KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Exploiting Symmetry in Model Expansion for Predicate and Propositional Logic

Jo Devriendt

Supervisor:
Prof. dr. M. Denecker

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

February 2017

Exploiting Symmetry in Model Expansion for Predicate and Propositional Logic

Jo DEVRIENDT

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Computer Science

February 2017

I love deadlines.
I like the whooshing sound they make as they fly by.

— Douglas Adams

I love pigeons.
I like the whooshing sound they make as they fly by.

— Bart Bogaerts

Preface

For four and a half years I had an amazing time exploring and experiencing the world of academia. To me, it was a place of wonder, and a humbling one at that... Wherever I went, there was always someone helpful who, after a couple of minutes of explaining the problem I had been struggling with for days, could figure out a solution on the spot.

I am grateful for all the help I got, and for that I need to thank Marc first, who proved to be an approachable and inspiring promotor. Without Maurice and Bart as industrious co-authors I would not stand here today: thanks for enduring my often chaotic work pace. I also want to thank Gerda, Patrick, and my colleagues at the KRR, DTAI and CODeS research groups for the many instructive discussions, and the smooth collaboration on various projects and teaching duties.

Also, I treasure the many days and evenings spent with friends, be it in Brugge, Leuven, Gent, Blankenberge, Zedelgem, de Westhoek, Brussel, Kortrijk, Oordegem, Koolkerke, Tielt, Lubbeek, Paris, Darmstadt or the Ardennes. The joy and escape they provided were soothing for an often spinning mind.

Of course, I want to thank my family for supporting me through periods that were more stressful than I could handle on my own. And Veronika, thanks for your boundless patience.

Finally, I am grateful to the KU Leuven and the Research Foundation Flanders (FWO) for providing the opportunity to "avoid real life" for all these years ;)

It is astounding that after four and a half years of hard work, the only thing I leave behind is a small booklet of less than 170 pages. It contains all I know on symmetry in propositional and predicate logic. I wish you, the reader, an insightful journey through its chapters.

Abstract

Many combinatorial problems exhibit *symmetry*, a transformational property that does not fundamentally alter the nature of a problem. For instance, renaming a set of identical trucks in a routing problem, mirroring or rotating a chessboard onto itself, or the automorphisms of an input graph give rise to symmetry. These symmetry properties often hinder an algorithm solving a combinatorial problem, as it wastefully investigates different configurations of essentially the same solution.

In this thesis, we set out to explore symmetry properties in model expansion problems for both predicate and propositional logic. The goal is to equip automated systems with the necessary tools to adequately handle symmetry. By investigating both symmetry detection and symmetry exploitation, we present symmetry exploitation techniques that rely only on problem specifications as input, without additional symmetry information. Systems equipped with these techniques are able to tackle a wider range of problems with less human input – the eventual goal of any artificial intelligence.

First, we present BreakID, a novel symmetry breaking preprocessor for propositional logic. BreakID's core idea is to investigate structural properties of the symmetry group of a problem, and to adjust any generated symmetry breaking formulas accordingly. Furthermore, we also added usability features and technical optimizations that allow symmetry detection and subsequent breaking for a broader range of propositional formulas, both in a Boolean satisfiability and an answer set programming context. Experimental results show that BreakID improves on Shatter and sbass, the previous state-of-the-art static symmetry breaking preprocessors for propositional logic.

Second, we investigate two new dynamic symmetry handling algorithms for propositional solvers, based on symmetrically deriving logical consequences.

The first algorithm, SP, focuses on propagating literals that are symmetrical to already propagated literals. SP is based on the notion of *weak activity*, a generalization of the constraint programming notion of *activity*. Experiments show that this approach is effective, but can be improved by also performing symmetrical propagations not based on the weak activity status of a symmetry. This leads to the second algorithm, SEL, which derives symmetrical explanation clauses and, as such, is a form of *symmetrical learning*. Even though SEL is a straightforward form of symmetry handling, we provide experiments that indicate it is competitive to BreakID's advanced symmetry breaking.

However, working on a propositional level burdens the algorithms that detect symmetry and derive symmetry group information, as these properties are more readily available at the predicate level. Hence thirdly, we investigate how symmetry manifests itself in first-order logic. We propose the notion of *local domain symmetry*, a form of symmetry in predicate logic theories that captures a broad range of symmetry groups occurring in practical problems. Based on theoretical properties of local domain symmetry, we design efficient ways of both detecting and breaking it. Our implementation in IDP outperforms other approaches in symmetry detection time, and in symmetry breaking power for problems with large row interchangeability groups. Moreover, our approach is one of the few automated approaches that detects symmetry at the predicate level. The modest price to pay is that some forms of symmetry are not captured by the notion of local domain symmetry.

