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 : 18 DÉCEMBRE 2019

DEVANT LE JURY COMPOSÉ DE :

Rapporteurs :	PASCAL FONTAINE	Maître de conférences, Université de Lorraine
	LAURE PETRUCCI	Professeur, Université Paris 13
Examinateurs :	JEAN-MICHEL COUVREUR	Professeur, Université d'Orléans
	EMANUELLE ENCRENAZ	Maître de conférences, Sorbonne Université
	BART BOGOERTS	Assistant Professor, Vrije Universiteit Brussel
Directeurs :	SOUHEIB BAARIR	Maître de conférences, Université Paris Nanterre
	FABRICE KORDON	Professeur, Sorbonne Université

À Seref, Keziban, Didem et Ilayda.

Résumé Long

Cette thèse traite la résolution du problème de satisfaisabilité booléenne (SAT). SAT permet de résoudre des problèmes importants dans différents domaines tels que la planification [33], la biologie [42], la vérification de logiciel et de matériel [8], de raisonnement automatique [27] etc. L'evolution des eurs évolutions au cours des dernières décennies leur ont permis de traiter des problèmes de plus en plus complexes. Des travaux récents ont réussi a prouver à l'aide d'un solveur SAT, une borne maximum pour le problème de coloration des triplets pythagoriciens, avec une preuve de 200 TB [27].

Étant donné un problème SAT, l'objectif est de déterminer si il est possible de satisfaire toutes les contraintes du problème, si c'est le cas on dit que le problème est satisfaisable, si non il est insatifaisable. Ce calcul est effectué par un solveur SAT qui répond SAT lorsque la formule est satisfaisable et UNSAT dans le cas contraire. SAT a été le premier problème ayant été prouvé NP-Complet [13]. Cela signifie que l'on ne connait pas d'algorithme capable de résoudre le problème avec une complexité polynomiale.

Malgré cette complexité, les solveurs SAT sont capables de résoudre de plus en plus de problèmes complexes. Ce succès vient de l'introduction de différentes heuristiques sophistiquées et de l'algorithme de résolution utilisé, « Conflict Driven Clause Learning» (CDCL). Il est basé sur l'algorithme DPLL nommé suivant ses auteurs Davis, Putnam, Logemann et Loveland [16], l'un des premier algorithme avec une utilisation non intensive de la mémoire.

Le CDCL peut être vu suivant l'exploration d'un arbre binaire avec au total 2^n branches, avec n le nombre de variables du problème. La première étape consiste à choisir une variable dite de décision puis, d'effectuer les déductions logiques à partir de l'affection. Si l'algorithme se trouve dans une situation de conflit, c'est-à-dire, que l'affectation courante n'est pas capable de satisfaire au moins une contrainte du problème, l'algorithme va calculer une clause dite de conflit qui permet d'élaguer cet espace de recherche. Avec cette contrainte le solveur effectue un *backjump*, c'est à dire, qu'il remonte dans l'arbre de recherche pour explorer une autre branche. Si aucun conflit n'est présent, l'algorithme choisit à nouveau une variable de décision. L'algorithme termine soit lorsque toutes les variables sont affectées auquel cas le problème est satisfaisable, soit lorsque un conflit est apparu avec uniquement des déductions logiques auquel cas le problème est non satisfaisable. La Figure 1 présente sous forme d'un organigramme le fonctionnement de l'algorithme CDCL.

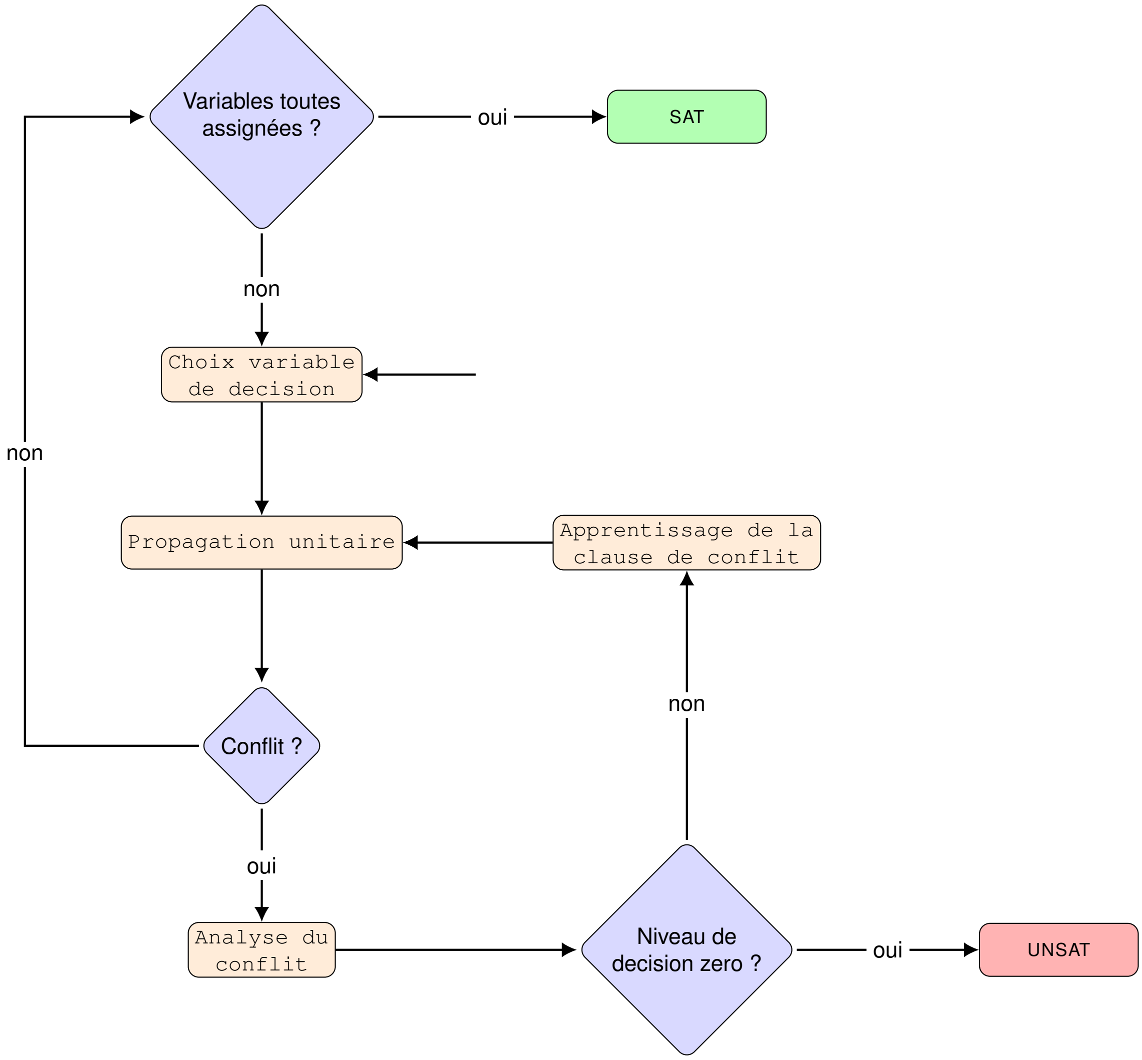


Figure 1: Organigramme de l'algorithme CDCL

Certains problèmes présentent des symétries, dans ce cas, des branches de l'espace de recherche sont identiques à une permutation près, on dit que cet espace de recherche est isomorphe. Si une branche de l'espace de recherche est solution du problème alors toutes les branches symétriques sont également des solutions. Dans le cas inverse si il n'existe pas de solution dans une branche alors il en existe pas dans aucune des branches symétriques. En parcourant les espaces isomorphe, le solveur effectue donc du travail inutile.

Pour illustrer ce problème, prenons comme exemple le problème des tiroirs (*pigeonhole problem*) dans lequel nous avons un ensemble de pigeons qui doivent êtres attribués à des nids différents. Dans ce problème il y a un nid de moins que le nombre de pigeons. Le but de ce problème est de déterminer s'il est possible d'attribuer un nid différent à chaque pigeon. La figure 2 présente une instance de ce problème.

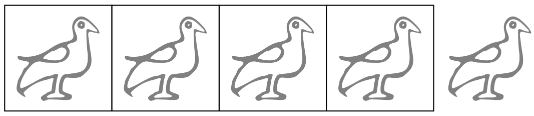


Figure 2: Représentation graphique d'une instance du problème des pigeons (5 pigeons, 4 nids)

Pour un humain, la réponse à ce problème est évidente, mais un solveur de l'état de l'art va parcourir toutes les combinaisons possibles de couples (pigeon, nid) et cela le mène à une explosion combinatoire. Pour cette raison, résoudre ce problème avec un solveur SAT standard s'avère très chronophage et même impossible dans des temps raisonnables pour un nombre de pigeons supérieur à 15.

De manière plus générale, une symétrie est une transformation qui laisse un objet (ou un aspect de l'objet) inchangé. Les symétries sont généralement définies comme une propriété syntaxique d'un problème lorsque leur présence est inhérente à l'encodage du problème. Dans ce cas, une permutation des variables préserve la spécification originale du problème. Dans le cas où les symétries sont indépendantes d'une représentation particulière du problème, il s'agit de symétries sémantiques.

La présence de symétrie dans un problème force l'algorithme de recherche à explorer en vain l'espace de recherche symétrique et entrave considérablement ses performances. La rupture de symétrie est une approche qui évite au solveur de visiter les espaces de recherche isomorphes.

Pour pouvoir exploiter les symétries, la première étape consiste à les trouver. Dans le contexte de la satisfaction booléenne, la détection des symétries syntaxiques se fait tout d'abord par la transformation de la spécification en un graphe coloré et ensuite à l'application d'un outil d'automorphisme de graphe sur celui-ci. Différents outils traitent de ce problème dans l'état de l'art, tel que, `bliss` [30], `saucy3` [31], …

Lorsque les symétries sont obtenues, la façon la plus courante pour les exploiter est d'utiliser une approche de rupture de symétrie statique. Elle est dite statique, car le traitement de cette approche est effectuée avant la résolution du problème SAT. Ceci consiste à prendre le problème symétrique en entrée et à produire une formule à satisfaction équivalente, en éliminant les symétries présentes. Pour produire une formule équivalente sans présence de symétries, le problème est augmenté par des contraintes de rupture de symétries ou *symmetry breaking predicates (sbp)* en anglais. Celles-ci empêchent le solveur d'explorer les espaces de recherche isomorphes. Si l'on considère l'exemple précédent avec les pigeons, le premier pigeon est attribué a un exactement un nid, la symétrie est alors «rompue».

Plusieurs outils, tels que `Shatter` [1] et `BreakID` [18], utilisent cette technique pour accélérer le calcul du solveur en présence de symétries. En général, cette approche aboutit à de bons résultats dans différentes instances symétriques, cependant elle possède des défauts. Parmi eux, nous pouvons citer le nombre de contraintes ajoutées qui peut être exponentiel par rapport à la taille du problème. Ceci a pour conséquence de ralentir l'algorithme principal du solveur. De plus, étant donné que ce calcul est effectué avant le lancement du solveur, cette approche peut être difficilement combinée avec d'autre technique de rupture de symétries. Aussi, cette approche ne peut pas distinguer les contraintes originales du problème avec les contraintes de rupture de symétrie. Avec cette information, le solveur peut modifier ses heuristiques pour augmenter ses performances. Pour ces raisons, certains problèmes avec de nombreuses symétries ne peuvent pas être traités efficacement avec cette approche.

