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

Part I

State-of-the-art

CHAPTER 1

INTRODUCTION

Nowadays, computers are powerful and used in many applications in different domains. One of these domains is critical application that run in planes, cars, trains, etc. Software present in these machines must be correct and exempt of bugs. Proving the correctness of these software is a difficult problem.

can leads to combinatorial explosion. Over the years, computer scientists have developed many techniques to solve these kinds of problems like *constraint programming* (CP) [30], *Propositional Satisfiability* (SAT) [6], *Satisfiability Modulo Theory* (SMT) [4]. In this thesis, we focus on solving propositional or Boolean formula, it consists of deciding whether all constraints of a formula can be satisfiable and answer SAT, or it is not possible to satisfy all constraints and answer UNSAT.This problem may appear to be simple but cannot be handled efficiently at the moment. This is due to the complexity of the problem which is proved NP-complete in 1971. Many industrial applications problems can be transformed into a SAT problem. Improving the performance of tools that resolve this problem is an important challenge.

Over the last decades, SAT solver can handle more and more complicated problems in different domains: like *formal methods*: hardware model checking, software model checking, etc.; *artificial intelligence*: planning [21]; *game resolution*: sudoku, n-queens, *Bioinformatics* : Haplotype inference, *design automation* : equivalence checking. A recent work solve the Pythagorean triple, an old mathematical problem which has been resolved with a SAT solver and produce a huge proof of 200 TB.

This success comes from the introduction of sophisticated heuristics and optimization of the solving algorithm called Conflict Driven Clause Learning (CDCL) algorithm. It is based on the first non memory intensive algorithm named by its authors Davis, Putnam, Logemann, and Loveland (DPLL). Unfortunately, some problems still intractable for state-of-the-art SAT solvers. But some of them exhibits symmetries that can be exploited by the solver to accelerate the overall solving time. At its most basic, symmetry is some transformations of an object that leaves it unchanged. Symmetries is common in real life, if we take some butterfly, it has exactly the same halves. In the case of satisfiability problems, it maps a solution of a problem to another. Ignoring these properties forces the solver to explore equivalent search space and it is a loss of time and energy. Considering the butterfly example if we search a pattern and it is not present, it is completely absurd to verify the other side.

In the literature, some works exist and tackle the symmetry problems. But, the first step to exploit symmetries is to find them. For this purpose, it exists technique that use graph isomorphism. When symmetries are found, the most common approach to exploit it is *static symmetry breaking*. It takes the symmetric problem as input and produces a satisfiability equivalent formula without symmetries. This transformation is made without any change of existing solvers. This approach works well on many applications but stuck on some highly symmetrical ones.

Another approach to handle symmetry is *dynamic symmetry breaking*, it is included in the solver. It observes his behavior and use symmetry properties to avoid visiting symmetric search space. These approaches will be clearly explained all along this thesis.

1.1 Contributions

Understanding state of the art techniques in symmetry breaking allow us to improve it. 2 majors contributions are detailed in this thesis. The first one use the force of static symmetry breaking and apply it dynamically to avoid its drawback. It add an opportunistic symmetry controller that avoid visiting symmetric search space. This idea allow us to solve very hard symmetric problems. The second contribution use the previous one and combines it with state of the art dynamic symmetry breaking approach and take the best of 2 worlds. This combination leads to important theoretical step for the usage of *partial symmetry breaking* with the usage of *local symmetries*.

1.2 Structure of the manuscript

CHAPTER 2

THE BOOLEAN SATISFIABILITY PROBLEM

Contents

In this thesis, our goal is to exploit the symmetrical properties of SAT problems. Before, we get to the heart of the matter, we first introduce the Boolean satisfiability (SAT) problem. This problem is a propositional formula representing the constraints of a system. A tool that aims to satisfy such formulas is called a SAT solver.

2.1 SAT basics

The satisfiability problem is constituted of *Boolean* or *propositional variable*. It has two possible values, true or false (noted respectively $\top$ or $\bot$). We call *literal*, a propositional variable or its negation. For a given variable x, the positive literal is represented by x and the negative one by $\neg x$. Given a formula φ, we denote $\mathcal{V}_\varphi$ ($\mathcal{L}_\varphi$) the set of variables (literals) used in the formula (the index in $\mathcal{V}_\varphi$ and $\mathcal{L}_\varphi$ is usually omitted when clear from context). To build more complex formula, it exists different operators, $\neg, \vee$ and $\wedge$ that are respectively negation, disjunction and conjunction. Others operators like $\Rightarrow, \Leftrightarrow$ and $\oplus, \cdots$ respectively called implication, equivalence and exclusive disjunction (xor), $\cdots$ can be expressed with the basic ones. For example $a \Rightarrow b$ can be expressed as $\neg a \vee b$. To ensure the priority of these operators, constraints are expressed between parenthesis. In the absence of its, the following priority order applies (from the highest to the lowest priority): negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$).

The value given to each variable of a formula is called an *assignment* noted α and defined as follows:

$$\alpha : \mathcal{V} \mapsto \{\top, \bot\}$$

As usual α is said *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. For example, $\alpha = \{\neg x_1, x_3\}$ means that x_1 have the false value and x_3 have the true value. The set of all (possibly partial) assignments of $\mathcal{V}$ is noted $Ass(\mathcal{V})$. A *truth table* gives an evaluation of all possible assignments for a given formula. Table 2.1 shows the evaluation of negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$) operators. For convenience, true value ($\top$) is also represented by 1 and false value ($\bot$) is represented by 0. When a formula is always true independently from the assignment, it is called a *tautology* $x \vee \neg x$ is an example of tautologous formula.

x	y	$\neg x$	$x \vee y$	$x \wedge y$
0	0	1	0	0
0	1	1	1	0
1	0	0	1	0
1	1	0	1	1

Table 2.1: Truth table of basic operators

Normal forms

In Boolean logic, it exists some structural properties, called *normal form*. To present them, we first need to introduce *clause* and *cube* concepts. A *clause* ω is a finite disjunction of literals:

$$\omega = \bigvee_{i=1}^{k} l_i \text{ , or the set of its literals } \omega = \{l_i\}_{i \in [\![1,k]\!]}$$

A *cube* γ is a finite conjunction of literals represented by:

$$\gamma = \bigwedge_{i=1}^{k} l_i$$

With respect to its size, a clause is said to be *unary, binary, ternary, n-ary* if it contains respectively one, two, three, or n literals.

The clause form has a property called *subsumption*. When a clause ω_1 is a subset of another clause ω_2 noted $\omega_1 \subset \omega_2$. And any assignment that satisfies ω_1 also satisfies ω_2, so ω_2 is *redundant* towards ω_1 and so can be removed of the formula.

Conjunctive Normal Form (CNF) of a formula is a finite conjunction of clauses:

$$\varphi = \bigwedge_{i=1}^{k} \omega_i$$

Disjunctive normal form (DNF) of a formula is finite disjunction of cubes:

$$\varphi = \bigvee_{i=1}^{k} \gamma_i$$

The following table is a summary of laws about Boolean formulas that allow to transform any formula to a normal form.

associativity of $\vee$	$(x \vee y) \vee z \equiv x \vee (y \vee z)$
commutativity of $\vee$	$x \vee y \equiv y \vee x$
identity	$x \vee \bot \equiv x$
domination	$x \vee \top \equiv \top$
idempotent	$x \vee x \equiv x$
distribution over $\wedge$	$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$
associativity of $\wedge$	$(x \wedge y) \wedge z \equiv x \wedge (y \wedge z)$
commutativity of $\wedge$	$x \wedge y \equiv y \wedge x$
identity	$x \wedge \top \equiv x$
domination	$x \wedge \bot \equiv \bot$
idempotent	$x \wedge x \equiv x$
distribution over $\vee$	$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$
negation 1	$x \vee \neg x \equiv \bot$
negation 2	$x \wedge \neg x \equiv \top$
double negation	$\neg(\neg x) \equiv x$
De Morgan 1	$\neg x \vee \neg y \equiv \neg(x \wedge y)$
De Morgan 2	$\neg x \wedge \neg y \equiv \neg(x \vee y)$

Table 2.2: Set of laws of operators

Every formula can be transformed into a normal form with different complexity and the resulting formula is *equivalent.* In other words, every assignment α that satisfies formula φ also models the resulting formula ψ and vice-versa, denoted by $\varphi \equiv \psi$. Also, we say a formula ψ is a *logical consequence* of a formula φ is every model of φ is also a model of ψ and is denoted by $\varphi \models \psi$.

Conjunctive normal form is the input form of state-of-the-art solvers. Any propositional formula can be transformed in CNF form with a polynomial complexity. Conversely, DNF form have an exponential memory complexity during the transformation. Note that each cube in the problem in DNF form is a solution in the equivalent CNF formulas. Hakan: Sharp SAT = transform CNF to DNF

An NP-complete problem

The SAT problem is the first NP-complete algorithm proven by Stephen Cook in 1971 [7]. NP-completeness means that a SAT problem can be solved with a non-deterministic Turing 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 problems in theoretical computer science is the P versus NP problem. This question is one of the seven millennium prize problems.

Particular forms easy to solve

In some particular forms, SAT problems can be computed with polynomial algorithm.

2-SAT [3].l In this particular form, the given CNF formula contains only binary clauses. In this case, it suffices to create a graph in which each clause is transformed into implication. For example, the clause $x \vee y$ will be transformed into $\neg x \Rightarrow y, \neg y \Rightarrow x$. After computing *strong connected component* on this graph, to be looking for the positive and negative forms of the same variable, in the same strong connected component suffices to determine the satisfiability of the formula. If it is the case, the formula is UNSAT. Otherwise a solution can be deduce and the problem is SAT. This algorithm can be computed in linear time complexity.

Horn SAT. In this particular form, the given CNF formula contains only Horn clauses. It exists three form of Horn clause: *strict Horn clause* that contains only one positive literal and at least one negative literal, *positive Horn clause* that contains only one positive literal and no negative literals *negative Horn clause* that contains only negative literals. To solve this particular form of formula, it suffices to apply *Boolean constraint propagation* (BCP) or *unit propagation* explained thereafter in section 2.1 until fix point. Roughly speaking, It satisfies all unit clauses in cascade. Either an empty clause was deduced and the problem is UNSATor the fix point is reached and the formula is SAT. Like 2-SAT, this algorithm can also be computed in linear time complexity.

XOR SAT. In this particular form, each clause contains xor ($\oplus$) operator rather than or ($\vee$). This problem can be seen as a system of linear equations. Gaussian elimination allows to solve this kind of problem is polynomial time.

The membership of polynomial class (P) of the particular form of satisfiability problems describes above is a special case of Schaefer's dichotomy theorem [32].

Some related problems

Different kind of problems related to SAT is presented is this section. One of them is sharp-SAT (#SAT), its purpose is to count the number of solutions in a CNF.

Another related problem is maximum satisfiability problem (MAX-SAT). In this case, the problem is to find the maximum subset of clauses that can be satisfied for a formula. Different variants of this problem exist. For example, some constraints must be satisfied (hard clauses) and MAX-SAT is applied on the remaining clauses called *soft* clauses.