Finally, we establish a promising link between local search algorithms and symmetries by noting that a local search neighborhood for a problem can be constructed from its quasisymmetry group. These quasisymmetries can be automatically detected with previously established techniques, leading to a novel automated local search approach with good initial performance.

Summarized, our work provides new insights in how symmetry can be detected and exploited without human interaction. These ideas, rooted in propositional and predicate logic, are useful for anyone designing automated combinatorial problem solving systems, be they affiliated with constraint programming, operations research, Boolean satisfiability solving, answer set programming, or related fields.

Beknopte samenvatting

Veel combinatorische optimalisatieproblemen vertonen *symmetrie*, een eigenschap die problemen transformeert, maar die de aard van een probleem niet fundamenteel verandert. Bijvoorbeeld, een poule identieke vrachtwagens hernoemen in een planningsprobleem, een schaakbord spiegelen of roteren, of een wiskundige grafe afbeelden op zichzelf, geeft aanleiding tot symmetrie. Een probleem dat veel symmetrie bevat, stelt een uitdaging aan een algoritme dat het probleem moet oplossen, aangezien het algoritme dreigt verloren te lopen in de vele verschillende configuraties van fundamenteel dezelfde oplossing.

In deze thesis onderzoeken we symmetrie in de context van modelexpansieproblemen in zowel propositionele logica als predikatenlogica. Het doel is om geautomatiseerde systemen die modelexpansieproblemen oplossen te voorzien van het nodige gereedschap om met symmetrie om te gaan. Door het bestuderen van zowel detectie als uitbuiting van symmetrie, ontwikkelen we technieken die als input enkel een probleemspecficatie krijgen, en geen aanvullende symmetrie-informatie Systemen die deze technieken implementeren kunnen een groter aantal problemen aan met minder menselijke hulp – het uiteindelijk doel van artificiële intelligentie.

Ten eerste presenteren we BreakID, een nieuwe symmetriebrekende preprocessor voor propositionele logica. BreakIDs kernidee is om structuureigenschappen van de symmetriegroep van een probleem uit te pluizen en om de opgestelde symmetriebrekingsformules hiermee te verfijnen. Daarnaast voegden we ook bruikbaarheidsfeatures en technische optimalisaties toe, zodat symmetriedetectie en -breking nuttig is voor een groter aantal propositionele formules, zowel in de context van *Boolean satisfiability* als van *answer set programming*. We presenteren experimenten die aangeven dat BreakID krachtiger is dan Shatter en sbass, de voormalige *state-of-the-art* preprocessoren voor statische symmetriebreking in propositionele logica.

Ten tweede onderzoeken we een nieuwe dynamische aanpak van symmetrie voor propositionele *solvers*, gebaseerd op het symmetrisch afleiden van logische gevolgen. Een eerste idee, SP, propageert *literals* die symmetrisch zijn aan reeds gepropageerde *literals* tijdens het zoekproces. Hiervoor introduceren we de notie van *weak activity*, een veralgemening van de *constraint programming* notie van *activity*. Onze experimenten tonen aan dat deze aanpak nuttig is, maar ook verbeterd kan worden door symmetrische propagaties uit te voeren onafhankelijk van de activiteitsstatus van een symmetrie. Dit leidt tot een tweede idee, SEL, dat symmetrische explanation clauses afleidt, en op die manier een vorm van symmetrisch leren is. Hoewel SEL een relatief simpele aanpak voor symmetrie is, tonen verdere experimenten aan dat het competitief is met BreakIDs geavanceerde symmetriebrekingstechnieken.

