

Dynamic symmetry breaking

Another disadvantage is that the solver is influenced by SBPs and explore the search space with a different manner and can affect performance negatively Hakan: FIND A REF

In the literature, different approach of dynamic symmetry breaking are used to reduce the search space of the sat solver in different ways. Some of them *inject* constraints to allow only one representative assignment of each orbit like the static approach and others accelerate the propagation of variable using symmetrical properties. In this section, we present different approach to use symmetrical properties of a problem dynamically.

SymChaff

One of the first dynamic symmetry breaking approach is *SymChaff* [25] and is applicable only on special groups where all couple of variables are symmetric. The idea of this approach is to treat each orbit like a *symbolic variable*, i.e. instead of considering a single variable, all symmetric variables are considered at the same time and so backtracked at the same time. In this special case of groups the number of orbits is easy to compute but the order in which they would be applied has a tremendously impact of the solver performance. In the general case, when we take any groups computing the number of orbit will be very difficult and this approach will be intractable.

Symmetry Propagation

A different approach can be used to reduce search space using symmetries is *symmetry propagation* [11]. The general idea of this approach is to propagate symmetrical literals of those already propagated. In other words, it accelerate the tree traversal by “transforming some guessing (decisions) to deductions (propagation)”. Indeed, problem that presents symmetries makes possible to deduce some value for the variables that would be guessed if symmetry properties were ignored. These deductions will reduce the overall tree traversal depth and hence eventually accelerate the solving process.

To explain this approach, some definitions are required.

Hakan: Mettre logical conqequence quand on explique CDCL + Expliquer CDCL plus en detail avec les learnt et les raisons

Definition 2 (Logical consequence)**.** *A formula ϕ is a logical consequence of a formula φ denoted by $\varphi \models \phi$ if for all assignment α satisfying φ, it satisfies also ϕ. Two formulas are* logically equivalent *if each is a logical consequence of the other.*

Proposition 2 (Symmetry propagation)**.** *Let φ be a formula, α an assignment and l a literal. If g is a symmetry (permutation) of $\varphi \cup \alpha$ and $\varphi \models \{l\}$, then $\varphi \cup \alpha \models g.\{l\}$ is also true.*

In other words, if a literal l was propagated by the solver and g is a *valid* symmetry for the sub problem $\varphi \cup \alpha$ (in which all satisfied clauses and false literals are removed), so , the solver can also propagate the symmetrical of l. The problem here is to determinate which symmetries are valid for the formula $\varphi \cup \alpha$.

Definition 3 (Active symmetry)**.** *A symmetry g is called active under a partial assignment α if $g.\alpha = \alpha$*

The Definition 3 leads to the following proposition:

Proposition 3. *Let φ a formula and α a partial assignment. Let g a symmetry of φ, if g is active under the assignment α, then g is also a symmetry of $\varphi \cup \alpha$*

The previous proposition states that an active symmetry g for a partial assignment α still valid for the formula $\varphi \cup \alpha$. So when a literal l is propagated, and a symmetry g is active for a partial assignment α, the solver can also propagate $g.l$. Moreover, the group theory allow to compose permutations with the composition operator $\circ$ and the composition of two active symmetries is also an active symmetries so the solver can also propagate $g^2.l, g^3.l, \ldots$

Hakan: Peut etre expliquer les sym conflicts

The authors of symmetry propagation improve the active symmetries, introducing *weakly active* symmetries.

Definition 4 (Weakly active symmetry)**.** *Let φ a formula and (δ, α, γ) a state of a CDCL solver in which δ is the set of decisions, α is the current assignment and γ the reasons of the learned clauses. Then a symmetry g is weakly active if $g.\delta \subseteq \alpha$*

This definition leads to the following proposition:

Proposition 4. *Let φ be a formula, α an assignment. If there exists a subset $\delta \subseteq \alpha$ and a symmetry g of φ such that $g.\delta \subseteq \alpha$ and $\varphi \cup \delta \models \varphi \cup \alpha$, then g is also a symmetry of $\varphi \cup \alpha$.*

Hakan: Proof

In other words, we can detect with a minimal effort, the symmetries of $\varphi \cup \alpha$ by keeping track of the set of variables δ, which are in a state-of-the-art complete SAT solving algorithms, the set of decision variables. Obviously, a weakly active symmetry can also propagate the symmetrical literals of a propagated one. Moreover, weakly active symmetries allows more propagation and so is more efficient. Note that if a weakly active symmetry want to propagate a symmetrical literal which are already affected to the opposite value, this leads to a symmetry conflict and the solver backtracks to propagate the symmetrical value correctly.

Hakan: Mettre des tableaux, courbes etc ... Hakan: Courbe VS static an no sbp
Hakan: Conclu SP, depend on the solver choice

Symmetry propagation gives good performances on many symmetric instances. The overall performance of the symmetry propagation is intrinsically related to the decision heuristics of the underlying SAT solv1er.