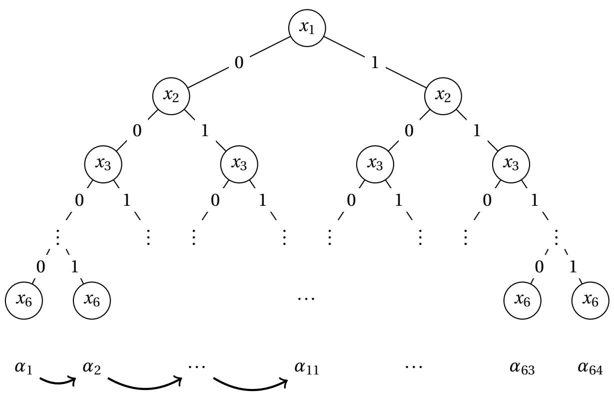
The last related problem is quantified Boolean formula (QBF) where the quantifiers $\exists$ and $\forall$ are present in the formula. For example, $\forall x \exists y \exists z (x \vee y) \wedge z$. This particular form is a generalization of the SAT problem with PSPACE complexity.

Solving a SAT problem

Two kinds of algorithms exist to solve satisfiability problems. First, the *incomplete* algorithm [6] which does not provide any guarantee that will eventually report either any satisfiable assignment or declare the formula 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. For a propositional formula with n variables, it leads to 2^n assignments in the worse case. Figure 2.1 illustrates the search tree for a given problem with six variables. The presented formula in the figure have 6 clauses, with 2 ternary clauses and 4 binary clauses. It will be used as example in different algorithm presented below. This formula is SAT, $\alpha_{11} = \{\neg x_1, \neg x_2, x_3, \neg x_4, x_5, \neg x_6\}$ is a solution of the problem. This naive algorithm will check 10 assignments before finding the solution. In the general case, due to the number of variable in problems, this algorithm is intractable.



$\omega_1 = \{x_1, x_2, x_3\}$

$\omega_2 = \{x_4, x_5, x_6\}$

$\omega_3 = \{\neg x_1, \neg x_5\}$

$\omega_4 = \{\neg x_2, \neg x_4\}$

$\omega_5 = \{\neg x_3, \neg x_4\}$

$\omega_6 = \{\neg x_3, \neg x_6\}$

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

Davis Putnam Logemann Loveland algorithm (DPLL)

One of the first non-memory-intensive algorithm developed to solve SAT problems is the Davis Putnam Logemann Loveland algorithm (DPLL) [9]. It explores a binary tree using depth first search as shown in Algorithm 1. The construction of the tree relies on the *decision* made (on line 8) then, each possible value are checked in recursive call, true value on line 9 and false value on line 11. When a leaf find a conflict (line 5) which means that the formula cannot be satisfied with the current assignment, other branches are explored. By recursive construction of the algorithm, when each value of a literal reach to a conflict, solver *backtracks* at most one level, this fact is called *chronological backtracking*. When the conflict occurs at the top of the tree, it means that the formula cannot be satisfied and the solver reports UNSAT (line 13). However, if the formula is empty in any branch, it means that current assignment satisfy the whole formula and the solver reports it on line 10 or 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 ⊥;                                    // Conflict
6      if φ = {} then
7          return α ;                                   // φ is SAT
8      x ← assignDecisionLiteral();
9      if α ← DPLL(φ ∪ {x}, α) then
10         return α
11     if α ← DPLL(φ ∪ {¬x}, α) then
12         return α
13     return UNSAT;                                    // φ is UNSAT
```

Algorithm 1: The DPLL algorithm.

An important function in the DPLL algorithm is `unitPropagation` (line 3). It is presented in Algorithm 2. This function set value to unit clause in order to satisfy them until fix point is reached. Either, they are no more unit clause in the formula or an inconsistency is found which means that current assignment cannot satisfy the formula. In the later case, the solver will backtrack and another branch will be explored.

When DPLL algorithm is executed on the formula of Figure 2.1, after making decisions on literals $\neg x_1$ and $\neg x_2$, unit propagation detects that x_3 must be assigned to true. This propagation prevents to explore multiple assignments. Effectively, when x_3 is set to false value, the clause ω_1 is not satisfied and it remains 3 variables and so 2^3 possible assignments (from α_1 to α_8). Moreover, in the DPLL algorithm, application of unit propagation until the fix point is reached leads to a solution. `assignDecisionLiteral` is an important procedure that responsible of choosing the variable that divides the search space. Its objective is to find a literal that will generate a maximum of unit propagation. Intuitively, decision