Symmetrie enkel aanpakken op een propositioneel niveau maakt het echter moeilijker om symmetrie te detecteren en specifieke eigenschappen van symmetriegroepen af te leiden. Deze informatie is duidelijker aanwezig op een predikaatniveau. Daarom onderzoeken we, ten derde, hoe symmetrie zich manifesteert in eerste-ordelogica. Hiertoe stellen we de notie van *local domain* symmetrie op, een vorm van symmetrie in predikatenlogica die een brede klasse aan vaak voorkomende symmetriegroepen omvat. Door verdere theoretische eigenschappen van local domain symmetrie te bewijzen, kunnen we efficiënte symmetriedetectie- en -brekingsmechanismes opstellen. Deze ideeën implementeren we in de eerste-ordemodelexpansieroutines van IDP, wat leidt tot snellere symmetriedetectie en krachtigere symmetriebreking dan alternatieve technieken. Het belangrijkste nadeel van onze aanpak op predikaatniveau is dat niet elke vorm van symmetrie als local domain symmetrie kan uitgedrukt worden.

Ten slotte tonen we aan dat er een veelbelovend verband bestaat tussen lokale zoekalgoritmes en symmetrie; een *neighborhood* voor een lokaal zoekalgoritme kan immers opgebouwd worden uit de quasisymmetriegroep van een probleem. Zoals in vorige hoofdstukken aangetoond, kunnen deze quasisymmetrieën automatisch gedetecteerd worden, wat tot een veelbelovende methode om automatisch neighborhoods op te stellen leidt. Experimenten tonen aan dat een initiële implementatie van geautomatiseerd lokaal zoeken in IDP al performantiewinst geeft.

Samengevat levert ons werk nieuwe inzichten in hoe symmetrie gedetecteerd en uitgebuit kan worden zonder menselijke inbreng. Deze ideeën in propositionele logica en predikatenlogica zijn nuttig voor alle systemen die combinatorische problemen geautomatiseerd oplossen, ook die in het veld van constraint programming, operations research, Boolean satisfiability solving, answer set programming, etc.

List of Symbols and Abbreviations

γ	Set of decision literals.
δ	Interchangeable subdomain.
$\mathcal{E}$	Explanation clause function.
$Grp(\mathcal{P})$	The permutation or symmetry group generated by $\mathcal{P}$.
λ	Learned clause store.
$\preceq_\alpha$	Lexicographic order over a set of assignments.
$LL^{\preceq_D}(\sigma)$	The lex-leader constraint for symmetry σ based on domain order $\preceq_D$.
$\preceq_\chi$	Order over the propositional vocabulary χ.
φ	A logical formula.
π	A permutation of literals or its induced symmetry.
$\mathbb{G}$	A permutation or symmetry group.
$\mathcal{P}$	A set of permutations or symmetries.
$\mathcal{SC}$	Set of symmetrical explanation clauses.
$\downarrow$	$\varphi \downarrow \alpha$ represents φ under α, the formula obtained by conjoining φ with all literals in assignment α.
l	A literal.
$d.i$	Argument vertex for domain element d and argument position index i.
$S\|i$	The ith argument position of symbol S.

$\overline{\chi}$	The set of literals over propositional vocabulary χ.
D	A domain.
$dom(I)$	The domain of a structure I.
σ_π	Global domain symmetry induced by domain permutation π.
IDP	A knowledge base system
$\mathbb{G}_\delta^A$	Subdomain interchangeability group induced by argument set A and subdomain δ.
σ_A^π	Local domain symmetry induced by argument set A and domain permutation π.
Co	Column in a variable matrix.
Ro	Row in a variable matrix.
MX	Model eXpansion
SP	Symmetry Propagation
o	An objective function.
MO	Model Optimization
χ	A vocabulary of Boolean variables for SAT problems or ground ASP programs.
sbf	A symmetry breaking formula.
SEL	Symmetrical Explanation Learning
Γ_D	A set of structures with domain D.
Γ^I	A set of structures extending I.
I	A first-order structure.
I_{in}	An input structure for a model expansion problem.
I_{in}^*	A decomposed input structure for a model expansion problem.
I_{out}	An output structure for a model expansion problem.
σ	A first-order symmetry.
$symb(\mathcal{T})$	The set of symbols occurring in a theory $\mathcal{T}$.