Une autre approche dite rupture de symétrie dynamique consiste à utiliser les symétries durant l'algorithme de recherche du solveur SAT, plus précisément, à modifier son comportement pour exploiter les propriétés de symétrie du problème. Cette approche consiste à déduire des faits symétriques par rapport aux déductions effectuées par le solveur. Lorsque ces faits sont ignorés, le solveur va explorer en vain l'espace de recherche symétrique. Ces déductions réduisent le nombre de décisions, qui sont des suppositions du solveur choisies de manière heuristique et, augmentent le nombre de propagations qui sont les déductions logiques faites par le solveur. Cette approche transforme donc les suppositions du solveur en déductions logiques. Différents outils utilisent cette approche, nous pouvons citer, *Symmchaff* qui n'exploite que certains types spécifiques de symétries, *Symmetry Propagation (SP)*, *Symmetry Learning Scheme (SLS)* et *Symmetry Explanation Learning (SEL)* qui ajoutent les symétriques des clauses apprises pour permettre les déductions symétriques. Étant donné que cette approche est dynamique, il est possible pour le solveur d'intégrer des heuristiques spécifiques ou encore de combiner les différentes techniques de rupture de symétrie.

Cette thèse vise à améliorer l'existant et rendre les solveurs plus performants en présence de symétries et propose différentes contributions allant dans ce sens.

Notre première contribution mime le comportement de la rupture de symétrie statique, mais opère dynamiquement, pendant l'exécution du solveur. On ajoute dans le solveur un composant de symétrie opportuniste qui va détecter que le solveur parcours un espace de recherche symétrique et va ajouter à celui-ci une contrainte appelée « effectective symmetry breaking predicate » qui va empêcher le solveur de rester dans l'espace de recherche en question. Cela a pour conséquence de réduire le nombre de contraintes et donc ne ralentit pas les solveur.

Ce composant de symétrie est fourni sous forme d'une bibliothèque codée en C++ et se nomme cosy. Elle peut s'interfacer avec n'importe quelle solveur de type CDCL. Pour conduire nos expériences, nous l'avons interfacé avec un solveur de l'état de l'art nommé MiniSAT [22]. Au total, l' intégration de cosy ajoute environ 60 lignes de code et augmente le code de MiniSAT de 3%. cosy est « open source», fournit sous une licence GPLv3 et est disponible sur Github [1].

Pour évaluer notre approche, nous avons comparé une version du solveur MiniSAT combiné avec la bibliothèque cosy, que nous avons appelé MiniSym avec les solveurs de l'état de l'art. Ces expérimentations se sont effectuées sur les instances de la SAT Competiton [29] sur les six dernières années de 2012 à 2017 où bliss a réussi a trouver des symétries. Au total, nous avons obtenu 1350 instances.

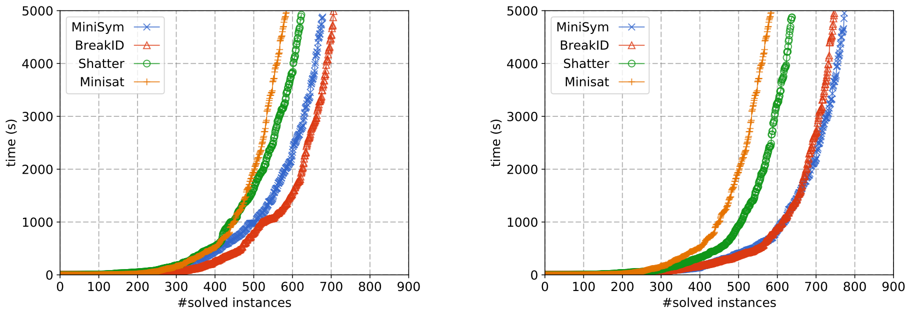
La figure 3 nous montre les résultats obtenus sous forme d'un cactus plot, dans lequel l'axe des abscisses nous montre le nombre d'instances réussies par chacun des solveurs et l'axe des ordonnées nous montre le temps de calcul des solveurs. Nous avons conduit ces expériences avec deux outils d'automorphisme de graphe, à savoir saucy3 à gauche de la figure et bliss à droite de la figure.

Comme nous pouvons le constater, les résultats sont meilleurs avec l'utilisation de l'outil d'automorphisme de graphe bliss. La principale différence entre les deux outils est le nombre de permutations obtenues. bliss nous donne plus de permutations que saucy3, cela permet a notre outil MiniSym d'atteindre un nombre d'instances résolues de 775 alors que le deuxième meilleur outil BreakID atteint un nombre d'instances résolues de 749 avec bliss. Toutes les expériences étaient limitées à une durée à 5000 secondes.

Ces expériences nous démontrent que notre approche est aussi performante que les approches de rupture de symétrie statique.

Malgré les très bons résultats obtenus par notre approche, certains problèmes qui sont résolus très rapidement par les approches de rupture de symétrie dynamique tel que Symmetry Propagation (SP) ne pouvaient toujours pas être traité par notre approche et vise-versa.

[1] https://github.com/lip6/cosy



(a) with `saucy3`

(b) with `bliss`

Figure 3: cactus plot total number of instances

SP est une approche qui a pour but d'accélérer la traversée de l'espace de recherche en déduisant des faits symétriques à partir des déductions effectuées par le solveur. À l'inverse notre approche consiste à éliminer les espaces de recherche symétriques. Notre deuxième contribution consiste à déterminer si cette combinaison est possible. Elle se résume donc à la question suivante: Est-il possible d'accélérer la traversée de l'espace de recherche tout en éliminant les espaces de symétriques ?

Pour que cette approche soit correcte, la contrainte qu'il faut absolument respectée est que la symétrie utilisée pour déduire les faits symétriques doit être valide dans le problème. En effet, les clauses ajoutées pour éliminer l'espace de recherche symétrique vont rompre cette symétrie. Cette dernière ne peut donc plus être utilisée pour propager les faits symétriques. Une approche naïve est de supprimer chacune des symétries dès lors qu'une contrainte de rupture de symétrie a été ajoutée. Le problème avec cette approche est que l'ensemble vide est très vite atteint et donc que plus aucune déduction symétrique ne peux être faites. Notre approche consiste à traquer les clauses utilisées par le solveur et à connaître à tout instant l'ensemble des symétries valide grâce à l'introduction dite de la notion de symétrie locale pour chaque clause.

Les expérimentations pour évaluer les performances de l'approche se sont effectuées sur les instances de la SAT Competiton sur sept années (de 2012 à 2018) où `bliss` a trouver des symétries. Au total, nous avons obtenue un total de 1400 instances. Les tables 1 et 2 présentent respectivement les résultats obtenues four les problèmes SAT et UNSAT. La première colonne de chaque tableau énumère les classes de problèmes sur lesquelles nous avons effectué nos expériences: nous classons les problèmes en fonction du nombre de symétries qu'ils admettent. Une ligne notée "permutations X-Y (Z)" regroupe les

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

Table 1: Comparaison des approches sur les instances SAT.

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

Table 2: Comparaison des approches sur les instances UNSAT.

problèmes Z ayant entre X et Y générateurs (symétries). Les autres colonnes indiquent le nombre de problèmes résolus par chaque approche. Les trois solveurs comparés sont respectivement `minisat-Sp`, le solveur `MiniSAT` avec l'approche SP; `minisat-Sym` le solveur `MiniSAT` avec cosy et `minisat-SymSP` le solveur avec l'approche combiné.

Globalement, nous observons que l'approche combinée est efficace dans de nombreuses classes de problèmes symétriques. Pour les problèmes de SAT, la combinaison a de meilleurs résultats que les deux autres approches (4 problèmes de SAT en plus par rapport au meilleur des deux autres). Lorsqu'on examine les problèmes de l'UNSAT, les résultats sont plus mitigées. Cela est du au coût mis en place pour maintenir à jour les structures de chacune des deux approches de manière indépendante. Cependant l'approche combinée apporte de meilleure résultats dans le nombre total d'instances résolues.

Abstract

Nowadays, logic is omnipresent and it is used in different domains such as logic optimization, test pattern generation, formal verification and functional simulation, etc. One method to solve this kind of problem is satisfiability problem (SAT). SAT solvers are more and more powerful and can handle large problems which seemed to be infeasible few years ago. However, some problems present symmetries which force the solver to explore fruitlessly the symmetric part of the search space and hinders the performance. In this thesis, we set out to exploit symmetry properties of the problems in better ways. For this purpose, we propose two major contributions that aims to improve the state-of-the-art techniques and augment the number of solved instances. With an evaluation over the instances presented in the SAT competition, we show that our approach overcomes the state-of-the-art ones and is able to solve more instances.

Contents

CHAPTER 1

INTRODUCTION

The interest of using computers for logic deduction and reasoning can be traced in the nineteen centuries. In 1869, William Stanley Jevons designed and built the first machine doing logic inference. With the progress of computers, logic is used in different domains such as design automation process (logic optimization, test pattern generation, formal verification, functional simulation, etc). Nowadays, one of the methods used in Boolean reasoning is the automatic satisfiability (SAT). Given a propositional formula (generally the constraints of an encoded problem), SAT solving consists in deciding whether the formula is satisfiable (i.e., all constraints can be satisfied) or unsatisfiable (i.e., there is no way to satisfy all constraints at the same time). This computation is made by a SAT solver that answer SAT when the formula is satisfiable and UNSAT otherwise. SAT is the first problem that has been proven to be NP-complete in 1971 [13], this means that every NP problem can be solved by encoding it into a SAT one. Solving this problem in polynomial time is equivalent to the P versus NP, one of the seven millennium prize problems.

Despite this complexity, SAT solvers are becoming more and more powerful. Over the last decades, these can handle more and more complex problems in different domains: like *formal methods* such that: bounded model checking (BMC) [9]; *artificial intelligence*: planning decision [33]; *Bio informatics* : Haplotype inference [42]. In recent work, researchers have succeeded in proving, using a SAT solver, a maximum limit for the problem of coloring Pythagorean triples [27], with proof weighing 200 TB. The success of SAT comes from the introduction of sophisticated heuristics and optimization of the solving algorithm called

Conflict Driven Clause Learning (CDCL) algorithm [43]. It is based on the first non memory intensive algorithm named by its authors Davis, Putnam, Logemann, and Loveland (DPLL) [16].

Nevertheless, some problems have a huge search space and some of their instances cannot be handled. An example of such a problem can be the vehicle routing problem (VRP). It concerns the service of a delivery company, in which given a fleet of vehicles based in a depot, they must make rounds between several customers who have requested each a certain amount of goods. All clients visited by a vehicle refers to the tour of the vehicle. The goal of this problem is to find the tour that minimizes the delivery cost with different criteria monetary, distance, time, ... Finding the optimal solution for VRP problem is NP-Hard [55]. When we look more in detail on an instance of this problem, renaming the set of identical vehicles will give us exactly the same problem, this is called a *symmetry*. More precisely, a symmetry is a transformation that leaves an object (or some aspect of the object) unchanged. Symmetries are typically defined as a *syntactical* property of a problem when its presence is inherent to the encoding of the problem and so a permutation of variables preserves the original specification. In the case where symmetries are independent of any particular representation of the problem, we speak of *semantic* symmetry.

The presence of symmetries in a problem leads the search algorithm to fruitlessly explore symmetric search spaces and greatly hinders its performance. The approach that avoids the solver to visit these symmetrical search spaces is called *Symmetry breaking*. But to exploit symmetries, it is still necessary to find them. To achieve this in SAT, the detection of syntactical symmetry is done by transforming the specification in a colored graph and then apply a graph automorphism tool.

When symmetries are computed, the most common approach to exploit them is to use a *static symmetry breaking* technique. It takes the symmetric problem as input and produces a satisfiability equivalent formula by eliminating symmetries. This is done by augmenting the problem with constraints that force the solver to not explore the symmetric search spaces. This approach is a easy to integrate static symmetry breaking, no modification of the solver is necessary. In addition, this approach works well on many symmetric applications. However, some highly symmetric instances cannot be solved using this technique. Effectively the number of symmetry breaking constraints can be exponential in relation to the size of the problem and its presence slow down the solver instead of increasing them.

There are also another approach to handle symmetry called *dynamic symmetry breaking*. Here, the management of symmetries is done during the search and different approaches exist. First, the behavior of the solver is analyzed to avoid it to visit symmetric part of the search space and thus accelerates the resolution of the problem. Second, under some conditions some symmetrical facts can be deduced through symmetry from the state of the solver. This has the effect of accelerating the tree traversal of the solver and reduce the solving time.