```
function unitPropagation(φ: CNF formula, α assignment)
    returns CNF formula and assignment α
    while {l} ∈ φ and {} ∉ φ do
        // Remove all clauses containing l, all literals ¬l
        φ ← φ|_l
        α ← α ∪ {l}
    return φ, α
```

Algorithm 2: Unit propagation

literals can be viewed as '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 the section 16.

Conflict Driven Clause Learning (CDCL) algorithm

The principal weakness of DPLL algorithm is to make the same inconsistencies several times (principally due to chronological backtracking), leading to unnecessary CPU usage. Conflict Driven Clause Learning (CDCL) algorithm 3 is a sound and complete algorithm that overcomes principal the weakness of DPLL.

Algorithm 3 gives an overview of CDCL, Like DPLL, it walks on a binary search tree. Initially, the assignment is empty and the decision level that indicates the depth of the search tree, noted by dl, is set to zero. The algorithm first applies unit propagation to the formula φ for the assignment α (line 6). 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, its reason is analyzed and a clause called *conflict clause* is deduced (line 10). The work done in this procedure will be explained thereafter. This clause is *learnt* (line 12) (added to the formula). This clause is redundant w.r.t the current formula and so it does not change the satisfiability of φ. It also avoids encountering a conflict with the same causes in the future. The analysis is completed by the computation of a *backjumps level* , the assignment and decision level is updated (line 11). As the level can be much lower than the level of 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 $\bot$values. Figure 2.2 shows an assignments that leads to a conflict. First the solver choses $\neg x_1$ as a 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. To satisfy ω_1, x_3 needs to be set to $\top$ and to satisfy ω_5, needs to be $\bot$. As a variable cannot have both values, 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.

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

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

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 escape this situation, the solver needs to analyze the reason of the conflict with so-called *implication graph.* It represents the current state of the solver. It records all dependencies between variables and 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 by $l@dl(l)$ where l represents assigned literal and $dl(l)$ represents the decision level of the literal l. Root vertices , that have no incoming edges, represent decision literal. The remaining vertices represents propagations. Each incoming arc, labeled with a clause represents the *reason* of this propagation. This clause must be assertive, (i.e. all literals are false except one that is not yet assigned). Figure 2.3 shows the implication graph of the previous example (Figure 2.2).

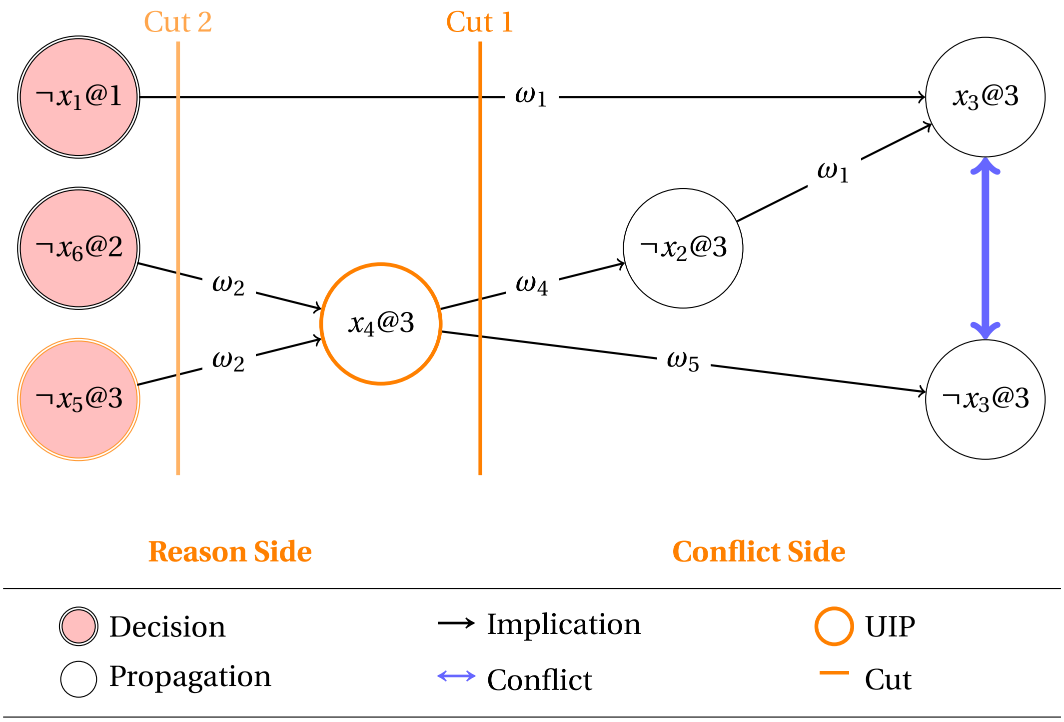


Figure 2.3: Implication graph

`analyzeConflict` procedure analyzes this graph to find the reason of the conflict. To do that, a search of *unique implication point* (UIP) is performed. UIP of the last decision level of the implication graph is a variable 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 (FUIP) 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 [35]. A UIP divides the implication graph in two sides with a *cut*, the *reason side* contains decision variables that is responsible of the contradiction and the *conflict side* that contains the conflict. 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 simply conflict clause) is produced. To build this clause, it suffices to negate the literals that have an ongoing arc to the cut that contains the UIP. In Figure 2.3, the produced clause will be $\omega_l = \{x_1, \neg x_4\}$. Since the information of this clause is redundant regarding the original formula. It can be added without any restriction. The conflict clause can be simplified using the implication graph to reduce its size, by detecting redundancies [34]. All leaned clause are store in clause database. `backjumpAndRestartPolicies` procedure is executed after producing the conflict clause. All responsible variables of the inconsistency are removed from the current assignment. Adding the conflict clause prunes search space that contains no solution. This is the key point of the CDCL algorithm. In our example Figure 2.3, the target decision level is 1. After backtrack, the conflict clause will be assertive, the first UIP is the only variable that have not a value and will be propagated in the next step of the algorithm. If a conflict implies only one level, the decision variable must be assigned to the opposite value at level 0. This means that this literal must be true without any decision.

Heuristics

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

Decision heuristics. Decision variable has a huge impact on the overall solving time. It impacts the number of propagations and so the depth of the search tree. The Variable State Independent Decaying Sum (VSIDS) [29] measure is one of the most famous decision heuristics and is used nowadays in almost all solvers; each variable has an activity and is increased by a multiplicative factor when it participates to the resolution of a conflict. Decision heuristics choose unassigned variable with the highest activity. Hakan: AVOIR The idea behind this heuristic 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 [24]) is a latest decision heuristic. It is a generalization of VSIDS and its goal is to optimize the *learning rate* (LR), defined as the ability to generate learned 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 the highest LRB is chose as a decision.

Hakan: AVOIR The idea behind this heuristics is to keep variables that used to generate learned clause in the search tree.

Restarts. Another important mechanism is *restart.* Basically, the solver abandons it current assignment and so start from the top of the tree, while maintaining other information notably learned clauses but also scores of variables in the decision heuristic. It prevents the solver to get stuck in 'hard' (heavy tailing [16] part of the search space and cannot escape due to backjump few levels after conflict resolution. Restart is best effort heuristics, hoping that, with more information, a better assignment was made. Hence, in practice, SAT solvers usually restart after a certain number of conflict. Empirically a solver with restart has a better result [17] and is today used in almost all state-of-the-art solvers.

Cleaning clause database. Storing all learned clauses will end up by a memory issue. So, we need to develop a apolicy to eliminate some of them. In the literatture, different criterion exists, the size of the clause is one of them and is very often used by solvers. Actually, a clause is really useful when it participates to unit propagation. The smaller the clause, the greater its chance. *Clause activity* is another heuristic, when a clause participate to conflict analysis its score is increment and lower activity will be removed. The last often used criteria is based on Literal Block Distance (LBD). It is a measure that computes the *quality* of a clause it is based on the number of decision levels present in the clause. Clauses with high value of LBD will bes deleted from the clause database.

Preprocessing / Inprocessing

In order to optimize solving time, some transformation can be applied to simplify the original formula. This is done by a *preprocessing* engine before the start of solving. When it is used 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*. Consider 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 a clause $\omega_3 = \{x_i, ..., x_j, y_i, ..., y_j\}$ which is called the *resolvent* as it results from solving 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 the computer will be limiting factor.

The subsumption aims at removing clauses. Consider 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 uses resolution rules and subsumption. The resolvent clause subsumes the original one. For example, $\omega_1 = \{x_1, \neg x_2, x_3\}$ and $\omega_2 = \{x_1, \neg x_2, x_3, x_4\}$, then the resolvent clause will be $\omega_3 = \{x_1, x_3\}$ which subsumes ω_2. This principle is implemented in `SatElite` [13] preprocessor engine and is used in almost all modern SAT solvers. Other simplification techniques exist such that *Gaussian elimination* which detects sub formula in a XOR-SAT form and solve it in a polynomial time. Moreover, this technique can also be used as inprocessing [33]. Some techniques exploit 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]. The usage of symmetries also adds relevant clauses 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 computers, one way to optimize the solving of a SAT problem is the exploitation of these cores. Actually, SAT problems are a good candidate for parallelism. *Portfolio* is a technique that launches several SAT solvers in parallel with different heuristics (decisions, restarts, ...) that communicates or not between them. When one of them finds a solution or finds that none exists, the overall computation is finished. Another technique to develop a parallel SAT solver is called *divide and conquer*. In this technique, the search space is divided dynamically. Several solvers cooperate to find a solution, each of them is 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 [22] [23]. Authors of this framework win the parallel tracks of SAT competition [1] in 2018.

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

CHAPTER 3

SYMMETRY AND SAT

Contents

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 way to cut off search space is the exploitation of symmetry. At its most basics, symmetry is a transformation of an object that leaves it unchanged. Symmetries is common in real life, if we take some butterfly, it has exactly the same halves. If we search a pattern on one halve and not found it, searching the same pattern in the other halve is completely absurd. But some of them are not symmetric and we must check pattern in every halves.

In this chapter, we show how to detect if the given formula presents symmetries and if it is the case, how to exploit them and so accelerate the solving time.

3.1 Group basics

As symmetries is a belongs to a branch of mathematics called theory group. This section gives 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 as 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 $a, b \in G$. A 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 groups $\langle G, *\rangle$, then

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

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 are 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 the 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 element in a group G can be expressed as a linear combination of a set of a 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 elements X being permuted. Two elements $x_1, x_2 \in X$ are equivalent if there exists a permutation $g \in G$ such that $g.x_1 = x_2$. The equivalence relation 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 definition 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 it 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 exist to tackle this problem like `saucy3` [20], `bliss` [19], `nauty` [27], etc.

There exists different ways to encode a SAT problem, which leads to different symmetries in these problems. 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 graphs and an automorphism tool is applied onto. Specifically, given a formula φ with m clauses over n variables, the graph is constructed as follows [6]:

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

Figure 3.1 shows the graph representation of a CNF. This problem has 6 variables and 11 clauses. So, the graph will have 12 + 11 = 33 vertexes where 12 represents literal vertexes (circle in the figure) and 11 represents the number of clause vertexes (square in the figure). The graph will also have 6 + 24 = 30 edges, 6 for Boolean consistency (red color in the figure) and 24 edges that rely clause 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 suffices to check:

$$\forall x \in supp(g), g.\neg x = \neg g.x$$

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

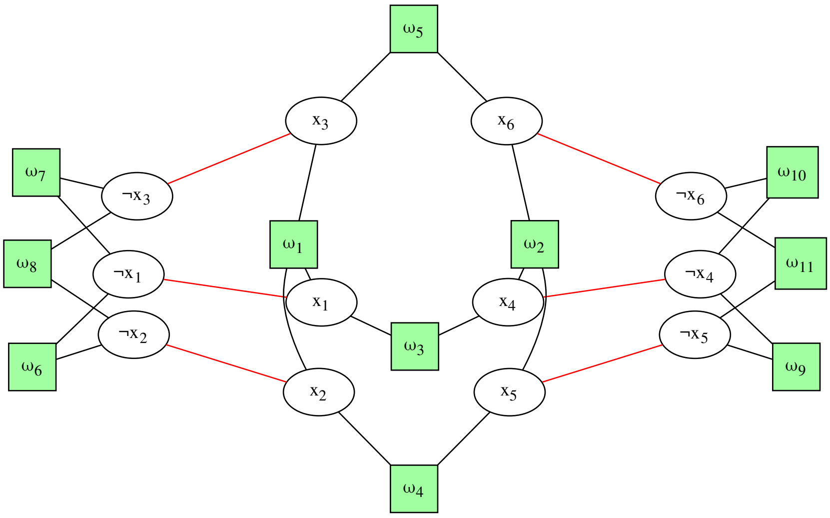


Figure 3.1: Example of constructed symmetry graph for a given CNF

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.

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

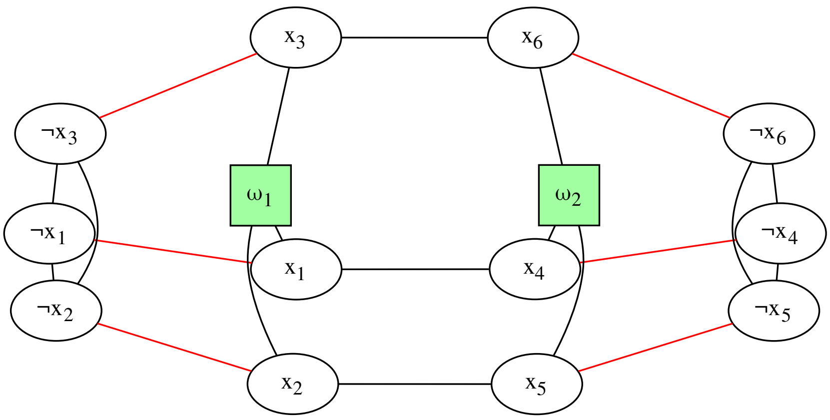


Figure 3.2: Example of constructed symmetry graph for a given CNF

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

$$\begin{aligned}
g_1 &= (x_2\ \ x_3)(x_5\ \ x_6)(\neg x_2\ \ \neg x_3)(\neg x_5\ \ \neg x_6)\\
g_2 &= (x_1\ \ x_2)(x_4\ \ x_5)(\neg x_1\ \ \neg x_2)(\neg x_4\ \ \neg x_5)\\
g_3 &= (x_1\ \ x_4)(x_2\ \ x_5)(x_3\ \ x_6)(\neg x_1\ \ \neg x_4)(\neg x_2\ \ \neg x_5)(\neg x_3\ \ \neg x_6)
\end{aligned}$$

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.

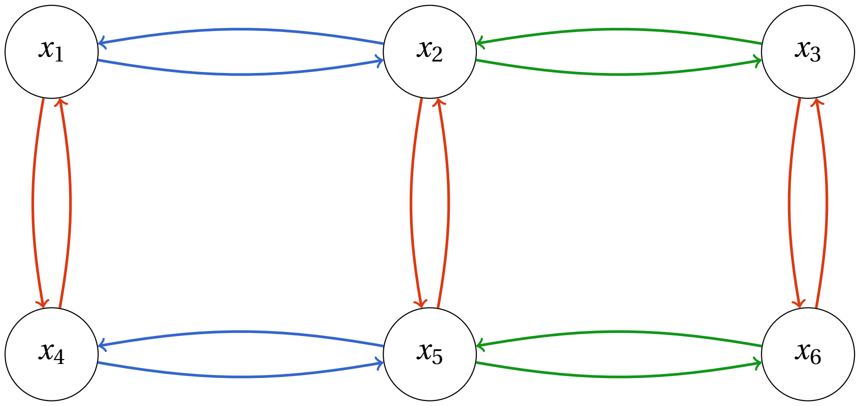


Figure 3.3: Graphical representation of an orbit

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.

Static symmetry breaking

Generally symmetry breaking allows SAT solver to avoid visit isomorphic search space. Visiting one branch is equivalent to visit each symmetric branch and so visiting this one is sufficient to determine satisfiability of the whole formula. Symmetrical branch are discarded by the solver with the addition of constraints. These constraint are not satisfied if the solver visit equivalent assignment. In other words, when symmetries of a problems are ignored, solver visit a branch multiple times. Suppose the *pigeonhole problems* (See fig. 3.4), where n pigeons are put into $n-1$ holes, with the constraint that each pigeon must be in a different hole is a highly symmetric problem. Indeed, all the pigeons (resp. holes) are exchangeable without changing the initial problem. Trying to solve it with a standard SAT solver, like MiniSAT [14], turns out to be very time consuming (and even impossible, in reasonable time, for high values of n). Here, such a standard solver ignores the symmetry property of the problem, and then potentially tries all variables combinations ; this eventually leads to a combinatorial explosion.