NP	The well-known complexity class.
P	The well-known complexity class.
$\mathcal{T}$	A first-order theory.
$\mathcal{T}^*$	A decomposed theory for a model expansion problem.
Σ	A first-order vocabulary.
Σ_{in}	An input vocabulary for a model expansion problem.
Σ_{in}^*	A decomposed input vocabulary for a model expansion problem.
Σ_{out}	An output vocabulary for a model expansion problem.
Σ^*	A decomposed vocabulary for a model expansion problem.
AP	Argument Position vertex
ASP	Answer Set Programming
CP	Constraint Programming
DE	Domain Element vertex
DPG	Domain Permutation Graph
IT	Interpretation Tuple vertex
MP	Mathematical Programming
SAT	Boolean satisfiability
SMT	Satisfiability Modulo Theories
TSP	Traveling Salesman Problem
TTP	Traveling Tournament Problem

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Broadly, the goal of a computer scientist is to study how computing machinery can help people; how it can solve their problems, manage their data, or connect them to the world. In this thesis, we focus on the problem solving part, more particularly, on solving *combinatorial problems* with computers.

A combinatorial problem is characterized as the problem of picking, from a typically finite set of candidate solutions, one or more solutions *satisfying* certain requirements or *constraints*. Sometimes, no such satisfying solution exists. Other times, a lot exist, but one wants to find the *optimal* one with regards to an optimization criterion.

A very simple example is the *pigeonhole problem*. The candidate solutions for the pigeonhole problem are assignments of a set of pigeons to a set of holes, and a satisfying solution is one that assigns a hole to each pigeon, such that no two pigeons share a hole. In essence, the pigeonhole problem asks a solver to derive Dirichlet's drawer principle, namely that you need at least as many holes as you have pigeons.

Over the years, computer scientists have developed many techniques to solve combinatorial problems with computers. Examples are *constraint programming* (CP) [8], *Boolean satisfiability solving* (SAT) [68], *answer set programming* (ASP) [66], *satisfiability modulo theories* (SMT) [13], *metaheuristics* [51] and *mathematical programming* [93].

At KU Leuven, the Knowledge Representation and Reasoning group developed the IDP system [25], a piece of software that – amongst other things – solves combinatorial problems.

1.2 The IDP system and its language

The IDP system is a *knowledge base system* [30]. The aim of a knowledge base system is to state domain knowledge in a radically declarative way, where domain *knowledge* is specified once as a *knowledge base*. Then, a user can solve multiple *problems* using builtin *inference methods*, without any modification to the knowledge base. As such, a knowledge base system is a radically declarative system, strictly separating knowledge from computation. Compared to other declarative approaches, a knowledge base discerns itself by supporting multiple *inference methods* acting on the same knowledge base.

The specification language of IDP is an instantiation of FO($\cdot$); it is based on classical first-order logic ("FO"), enriched with extensions ("$\cdot$") such as types, aggregates, arithmetic, inductive definitions and constructed types.

For this thesis, we abstract IDP as a *model expansion* (and *model optimization*) system for untyped first-order logic (FO), as this will greatly simplify matters. However, the ideas presented in the following chapters are applicable to the richer language FO($\cdot$) as well, exemplified by the fact that our implementations in IDP of the presented algorithms are still sound for IDP's extensions to FO.

An FO model expansion system takes as input (i) *global constraints* in the form of a *logical theory* and (ii) instance specific data as an *input structure*. As output, it returns a *model* satisfying the constraints and *expanding* the input structure. In the case of model optimization, IDP also aims to optimize a given *objective function*. We formalize these notions on a by-need basis in Section 5.2 and 6.2. These are not coincidentally located in the chapters on symmetry in the predicate logic context.

To calculate models, IDP makes use of a *ground-and-solve* approach, where, for a finite *domain* of relevant objects provided in the input structure, the predicate level theory and input structure are instantiated (or *grounded*) to quantor-free, often propositional formulas. MiniSAT(ID) [26], IDP's back-end solver, then employs SAT and CP technology to search for a satisfying assignment to the propositional theory, representing a valid model at the predicate level. This propositional solving step is a major motivation for the research on symmetry in a propositional logic context in Chapters 3 and 4.

A last remark on the IDP system is its support for *Prolog*-like rules that adhere to the *well-founded* semantics. These rules allow convenient modeling of recursive properties as well as *definitional* dependencies between symbols. The well-founded semantics is strongly related to the *stable semantics* for logical rules in an ASP specification [28]. As a result, the IDP system is closely related to ASP systems, not only because both have specification languages based on predicate logic, but also because of their support for *Prolog*-like rules.

1.3 Symmetry

As advanced as many combinatorial problem solving systems are, they still show significant weaknesses.