This thesis addresses the challenge of optimizing the solving of a SAT problem in presence of symmetries. In detail, my research exploits symmetry breaking during the solving. It provides two major contributions. The first one uses the strengths of static symmetry breaking approach and applied it dynamically to avoid the drawbacks of the approach. It adds an opportunistic symmetry controller that avoids visiting symmetric part of the search spaces. Benchmarks show that this makes it possible to solve very difficult symmetric problems. The second contribution uses the previous one and combines it with state-of-the-art dynamic symmetry breaking approach and so takes the best of two worlds. This combination leads to important theoretical step for the usage of *partial symmetry breaking* with the usage of *local symmetries*.

The remaining of this document is organized in 6 chapters. Chapter 2 describes the state-of-the-art for the Boolean satisfiability problem, Chapter 3 focuses on the symmetry present in SAT. Chapter 4 focuses on the first contribution that uses dynamically the symmetries. Chapter 5 describes our second proposal and Chapter 6 conclude the thesis. More precisely:

The Boolean Satisfiability Problem The goal of Chapter 2 is to better understand what is SAT. It describes in detail the basics about propositional logic that will be used in the rest of the manuscript. Satisfiability is a hard problem but some particular forms that are easier to solve are presented such as 2-SAT, Horn SAT and Xor-SAT. This chapter also describes the original solving algorithm called DPLL, and the nowadays used one called Conflict Driven Clause Learning algorithm (CDCL). This last can handle sophisticated problems, thanks to different heuristics, an overview of which will be presented. Finally, with the presence of multi core machines, an overview of the state-of-the-art parallel SAT solving are presented.

Symmetries and SAT The goal of Chapter 3 is to better understand what is a symmetry and its usage in the SAT context. For this purpose, we first present group theory and the notation used in the rest of the manuscript. This chapter also presents the process to find the (syntactic) symmetries of a SAT problem. This computation involves the creation of a graph from the problem and the computation of an automorphism tool. After obtaining the symmetries, the second part presents how to exploit them for reducing the search space of the solver. The two major approaches that are the static symmetry breaking approach and the dynamic symmetry breaking approach. Static symmetry breaking is so far the most popular approach to take advantage of symmetries. It relies on a symmetry preprocessor which augments the initial problem with constraints that force the solver to consider only a few configurations among the many symmetric ones. Dynamic symmetry breaking exploits the symmetries during the computation of the SAT solver to accelerate the tree traversal of the SAT solver using symmetrical facts or avoids symmetric configurations like in the static approach.

Between Static and Dynamic Chapter 4 describes our efficient dynamic symmetry breaking approach. The first part explains our algorithm, a new way to handle symmetries that avoid the main problem of the current static approaches. Our proposal has been implemented in state-of-the-art SAT solver called `MiniSAT` [22]. The second part presents the extensive experiments on the benchmarks of last six SAT competitions, which show that our approach is competitive with the best state-of-the-art static symmetry breaking solutions. The last part presents different heuristics that can improve the performance of our algorithm.

Compose dynamic symmetry handling Chapter 5 describes the theoretical and practical aspects of combining two existing symmetry breaking approach with the introduction of *local symmetries*. An extensive experiments show that the hybrid approach is better than each approach individually. The local symmetries allows to combine another symmetry breaking approach. Finally, Chapter 6 concludes this manuscript and discusses different directions we have identified for future works.

Part I

State-of-the-art

CHAPTER 2

THE BOOLEAN SATISFIABILITY PROBLEM

Contents

In this thesis, our goal is to exploit the symmetry properties of SAT problems. Before, we get to the heart of the matter, we first introduce the Boolean satisfiability (SAT) problem.

2.1 SAT basics

The goal of SAT is to determine whether a propositional formula is satisfiable (i.e, all constraint can be satisfied) or unsatisfiable (i.e there us no way to satisfy all constraint at the same time). The formula is constituted of *Boolean* or *propositional variables*, i.e. each variable 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 $\mathcal{V}_\varphi$ (respectively $\mathcal{L}_\varphi$) the set of variables (respectively literals) used in the formula (the index in $\mathcal{V}_\varphi$ and $\mathcal{L}_\varphi$ is usually omitted when clear from context). To build complex formulas, it is sufficient to use, $\neg, \vee$ and $\wedge$ which are respectively negation, disjunction and conjunction. The remaining operators like, $\Rightarrow, \Leftrightarrow$ and $\oplus, \cdots$ expressed using combinations of the basic ones. For example, $a \Rightarrow b$, can be expressed by $\neg a \vee b$. Every binary operator adds a pair of parentheses to define explicitly the semantic of the formula. In the absence of parentheses, the following priority order applies (from the highest to the lowest priority): negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$). An *assignment*, noted α, is defined as the function that assigns a value to each variable of φ.

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

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 is set to false value and x_3 is set to 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 the negation ($\neg$), the conjunction ($\wedge$), and the 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

A formula is said to be *satisfiable* (SAT) if there is at least one assignment that satisfies it; otherwise the formula is *unsatisfiable* (UNSAT). In order to compare different formulas, we define here the concepts of logical equivalence and logical consequence respectively defined in Definition 2.1 and Definition 2.2.

Definition 2.1: Logical equivalence

Two formulas φ and ψ are equivalent iff every assignment α that satisfies formula φ also models the formula ψ and vice versa, denoted by $\varphi \equiv \psi$.

Definition 2.2: Logical consequence

A formula ψ is a *logical consequence* of a formula φ if every model of φ is also a model of ψ and is denoted by $\varphi \models \psi$.

Normal forms

In Boolean logic, there are some particular form of formula, called *normal form*. To introduce some of them, we first need to present the concepts of *cube* and *clause*.

Definition 2.3: Cube

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

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

Definition 2.4: Clause

A *clause* ω is a finite disjunction of literals represented equivalently by:

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

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. Clauses have the following property that can be exploited to simplify the formula. When a clause ω_1 is a subset of another clause ω_2, noted $\omega_1 \subset \omega_2$, we say that ω_1 subsumes ω_2. And any assignment that satisfies ω_1 will also satisfy ω_2. So, ω_2 is *redundant w.r.t.* can be removed from the formula.

Definition 2.5: Conjunctive Normal Form

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

$$\varphi = \bigwedge_{i=1}^{k} \omega_i \text{ (or by the set of its clauses } \varphi = \{\omega_i\}_{i\in[\![1,k]\!]})$$

Definition 2.6: Disjunctive normal form

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

$$\varphi = \bigvee_{i=1}^{k} \gamma_i \text{ (or by the set of its cubes } \varphi = \{\gamma_i\}_{i\in[\![1,k]\!]})$$

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

Associativity laws	$(x \vee y) \vee z \equiv x \vee (y \vee z)$ $(x \wedge y) \wedge z \equiv x \wedge (y \wedge z)$
Commutativity laws	$x \vee y \equiv y \vee x$ $x \wedge y \equiv y \wedge x$
Identity laws	$x \vee \perp \equiv x$ $x \wedge \top \equiv x$
Domination laws	$x \vee \top \equiv \top$ $x \wedge \perp \equiv \perp$
Idempotent laws	$x \vee x \equiv x$ $x \wedge x \equiv x$
Distributive laws	$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$ $x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$
Negation laws	$x \vee \neg x \equiv \top$ $x \wedge \neg x \equiv \perp$
double negation law	$\neg(\neg x) \equiv x$
De Morgan's laws	$\neg x \vee \neg y \equiv \neg(x \wedge y)$ $\neg x \wedge \neg y \equiv \neg(x \vee y)$

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 satisfiability *equivalent*. Conjunctive normal form is the input form of state-of-the-art SAT solvers. Any propositional formula can be transformed in CNF form with polynomial time [50]. Conversely, DNF form has an exponential memory complexity during the transformation [15]. Note that each cube in the problem in DNF form is a solution in the equivalent CNF formula.

An NP-complete problem

The SAT problem is the first NP-complete problem proven by Stephen Cook in 1971 [13]. 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 to be NP-hard (non-deterministic polynomial-time hard) if any problem can be reduced to this problem in polynomial time. If someone finds an algorithm that solves a SAT with a polynomial time algorithm, this answer one of the most important unsolved problems in theoretical computer science: the P versus NP problem, that is also one of the seven millennium prize problems [11].

Some easy to solve forms

Some particular instances of the SAT problem can be computed in polynomial time.

2-SAT [4]. In this particular form, the given CNF formula contains only binary clauses. In this case, a graph in which each clause is transformed into implication is created. For example, the clause $x \vee y$ will be transformed into $\neg x \Rightarrow y$ and $\neg y \Rightarrow x$. Then, a *strong connected component* (SCC) is computed to determine the satisfiability of the formula. If the same variable is present in both its positive and negative values in the same SCC then the formula is declared unsatisfiable, otherwise a solution can be deduced and so the problem is satisfiable. This algorithm can be computed in linear time complexity.

Horn SAT. [20] In this particular form, the given CNF formula contains only Horn clauses. There are three forms of Horn clauses:

- *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, in section 2.1 until a fix point is reached. Roughly speaking, it satisfies all unit clauses in cascades. At the end of the procedure, either an empty clause is deduced and the problem is declared UNSAT or the fix point is reached and the formula is declared SAT. Like 2-SAT, this algorithm can also be computed in linear time complexity.

XOR SAT. [47] 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 is an algorithm that allows to solve this kind of problem is polynomial time, more exactly in $O(n^3)$.

Some related problems

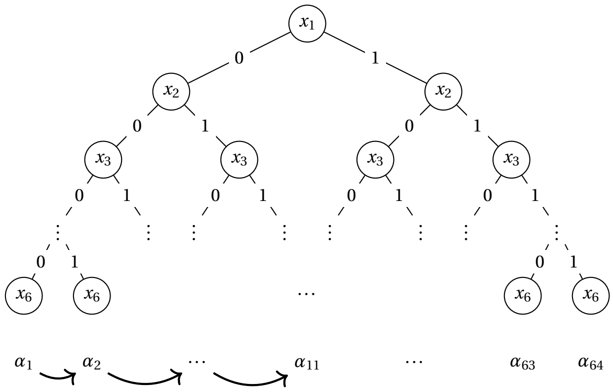
A different kind of problems are related to SAT. One of them is sharp-SAT (#SAT) [56], its purpose is to count the number of satisfiable assignments in a formula. Another related problem is *maximum satisfiability problem* (MAX-SAT) [10]. In this case, the problem is to find the maximum subset of clauses 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 [24].

Solving a SAT problem

Two kinds of algorithms exist to solve satisfiability problems. First, the *incomplete* algorithms [32] which do not provide any guarantee that they will eventually report, either a satisfiable assignment or declare the formula unsatisfiable. This kind of algorithm is out of scope of this thesis. Second, the *complete* algorithms, which provide a guarantee that if an assignment exists, it will be found or 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, we have to verify, in the worse case, 2^n assignments. The algorithm first tries an assignment, for example all literals are set to false and then if the formula is not satisfiable, it tries other assignments. This algorithm is finished when either a satisfiable assignment is found and so the formula is declared SAT or all assignments are not satisfying the formula and so the formula is declared UNSAT. Figure 2.1 illustrates the search tree for a given problem with six variables. The formula presented in the figure has 6 clauses, with 2 ternary clauses and 4 binary clauses. This formula is SAT, the assignment $\alpha_{11} = \{\neg x_1, \neg x_2, x_3, \neg x_4, x_5, \neg x_6\}$ is a solution of the problem. A naive algorithm will check 10 assignments before finding the solution. In the general case, due to the number of variables in problems, this algorithm will be 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) [16]. It explores a binary tree using depth first search, as shown in Algorithm 1. The construction of the tree relies on a *decision* variable that is chosen on line 8. Both value of this variable is checked, the true value on line 9 and the false value on line 11. When a leaf of the tree is inconsistent (i.e. a variable needs to be set to true and false at the same time), called a *conflict* (line 5), the opposite value of the decision is explored. Recursively, when both value of a variable reach a conflict, the solver *backtracks* one level (*chronological backtracking*), i.e. try the opposite value of the previous decision. When the top of the tree is reached and a conflict occurs, it means that the formula cannot be satisfied and the solver reports UNSAT (line 13). However, if the formula is empty in any branch, this means that the current assignment satisfies the whole formula and the algorithm reports it on line 10 or 12. An important function in the DPLL algorithm is the Boolean constraint propagation (BCP) also called unit propagation is presented in (line 3). This function forces the values of unit clauses in order to satisfy them until a fix point is reached and will end either, there are no more unit clauses in the formula or an inconsistency is found this means that the current assignment cannot satisfy the formula. In the later case, the solver will backtrack and another branch will be explored.

```
function DPLL(φ: CNF formula, α assignment)
    returns an assignment if φ is SAT and UNSAT otherwise
  φ, α ← unitPropagation(φ, α);
  if {} ∈ φ then
    return ⊥;                                   // Conflict
  if φ = {} then
    return α;                                   // φ is SAT
  x ← assignDecisionLiteral();
  if α ← DPLL(φ ∪ {x}, α) then
    return α
  if α ← DPLL(φ ∪ {¬x}, α) then
    return α
  return UNSAT;                                 // φ is UNSAT
```

Algorithm 1: The DPLL algorithm.