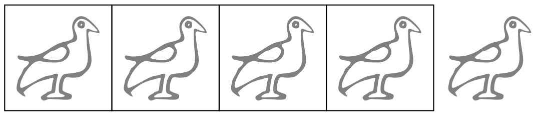


Figure 3.4: Graphical representation of pigeonhole problem

This section explains how to exploit statically symmetrical properties of a SAT problem. This can be summarize with the following two questions: 1) How to choose branch that are equivalent to all symmetric ones? 2) How to generate constraints that forbid symmetrical assignments?

To answer the first part of the problematic: how to choose branch that are equivalent to all symmetric ones ? We need to introduce an ordering relation between 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 words, the prefix of both assignment is equal 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 symmetric SAT problem would be to explore only one assignment per orbit (for instance each lex-leader). Figure 3.5 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* [8] and prevent other assignments from being explored.

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

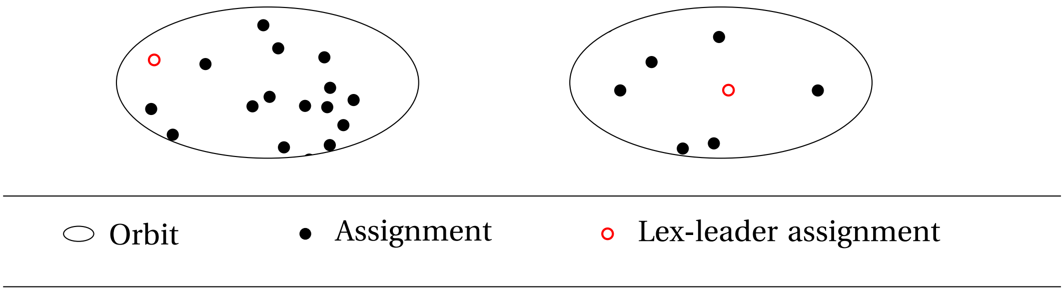


Figure 3.5: Show lex-leader per orbit

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 permutations $g \in G_\varphi$ 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 [26]. 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 cases, 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 principle (unit propagation). Hence, slow down overall time computation. Conversely, partial symmetry breaking adds few constraints and bring often considerable reduction of the search space. Generally, the set of generators produced by automorphism tool is chosen as a subgroup. Partial symmetry breaking gives a good trade off between the 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:*

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

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

Generation of lex-leader constraints proposed by Crawford et al. [8] is defined as follows:

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

Figure 3.6 shows an example of generated clauses for the permutation g_3 of the previous example and a 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.

$$\begin{array}{lll} \text{Order} & : & x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5 \prec x_6 \;\mid\; (\perp < \top) \\ \text{Permutation} & : & g_3 = (x_1 \;\; x_4)(x_2 \;\; x_5)(x_3 \;\; x_6)(\neg x_1 \;\; \neg x_4)(\neg x_2 \;\; \neg x_5)(\neg x_3 \;\; \neg x_6) \end{array}$$

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 3.6: 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 encodes the first equality. Following clauses encode a compact lex-leader for a permutation:

$$\begin{array}{ll|ll} \neg y_i \vee \neg x_{i-1} \vee \neg x_i \vee g.x_i & 1 \leq i \leq n & \neg y_i \vee \neg x_{i-1} \vee \neg y_{i+1} & 1 \leq i \leq n \\ \neg y_i \vee g.x_{i-1} \vee \neg x_i \vee g.x_i & 1 \leq i \leq n & \neg y_i \vee g.x_{i-1} \vee \neg y_{i+1} & 1 \leq i \leq n \end{array}$$

Figure 3.7 shows the compact encoding of generated clauses. This form grows 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 : $x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5 \prec x_6 \quad | \quad (\bot < \top)$
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 3.7: 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.

Hakan: Mette des espaces pour les nombres et texte pour expliquer si tableau ok
Hakan: Mettre total de clause et vars et pourcentage d'augmentation

Instances	#vars	#clause	#sbp	#auxiliary variables
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 [11] 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 a specific type of symmetry called *row (column) interchangeability*, when a subset of variables is structured as a two-dimensional matrix. Each row (column) is interchangeable with the symmetries. This form of symmetry is common in different kind of problem like pigeon hole problems in which pigeons and holes are interchangeable or in the delivery system in which trucks of a fleet are interchangeable. Usage of row (column) interchangeability can significantly improve SAT performance. Effectively symmetries can be eliminated by the addition of only a linear number of symmetry-breaking constraints [15]. 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 give only the set generators which contains no information on the structure of the group. Authors of `BreakID` [11] develop an algorithm to detect this specific structure and exploit it.

Binary lex-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*.

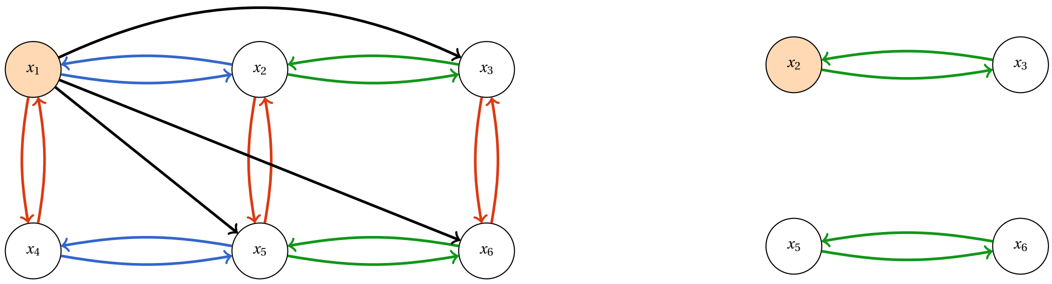
Figure 3.8 shows application of the generation of binary clauses. In the example, considered group has three permutations and its graphical representation is showing. Given the lexicographic order, the smallest variable is x_1 and all other variables are in its orbits. According to the ordering relation, five symmetry breaking predicates are generated with the formula $\neg x_1 \vee g.x_1$. Then, subgroup that stabilizes 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 fewer binary clauses are generated. An incremental order is proposed to optimize the number of generates binary clauses. First, orbit of all variables is computed and the variable with fewest number of occurrence is choosing among the biggest orbit. The idea is that biggest orbit produces more clauses and the variable appearing in few permutations reduce the

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

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

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



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

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

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

Figure 3.8: Generation of binary symmetry breaking predicates

number of stabilized permutations. This procedure is applied until trivial group is reached. At the end, remaining variables are added to the order.

BreakID

As summary, `BreakID` combines three idea. First it searches if produced generators by the automorphism tool has the special form row interchangeability and if that is the case exploit it. This first point establish a prefix of an order for the lex-leader. Secondly this order will be completed to generate a maximum number of binary lex-leader constraints. Thirdly, the order is supplemented with leftover variables until it is total and classical lex-leader constraints are added to the original formula.

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 is often too large to be effectively handled by a SAT solver [26]. 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 [6]. An extremely important point for static symmetry breaking is the chosen lexicographic order. Variable ordering may impact the number of generated constraints and so the performance of the underlying SAT solver. Different orders are studied in the literature. One of the simplest orders was the sorted variables according to their numbers. Some other order exists and exploit structural properties of the problem. Some recent works brings some optimizations and solves more instances. Despite these optimizations and the good reduction of the search space with symmetries, some formula that exhibit symmetries still intractable for a state-of-the-art SAT solver. Moreover, a disadvantage of static symmetry breaking is that the solver is influenced by SBPs and explore the search space with a different manner and can affect performance negatively.

Dynamic symmetry breaking

Dynamic symmetry breaking approaches aims to exploits symmetries during the solving by altering search space dynamically. This is possible with the integration of symmetry breaking approach inside existing SAT solver. It uses symmetries presents in the formula to propagate symmetrical literals. As consequence, the solver reduces the number of decisions, that are chosen heuristically and increase the number of propagation that are consequence of the formula. In other words, symmetries allows to transform some "guesses" as "deductions". And so, improves performance of the underlying SAT solver. In the literature, different approaches of dynamic symmetry breaking exists, this section presents some of them.

SymChaff

One of the first dynamic symmetry breaking approaches is *SymChaff* [31] and is applicable only on special groups where all couple of variables are symmetric section 3.4. The idea of this approach is to treat each orbit like a *symbolic variable*, i.e. instead of considering a single variable (branching), all symmetric variables are considered at the same time and so backtracked at the same time (k-branching). 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 tremendous impact of the solver performance. In the general case, when we take any groups computing the number of orbits 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* [12]. The general idea of this approach is to propagate symmetrical literals of those already propagated. In other words, it accelerates 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.

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 also satisfies 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 allows to compose permutations with the composition operator $\circ$ and the composition of two active symmetries is also an active symmetry so the solver can also propagate. $g^2.l, g^3.l, ...$

Devriendt et al improves the active symmetries in the SAT context, 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 also is a symmetry of $\varphi \cup \alpha$.*

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 allow more propagation and so is more efficient. Note that if a weakly active symmetry wants to propagate a symmetrical literal which are already affected to the opposite value, this leads to a symmetry conflict and the solver backtrack to propagate the symmetrical value correctly.

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 solver.

Note that, this approach don't discard any assignments like in the static approach where not lex-leader assignment were eliminated by symmetry breaking predicates.

Symmetry Explanation Learning

Another approach to exploit symmetry without removing any satisfiable assignment of the problem is *Symmetry Explanation Learning* [10] (SEL). Symmetries of a formula leaves this one invariant. Moreover all learned clauses are logical consequence of the problem, symmetric of these clauses are also valid. Unlike Symmetry explanation scheme [5] (SLS) where all symmetrical learned clauses are added to the clause database. The idea of this approach is to learn useful symmetrical variant of learned clauses. A clause is said useful if it participates to the unit propagation or conflict analysis. Computing all symmetrical learn clauses will create a huge overhead and will be intractable on real problems.

Symmetry Explanation Learning uses the following fact: on the unit propagation, propagated literals has a reason clause which is assertive. Generally , symmetries permute only few literals in a clause and so symmetrical clauses may also be assertive and participate to unit propagation. These clauses are stored in different learning scheme and treated separately. Solver promotes these clause uniquely where they are effectively useful at the end of unit propagation. As unit propagation is done until fix point, it ensures that no duplicate clause is added in the problem. To limit memory impact, symmetrical clauses are removed when the propagated literal responsible of the computation is unaffected.

SEL provides some interesting properties. First, the authors proves that its propagation are a super-set of the one provided by symmetry propagation. It also no need to track any status of symmetries as opposed to symmetry propagation. Like symmetry propagation no satisfying assignment are discarded. As disadvantage, SEL may flood the solver if the used set of symmetries is big and take time to compute symmetrical clauses.

Conclusion

Dynamic symmetry handling

Part II

Contributions

CHAPTER 4

Between Static and Dynamic Symmetry Breaking

Contents

This chapter presents our first contribution published in TACAS 2018 conference 4.

4.1 General idea

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 entirely computed. Even in more favorable situations, the size of the generated *sbp* is often too large to be effectively handled by a SAT solver [26]. 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 [6]. Besides, these approaches are preprocessors, so their combination with other techniques, such as *symmetry propagation* [12], can be very hard. Also, tuning their parameters during the solving turns out to be tough. For all these reasons, some classes of SAT problems cannot be solved yet despite the presence of 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* permutations 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 track the status of a permutation during the solving. Let us, first, recall that the *support* of a permutation g, $supp_g$, the set $\{v \in \mathcal{V} \mid g.v \neq v\}$.

Proposition 5. *Let $\alpha \in Ass(\mathcal{V})$ be an assignment, $g \in \mathfrak{S}$ a permutation and $supp_g \subseteq \mathcal{V}$ 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 $U = \{v', \neg v' \mid v' \in \mathcal{V}_g \text{ and } v' \preceq v\}$. 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]. Specifically, 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. Principle

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

of CDCL is described in section 6, algorithm 4 explains how to extend it with a *symmetry controller* component which guides the behavior of CDCL algorithm depending on the status of symmetries.