For instance, many cannot tackle the aforementioned pigeonhole problem. More specifically, even for small numbers of pigeons and holes, they need huge, unpractical amounts of computing time to decide that there exists no satisfying solution to the problem if the number of holes is smaller than the number of pigeons [54].[1] Current solvers essentially cannot bypass the factorial enumeration of all possible pigeon-to-hole mappings.

This is surprising, as a human agent can easily conclude via a counting argument that n pigeons do not fit into $m < n$ holes. One way of addressing this weakness in combinatorial solvers is via **symmetry**.

Very generally, symmetry is a transformational property of some given combinatorial problem: it transforms candidate solutions to the problem in other candidate solutions, preserving the property of whether a candidate solution is an actual solution satisfying the constraints of the problem. Many problems exhibit symmetry. Symmetries are, for instance, renaming a set of identical trucks in a routing problem, mirroring or rotating a chess board onto itself, or the automorphisms of an input graph. Or permuting the pigeons or holes in a pigeonhole problem. Symmetry has been extensively studied for languages and systems that allow to model and solve combinatorial problems [52, 85, 14, 27, 67].

Symmetry is typically defined as a *syntactical* property of problem specifications, i.e., a symmetry is a permutation of variables and/or values that preserves the original specification. Sometimes, a *semantic* notion of symmetry is given. In this case, a symmetry is a permutation of a set of candidate solutions to the combinatorial problem that preserves *satisfiability*: whether the candidate solution is an actual solution to the combinatorial problem. The syntactical notion is useful in a context of *symmetry detection*, where one wants to derive

[1] A related, even more problematic problem is the *clique coloring problem* [20].

symmetry properties of a given problem specification. The semantical notion is more general, and allows for stronger theoretical results. In this thesis, we always use the semantic notion of symmetry, with the theorems useful for symmetry detection instead specifying useful syntactical properties.

Often, we will talk about *symmetry groups*, which is the algebraic group formed by symmetries under the composition relation.

Research on symmetry of combinatorial problems poses itself two basic questions: how to *detect* symmetry, and how to *exploit* it to speed up the solving process.

1.3.1 Symmetry detection

Symmetry detection is typically done by reducing the combinatorial problem specification to some low-level (*ground* or *Boolean*) form, transforming this specification to a colored graph, and letting specialized tools [71, 60, 58] derive a set of *generators* for the *automorphism* group of this colored graph. By construction of the colored graph, these generators can be converted back to a set of generators for a symmetry group of the low-level specification. Though often effective [4, 41], this approach has the drawback of having to reduce a high-level specification to a low-level one, often blowing up the size of the specification. In addition, solving a graph automorphism problem currently requires running a worst-case exponential algorithm. A third drawback is that the derived symmetry group generators contain little information on the structure of the detected symmetry group, while this information often proves useful in later exploiting the detected symmetry.

There exist symmetry detection approaches that follow another path. Firstly, one can detect symmetry on small instances of the combinatorial problem, and use theorem proving techniques to weed out the symmetries that do not hold for all instances [74]. The drawback of this approach is that the theorem proving step is very costly, and only a limited range of symmetry can be detected. Secondly, some predicate logic-based specification languages exhibit trivial interchangeability of sets of objects, as there is no way to distinguish between different objects in the specification language. This interchangeability leads to a particular form of symmetry, which can be detected efficiently [100, 9, 23, 94, 27]. However, this technique is restricted to one particular form of symmetry. Lastly, one can detect symmetry on a "constraint by constraint" basis, where each constraint (or formula) has certain symmetry properties, and the conjunction of two constraints only leaves the intersection of both symmetry groups for each constraint [97]. This is an interesting approach, which arguably captures some of the aforementioned notions of symmetry detection. Its biggest drawback is that it is not implemented in existing combinatorial solvers, as the symmetry properties of each language feature supported by the solver need to be formalized.

This thesis improves on existing symmetry detection approaches in several regards. On the classical, low-level symmetry detection approach, in Chapter 3, we infer and exploit structure of the symmetry group generated by the symmetry generators detected by the graph automorphism approach. This results in more information to exploit symmetry formulas, leading to better performance. In Chapter 5, we detect symmetry for first-order logic specifications without *grounding* them to a low-level, propositional theory. This has the advantage of more efficient symmetry detection, as well as retaining high-level information on the structure of the detected symmetry group.

1.3.2 Symmetry exploitation

As exemplified by the performance of combinatorial solvers on the pigeonhole problem, symmetry poses a challenge to such systems. If left unadressed, for a problem exhibiting strong symmetry, a combinatorial search engine might visit each of the (potentially exponentially many) symmetric parts of the search space, and hence waste valuable time rediscovering already known information.