When DPLL is executed on the formula of Figure 2.1, after making decisions on literals

```
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

$\neg x_1$ and $\neg x_2$, unit propagation detects that x_3 must be assigned to true. This propagation prevents to explore non-interesting assignments. Actually, when x_3 is set to false, the clause ω_1 cannot be satisfied and as it remains 3 variables and so 2^3 possible assignments (from α_1 to α_8). These assignments will never be checked. The performance of the DPLL algorithm is highly impacted by the decision variable chosen. `assignDecisionLiteral` is the procedure responsible of choosing it. Its objective is to find a literal that will generate a maximum of unit propagations. Intuitively, decision literals can be viewed as "guesses" and propagated literals can be viewed as "deductions". Finding the optimal variable that will generate the maximum number of propagation is NP-Hard [10]. Different heuristics exists to choose the decision variable, some of them are presented in 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) [43] is a sound and complete algorithm that overcomes this weakness. 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 (from line 7 to line 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 are updated (line 11). As the level can be much lower than the current level, this is called *non-chronological backtracking* or *backjump*. 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.

```
1  function CDCL(φ: CNF formula)
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      (φ, α) ← unitPropagation(φ|α, α);
7      if {} ∈ φ then                          // A conflict occurs
8        if dl = 0 then
9          return ⊥;                           // φ is UNSAT
10       ω ← analyzeConflict();
11       (dl, α) ← backjumpAndRestartPolicies();
12       φ ← φ ∪ {ω};
13     else
14       α ← α ∪ assignDecisionLiteral();
15       dl ← dl + 1;
16   return ⊤;                                 // φ is SAT
```

Algorithm 3: The CDCL algorithm.

Conflict Analysis

A conflict is an inconsistency discovered by the solver, a situation that requires for a variable to be set simultaneously to the true and false values. Figure 2.2 shows an example that leads to a conflict. First, the solver chooses $\neg x_1$ as a decision (noted D $(\neg x_1)$ in the figure) then, $\neg x_6$ and, then $\neg x_5$. This last one propagates x_4 (marked with P (x_4) in the figure), 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, it needs to be set to $\bot$. As a variable cannot have both values, a conflict appears (noted C in the figure). Applied another time, this serie of decisions would provoke the same propagation

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

$$\alpha = \mathrm{D}\,(\neg x_1) \to \mathrm{D}\,(\neg x_6) \to \mathrm{D}\,(\neg x_5) \to \mathrm{P}\,(x_4) \to \mathrm{P}\,(\neg x_2) \longrightarrow \mathrm{C}$$

Figure 2.2: Decisions/Propagations that leads to a conflict

and leads to the same conflict. To escape this situation, one needs to analyze the situation and feed the algorithm with the information that prevent it to do the same mistake again. This is done by use of the so-called *implication graph*. It represents the current state of the solver and records all dependencies between variables. It is updated when a variable is assigned (on decision/propagation), or unassigned (on backjumping operation). 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 vertice , that have no incoming edges, represent decision literal. The remaining vertices represent propagations. Each incoming arc, labeled by 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). `analyzeConflict` procedure analyzes this graph to find the reason of the conflict. To do that, a search of a *unique implication point (UIP)* is performed. A 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 UIPs 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 FUIP provides the smallest set of assignment that is responsible for the contradiction [57]. A

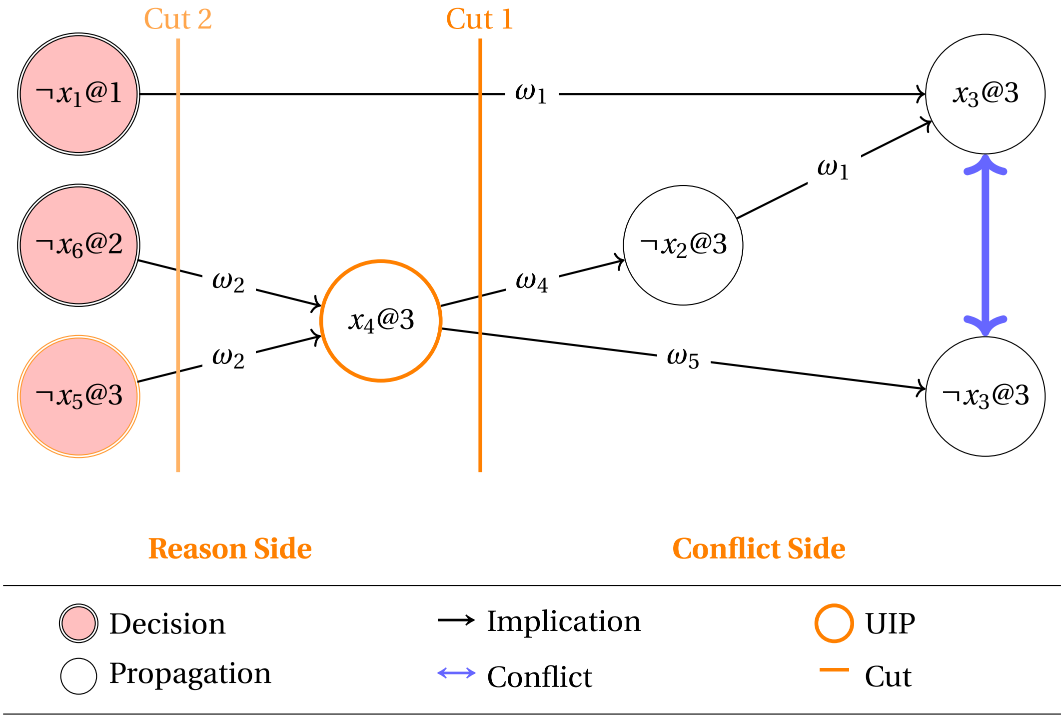


Figure 2.3: Implication graph

UIP divides the implication graph in two sides with a *cut*; the *reason side* contains decision variables that are responsible of the contradiction and the *conflict side* that contains the conflict. A UIP is always in 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 learned 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 [53]). All learned clauses are stored in a clauses database.

`backjumpAndRestartPolicies` procedure is executed after producing the conflict clause. All variables from the highest decision level to the current decision level are unassigned and so the current decision level and assignment are updated accordingly. 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. Adding the conflict clause prunes the search space that obviously contains no solution. This is the key point of the CDCL algorithm and the big difference with DPLL algorithm. In our example Figure 2.3, the target decision level is 1. After backtracking, the conflict clause will be assertive and the

FUIP is the only variable that has not a value and so will be propagated in the next step of the algorithm.

Heuristics

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

Decision heuristics . Decision variable has a huge impact on the overall solving time. It influences the number of propagations and so the depth of the search tree. The Variable State Independent Decaying Sum (VSIDS) [48] 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 the unassigned variable with the highest activity. Learning rate based branching (LRB [39]) is the 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 chosen as a decision.

Restarts Another important mechanism is *restart*. Basically, the solver abandons its current assignment and restart from the top of the tree, while maintaining some information, like learned clauses, scores of variables, etc. The restart prevents the solver to get stuck in the same part of the search space (phenomenon known as heavy tailing [25]). Detecting this phenomenon has been widely treated in the literature [5, 7]. These strategies are based on counting the number of conflicts or on the monitoring the current search's depth. Theoretically, a solver with restarts has a better result [28] 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 and the cost of unit propagation will increase. So, the solver needs to develop a policy to eliminate some of them. These clauses are redundant with regards to the initial problem, removing them will not affect the satisfiability of the formula. In the literature, different criteria exist. The size of the clause is one of them and is very often used by solvers. A small clause has better chance to participate to the unit propagation and so be useful for the solving. As a consequence, large clauses are removed.
Clause activity is another criterion, a clause augments its activity when it participates to

conflict analysis. Clauses with lower activity that are not implied in the resolution of conflicts are removed. The last, often, used criterion is based on the Literal Block Distance (LBD) measure. It is a measure that computes the *quality* of a clause. It is based on the number of decision levels present in the clause. The more a clause has a high value of LBD and the weaker its quality is, and so will be deleted from the clause database.

In current state-of-the-art solvers, multiple criteria are used and half of the learned clauses are removed during the clause database cleaning process.

Preprocessing / Inprocessing

In order to optimize solving time, some transformation can be applied to simplify the original formula. This is done by *preprocessing* the formula before the start of the 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 the *resolution inference rule* [49]. Consider the 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 $\neg x_1$. *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* uses resolution rules and subsumption. at the same time, 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 the `SatElite` [21] 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 [47]. This technique can also be used in inprocessing [52]. 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 uses the *community structure* of the formula to find good clauses to add into. A preprocessor engine doing that is `modprep` [3].

Optimizing SAT solving

With the emergence of multi-core architectures and the increasing power of computers, one way to optimize the solving of a SAT problem is the exploitation of these cores. *Portfolio*, first introduced in *ManySAT* [26], 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 and submitted to different solvers that cooperate to find a solution and used in different solvers like, for example [12, 38]. 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 [36] [37]. Another way to optimize the solving time of SAT solver is the exploitation of symmetries. The rest of this manuscript will detail how this allows to improve the performance of SAT solvers in the presence of symmetries in the original formula.

CHAPTER 3

SYMMETRY AND SAT

Contents

Despite the NP-Completeness character of the SAT problem, state-of-the-art solvers are able to treat many industrial problems.This is mainly due to the capacity of SAT solvers to prune search space using, for instance , learnt clauses. Another way to accelerate the solving is the exploitation of symmetries. These are common in real life.
Consider the problem of searching for a pattern in butterfly wings. Most butterflies have an identical pair of wings. After checking that both wings are symmetric (process called symmetry detection), the pattern can be searched for only one wing. Searching this pattern in the other wing is useless (process called symmetry exploitation). In this chapter, we show

how to detect that a given formula has symmetries and how to exploit them to accelerate the solving in the SAT context.

3.1 Group theory basics

Since symmetries belong to a branch of mathematics called group theory. This section gives us an overview of this.

Groups

Definition 3.1: Group

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* (i.e. $a * b = b * a$, for $a, b \in G$) is not a required axiom. If it satisfies it, the group is said *abelian*. The last definition leads to important properties: 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 a has two inverses, denoted b and c in groups $\langle G, * \rangle$, then,

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

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

Definition 3.2: Subgroup

Given a group G, a *subgroup* is a non-empty subset of G which is also a group with the same binary operation. We denote $H \leq G$, the subgroup H of G.

A group has at least two subgroups:

1. *trivial* subgroup: the subgroup composed of the identity element $\{e\}$. (All other subgroups are *non-trivial*)
2. *improper* subgroup: the subgroup composed of itself. (All other subgroups are *proper*).

Definition 3.3: Generators of a group

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

In other words, to obtain the group, it is sufficient to compose all permutations in the generators set until a fix point. So the generators are a compact representation of a group.

Permutation group

Definition 3.4: Permutation

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

Example: given a set $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$,

$$g = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ x_2 & x_3 & x_1 & x_4 & x_6 & x_5 \end{pmatrix}$$

In this example, 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

$$g = (x_1 \;\; x_2 \;\; x_3)\;(x_5 \;\; x_6)$$

Definition 3.5: Support of a permutation

A *support* of the permutation g, noted $supp_g$, is the elements that are not mapped to themselves:

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

Definition 3.6: Stabilized variable over permutation

A variable x is *stable* by a permutation g iff $x \notin supp_g$.

Definition 3.7: Permutation Group

A set of permutations of a given set X form a group G_X with the composition operation $(\circ)$ called *permutation group*.

Definition 3.8: Symmetric Group

The set of **all** permutations of a set X is the *symmetric group* of X and is noted $\mathfrak{S}(X)$.

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 partitions X into *equivalence classes* referred to as the *orbits* of X under G. The orbit of an element x under group G (or simply orbit of x when clear from the context) is the set. $[x]_G = \{g.x \mid g \in G\}$

3.2 Symmetries in SAT