```
1  function CDCLSym(φ: CNF formula, SymController: symmetry controller)
     returns ⊤ if φ is SAT and ⊥ otherwise
2    dl ← 0                                          // Current decision level
3    α ← ∅
4    while not all variables are assigned do
5        isConflict ← unitPropagation()
6        SymController.updateAssign(α)
7        isReduced ← SymController.isNotLexLeader(α)
8        if isConflict || isReduced then
9            if dl == 0 then
10               return ⊥                            // φ is UNSAT
11           if isConflict then
12               ω ← analyzeConflict()
13           else
14               ω ← SymController.generateEsbp(α)
15           φ ← φ ∪ {ω}
16           dl ← backjumpAndRestartPolicies()
17           SymController.updateCancel(α)
18       else
19           α ← α ∪ assignDecisionLiteral()
20           dl ← dl + 1
21   return ⊤                                        // φ is SAT
```

Algorithm 4: the CDCLSym SAT Solving Algorithm.

The symmetry controller is initially given a set of symmetries G [2]. It observes the behavior of the SAT engine and updates its internal data according to the current assignment, to keep track of the status of the symmetries. This observation is *incremental*: whenever a literal is assigned or canceled, the symmetry controller updates the status of all the symmetries. This corresponds to lines 6 and 17 of Algorithm 3. When the controller detects that the current assignment cannot be a *lex-leader* (line 7), it generates the corresponding *esbp* (line 14).

In the remainder of this section, functions composing the symmetry controller are detailed.

[2] The generators of the group of symmetries.

Symmetries Status Tracking.

The `updateAssign`, `updateCancel` and `isNotLexLeader` functions (Algorithm 5) track the status of symmetries based on Proposition 5 ; there, resides th core of our algorithm.

All these functions rely on the pt structure: a map of variables indexed by permutations. Initially, $pt[g] = \min_{\prec}(supp_g)$ for all $g \in G$ according to the ordering relation and all permutations are marked *active*.

For each permutation, g, the symmetry controller keeps track of the smallest variable $pt[g]$ in the support of g such that $pt[g]$ and $g^{-1}(pt[g])$ does not have the same value in the current assignment. If one of the two variables is not assigned, they are considered to have different values.

When new literals are assigned, only active symmetries need to have their $pt[g]$ updated (line 2). This update is done thanks to a while loop (lines $4-5$).

When literals are canceled, we need to update the status of symmetries for which some variable v before $pt[g]$, or $g^{-1}(v)$, becomes unassigned (lines $9-10$). Symmetries that were inactive may be reactivated (line 11).

The current assignment is not a *lex-leader* if some symmetry g is a reducer. This is detected by comparing the value of $pt[g]$ with the value of $g^{-1}(pt[g])$ (line 16). The function `isNotLexLeader` also marks symmetries as *inactive* when appropriate (lines $18-19$).

Generation of the *esbp*.

When the current assignment cannot be a *lex-leader*, some symmetry g is a reducer. The function `generateEsbp` computes the $\eta(\alpha, g)$ defined in Definition 7, which is an effective symmetry-breaking predicate by Proposition 6. This will prevent the SAT engine to explore further the current partial assignment.

Illustrative example

Let us illustrate the previous concepts and algorithms on a simple example. Let the ordering relation $x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5 \prec x_6 \mid \bot < \top$, and the two last previous generators. $G = \{g_2 = (x_1\ \ x_2)(x_4\ \ x_5), g_3 = (x_1\ \ x_4)(x_2\ \ x_5)(x_3\ \ x_6)\}$ (written in cycle notation with opposite cycles omitted). Their respective supports sorted according to ordering relation are, $supp_{g_2} = \{x_1, x_2, x_4, x_5\}$ and $supp_{g_3} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$.

On the assignment $\alpha = \emptyset$, both permutations are active and $pt[g_1] = pt[g_2] = x_1$. When the solver updates the assignment to $\alpha = \{x_4\}$, both permutations remain active and $pt[g_2] = pt[g_3] = x_1$. On the assignment $\alpha = \{x_4, x_1\}$, the symmetry controller updates $pt[g_3]$ to x_2, while $pt[g_2]$ remains unchanged. On the assignment $\alpha = \{x_4, x_1, \neg x_2\}$, $g_2.\alpha = \{x_5, x_2, \neg x_1\}$, which is smaller than α (because $x_1 \in \alpha$ and $\neg x_1 \in g.\alpha$): g_2 is a reducer of α. The symmetry controller then generates the corresponding *esbp* $\omega = \{\neg x_1, x_2\}$.