The most common way to prevent this is **static symmetry breaking**. This approach posts a so-called *symmetry breaking constraint* (or formula) that eliminates certain candidate solutions, while guaranteeing that for each eliminated candidate solution, at least one symmetrical candidate solution satisfies the symmetry breaking constraint [24, 1]. This way, combinatorial solvers are guided to only visit asymmetrical parts of the search space.

More formally, a symmetry group imposes an equivalence relation on the set of candidate solutions: two solutions are equivalent if there exists a symmetry mapping one to the other. Similarly, the set of candidate solutions can be partitioned into equivalence classes (*symmetry classes*). Then, a symmetry breaking constraint is *sound* if it is satisfied by *at least* one candidate solution in each symmetry class. Similarly, it is *complete* if it is satisfied by *at most* one candidate solution in each symmetry class [99]. The goal of static symmetry breaking then becomes to post small and sound symmetry breaking formulas that still are complete.

Typically, static symmetry breaking imposes some order on the set of candidate solutions, and the symmetry breaking constraint then states that a satisfying solution should not be larger than its symmetrical image under some given symmetry. As such, this constraint is sound, as the "smallest" candidate solution in a symmetry class satisfies it. However, it is typically not complete.

This is one place where knowledge on the type and structure of a symmetry group is useful: for many important types of symmetry, a small and complete symmetry breaking constraint has been devised. We take advantage of this fact in Chapters 3 and 5, where our static symmetry breaking approach can guarantee symmetry breaking completeness for a particular type of symmetry.

Another way to handle symmetry in combinatorial solvers is by **dynamic symmetry handling**. The idea here is to not modify the problem specification, but adjust the underlying solving algorithm to be aware of symmetry, and exploit this information. Dynamic symmetry handling is less restrictive than static symmetry breaking: it does not fundamentally alter the problem by removing possibly interesting or easy-to-find solutions. A more widely used term is "dynamic symmetry *breaking*", but since some of these techniques (including the ones we present in Chapter 4) instead "conserve" symmetry during search, we avoid this misnomer.

Dynamic symmetry handling techniques are diverse. Some immediately interfere with a solver's decision heuristic [84], others derive symmetrical information such as *propagations* or *learned constraints* [87, 16, 73], or simply add symmetry breaking formulas during search [3].

In Chapter 4, we investigate two dynamic symmetry handling techniques which, at their core, derive logical consequences symmetrical to logical consequences derived by the underlying search algorithm. We argue that this is a simple and effective way of dynamic symmetry handling, and link it to existing dynamic symmetry handling approaches.

Apart from studying symmetry to address the potential combinatorial blowup of search engines on highly symmetrical problems, in Chapter 6 we identify a new use case for symmetry: **automated local search**. The idea is to use symmetry properties of a problem to construct *neighborhoods*, which allow an incomplete search algorithm to "move" from one satisfying solution to another, aiming to improve some external objective function. To our knowledge, this is one of few known approaches to automate local search, efficiently solving combinatorial problems without human interaction.

1.4 Summary

In this thesis, we continue research on symmetry of model expansion problems for propositional and predicate logic specifications. The reason we use this context is because it is the natural environment of the knowledge base system IDP. Also, propositional and predicate logic form the theoretical foundations of

many languages specifying combinatorial problems, as well as systems solving them. We focus on providing fully integrated approaches, where the often separated tasks of detecting symmetry and exploiting it are tuned towards each other.

The remainder of this text is divided into five chapters. Chapter 2 refreshes concepts employed throughout the other chapters. Chapter 3 is closest to existing work, and provides a new preprocessor that detects and breaks symmetry in propositional theories. Chapter 4 takes a closer look at dynamic symmetry handling through *symmetry propagation* and *symmetrical learning*. Chapter 5 contains the main result of this thesis. Here, we investigate symmetry properties of first-order logic theories, identify important classes of symmetry and provide the theoretical foundations to efficiently detect and break the corresponding symmetry. Chapter 6 investigates a surprising link between local search and symmetry, employing symmetry detection methods to automatically derive the necessary input for local search algorithms.

Chapter 2

Preliminaries

Before moving on to the main matter, we give some basic concepts that make a recurring appearance throughout this thesis.