The previous mathematical definitions of group theory can be applied to a CNF formula. The symmetric group of permutations of $\mathcal{V}$ (i.e., bijections from $\mathcal{V}$ to $\mathcal{V}$) is noted $\mathfrak{S}(\mathcal{V})$. The group $\mathfrak{S}(\mathcal{V})$ naturally acts on the set of literals: for $g \in \mathfrak{S}(\mathcal{V})$ and a literal $\ell \in \mathcal{L}$, $g.\ell = g(\ell)$ if ℓ is a positive literal, and $g.\ell = \neg g(\neg \ell)$ if ℓ is a negative literal. The group $\mathfrak{S}(\mathcal{V})$ acts on assignments (possibly partial) of $\mathcal{V}$ as follows:

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

The set of symmetries of φ is noted G_φ and is a subgroup of $\mathfrak{S}(\mathcal{V})$. Symmetry of a formula φ preserves the satisfaction, for every *complete* assignment α:

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

These symmetries can be obtained either *syntactically* or *semantically*. Semantic symmetries are independent of any particular representation of the problem. Conversely, syntactic symmetries depend of the encoding of the problem and can lead to different symmetries. We say that $g \in \mathfrak{S}(\mathcal{V})$ is a *symmetry of* φ if the following conditions hold:

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

3.3 Symmetry detection in SAT

For the detection of symmetries in SAT, we first introduce the notion of graph automorphism. Given a colored graph $Gr = (V, E, \gamma)$, with a set of vertices set $V \in [1, n]$, a set of edges E and γ a mapping: $V \to C$, where C is a set of *colors*. An automorphism of Gr is a permutation on its vertices, $aut : V \to V$, such that:

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

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 [34, 54]. Several tools exist to handle this problem like `saucy3` [31], `bliss` [30], `nauty` [44], etc. To find symmetries in a SAT problem, the formula is encoded in a colored graph and an automorphism tool is applied on it. In particular, given a formula φ with m clauses and n variables, the graph is constructed as follows [10]:

- *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;

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

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

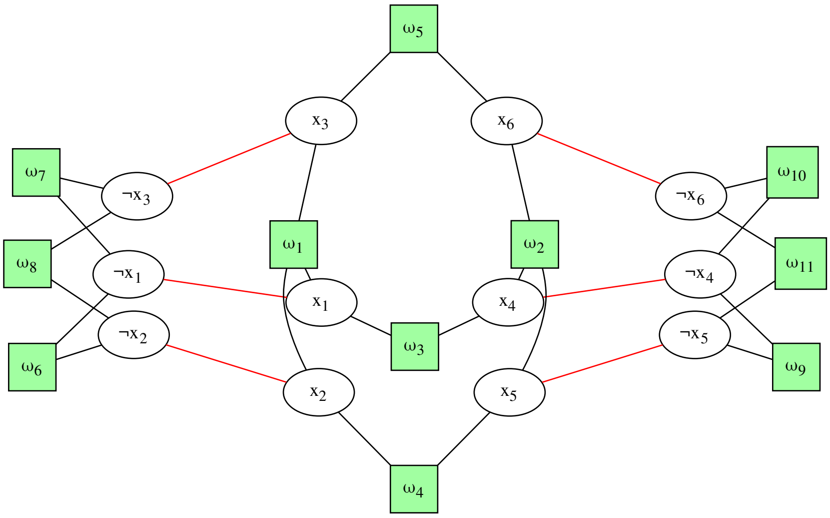


Figure 3.1: Example of constructed symmetry graph for a given CNF

Figure 3.1 shows the graph representation of a CNF. This problem has 6 variables and 11 clauses. So, the graph will have 12 + 11 = 33 vertices, where 12 represent the number of literal vertices (circles in the figure) and 11 represents the number of clause vertices (squares in the figure). The graph will also have 6 + 24 = 30 edges, 6 edges for Boolean consistency (red edges in the figure) and 24 edges that rely clause vertices to the literals. An optimization to reduce the number of graph vertices is possible. It is achieved by modeling binary clause using graph edges instead of graph vertices. However, in some particular cases, it can produce spurious permutations (i.e. Boolean consistency is not respected [2]). To ensure that the permutation is valid, the following condition must be satisfied:

$$\forall x \in supp_g, g.\neg x == \neg g.x$$

In other words, we check if the image of the negation of x is equals to the negation of the image of x, or each element x in the support of the permutation. This optimization reduces considerably the size of the graph, and accelerates the symmetry detection. In the previous example, we can remove 12 nodes and 12 edges. More generally, we can remove from the graph as many nodes and edges as those are binary clauses on the formula. Figure 3.2 represents the optimized graph for the detection of automorphism.

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

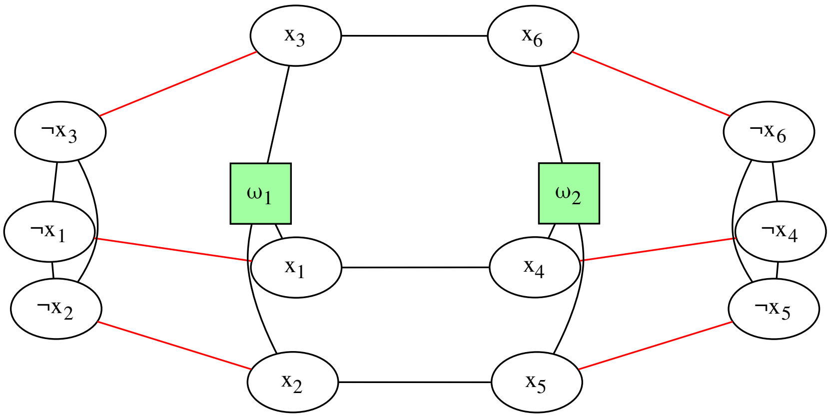


Figure 3.2: Example of constructed symmetry graph for a given CNF

After the construction of such a graph, it is given to an automorphism tool. This last will produce its set of generators. With the previous graph, the following generators are obtained using `bliss` as the automorphism tool:

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

3.4 Usage of symmetries

To illustrate the usage of symmetries, consider the *pigeonhole problems* (see Figure 3.3), where n pigeons are put into $n-1$ holes, with the constraint that each pigeon must be in a different hole. This is a highly symmetric problem. Indeed, all the pigeons (resp. holes) are exchangeable without changing the initial problem. The search algorithm explores fruit-

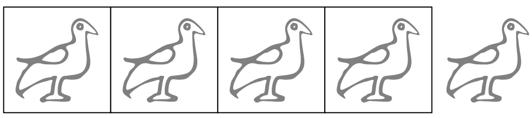


Figure 3.3: Graphical representation of an instance of the pigeonhole problem (5 pigeons, 4 holes)

lessly the symmetric search space, i.e, tries all possible combinations of couples (pigeon, hole). Solving this problem with a standard SAT solver, like MiniSAT [22], turns out to be very time consuming (and even impossible, in a reasonable time, for high values of n). To avoid this combinatorial explosion, a technique called *symmetry breaking* allows a SAT solver to avoid the visit of symmetric search space. For this purpose, there are two principles , known as *static symmetry breaking* and *dynamic symmetry breaking*. In the general case, visiting one assignment for each orbit is sufficient to determine the satisfiability of the whole formula. So, in the first case, symmetry breaking constraints that invalidate symmetric assignments are added to the initial problem before the start of solving (statically). The second one alters the search space during the solving (dynamically), it will like in the static approach prune symmetric search space or it can accelerate tree traversal using symmetrical facts. In fact, some decisions will be transformed into propagations and so accelerate the overall solving time.

In the following sections, we present in detail the two principles.

Static symmetry breaking

This section explains how to statically exploit symmetrical properties of a SAT problem. In this approach, only one assignment (branch) from each orbit is visited and all others are omitted.This leads us to the following questions:

1. How to generate constraints that forbid symmetrical assignments?
2. How to choose branch that is equivalent to all symmetric ones?

To answer question 2, we need to introduce an ordering relation between assignments

Definition 3.9: 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 $v \in \beta$ [a].

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

In other words, if the prefix of both assignments 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 by the following proposition:

Proposition 3.1: 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 proposal is a direct result of the definition 3.9. □

Given a formula φ and its group of symmetries 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).

To answer the first question, the set of 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 that has 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 $g \in G_\varphi$ produces a sound and complete set of symmetry breaking predicates also called *full symmetry breaking*. In this case, only the lex-leader assignment will be visited for each orbit. However, the size of the *sbps* can be exponential in the number of variables of the problem and so, they cannot be totally computed. To overcome this problem, only a subgroup is considered, in this case conjunction of LL_g for $H \subset G_\varphi$ (such that g is a permutation of the subgroup results a set of symmetry breaking predicates that aims at visiting at least one assignment per orbit and is called *partial symmetry breaking*. In this situation, several assignments per orbit can be visited but often bring a considerable reduction of the search space. Partial symmetry breaking gives a good trade-off between the number of generated constraints and the reduction of the search space. In the partial and full symmetry breaking, the set of symmetry breaking predicates generated is denoted by ψ.

Theorem 3.1: Satisfiability preservation SBPs

Let φ be a formula and ψ the computed *SBPs* for the set of symmetries in G_φ:

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

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

The generation of lex-leader constraints proposed by Crawford et al. [14] 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 3.4 shows an example of the generated clauses for the permutation g_3 of the previous example and a lexicographic order. The last constraint of the figure produces tautological clauses. Actually variables x_1, x_4 are present with both polarities. The constraints of other variables also produce tautological clauses.

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

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

Figure 3.4: Example of generated SBPs for one permutation

Here, the number of clauses generated per constraint increase exponentially by the cardinality of the support of the permutation. Hence, Aloul et al [1] proposed a more compact representation of *sbps* based on the creation of auxiliary variables. These variables encode equality of literals and are disjoint from the support of the permutation. Following clauses encode a compact lex-leader for a permutation:

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

where $\{y_0, \cdots, y_n\}$ is the set of auxiliary variables, y_0 is a unit clause that encodes the first equality and $\{x_1, ..., x_n\}$ be the set of variables sorted with the lexicographic order.

Figure 3.5 shows the compact encoding of generated constraints. This form grows linearly w.r.t. the number of variables. The auxiliary variables that encode the equality of two literals provide this reduction. Three auxiliary variables are introduced in this example x_7, x_8, x_9 such that x_7 encodes the equality of x_1 and x_4, x_8 encodes the equality of x_2 and x_5, and x_9 encodes the equality of x_3 and x_6. `Shatter` is a tool [1] for partial sym-

Order : $x_1 \prec x_2 \prec x_3 \prec x_4 \prec x_5 \prec x_6$; $(\perp < \top)$
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
$x_1 = x_4 \Rightarrow x_2 \preceq x_5$	$\neg x_7 \vee \neg x_1 \vee \neg x_2 \vee x_5$
	$\neg x_7 \vee \neg x_1 \vee x_8$
	$\neg x_7 \vee x_4 \vee \neg x_2 \vee x_5$
	$\neg x_7 \vee x_4 \vee x_8$
$x_1 = x_4 \wedge x_2 = x_5 \Rightarrow x_3 \preceq x_6$	$\neg x_8 \vee \neg x_2 \vee \neg x_3 \vee x_6$
	$\neg x_8 \vee \neg x_2 \vee x_9$
	$\neg x_8 \vee x_5 \vee \neg x_3 \vee x_6$
	$\neg x_8 \vee x_5 \vee x_9$

Figure 3.5: Example of compact generated SBPs for one permutation

metry breaking. It computes a compact lex-leader *sbps* with the symmetries produced by `saucy3` [31]. The following table shows the number of symmetry breaking predicates and the number of auxiliary variables added to the original formula.

Table 3.1 presents the number of variables and clauses present in the formula and also the number of *sbps* generated and the number of auxiliary variables added on different

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

Table 3.1: Number of sbps generated on different problem categories

problem categories that are: battleship (the battleship puzzle), chnl (channel routing instances), fpga (routing of global wires in integrated circuits), hole (the pigeonhole problem), urq (randomized instance based on expanded graphs), xor (exclusive or chain).
We can observe that the number of produced *sbps* and added auxiliary variables can be much larger than respectively the number of initial clauses and variables.

Special form of the group

Some formulas exhibit a specific type of symmetry, called *row (column) interchangeability*. These are a subset of variables structured as a two-dimensional matrix. Each row (column) is interchangeable, so, all variables of a row (column) permute with any other one. This form of symmetry is common in different kinds of problems like the pigeon hole problem in which pigeons and holes are interchangeable. The usage of row (column) interchangeability can significantly improve SAT performance. Actual symmetries can be eliminated by the addition of only a linear number of symmetry-breaking constraints [23]. To ensure this linear number of constraints, one condition must be satisfied : the lexicographic order of variables needs to respect the structure of the matrix. In practice, automorphism tools give only set of generators that contains no information on the structure of the group. The authors of `BreakID` [18] developed an algorithm to detect this specific structure and exploit it.

Binary lex-leader constraints