```
function updateAssign(α: assignment)
    foreach active g ∈ G do
        v ← pt[g];
        while {v, g⁻¹(v)} ⊆ α or {¬v, ¬g⁻¹(v)} ⊆ α do
            v ← next variable in 𝒱_g;
        pt[g] ← v
function updateCancel(α: assignment)
    foreach g ∈ G do
        u ← min{v ∈ 𝒱_g | {v, ¬v} ∩ α = ∅ or {g⁻¹(v), ¬g⁻¹(v)} ∩ α = ∅};
        if u ≤ pt[g] then
            mark g as active;
            pt[g] ← u;
function isNotLexLeader(α: assignment)
    foreach active g ∈ G do
        v ← pt[g];
        if {v, ¬g⁻¹(v)} ⊆ α then
            return ⊤;                              // g is a reducer
        if {¬v, g⁻¹(v)} ⊆ α then
            mark g as inactive ;                   // g can't reduce α or its
                                                      extensions
    return ⊥
function generateEsbp(α: assignment) returns ω: generated esbp
    ω ← {};
    g ← the reducer of α detected in isNotLexLeader;
    v ← min(𝒱_g);
    u ← pt[g];
    while u ≠ v do
        if v ∈ α then ω ← ω ∪ {¬v} else ω ← ω ∪ {v};
        if g⁻¹(v) ∈ α then ω ← ω ∪ {¬g⁻¹(v)} else ω ← ω ∪ {g⁻¹(v)};
        v ← next variable in 𝒱_g
    ω ← ω ∪ {¬v, g⁻¹(v)};
    return ω
```

Algorithm 5: the functions keeping track of the status of the symmetries and generating the *esbp*.

4.2 Implementation and Evaluation

In this section, we first highlight some details on our implementation of the symmetry controller. Then, we experimentally assess the performance of our algorithm against three other state-of-the-art tools.

cosy: an efficient implementation of the symmetry controller

We have implemented our method in a C++ library called cosy (1630 LoC). It implements a symmetry controller as described in the previous section, and can be interfaced with virtually any CDCL SAT solver. cosy is released under GPL v3 license and is available at `https://github.com/lip6/cosy`.

Heuristics and Options.

Let us recall that finding the optimal ordering of variables (with respect to the exploitation of symmetries) is NP-hard [25], so the choice for this ordering is heuristic. cosy offers several possibilities to define this ordering:

- a naive ordering, where variables are ordered by the lexicographic order of their names;
- an ordering based on occurrences, where variables are sorted according to the number of times they occur in the input formula. The lexicographic order of variable names is used for those having the same number of occurrences;
- an ordering based on symmetries, where variables belonging to the same orbit (under the given set of symmetries) are grouped together. Orbit are ordered by their numbers of occurrences.

The ordering of assignments we use in this paper orders positive literals before negative ones (thus, $\top < \bot$), but using the converse ordering does not change the overall method. However, it can impact the performance of the solver on some instances, so that it is an option of the library. All the symmetries we used for the presentation of our approach are permutations of variables. Our method straightforwardly extends to permutations of literals, also known as *value permutations* [6].

Integration in `MiniSAT`.

We show how to integrate cosy to an existing solver, through an example of `MiniSAT` [14]. First, we need an adapter that allows the communication between the solver and cosy (30 LoC). Then, we adapt Algorithm 3 to the different methods and functions of `MiniSAT`. In particular, the function `updateAssign` is moved into the `uncheckEnqueue` function of `MiniSAT` (2 LoC). The `updateCancel` function is moved to the `cancelUntil` function of `MiniSAT` that performs the backjumps (2 LoC). The `isNotLexLeader` and `generateEsbp` functions are integrated in the `propagate` function of `MiniSAT` (30

LoC). This is to keep track of the assignments as soon as they occur, then the *esbp* is produced as soon as an assignment is identified as not being *lex-leader*. Initialization issues are located in the main function of `MiniSAT`(15 LoC). The integration of **cosy** increases `MiniSAT` code by 3%.

Evaluation

This section presents the evaluation of our approach. All experiments have been performed with our modified `MiniSAT` called `MiniSym`. The symmetries of the SAT problem instances have been computed by two different state-of-the-art tools `saucy3` [20] and `bliss` [19]. For a given group of symmetries, the first tool generates less permutations to represent the group than the second one, but it is slower than the other one. We selected from the last six editions of the SAT contests [18], the CNF instances for which `bliss` finds at least 2% of the variables are involved in some symmetries that could be computed in at most 1000*s* of CPU time. We obtained a total of 1350 symmetric instances (discarding repetitions) out of 3700 instances in total. All experiments have been conducted using the following conditions: each solver has been run once on each instance, with a time-out of 5000 seconds (including the execution time of the symmetries generation except for `MiniSAT`) and limited to 8 GB of memory. Experiments were executed on a computer with an Intel Xeon X7460 2.66 GHz featuring 24 cores and 128 GB of memory, running a Linux 4.4.13, along with g++ compiler version 6.3. We compare `MiniSym` using the occurrence order, value symmetries, and without *lex-leader* forcing, against:

- `MiniSAT`, as the reference solver without symmetry handling [14];
- `Shatter`, a symmetry breaking preprocessor described in [1], coupled with the `MiniSAT` SAT engine;
- `BreakID`, another symmetry breaking preprocessor, described in [11], also coupled with the `MiniSAT` SAT engine.

Each SATsolution was successfully checked against the initial CNF. For UNSATsituations, there is no way to provide an UNSATcertificate in presence of symmetries. Nevertheless, we checked our results were also computed by the other measured tools. Unfortunately, out of the 1350 benchmarked formulas, we have no proof or evidence for the 15 UNSATformulas computed by `MiniSym` only. Results are presented Tables in 4.1, 4.2, and 4.3. We report the number of instances solved within the time and memory limits for each solver and category. We separate the UNSAT instances (Table 4.1) from the SAT ones (Table 4.2). Besides the reference with no symmetry (column `MiniSAT`), we have compared the performance of the three tools when using symmetries computed by `saucy3` (see Table 4.1a and Table 4.2a), and `bliss` (see Table 4.1b and Table 4.2b). Rows correspond to groups of instances: from each edition of the SAT contest, and when possible, we separated applicative instances (app$\langle x\rangle$ where $\langle x\rangle$ indicates the year) from hard combinatorial ones (hard$\langle x\rangle$). This separation was not possible for the editions 2015 and 2017 (all2015 and all2017). The

Benchmark	MiniSAT	Shatter	BreakID	MiniSym
app2016 (134)	18	19	**20**	17
app2014 (161)	23	23	22	**24**
app2013 (145)	6	8	8	**10**
app2012 (367)	115	115	**120**	**120**
hard2016 (128)	8	17	**50**	42
hard2014 (107)	9	24	**30**	29
hard2013 (121)	12	24	**48**	29
hard2012 (289)	86	84	88	**93**
all2017 (124)	8	14	**15**	14
all2015 (65)	9	8	8	**10**
TOTAL (no dup)	261	302	**371**	345

(a) With saucy3

Benchmark	MiniSAT	Shatter	BreakID	MiniSym
app2016 (134)	18	**21**	18	19
app2014 (161)	23	21	20	**24**
app2013 (145)	6	7	10	**11**
app2012 (367)	115	106	114	**123**
hard2016 (128)	8	11	**79**	77
hard2014 (107)	9	45	40	**53**
hard2013 (121)	12	51	**56**	54
hard2012 (289)	86	69	90	**93**
all2017 (124)	8	14	**15**	**15**
all2015 (65)	**9**	7	8	8
TOTAL (no dup)	261	324	415	**439**

(b) With bliss

Table 4.1: Comparison of different approaches on the UNSATinstances of the benchmarks of the six last editions of the SAT competition.

Benchmark	MiniSAT	Shatter	BreakID	MiniSym
app2016 (134)	20	**22**	21	20
app2014 (161)	**24**	**24**	**24**	22
app2013 (145)	34	35	35	**43**
app2012 (367)	121	112	119	**126**
hard2016 (128)	**0**	**0**	**0**	**0**
hard2014 (107)	14	**17**	**17**	14
hard2013 (121)	23	23	**24**	22
hard2012 (289)	135	141	**143**	138
all2017 (124)	23	20	26	**27**
all2015 (65)	**7**	5	**7**	6
TOTAL (no dup)	325	323	**337**	335

(a) With saucy3

Benchmark	MiniSAT	Shatter	BreakID	MiniSym
app2016 (134)	20	20	**22**	20
app2014 (161)	**24**	**24**	23	22
app2013 (145)	**34**	32	30	33
app2012 (367)	**121**	112	120	118
hard2016 (128)	**0**	**0**	**0**	**0**
hard2014 (107)	14	14	17	**18**
hard2013 (121)	23	24	**26**	25
hard2012 (289)	135	134	141	**142**
all2017 (124)	23	25	26	**29**
all2015 (65)	**7**	5	6	6
TOTAL (no dup)	325	316	334	**336**

(b) With bliss

Table 4.2: Comparison of different approaches on the SATinstances of the benchmarks of the six last editions of the SAT competition.

total number of instances for each bench is indicated between parentheses. For each row, the cells corresponding to the tools solving the most instances (within time and memory limits) are typeset in bold and grayed out. Table 4.3 shows the cumulative and average PAR-2 times of the evaluated tools. We observe that MiniSym with saucy3 solves the most

Solver	PAR-2 sum	PAR-2 avg
MiniSAT	8 074 348	5 981
Shatter	7 770 434	5 756
BreakID	6 909 999	5 119
MiniSym	7 229 700	5 355

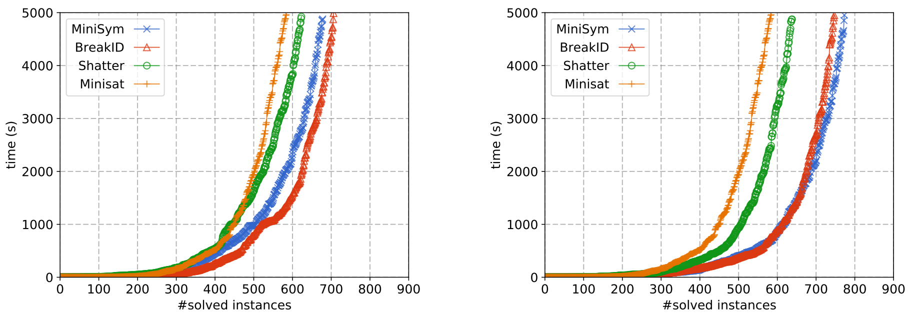
(a) With saucy3

Solver	PAR-2 sum	PAR-2 avg
MiniSAT	8 074 348	5 981
Shatter	7 517 556	5 569
BreakID	6 444 954	4 774
MiniSym	6 245 448	4 626

(b) With bliss

Table 4.3: Comparison of PAR-2 times (in seconds) of the benchmarks on the six last editions of the SAT competition.

instances in only half of the UNSATcategories. However, with bliss, MiniSym solves the most instances in all but four of the UNSATcategories ; it then also solves the highest number of instances among its competitors. This shows the interest of our approach for UNSATinstances. Since symmetries are used to reduce the search space, we were expecting that it will bring the most performance gain for UNSATinstances. The situation for SATinstances is more mitigated (Table 4.2), especially when using `saucy3`. Again, this is not very surprising: our method may cut the exploration of a satisfying assignment because it is not a *lex-leader*. This delays the discovery of a satisfying assignment. The other tools suffer less from such a delay, because they rely on symmetry breaking predicates generated in a pre-processing step. Also, when seeing the global results of `MiniSAT`, we can globally state that the use of symmetries in the case of satisfiable instances only offers a marginal improvement. We observe that performances our tool are better with `bliss` than with



(a) with `saucy3` (b) with `bliss`

Figure 4.1: cactus plot total number of instances

`saucy3` (see fig 4.1). We explain it as follows: `saucy3` is known to compute fewer generators for the group of symmetries than `bliss`. Since, the larger the symmetries set is, the earlier the detection of an *evidence* that an assignment is not a *lex-leader* will be, we generate less symmetry-breaking predicates (only the effective ones). This is shown in Table 4.4; `MiniSym` generates an order of magnitude fewer predicates than `BreakID`.

Number of SBPs	`BreakID`	`MiniSym`
UNSAT(316)	12 088 433	1 579 623
SAT(312)	13 839 689	359 352

(a) With `saucy3`

Number of SBPs	`BreakID`	`MiniSym`
UNSAT(399)	2 576 349	913 339
SAT(320)	12 179 513	457 452

(b) With `bliss`

Table 4.4: Comparison of the number of generated SBPs each time `BreakID` and `MiniSym` both compute a verdict (number of verdicts between parentheses).

We also conducted experiments on highly symmetrical instances (all variables are involved in symmetries), whose results are presented in Table 4.5. The performance of `BreakID` on this benchmark is explained by a specific optimization for the *total symmetry groups* that are found in these examples, that is neither implemented in `Shatter` nor in `MiniSym`.

However, the difference between `BreakID` and `MiniSym` is rather thin when using `bliss`. Our tool still outperforms `Shatter` on this benchmark.

Benchmark	MiniSAT	Shatter	BreakID	MiniSym
battleship(6)	**5**	**5**	**5**	**5**
chnl(6)	4	**6**	**6**	**6**
clqcolor(10)	3	4	5	**6**
fpga(10)	6	**10**	**10**	**10**
hole(24)	10	12	**23**	11
hole shuffle(12)	1	2	**12**	3
urq(6)	1	2	**6**	2
xorchain(2)	1	1	**2**	**2**
TOTAL	31	42	**69**	45

(a) With `saucy3`

Benchmark	MiniSAT	Shatter	BreakID	MiniSym
battleship(6)	5	5	5	**6**
chnl(6)	4	**6**	**6**	**6**
clqcolor(10)	3	5	8	**10**
fpga(10)	6	**10**	**10**	**10**
hole(24)	10	**24**	**24**	23
hole shuffle(12)	1	3	**7**	4
urq(6)	1	2	**6**	5
xorchain(2)	1	1	**2**	**2**
TOTAL	31	56	**68**	66

(b) With `bliss`

Table 4.5: Comparison of the tools on 76 highly symmetric UNSAT problems.

4.3 Related Works

Usage of symmetry property dynamically allows the solver to adapt classical heuristics and symmetry based one on the fly. For example, some restart heuristics are based on the number of conflicts, taking into account injection of ESBP may impact the performance of the overall SAT solver.

Adapt heuristics dynamically

Other heuristics on the symmetry handling may increase the performance. We present here some of them. In some cases multiple permutations are reducer at the same time, and each one generate different symmetry breaking constraints. The backtrack level of the solver and the pruning capacity depends heavily on the chosen one. In the provided library, first reducer permutation generates the constraints. Inject symmetry breaking predicates when it was detected at the beginning of the unit propagation or at the end will change the solver behavior. In the first case, ESBP is more important of a classical conflict if bot occurs and the inverse on the second case. We can even ignore the conflict and just add the constraints to clause database. This prevents the solver to get multiple time on non-minimal part of search. We can allow to go one time for satisfiable for example. Effectively, if the solution is easy to find in this symmetrical branch and much harder on the lex leader, it may have a positive impact.

Change the Order Dynamically

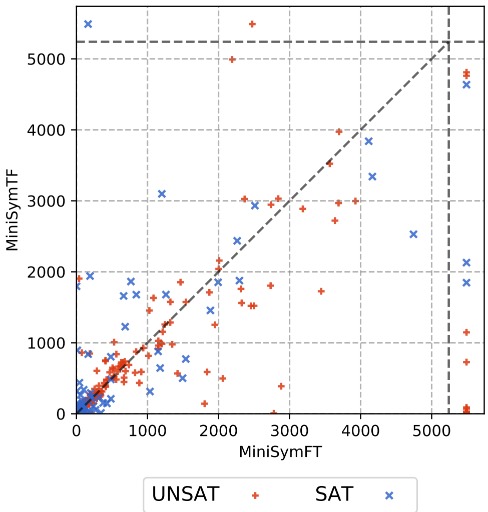
As seen before, ordering relation between variable influence lex-leader and the generated constraints. This order is chosen heuristically, changing this order dynamically is possible but with some requirement. All generated constraints and all deduced clause from a symmetry breaking predicates needs to be completely removed. If these constraints are not removed, inconsistencies may appear and a satisfiable problem become unsatisfiable.

Impact of the sign in variable ordering

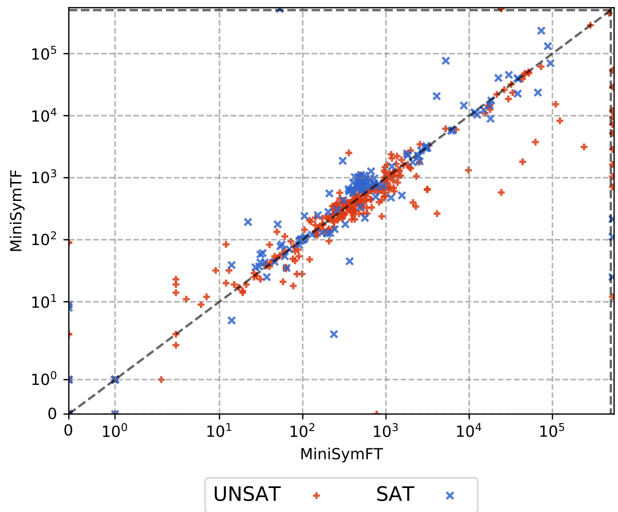
With the same variable ordering, swapping the value thus, $\top < \bot$ or $\bot < \top$ may impact drastically the performance of the solver.

To illustrate it, we execute hole100, the pigeonhole problem with 100 holes and 101 pigeons with the increase order and change only the sign. With $\bot < \top$, the solver generates 20 619 ESBP and takes 13.8 seconds to solve it. With the inverse order, $\top < \bot$, it generates 33 263 ESBP ans solve it in 93.4 seconds.

Following figures 4.6 show this difference on 500 symmetric instances with a scatter plot that compare orders, `MiniSymFT` is $\top < \bot$ and `MiniSymTF` is $\bot < \top$. On the left, we compare the duration of the solver, `MiniSymTF` is more efficient on some UNSATinstances. The right figure shows the number of generated ESBP by solvers in log scale. On some instance, it will generate approximately the same number of ESBP. But the difference can an order of magnitude higher. A high number of ESBP is not necessarily better. If few number of ESBP cut-off a huge search space it can be sufficient for the solver to prove the existence or not of a solution, variable ordering impact also.



(a) Time



(b) Number of ESBP

Table 4.6: Comparison of the order with different sign on 500 symmetric instances.

`MiniSymTF` is generally better and is the default choice of the library. But on some cases, it can be worth than the opposite order like in the hole problem. With a specific application, both sign orders can be applied to evaluate which one is better.

4.4 Conclusion

SymmSAT uses same principles as static symmetry breaking but operates dynamically by injecting effective symmetry breaking during the search. This overcomes, the main problem of the static approaches , that they generate many *sbp* that is not effective in the solving (size of the generated formulas, overburden of the unit propagation procedure, etc.). The idea we bring is to break symmetries *on the fly*: when the current partial assignment cannot be a prefix of a *lex-leader* (of an orbit), an *esbp* that prunes this forbidden assignment and all its extensions are generated. This approach is implemented in the C++ library called cosy. It is an off-the-shelf component that can be interfaced with virtually any CDCL SAT solver. cosy is released under GPL license and is available at `https://github.com/lip6/cosy`.

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.

CHAPTER 5

COMPOSE DYNAMIC SYMMETRY HANDLING

Contents

5.1 Composition of SP and SymmSAT

Recently, we developed an approach that reuses the principles of the static approaches, but operates dynamically (namely, the effective symmetry breaking approach [28]): the symmetries are broken during the search process without any pre-generation of the *sbp*. The main advantage of this technique is to cope with the heavy (and potentially blocking) pre-generation phase of the static-based approaches. It also gives more flexibility for adjusting some parameters on the fly. Nevertheless, we also observed that many formulas easily solved by the pure dynamic approaches remained unsolvable by our approach and vice versa. This is particularly true with the *symmetry propagation* technique developed by Devriendt et al. [12]. Hence, our goal is to explore the composition of our algorithm with the *symmetry propagation* technique in a new approach that would mix the advantages of the two classes of techniques while alleviating their drawbacks. At first sight, the two approaches appear to be orthogonal, and hence could be mixed easily. However, as we show in the rest of this section, this is not completely true: both theoretical and practical issues have to be analyzed and solved to get a running complementary. Since the approach based on symmetry propagation (later called SPA) focuses on accelerating the tree traversal and the approach based on effective symmetry breaking (later called ESBA) targets to prune the tree traversal, the question of combining these approaches, to solve a formula φ, can be reformulated as:

is it possible to accelerate the traversal while pruning the tree?

Theoretical foundations

To answer the previous questions, we analyze the evolution of φ during its solving. In ESBA, φ evolves, incrementally, to an equi-satisfiable formula of the form $\varphi \equiv \varphi \cup \varphi_e \cup \varphi_d$, where φ_e is a set of injected esbps and φ_d is a set of deduced clauses (logical consequences). Both sets are modified continuously during the solving. Hence, to be able to compose ESBA with SPA, we have to consider the symmetries of $\varphi' = \varphi \cup \varphi_e \cup \varphi_d$ as allowed permutations in place of those of φ. A first naive solution could be to recompute, dynamically, the set of symmetries of $\varphi \cup \varphi_e \cup \varphi_d$ for each new $\varphi_e \cup \varphi_d$, but this would be an intractable solution generating a huge complexity. A computationally less expensive solution would be to keep track of all globally unbroken symmetries as the clauses of φ_e are injected during the solving process: considering formula φ and a set of esbps φ_e then the set of global unbroken symmetries is:

$$GUS = \bigcap_{\omega_e \in \varphi_e} Stab(\omega_e) \cap S(\varphi)$$

Where $Stab(\omega_e) = \{g \in \mathfrak{S}\mathcal{V} \mid \omega_e = g.\omega_e\}$ is the stabilizer set of ω_e and $S(\varphi)$ is the set of symmetries of φ. Since $\varphi \cup \varphi_e \models \varphi_d$, then GUS is a valid set of symmetries for $\varphi \cup \varphi_e \cup \varphi_d$. Then, (1) each time a new set of esbp clauses is added, its stabilizer will be used to reduce GUS; (2) conversely, when a set of esbp clauses is reduced[1], GUS cannot be enlarged by the recovered broken symmetries because of the retrieved set: *at that point, we do not know which symmetries become valid*! As a consequence, the set of globally unbroken symmetries will converge very quickly to the empty set. At this point, SPA will be blocked for the rest of the solving process without any chance to recover. Therefore, this solution is of limited interest in practice. We propose here to improve the aforementioned solution by alleviating the issue cited in point (2). We first present the intuition, then we will detail and formalize it. Consider formula φ' as before. It can be rewritten as:

$$\varphi' = \varphi \bigcup_i (\varphi_e^i \cup \varphi_d^i) \text{ , such that } \varphi_e \cup \varphi_d = \bigcup_i (\varphi_e^i \cup \varphi_d^i) \text{ and } \varphi \cup \varphi_e^i \models \varphi_d^i \text{ for all } i$$

So, $GUS_i = \bigcap_{\omega_e \in \varphi_e^i} Stab(\omega_e) \cap S(\varphi)$ is a valid set of symmetries for the sub-formula $\varphi \cup \varphi_e^i \cup \varphi_d^i$, and GUS can be obtained by $GUS = \bigcap_i GUS_i$. If some esbp clauses are added to φ', then the

[1] In classical CDCL algorithm, this can be due to a back-jump or a restart.

new GUS is computed as described in (1). The novelty here comes with the retrieval of some set of clauses: by keeping track of the symmetries associated to each sub-formula (GUS_i), it is now easy to recompute a valid set of symmetries for φ' when some set $\varphi_e^k \cup \varphi_d^k$ is retrieved. It suffices to operate the intersection on the valid symmetries of the rest of the sub-formulas: $GUS = \bigcap_{i \neq k} GUS_i$. Just say your approach keeps track of a set of particular symmetries for each clause. For a deduced clause, this set of symmetry captures which esbp's were involved in a deduced clause's derivation. The intersection of these sets is a superset of the globally unbroken symmetries, and a strict superset after clause deletion.

Local Symmetries

The general and formal framework that embodies the above idea is given by the following. It first relies on the notion of *local symmetries* that we introduce in definition 8.

Definition 8. *Let φ be a formula. We define $L_{\omega,\varphi}$, the set of local symmetries for a clause ω, and with respect to a formula φ, as follows:*

$$L_{\omega,\varphi} = \{g \in \mathfrak{S}\mathcal{V} \mid \varphi \models g.\omega\}$$

$L_{\omega,\varphi}$ is local since the set of permutations applies locally to ω. It is then straightforward to deduce the next proposition that gives us a practical framework to compute, incrementally, a set of symmetries for a formula (by using the intersection of all local symmetries).

Proposition 7. *Let φ be a formula. Then, $\bigcap_{\omega \in \varphi} L_{\omega,\varphi} \subseteq S(\varphi)$.*

Proof. Let φ be a formula. Then, $\forall \omega \in \varphi, \forall g \in L_{\omega,\varphi}, \varphi \models g.\varphi$. So, $\forall g \in \bigcap_{\omega \in \varphi} L_{\omega,\varphi}, \varphi \models g.\varphi$. This is combined with the fact that the number of satisfying assignments for a formula is not changed by permuting the variables of the formula, we have $g.\varphi \models \varphi$. Hence $\varphi \equiv g.\varphi$, and $g \in S(\varphi)$ (by definition). □

Using this proposition, it becomes easy to reconsider the symmetries on-the-fly: each time a new clause ω is added to the formula φ, we can just operate an intersection between $L_{\omega,\varphi}$ and $\bigcap_{\omega' \in \varphi} L_{\omega',\varphi}$ to get a new set of valid symmetries for $\varphi \cup \{\omega\}$. Proposition 8 establishes the relationship between the local symmetries of a deduced clause and those of the set of clauses that allow its derivation.

Proposition 8. *Let φ_1 and φ_2 be two formulas, with $\varphi_2 \subseteq \varphi_1$. Let ω be a clause such that $\varphi_2 \models \omega$. Then, $(\bigcap_{\omega' \in \varphi_2} L_{\omega',\varphi_1}) \cup Stab(\omega) \subseteq L_{\omega,\varphi_1}$;*

Proof. Let us consider a clause ω and a permutation $g \in (\bigcap_{\omega' \in \varphi_2} L_{\omega',\varphi_1}) \cup Stab(\omega)$. Since, $\varphi_2 \models \omega$, then $g.\varphi_2 \models g.\omega$. Since $\varphi_1 \models \varphi_2 (\varphi_2 \subseteq \varphi_1)$, and $g \in (\bigcap_{\omega' \in \varphi_2} L_{\omega',\varphi_1}) \cup Stab(\omega)$, then we have $\varphi_1 \models g.\varphi_2$ (from Def. 8). Hence, $\varphi_1 \models g.\varphi_2 \models g.\omega$, and then, $g \in L_{\omega,\varphi_1}$ (by definition). □

Algorithm

This section shows how to integrate the propositions developed in the previous section as the basis of our combo approach in a concrete Conflict-Driven Clause Learning (CDCL)-like solver. First recall the algorithm of symmetry propagation used for the combination of two approaches.