One optimization of symmetry propagation is the following proposition, as seen in Section Hakan: DETERMINE SECTION SAT LEARNING each propagated clause has a reason which is an assertive clause. If the symmetrical clause is also an assertive one, this clause can be added in the formula without any requirements (even if the permutation is not weakly active). The added symmetrical clause will participate also to unit propagation and propagate the symmetrical literal.

Symmetry Explanation Learning

Another approach to exploit symmetry without removing any satisfiable assignment of the problem is *Symmetry Explanation Learning* [9]. This approach aim to learn useful symmetrical variant of clauses only where they are used by the solver. A useful clause is a clause that participate to the unit propagation or conflict. A naive approach to ensure that is to compute the symmetrical clause of each learned clauses and check if it is useful. In addition, due to size of a group a clause can have many symmetrical ones. the naive approach will have an important overhead and will be intractable on reel problems. SEL have different optimization based on the following facts. First, On the unit propagation, propagated literals has a reason clause which are assertive, and in the general case, symmetries permutes only few literals of the clause so symmetrical clauses can also be assertive and so useful. Secondly, avoiding to add identical clauses to the problem, symmetrical clauses ares stored in different learning scheme and used when classical unit propagation is finished. This ensure that no duplicate clause is added in the problem.

CHAPTER 4

SYMMSAT

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 [21]. 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 [5]. Besides, these approaches are preprocessors, so their combination with other techniques, such as *symmetry propagation* [11], 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. 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*. It 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 and *effective symmetric breaking predicates* (*esbp*).

Definition 5 (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 5 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 5. *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 6 (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:*

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

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

Definition 7 (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 5. 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 6. *$\eta(\alpha, g)$ 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 2 (Satisfiability preservation)**.** *Let φ be a formula and ψ an esp for some assignment α under $g \in S(\varphi)$. Then,*

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

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 $\beta \models \psi$.

□

Algorithm

This section describes how to augment the state-of-the-art CDCL algorithm with the aforementioned concepts to develop an efficient symmetry-guided SAT solving algorithm. 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. We first recall how the CDCL algorithm works. We then explain how to extend it with a *symmetry controller* component which guides the behavior of CDCL algorithm depending on the status of symmetries.

Conflict-Driven Clause Learning (CDCL) algorithm was already presented in Algorithm 3

epicted in Algorithm **??** (in black) is the same Algorithm . The parts in red (grey in B&W printings) should be ignored for the moment.

Hakan: Change Font !!

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 [11]. 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.

Experiments

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

```
function CDCLSym (φ: CNF formula, SymController: symmetry controller)
  returns ⊤ if φ is SAT and ⊥ otherwise
    dl ← 0                                            // Current decision level
    while not all variables are assigned do
        isConflict ← unitPropagation()
        SymController.updateAssign(currentAssignment())
        isReduced ←
         SymController.isNotLexLeader(currentAssignment())
        if isConflict || isReduced then
            if dl == 0 then
                return ⊥                                      // φ is UNSAT
            if isConflict then
                ω ← analyzeConflict()
            else
                ω ← SymController.generateEsbp(currentAssignment())
            addLearntClause(ω)
            dl ← backjumpAndRestartPolicies()
            SymController.updateCancel(currentAssignment())
        else
            assignDecisionLiteral()
            dl ← dl + 1
    return ⊤                                                  // φ is SAT
```

Algorithm 4: the CDCLSym SAT Solving Algorithm.

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. Ansótegui, J. Giráldez-Cru, J. Levy, and L. Simon. Using community structure to detect relevant learnt clauses. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 238–254. Springer, 2015.

[3] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

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

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

[6] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[7] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. *KR*, 96:148–159, 1996.

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

[9] J. Devriendt, B. Bogaerts, and M. Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 83–100. Springer, 2017.

[10] J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker. Improved static symmetry breaking for sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 104–122. Springer, 2016.

[11] 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.

[12] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *International conference on theory and applications of satisfiability testing*, pages 61–75. Springer, 2005.

[13] P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *International Conference on Principles and Practice of Constraint Programming*, pages 462–477. Springer, 2002.

[14] C. P. Gomes, B. Selman, and N. Crato. Heavy-tailed distributions in combinatorial search. In *International Conference on Principles and Practice of Constraint Programming*, pages 121–135. Springer, 1997.

[15] J. Huang et al. The effect of restarts on the efficiency of clause learning. In *IJCAI*, volume 7, pages 2318–2323, 2007.

[16] 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.

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

[18] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon. Painless: a framework for parallel sat solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 233–250. Springer, 2017.

[19] L. Le Frioux, S. Baarir, J. Sopena, and F. Kordon. Modular and efficient divide-and-conquer sat solver on top of the painless framework. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 135–151. Springer, 2019.

[20] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki. Learning rate based branching heuristic for sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140. Springer, 2016.

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

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

[23] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

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

[25] A. Sabharwal. Symchaff: A structure-aware satisfiability solver. In *AAAI*, volume 5, pages 467–474, 2005.

[26] M. Soos. Enhanced gaussian elimination in dpll-based sat solvers. In *POS@ SAT*, pages 2–14, 2010.

[27] N. Sörensson and A. Biere. Minimizing learned clauses. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 237–243. Springer, 2009.

[28] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.