`BreakID` tries to generate a maximum number of binary lex-leader constraints. The first lex-leader constraint generated by each permutation is a binary clause. Enumerating the whole symmetry group will generate many binary clauses but will be time consuming. To avoid this enumeration, the graph structure of the orbit is exploited: as the orbit can be seen as a strongly connected component, there must exist a permutation that permutes a variable (for example the smallest variable according to the lexicographic order) of an orbit with each of the other variables of the same orbit. This allows us to generate as many binary lex-leader constraints as the size of the orbit. In addition, constructing a sequence of subgroups that stabilize the smallest variable allows the generation of new binary *sbps*. This sequence ends when a trivial subgroup is reached and is called a *stabilizer chain*. Figure 3.6 shows the application of the stabilizer chain. In the example, the considered group has three permutations and its graphical representation is shown. Given the lexicographic order, the smallest variable is x_1 and all other variables are in its orbit. According to the ordering relation, five *sbps* are generated, one *sbp* for each variable of the orbit except the smallest one. Then, the subgroup that stabilizes x_1 is computed. It contains only one permutation (g_2). As x_2 is the smallest variable according to the lexicographic order, the constraint $\neg x_2 \vee x_3$ is generated. The stabilizer chain leads to a trivial group and no more binary clauses are generated. In total, six binary clauses are generated without adding any auxiliary variables. Moreover, a property can be observed, when the smallest variable has the greatest value ($\top$ in this case), all variables in the orbits must have the same value. The

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

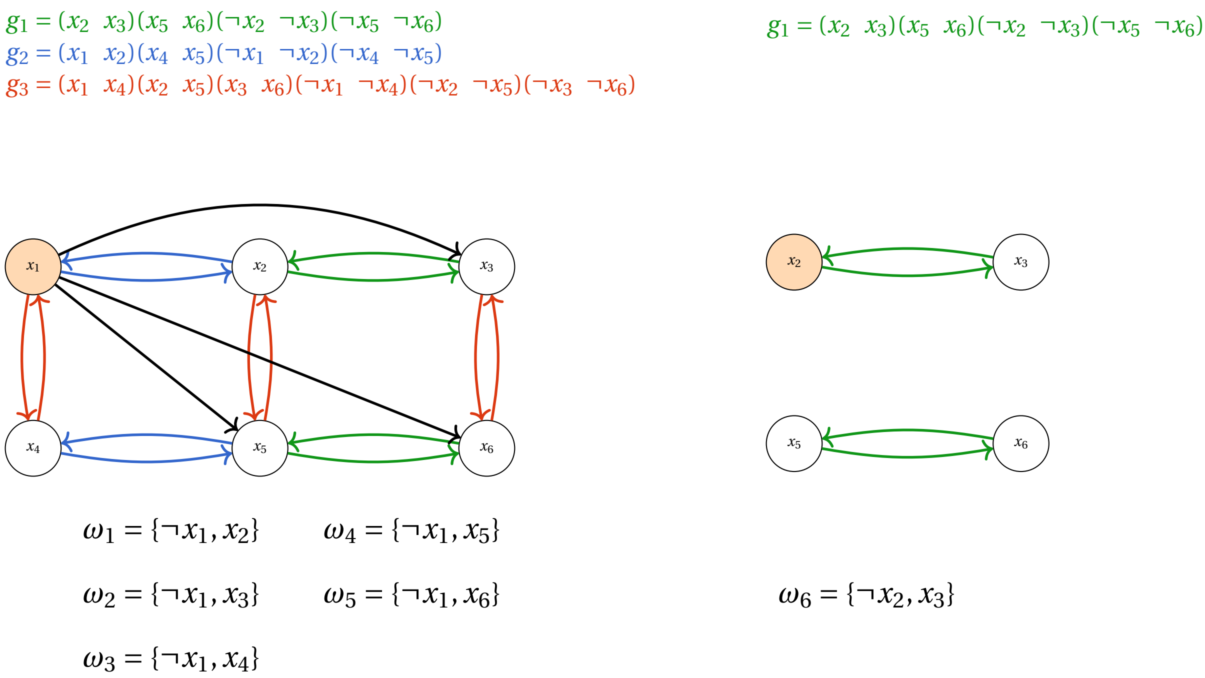


Figure 3.6: Generation of binary symmetry breaking predicates

size of the stabilizer chain is heavily dependent on the chosen lexicographic order. To avoid reaching trivial subgroups quickly an incremental order is proposed. It uses the size of the orbit and occurrences of variables in the set of generators: the biggest orbit produces more binary clauses and variables with few occurrences allow to disable less generators.

BreakID

As summary, `BreakID` combines three ideas:i) It searches if produced the generators by the automorphism tool have the row interchangeability special form and exploit; ii) It generates a maximum number of binary lex-leader constraints; iii) The classical *sbps* are generated.

Conclusion

Static symmetry breaking approach acts as a preprocessor that augment the initial formula with *sbps*. These constraints avoid the exploration of isomorphic search spaces. In the general case, the number of these clauses is often too large to be handled effectively by a SAT solver [41]. 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 will depend heavily on the symmetries chosen heuristically. [10]. An important point in static symmetry breaking is the chosen lexicographic order. Variable ordering may impact the number of generated constraints and hence the performance of the underlying solver. Different orders are studied in the literature. One of the simplest order is the lexicographical order. Some other existing orders exploit the structural properties of the problem [18]. Combining the generation of binary *sbps* with the exploitation of these properties allows state-of-the-art solver to solve more symmetric instances. Despite these optimizations and the good reduction of the search space with symmetries, some formula that exhibit symmetries are 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*. Internal heuristics consider these clauses as the original clauses, so the solver explores the search space with a different manner and affects performance negatively.

Dynamic symmetry breaking

Dynamic symmetry breaking approaches aims at exploiting the symmetries during the solving by altering the behavior of the solver. During the solving, the solver uses symmetries (present in the formula) to remove symmetric assignments or to propagate symmetrical facts. Propagating symmetrical facts has the consequence of reducing the number of decisions that are chosen heuristically and increase the number of propagations. In other words, symmetries transform some “guesses” into “deductions”. So, it improves the performance of the underlying solver. In the literature, different approaches of dynamic symmetry breaking exist, this section presents the most important of them.

SymChaff

One of the first tools for dynamic symmetry breaking is *SymChaff* a structure-aware SAT solver [51] and applies only on particular groups (section 3.4). To take part of this property, instead of using the classical decision heuristic that chooses exactly one variable, all symmetric variables are considered at the same time (k-branching). Roughly speaking, all variable in the same orbit are assigned/unassigned at the same time. So, all possible valuation of the orbit can be checked once. In this approach, only the number of true and false literal matters and computing the number of possible valuations is trivial in this particular form of group. For example, consider the permutations: $g_1 = (1\,2)$, only one of the valuations FT (variable 1 assigned to the false value and variable 2 assigned to the true value) or the reverse assignment (TF). So, one true value and one false value must be checked in addition to the FF and TT assignments to determine the satisfiability of the formula. The order in which the valuations are checked has a tremendous impact on solver performance. This approach has good results when the group of symmetries presents a particular form. In the general case, when we consider any group, computing the number of possible valuation will be very difficult and this approach is not applicable.

Symmetry Propagation

A different approach can be used to accelerate the tree traversal using symmetrical facts during the solving. One of them is *symmetry propagation* (SP) [19]. 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 guess (decisions) to deductions (propagations)". These deductions will reduce the overall tree traversal depth and hence will eventually accelerate the solving process. To explain this approach, let first give some definitions.

Definition 3.10: Logical consequence

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

Proposition 3.2: 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 $\varphi \cup \alpha \models g.\{l\}$.

In other words, if a literal l is 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) then, the solver can also propagate the symmetric of l. The problem here is to determinate which symmetries are valid for the formula $\varphi \cup \alpha$.

Definition 3.11: Active symmetry

A symmetry g is called active under a partial assignment α if $g.\alpha = \alpha$

Definition 3.11 leads to the following proposition:

Proposition 3.3

Let φ a formula and α a partial assignment. Let g be 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 is 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, and the composition of two active symmetries is also an active symmetry, so the solver can also propagate. $g^2.l, g^3.l, ...$

Active symmetries need strong requirements and so their applications are limited. Devriendt et al [19] improved the notion of active symmetries in the SAT context by introducing the notion *weakly active* symmetries that relax some constraints.

Definition 3.12: Weakly active symmetry

Let φ be 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 3.4

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 minimal effort the symmetries of $\varphi \cup \alpha$ by keeping track of the set of variables δ, which are in 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 propagations and so are more efficient. 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.

Symmetry Explanation Learning

Another approach to exploit symmetry without removing any satisfiable assignment of the problem is *Symmetry Explanation Learning* [17] (SEL). Symmetries of a formula leave this last invariant. Moreover, all learned clauses are logical consequences of the problem, symmetric of these clauses are also valid. Unlike Symmetry explanation scheme [6] (SLS) where all symmetrical learned clauses are added to the clause database. The idea of this approach is to learn useful symmetrical variants of learnt clauses. A clause is said to be useful if it participates to the unit propagation or conflict analysis. Computing all symmetrical learnt clauses will create a huge overhead in terms of computation time and memory. Its usage will be limited on huge problems. To avoid the previous drawback, SEL uses the following fact: i) All modern CDCL solvers need to maintain the implication graph and so store the reason of the propagated literals and obviously this reason is assertive. ii) Symmetries permute only few literals in a clause and so the probability that symmetrical clauses are also assertive is high. So, symmetrical clauses may also participate to the unit propagation. iii) These clauses are stored in different learning database and treated separately. The solver promotes these clauses when they are effectively useful at the end of unit propagation. As unit propagation is done until fix point, it ensures no duplicate clause is added to the problem. iv) To limit memory impact, symmetrical clauses are removed when the propagated literal are unassigned. Moreover, SEL provides some interesting properties: first, the authors prove that SEL propagations are a super-set of the one provided by SP. It also does not need to track any status of symmetries (as opposite to SP). Like SP no satisfying assignment is discarded. Nonetheless, the negative point is that SEL may flood the solver if the used set of symmetries is big.

Conclusion

Dynamic symmetry breaking approaches exploit the symmetry property of the formula during the solving. It prevents the creation, as in static symmetry breaking, of potentially useless clause that increases the size of the original formula. Different approaches exist, one use k-branching that allows to visit only lex-leader assignment but can be applied only in particular form group. Others, use the symmetry to propagate symmetrical facts. Mainly, they transform decisions (guesses) into propagations (deductions), that accelerate the tree traversal and may improve the overall performance of the solver.Moreover, as they are integrated directly to the search engine, solvers can adapt their heuristics dynamically, like for example the restart. However, the integration of dynamic approach must be done carefully, CDCL algorithm is a highly optimized and fine-tuned search engin. The integration of symmetry breaking can slow down its core engine.

Part II

Contributions

CHAPTER 4

SYMMSAT: 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 static symmetry breaking approach constraints are added to the original problem that avoids the solver to visit symmetrical search space. But, 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 [41]. 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 [10]. Besides, these approaches are preprocessors, so their combination with other techniques, such as *symmetry propagation* [19], 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 easily 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 4.1: 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. The permutation g reduces the assignment and all its extensions). The permutation g is *inactive* on α when $\alpha < g.\alpha$ (so g cannot reduce α and all its extensions). A symmetry is said to be *active* with respect to α when it is neither inactive nor a reducer of α.

Proposition 4.1 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, $supp_g$, is the set $\{v \in \mathcal{V} \mid g.v \neq v\}$.

Proposition 4.1

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 $v \in supp_g$ such that:
   - $\forall\ v' \in supp_g$, 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 $v \in supp_g$ such that:
   - $\forall\ v' \in supp_g$, 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 4.2: 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 4.3: 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 4.1. Let $U = \{v', \neg v' \mid v' \in \mathcal{V}_g \text{ and } v' \preceq v\}$. We define $\eta(\alpha, g)$ as $(U \cup g.U) \setminus \alpha$.

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 following the Definition 4.3, we can deduce than $\eta(\alpha, g) = (U \cup g.U) \setminus \alpha = \{\neg x_1, \neg x_2, \neg x_3, x_4\}$.

Proposition 4.2

$\eta(\alpha, g)$ 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 4.1: Satisfiability preservation

Let φ be a formula and ψ an *esbp* for some assignment α under $g \in G_\varphi$. Then,

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

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

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

the backjump order is given by a simple injection of an *esbp* computed on the fly. Algorithm 4 explains how to extend the CDCL algorithm described in Line 6 with a *symmetry controller* component.