2.1 Permutation groups

A *permutation* is a bijection from a set to itself. We write permutations in *cycle notation*: $(a\ b\ c)(d\ e)$ is the permutation that maps a to b, b to c, c to a, swaps d with e, and maps all other elements to themselves. A *swap* is a non-trivial permutation that is its own inverse. In cycle notation, a swap consists only of cycles of length two. E.g., $(a\ b)(c\ d)$ is a swap over $\{a, b, c, d, e\}$, but $(a\ b\ c)(d\ e)$ is not.

Permutations form algebraic groups under the composition relation $(\circ)$. With π^k $(k \in \mathbb{N})$ we denote the n-fold composition of π with itself. π^{-1} denotes π's inverse. A set of permutations $\mathcal{P}$ is a set of *generators* for a permutation group $\mathbb{G}$ if each permutation in $\mathbb{G}$ is a composition of permutations from $\mathcal{P}$. The group $\mathit{Grp}(\mathcal{P})$ is the permutation group *generated* by all compositions of permutations in $\mathcal{P}$.

The *orbit* $\mathit{Orb}_{\mathbb{G}}(x)$ of an element x under a permutation group $\mathbb{G}$ is the set $\{\pi(x) \mid \pi \in \mathbb{G}\}$. The *support* $\mathit{Supp}(\pi)$ of a permutation π is the set of elements $\{x \mid \pi(x) \neq x\}$. The support $\mathit{Supp}(\mathbb{G})$ of a permutation group $\mathbb{G}$ is the union of the supports of permutations in $\mathbb{G}$.

A permutation group $\mathbb{G}$ imposes an *equivalence relation* on the set of elements S

being permuted: $x, y \in S$ are equivalent if there exists a permutation $\pi \in \mathbb{G}$ such that $y = \pi(x)$. Alternatively, two elements are equivalent if they share the same orbit. As with any equivalence relation, S can be partitioned into *equivalence classes*, with each equivalence class being the orbit of its constituents.

A permutation group $\mathbb{G}$ *stabilizes* an element x if $x \notin \mathit{Supp}(\mathbb{G})$. The *stabilizer subgroup* $\mathit{Stab}_{\mathbb{G}}(x)$ of a permutation group $\mathbb{G}$ for an element x is the group $\{\pi \in \mathbb{G} \mid \pi(x) = x\}$, or equivalently, the largest subgroup of $\mathbb{G}$ that stabilizes x.

2.2 Graph automorphisms

A *colored graph* is a tuple $G = (V, E, c)$, where V is a set, whose elements we call *nodes*, E is a binary relation on V; elements of E are called edges and c is a mapping $V \to C$ for some set C. The elements of C are called *colors*.

An automorphism of G is a permutation of its vertices $\pi \colon V \to V$ such that the following two conditions hold:

- $(u, v) \in E$ if and only if $(\pi(u), \pi(v)) \in E$ for each $u, v \in V$, and
- $c(v) = c(\pi(v))$ for each $v \in V$.

The graph automorphism problem is the task of constructing the graph automorphism group of a given (colored) graph. The complexity of this problem is conjectured to be strictly in between **P** and **NP** [11]. Several tools are available to tackle this problem, including SAUCY [60], NAUTY [71] and BLISS [58].

Chapter 3

Symmetry in Propositional Logic

Goal of the chapter
Symmetry breaking preprocessors for propositional logic have been around for quite some time. However, they are not used in the default configurations of any propositional reasoning engines. This means that, in general, the benefit of reducing the search space does not outstrip the overhead incurred for detecting and breaking symmetry. We surmise that this is due to insufficient exploitation of the *structure* of a detected symmetry group. By exploiting symmetry group structure, we could improve the performance of these preprocessors.

This chapter is based on work presented at the Fourth International Workshop on the Cross-Fertilization Between CSP and SAT – July 2014, Vienna, Austria [36]; the 19th International Conference on Theory and Applications of Satisfiability Testing – July 2016, Bordeaux, France [37]; the 9th Workshop on Answer Set Programming and Other Computing Paradigms – October 2016, New York City, USA [35].

3.1 Introduction

This chapter describes BreakID,[1] a preprocessor for propositional theories that detects symmetry and extends the propositional theory with effective symmetry breaking formulas. In essence, BreakID is an improvement to the state-of-the-art Shatter [4] – a symmetry breaking preprocessor for propositional theories in conjunctive normal form (CNF) for Boolean satisfiability solving (SAT) – and sbass [41] – a symmetry breaking preprocessor for ground answer set programs (ASP).