```
1  function CDCLSymSp(φ: CNF formula, spController: symmetry propagation controller)
2      returns ⊤ if φ is SAT and ⊥ otherwise
3      dl ← 0;                                          // Current decision level
4      α ← ∅;
5      while not all variables are assigned do
6          isConflict ← unitPropagation() ∧ spController.symPropagation();
7          if isConflict then
8              if dl = 0 then
9                  return ⊥;                                      // φ is UNSAT
10             ω ← analyzeConflict();
11             dl ← backjumpAndRestartPolicies();
12             φ ← φ ∪ {ω};
13             spController.cancelActiveSymmetries();
14         else
15             α ← α ∪ assignDecisionLiteral();
16             dl ← dl + 1;
17             spController.updateActiveSymmetries();
18     return ⊤;                                                   // φ is SAT
```

Algorithm 6: The CDCLSp algorithm. Blue (or grey) parts denote additions to CDCL.

CDCLSp (see Algorithm 6) implements SPA, and also has a structure similar to the one of CDCL. In this algorithm, the symmetry propagation actions are executed by the controller component (spController) through a call to the function symPropagation (line 6). This propagation is allowed only if the conditions are met. Such conditions are evaluated by tracking on-the-fly the status of the symmetries. This is implemented by functions updateSymmetries (line 17) and cancelSymmetries (line 13).

The algorithm we propose for the composed approach is presented in algorithm 7. Let us detail the critical points.

- Lines 13 and 16: when a conflict is detected, then the analyzing procedure is triggered. According to Proposition 8, the generated conflicting clause ω, should be as-