```
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, α) ← 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 4.1 ; 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 and 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 (line 9). 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 and 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 $\eta(\alpha, g)$ of Definition 4.3, the effective symmetry-breaking predicate of Proposition 4.2. This will prevent the CDCL 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 two generators:
$G = \{g_1 = (x_1\ \ x_2)(x_4\ \ x_5), g_2 = (x_1\ \ x_4)(x_2\ \ x_5)(x_3\ \ x_6)\}$ (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 $supp_{g_2} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$.

```
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*.

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

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 [40], 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* [10].

Integration in `MiniSAT`.

We show how to integrate cosy to an existing solver, through an example of `MiniSAT` [22]. 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` [31] and `bliss` [30]. 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 symmetric instance of the SAT contests [29] from 2012 to 2017, we call a symmetric instance a CNF instances for which `bliss` finds symmetries that could be computed in at most 1000 seconds 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 timeout 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 [22];

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 UNSAT instances of the benchmarks of the six last editions of the SAT competition.

- Shatter, a symmetry breaking preprocessor described in [1], coupled with the MiniSAT SAT engine;

- BreakID, another symmetry breaking preprocessor, described in [18], also coupled with the MiniSAT SAT engine.

Each SAT solution was successfully checked against the initial CNF. For UNSATsituations, there is no way to provide an UNSAT certificate 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 UNSAT formulas 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 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. PAR-2 measure is used in SAT competition, it corresponds to the sum of cumulative time of solved instances with 2 times t he timeout of unsolved instances.

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 SAT instances of the benchmarks of the six last editions of the SAT competition.

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

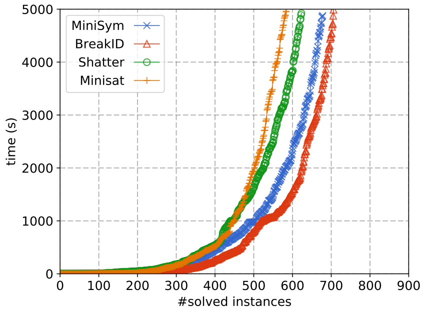
(a) With saucy3

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

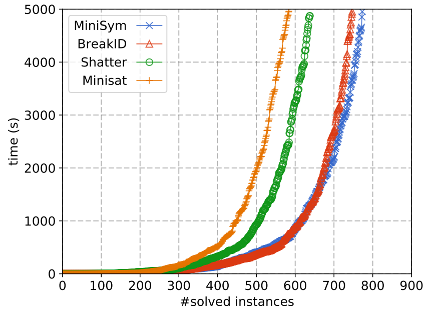
(b) With bliss

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

We observe that MiniSym with saucy3 solves the most instances in only half of the UNSAT categories. However, with bliss, MiniSym solves the most instances in all but four of the UNSAT categories; it then also solves the highest number of instances among its competitors. This shows the interest of our approach for UNSAT instances. Since symmetries are used to reduce the search space, we were expecting that it will bring the most performance gain for UNSAT instances. The situation for SAT instances 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 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



(a) with saucy3



(b) with bliss

Figure 4.1: cactus plot total number of instances

(only the effective ones). This is shown in Table 4.4; MiniSym generates an order of magnitude fewer predicates than BreakID. 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.

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).

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 Some optimization

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

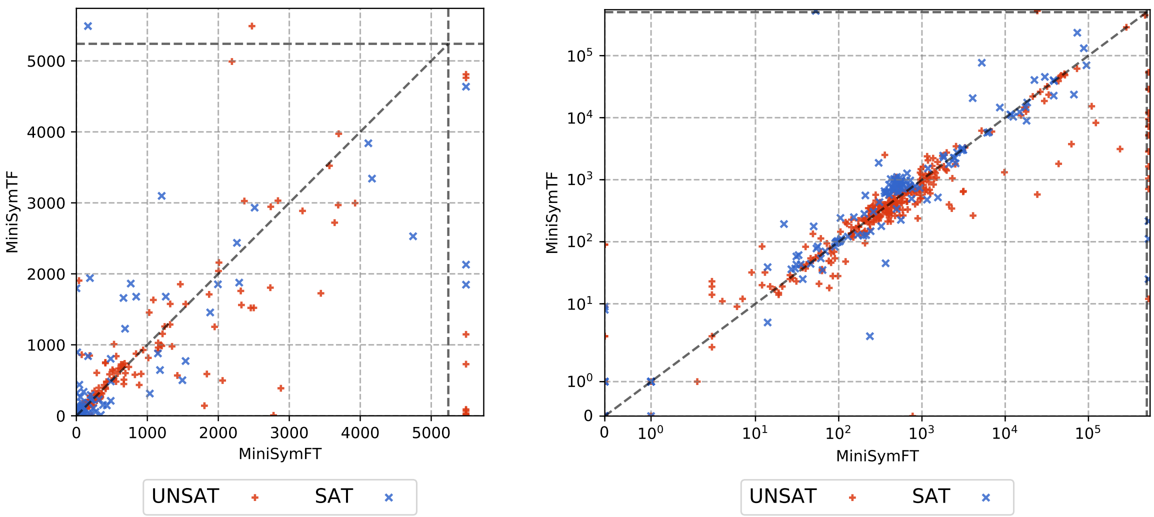
Other heuristics on the symmetry handling may increase the performance. We present here some of them. In some cases, multiple permutations can be reducers at the same time, and each one generates different symmetry breaking constraints. The backtrack and the pruning capacity depends heavily on the chosen constraint. In our library, the first reducer permutation generates the *esbp*. Another point concerns the injection of the symmetry breaking predicates. Two choices are possibles, first, before the unit propagation and second, at the end of the propagation. This choice will impact the solver behavior. In the first case, esbp takes the lead over the classical conflict (if its occurs). Conversely, in the second case, the classical conflict takes the lead over esbp. This can be, especially, useful on SAT problems because esbp can eliminate non lex-leader SAT assignment. To emphasize this behavior, the conflict of the esbp can be ignored in the sense that the conflict clause is just added into the clause database and so will participate to the next unit propagation. This gives to the solver the ability to find a solution symmetrical branch but avoid to get multiple times on non-minimal part of search. I can be useful if we know in advance that the problem is satisfiable.

Change the Order Dynamically

As seen before, the ordering relationship between variables influence the minimal value of each orbit (lex-leader) and the generated constraints. The symmetry controller is "waiting" for the solver that assigns the variables that allows it to decide if the current assignment is the lex-leader. The main idea to change dynamically the order, and so the lex-leader, is that the symmetry breaking order follows the decision heuristics of the solver, then symmetry controller can decide quickly if the current assignment is minimal. Changing this order dynamically is possible with some requirements: all esbps and all deduced clauses from a symmetry breaking predicates need to be removed. If these constraints are not deleted, the correctness of the algorithm is not guaranteed.

Impact of the sign in variable ordering

With the same variables ordering, swapping the value thus, $\top < \bot$ or $\bot < \top$ may impact drastically the performance of the solver. To illustrate it, consider the pigeonhole problem with 100 holes and 101 pigeons with the increasing order of the variables and change only the sign. With $\bot < \top$, the solver generates 20 619 esbps, and takes 13.8 seconds to solve it. With the reverse order ($\top < \bot$), it generates 33 263 esbps and solves it in 93.4 seconds. Figure 4.2 shows this difference on 500 symmetric instances with a scatter plot that compares the same variable order with $\top < \bot$ and $\bot < \top$, `MiniSymFT` is the solver in which $\bot < \top$, and `MiniSymTF` is the solver in which $\top < \bot$. On the left, we compare the computation time of the solver. As we can observe `MiniSymTF` is more efficient on some UNSAT instances (red points in the figure). The right figure shows the number of generated esbp by solvers in a log scale. On the large majority of instances, they generate approximately the same number of esbps. But the difference can an order of magnitude higher. This can be due to the time of execution of the solver and/or the impact of the sign of the constraints.



(a) Time

(b) Number of esbp

Figure 4.2: Comparison of the order with different signs on 500 symmetric instances.

`MiniSymTF` is generally better and is the default choice in the library. If it is running on a specific application, reverse order can be chosen if it performs better results.

4.4 Conclusion

SymmSAT uses same the principles as static symmetry breaking approaches but operates dynamically by injecting *effective symmetry breaking* during the search. This overcomes, the main problem of the static approaches, that they generate many *sbps* that are not effective in the solving (size of the generated formulas, overburden of the unit propagation procedure, etc.). The idea we brought it 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 SAT contests from 2012 to 2017 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

In the previous chapter, we presented an approach that reuses the principles of the static approaches, but operates dynamically (namely, the effective symmetry breaking approach [45]): the symmetries are broken during the search process without any pre-generation of the *sbps*. 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 when compared with the *symmetry propagation* technique developed by Devriendt et al. [19]. 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 [46]. 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 chapter, this is not completely true: both theoretical and practical issues have to be analyzed and solved to get a running complementarity. 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:

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

Where $Stab(\omega_e) = \{g \in \mathfrak{S} \mid \omega_e = g.\omega_e\}$ is the stabilizer set of ω_e and G_φ 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

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

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, $GUS_i = \bigcap_{\omega_e \in \varphi_e^i} Stab(\omega_e) \cap G_\varphi$ 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 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: $GUS = \bigcap_{i \neq k} GUS_i$.

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 5.1.

Definition 5.1: Local Symmetries

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:

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

$L_{\omega,\varphi}$ 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 5.1

Let φ be a formula. Then, $\bigcap_{\omega \in \varphi} L_{\omega,\varphi} \subseteq G_\varphi$.

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 $g \in G_\varphi$ (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 5.2 establishes the relationship between the local symmetries of a deduced clause and those of the set of clauses that allow its derivation.

Proposition 5.2

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. 5.1). 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. `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.

```
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, α) ← 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.

- Line 13: when a conflict is detected, then the analyzing procedure is triggered. According to Proposition 5.2, the generated conflicting clause ω, should be associated 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 Proposition 5.2).

  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 `cancelActiveSymmetries` of Algorithm 6 with the additional reactivation of the symmetries that have been broken (deactivated) by ESBPA. Technically speaking, each time a deduced literal

[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.

```
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, α) ← 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.

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 5.1 are active each time the `symPropagation` function is called. Technically speaking, each symmetry of the complement set (to G_φ) 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 `updateActiveSymmetries` of algorithm 6. The reason clause, ω_l, of each propagated literal, l, by the `unitPropagation` function is analyzed. Each symmetry of the complement set (to G_φ) of the set local symmetries of ω_r is marked *inactive*, if it is not already marked so. l becomes responsible of this deactivation.

Illustrative Example

Consider the following permutation $G = \{g_1 = (x_1 x_2)(x_3 x_4), g_2 = (x_3 x_4)(x_5 x_6)\}$, the lexicographic ordering relation of variables with $\top < \bot$ and the current assignment $\alpha = \{\neg x_7\}$. Suppose the permutation g_1 already generates the *esbp* $\omega_e = \{x_1 \neg x_2\}$ then, associated local symmetry is g_2 because it stabilizes ω_e ($g_2.\omega_e = \omega_e$). Then, suppose a conflict occurs and the resulting clause is $\omega_d = \{x_4\, x_7\}$. In the conflict analysis, this clause is deduced by original clauses and ω_e. So, it has the same valid symmetries as ω_e. As $g_2.\omega_d = \{x_3\, x_7\}$ is an assertive clause and g_2 is a valid permutation for this clause SP can propagate x_3 (the symmetrical of ω_d)

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

[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 0â€“20 (704)	194	197	**198**
Generators 20â€“40 (136)	33	**34**	**34**
Generators 40â€“60 (141)	28	28	**29**
Generators 60â€“80 (168)	**65**	64	**65**
Generators 80â€“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 0â€“20 (704)	**233**	220	226
Generators 20â€“40 (136)	50	**54**	**54**
Generators 40â€“60 (141)	75	**83**	**83**
Generators 60â€“80 (168)	**11**	**11**	10
Generators 80â€“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 [22]: `minisat-Sp`, for SPA; `minisat-Sym`, for ESBA; and `minisat-SymSP`, for the combo. Symmetries of the SAT problems have been computed by `bliss` [30]. We selected from the last seven editions of the SAT contest [29], 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 (from 2012 to 2018). 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 symmetrical 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 [29]. 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 measure 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.

To go further in our analyze, we also compared 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

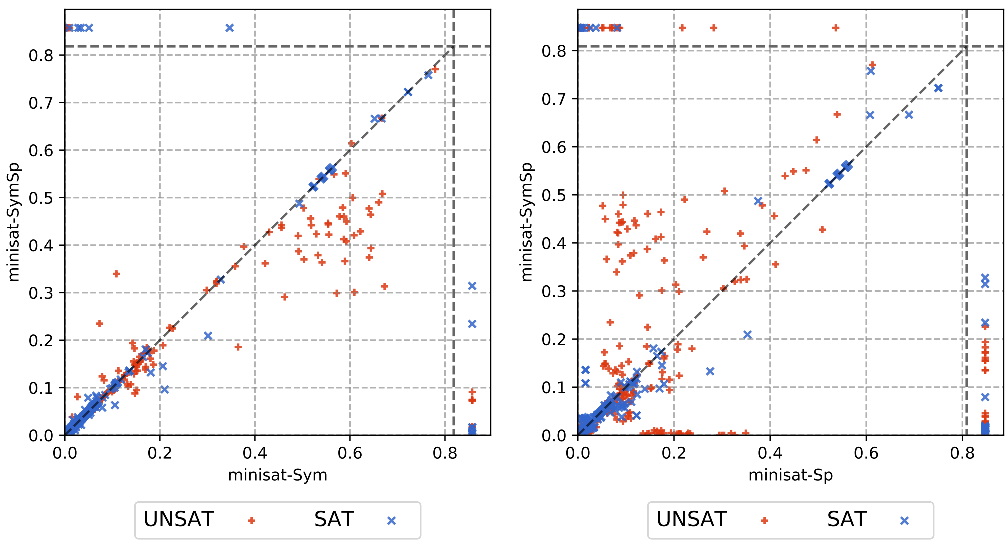


Figure 5.1: Comparison of the ratio between the number of decisions and the number of propagation for the combo w.r.t. ESBA and SPA.

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.

CHAPTER 6

CONCLUSION AND FUTURE WORKS

This thesis presented different approaches to increase the performance of solving the Boolean satisfiability problem (SAT) (see Chapter2) in presence of symmetries. Symmetries are common and can be found in different classes of problems like graph coloring, FPGA routing, etc. The presence of symmetries in a problem hinders the performance of the solver. It forces it to explore every symmetric branch of the search tree thus facing to a combinatorial explosion. Some trivial question for a human, like: Can 100 pigeons fit in 99 holes?, where each pigeon (hole) are symmetric becomes impossible for a state-of-the-art solver. To deal with symmetries in SAT problems, two approaches exists (see Chapter 3). The first one called static symmetry breaking approach, acts as a preprocessor augmenting the initial problem to prune symmetrical assignments of the search tree. The second one called dynamic symmetry breaking, acts during the solving. Like in the static approach, it prunes the symmetrical assignments of the search tree or accelerates its traversal using symmetrical facts. Each approach has its weaknesses and strengths. However, some highly symmetrical instances cannot be solved with state-of-the-art approaches. In this thesis, we improve current symmetry breaking approaches to deals with more symmetric formulas.

Our first symmetry based approach (see Chapter 4)introduces the notion of effective symmetry breaking predicates (esbp) that borrows the principle of static symmetry breaking approach but operating dynamically [45]. This approach overcomes the combinatorial explosion caused by the pre-generation of *sbp* in the state-of-the-art static approaches. An extensive evaluation shows that our approach improves on state-of-the-art static symmetry breaking approaches. The method is encapsulated in a library called `cosy` and can be integrated easily to any CDCL-like SAT solvers. It is released under GPL-v3 license and is available at `https://github.com/lip6/cosy`. Though easy to use and effective, this method cannot handle some problems that are easily solved by other dynamic symmetry breaking approach like Symmetry Propagation (SP) [19]. Chapter 5 presents our second contribution that aims to combine two dynamic symmetry breaking approach, our first algorithm with the SP approach. In terms of performance, the combined version not brings big difference compared to the use of esbp. Nevertheless it can solve few instances that cannot be solved with previous approaches. Overall, this work answers to the precise question: "Is it possible to accelerate the traversal while pruning the tree with symmetries?".We clearly show that the answer is yes, thanks to the notion of *local symmetries*.

6.1 Perspectives

Despite the new instances that have been resolved by our approach, yet remains to be done in this area. Indeed, some highly symmetric instances cannot be resolved by the state-of-the-art approaches and ours. This section gives some suggestions to improve symmetry based approaches to solve SAT problems. In the literature, other dynamic approach exists such as for example Symmetry Explanation Learning (SEL) [17]. Like SP approach, it aims to accelerate the tree traversal using symmetries. Like our combo approach with SP, a combination with SEL should be possible. Actually, SEL has fewer requirements than SP and author of these two papers demonstrates that SEL is a superset of SP. For this reason, this hybrid approach would probably give us much better performance and maybe solve some unsolved symmetric problems. An approach that mimics the previous work could create this new combination, thanks again to the local symmetries. To improve performances, we can also use parallel SAT solver. Due to the emergence of multi-core machines, many research has been done in this area. In the general case, cooperation based approach (divide and conquer) and competition base approach (portfolio) not use any symmetry breaking strategy and will explore the full search space. An integration of symmetry breaking may improve the overall performance of the solver. A theoretical and practical study may demonstrate if it increases performances and whether it is more efficient in portfolios or divide and conquer strategy.

SAT solvers are present in different tools that use it for checking different properties. For instance, Eclipse uses a SAT solver to check for dependencies conflict on installed packages [35]; a verification tools like Alloy[1] use a SAT solver to check properties and find counter-

[1] `http://alloytools.org/`

example for software modeling. Integration of our symmetry based SAT solver may improve the performance of such products.

All approaches we proposed suppose that we compute the symmetries of the problem before solving it. But, on some large instances, the computation of symmetries is a limiting factor. An optimal approach would be to find them during the solving or maybe find them "opportunistically" in the sense that we can stop the search of symmetries when some are found.

Another perspective of this work is the exploitation of partial symmetries i.e., the symmetries present only with partial assignment of the solver. The introduced local symmetries can be an entry point for this purpose. Partial symmetries widen the applicable scope of our techniques.

BIBLIOGRAPHY

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

[2] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(9):1117–1137, 2003.

[3] 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.

[4] 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.

[5] G. Audemard and L. Simon. Refining restarts strategies for sat and unsat. In *International Conference on Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012.

[6] 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.

[7] A. Biere. Adaptive restart strategies for conflict driven sat solvers. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 28–33. Springer, 2008.

[8] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems*, pages 193–207. Springer, 1999.

[9] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without bdds. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.

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

[11] J. A. Carlson, A. Jaffe, and A. Wiles. *The millennium prize problems*. American Mathematical Society Providence, RI, 2006.

[12] G. Chu, P. J. Stuckey, and A. Harwood. Pminisat: a parallelization of minisat 2.0. *SAT race*, 2008.

[13] 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.

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

[15] A. Darwiche and P. Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

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

[17] 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.

[18] 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.

[19] 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.

[20] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *The Journal of Logic Programming*, 1(3):267–284, 1984.

[21] 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.

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

[23] 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.

[24] M. R. Garey and D. S. Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

[25] 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.

[26] Y. Hamadi, S. Jabbour, and L. Sais. Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6:245–262, 2008.

[27] M. J. Heule, O. Kullmann, and V. W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 228–245. Springer, 2016.

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

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

[30] 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.

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

[32] H. A. Kautz, A. Sabharwal, and B. Selman. Incomplete algorithms. *Handbook of satisfiability*, 185:185–203, 2009.

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

[34] J. Kobler, U. Schöning, and J. Torán. *The graph isomorphism problem: its structural complexity*. Springer Science & Business Media, 2012.

[35] D. Le Berre and A. Parrain. On sat technologies for dependency management and beyond. 2008.

[36] 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.

[37] 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.

[38] M. Lewis, T. Schubert, and B. Becker. Multithreaded sat solving. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pages 926–931. IEEE Computer Society, 2007.

[39] 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.

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

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

[42] I. Lynce and J. Marques-Silva. Sat in bioinformatics: Making the case with haplotype inference. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 136–141. Springer, 2006.

[43] J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

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

[45] 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.

[46] H. Metin, S. Baarir, and F. Kordon. Composing symmetry propagation and effective symmetry breaking for sat solving. In *NASA Formal Methods Symposium*, pages 316–332. Springer, 2019.

[47] C. Moore and S. Mertens. *The nature of computation.* OUP Oxford, 2011.

[48] 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.

[49] J. A. Robinson et al. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[50] S. J. Russell and P. Norvig. Arti cial intelligence a modern approach. 1994.

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

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

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

[54] J. Torán. On the hardness of graph isomorphism. *SIAM Journal on Computing*, 33(5):1093–1108, 2004.

[55] P. Toth and D. Vigo. *The vehicle routing problem.* SIAM, 2002.

[56] L. G. Valiant. The complexity of computing the permanent. *Theoretical computer science*, 8(2):189–201, 1979.

[57] 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.