```
function CDCLSymSp (φ: CNF formula, symController: symmetry controller,
                    spController: symmetry propagation controller)
    returns ⊤ if φ is SAT and ⊥ otherwise
    dl ← 0;                                          // Current decision level
    α ← ∅;
    while not all variables are assigned do
        isConflict ← unitPropagation() ∧ spController.symPropagation();
        symController.updateAssign(α);
        isReduced ← symController.isNotLexLeader(α);
        if isConflict ∨ isReduced then
            if dl = 0 then
                return ⊥;                                    // φ is UNSAT
            if isConflict then
                ⟨ω, L = ⋂_{ω′∈φ1} L_{ω′,φ1} ∪ Stab(ω)⟩ ← analyzeConflictSymSp();
            else
                ⟨ω, L = Stab(ω)⟩ ← symController.generateEsbpSp(α);
            dl ← backjumpAndRestartPolicies();
            φ ← φ ∪ {ω};
            symController.updateCancel(α);
            spController.cancelActiveSymmetriesSym();
            spController.updateLocalSymmetries(L);
        else
            α ← α ∪ assignDecisionLiteral();
            dl ← dl + 1;
            spController.updateActiveSymmetriesSym();
    return ⊤;                                                // φ is SAT
```

Algorithm 7: The `CDCLSymSp` algorithm. Additions derived from `MiniSym` and `CDCLSp` are reported in red and blue (or grey). Additions due to the composition of the two algorithms are reported with a gray background.

sociated with the computation of its set of local symmetries. Thus, we update the classical `analyzeConflict` procedure to `analyzeConflictSymSp` that produces such a set: φ_1 contains all the clauses that are used to derive ω[2]. So, at the end of the conflict analysis we operate the intersection of a local symmetry of these clauses to get the set of local symmetries of ω. We can thus complete this set with the stabilizer set (see as Proposition 8).

In the classical algorithm of symmSAT, when a non lex-leader assignment is detected , then the esbp generation function, `generateEsbp`, is called. In the new algorithm this function is replaced by a new one called `generateEsbpSp`. In addition to compute the esbp clause ω, it produces the stabilizer set of ω[3].

- Line 20: `cancelActiveSymmetriesSym` extends function `cancelActiveSymmetrie` of Algorithm 6 with the additional reactivation of the symmetries that have been broken (deactivated) by ESBPA. Technically speaking, each time a deduced literal is unassigned, all symmetries that became inactive because of its assignment (see `updateLocalSymmetries` and `updateActiveSymmetriesSym` functions below) are *reactivated*.

- Line 21: `updateLocalSymmetries` is a new function of `spController`. It is responsible of updating the status of the manipulated symmetries so that only those respecting Proposition 7 are active each time the `symPropagation` function is called. Technically speaking, each symmetry of the complement set (to $S(\varphi)$) of the set L is marked *inactive* (it is a broken symmetry), if it is not already marked so. Here, the asserting literal of clause ω becomes responsible of this deactivation.

- Line 25: `updateActiveSymmetriesSym` extends function `updateActiveSymmetrie` of algorithm 6. The reason clause, ω_l, of each propagated literal, l, by the `unitPropagatior` function is analyzed. Each symmetry of the complement set (to $S(\varphi)$) of the set local symmetries of ω_r is marked *inactive*, if it is not already marked so. l becomes responsible of this deactivation.

Implementation

We have implemented our combo on top of the `minisat-SPFS`[4] solver, developed by the authors of SPA. This choice has been influenced by two points: (1) take advantage of the expertise used to implement the original SPA method; (2) the easiness of integrating our implementation of ESBA to any CDCL-like solver because it is an off-the-shelf library[5]. However, this choice has the drawback of doubling the representation of symmetries. This

[2] These are clauses of the *conflict side* of the implication graph when applying the classical conflict analysis algorithm.

[3] The only allowed local symmetries in case of an esbp, according to point 2 of section **??**.

[4] https://github.com/JoD/minisat-SPFS

[5] This library is released under GPL v3 license, see `https://github.com/lip6/cosy`.

Benchmark	minisat-Sp	minisat-Sym	minisat-SymSP
Generators 0–20 (704)	194	197	**198**
Generators 20–40 (136)	33	**34**	**34**
Generators 40–60 (141)	28	28	**29**
Generators 60–80 (168)	**65**	64	**65**
Generators 80–100 (51)	28	**34**	**34**
Generators >100 (200)	58	59	**60**
TOTAL no dup (1400)	406	416	**420**

Table 5.1: Comparison of the number of SAT problems solved by each approach.

Benchmark	minisat-Sp	minisat-Sym	minisat-SymSP
Generators 0–20 (704)	**233**	220	226
Generators 20–40 (136)	50	**54**	**54**
Generators 40–60 (141)	75	**83**	**83**
Generators 60–80 (168)	**11**	**11**	10
Generators 80–100 (51)	**11**	**11**	**11**
Generators >100 (200)	90	**109**	107
TOTAL no dup (1400)	470	488	**491**

Table 5.2: Comparison of the number of UNSAT problems solved by each approach.

can be a hard limit to treat certain big problems from the memory point of view. The implemented combo solver can be found at:
`https://github.com/lip6/minisat-SymSp`

Evaluation

This section compares our combo approach against ESBA and SPA. All experiments have been performed with a modified version of the well-known `MiniSAT` solver [14]: `minisat-Sp`, for SPA; `minisat-Sym`, for ESBA; and `minisat-SymSP`, for the combo. Symmetries of the SAT problems have been computed by `bliss` [19]. We selected from the last seven editions of the SAT contest [18], the CNF problems for which `bliss` finds some symmetries that could be computed in at most 1000s of CPU time. We obtained a total of 1400 SAT problems (discarding repetitions) out of the 4000 proposed by the seven editions of the contest. All experiments have been conducted using the following settings: each solver has been run once on each problem, with a time-out of 7200 seconds (including the execution time of symmetry generation) and limited to 64 GB of memory. Experiments were executed on a computer with an Intel® Xeon® Gold 6148 CPU @ 2.40 GHz featuring 80 cores and 1500 GB of memory, running a Linux 4.17.18, along with g++ compiler version 8.2. Tables 5.1 and 5.2 present the obtained results for SAT and UNSAT problems respectively. The first column of each table lists the classes of problems on which we operated our experiments: we classify the problems according to the number of symmetries they admit. A line noted "generators X-Y (Z)" groups the Z problems having between X and Y generators (i.e., symmetries). Other columns show the number of solved problems for each approach. Globally, we observe that the combo approach can be effective in many classes of symmetric problems. For SAT problems, the combo has better results than the two other approaches (4 more SAT problems when compared to the best of the two others) and this is despite the significant cost paid for the tracking of the symmetries' status. When looking at the UNSAT problems, things are more mitigated. Although, the total number of solver problems is greater than the best of the two others, we believe that the cost for tracking the symmetries' status has an impact on the performances. This can be observed on the first and last lines of Tables 5.2: when the number of generators is small (first line), the ESBA benefits greatly from the SPA. When the number of generators is high (last line), we see a small loss of the combo with respect to ESBA. It is also worth noting that the combo approach solved **8** problems that could not be handled by ESBA nor SPA. Table 5.3 compares the different techniques with respect to the PAR-2 and the CTI time measures. PAR-2 is the official ranking measure used in the yearly SAT contests [18]. CTI measures the Cumulative solving Time of the problem Intersection (i.e., 825 problems solved by all solvers). While PAR-2 value gives a global indication on the effectiveness of an approach, CTI is a good mean to evaluate its speed compared to other approaches. Hence, we observe that the combo has a better PAR-2 score, and this shows its effectiveness. However, it is the least fast when coming to solved intersection. This is clearly due to the double cost paid for tracking the symmetries' status (one for ESBA and the other for SPA). Having a unified management of symmetries tracking would probably reduce this cost.

Solvers	PAR2 (1400)	CTI (825)
`minisat-SymSp`	5,653,089	614,856
`minisat-Sym`	5,682,892	584,868
`minisat-Sp`	6,026,840	612,638

Table 5.3: Comparison of PAR-2 and CTI times (in seconds) of the global solving.

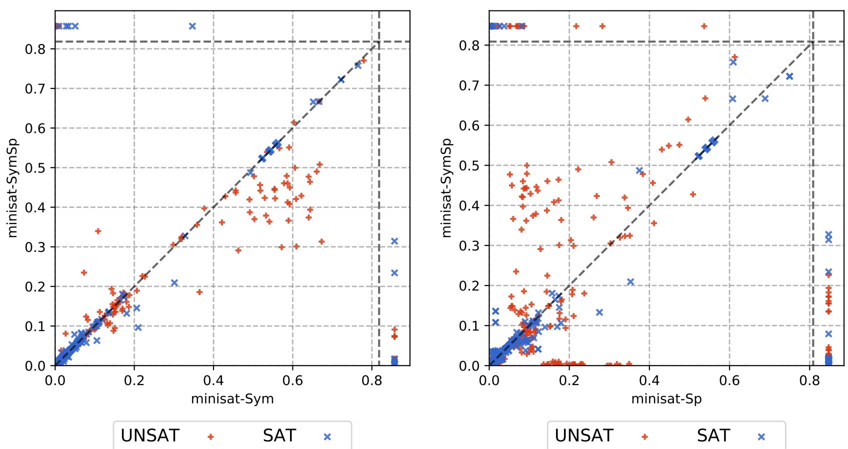


Figure 5.1: Comparison of the ratio between the number of decisions and the number of propagation for the combo w.r.t. ESBA and SPA.

To go further in our analyze, we also compare the ratio between the number of decisions and the number of propagation. This is a fair measure to assess the quality of a SAT solving approach: if the ratio is small, then this means that the developed algorithm is producing more deduced facts than making guesses, which is the best way to conclude quickly on a problem! The scatter plots of Fig.5.1 show a comparison between the aforementioned ratios. When comparing `minisat-Sp` to `minisat-SymSp` (right hand side scatter plot), we observe that the ratio goes in favor of `minisat-Sp` for the problems solved by both approaches. This is an expected result since the main objective of SPA is to minimize the number of decisions while augmenting the number of propagation. What is important to underline here is highlighted on the left hand side scatter plot: on a large majority of UNSAT problems, the ratio goes in favor of `minisat-SymSp` w.r.t. `minisat-Sym`. This confirms the positive impact of SPA when applied in conjunction with ESBA.

5.2 Another combo approach

Introducing local symmetries allow us to use it with another dynamic approach : *Symmetry Explanation Learning* (SEL). It computes the symmetrical learning clause and include it in the solver clauses when it can be used in the unit propagation or leads to a conflict. With local symmetries, each clause has a set of allowed symmetries, when a symmetry wants to add a symmetrical clause, it suffices to check if this permutation belongs to local symmetries. Hakan: Naive approach no stab .

5.3 Exploitation of local symmetries

Local symmetries seen previously allows to exploit symmetries in different ways. Effectively, we can consider symmetries on a part of the problem. Only clauses that are really used by the solver i.e in the implication graph can be processed. To obtain the symmetries of the current sub problem is the intersection of local symmetries.

CHAPTER 6

Conclusion And Future Works

In this thesis, we have presented a way to increase performance of solving Boolean satisfiability problem (SAT) on presence of symmetries.

Nowadays, SAT solvers can handled huge problems with thousands of variables and clauses. [redacted] It is primary due to efficiently cut off search space.

studies of detection and exploitation of symmetry breaking techniques.

Symmetries are

6.1 Perspectives

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] B. Benhamou, T. Nabhani, R. Ostrowski, and M. R. Saidi. Enhancing clause learning by symmetry in sat solvers. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, volume 1, pages 329–335. IEEE, 2010.

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

[7] 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.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] N. Eén and N. Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[15] 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.

[16] 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.

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

[18] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon. The international sat solver competitions. *AI Magazine*, 33(1):89–92, 2012.

[19] 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.

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

[21] H. A. Kautz, B. Selman, et al. Planning as satisfiability. In *ECAI*, volume 92, pages 359–363, 1992.

[22] 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.

[23] 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.

[24] 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.

[25] E. Luks and A. Roy. The complexity of symmetry-breaking formulas. *Annals of Mathematics and Artificial Intelligence*, 41(1):19–45, 2004.

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

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

[28] H. Metin, S. Baarir, M. Colange, and F. Kordon. Cdclsym: Introducing effective symmetry breaking in sat solving. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 99–114. Springer, 2018.

[29] 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.

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

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

[32] T. J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 216–226. ACM, 1978.

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

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

[35] 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.