Shatter and sbass follow the same workflow: represent a propositional theory by a colored graph, use the automorphism detection tool Saucy to derive generators for the graph's automorphism group, convert these automorphism generators to symmetry generators, and for each of these generators construct a *lex-leader* symmetry breaking formula [24] based on a lexicographical order on the set of Boolean assignments for the propositional variables in the theory. These symmetry breaking formulas can then be added to the propositional theory, after which an out-of-the-box solver will be hindered less by the symmetry present in the problem.

Though effective, this approach does not take into account any particular properties of the detected symmetry group. For instance, the pruning power of the symmetry formulas depends heavily on the chosen set of symmetry generators and whether or not compositions of these generators are eliminated as well [62].

To address this, BreakID presents two improvements with the common theme of adjusting the set of symmetry generators for which symmetry is broken, as well as optimizing the variable order at the heart of the lexicographic assignment order. The first improvement is to detect *row interchangeability symmetry* subgroups, for which a small set of generators exists such that their lex-leader constraints do break the subgroup completely. The second is to generate *binary symmetry breaking clauses* not based on individual generators, but on algebraic properties of the entire symmetry group. These improvements can be used in conjunction with each other and with traditional lex-leader symmetry breaking.

Next to this, BreakID also boosts some technical optimizations that were known in the literature, but omitted in Shatter (and sbass to a lesser extent). BreakID uses a more compact encoding of the lex-leader symmetry breaking constraint, published before but not experimentally investigated [85]. BreakID by default limits the size of symmetry breaking formulas, which is shown to reduce symmetry breaking formula overhead [2]. BreakID limits the symmetry

[1]Pronounced "Break it!" or "Break Idea".

detection time of SAUCY, as it sometimes takes too long to derive all generators of the colored graph automorphism group.

Together, these improvements and optimizations increase the robustness and efficacy of symmetry breaking preprocessing, as we show with extensive experiments.

In this chapter, we first explain BREAKID in the context of SAT, introducing the ASP particularities later.

More specifically, after some SAT preliminaries in Section 3.2, we explain BREAKID's main improvements in Sections 3.3, 3.4 and 3.5 respectively. Section 3.6 describes how these ideas are combined to form one symmetry detecting and breaking workflow. Section 3.7 contains an extensive experimental evaluation of BREAKID on SAT problems.

Section 3.8 and 3.9 continue by establishing a (ground) ASP context for BREAKID, while Section 3.10 motivates and explains BREAKID's approach to (ground) ASP symmetry detection. Section 3.11 provides an experimental evaluation for ASP problems, and a final discussion on this chapter is given in Section 3.12.

3.2 SAT preliminaries

Satisfiability problem Let χ be a set of Boolean variables and $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$ the set of Boolean values. For each $x \in \chi$, there exist two *literals*; the *positive* literal denoted by x and the *negative* literal denoted by $\neg x$. The negation $\neg(\neg x)$ of a negative literal $\neg x$ is the positive literal x, and vice versa. The set of all literals over χ is denoted $\overline{\chi}$. A *clause* is a finite disjunction of literals, and a *formula* is a finite conjunction of clauses (as usual, we assume formulas are in conjunctive normal form (CNF)). In the context of SAT, we make no distinction between propositional formulas or *theories*.

An *assignment* α is a mapping $\chi \to \mathbb{B}$. We extend α to literals as $\alpha(\neg x) = \neg\alpha(x)$, where $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{f} = \mathbf{t}$. An assignment α *satisfies* a formula φ – $\alpha \models \varphi$ – if at least one literal from each clause is mapped to $\mathbf{t}$ by α. If $\alpha \models \varphi$, we also say that φ *holds* in α. The Boolean satisfiability (SAT) problem consists of deciding whether there exists an assignment that satisfies a propositional formula.

Symmetry in SAT Let π be a permutation of a set of literals $\overline{\chi}$. We extend π to clauses: $\pi(l_1 \vee \ldots \vee l_n) = \pi(l_1) \vee \ldots \vee \pi(l_n)$; to formulas: $\pi(c_1 \wedge \ldots \wedge c_n) = \pi(c_1) \wedge \ldots \wedge \pi(c_n)$; to assignments: $\pi(\alpha)(\pi(l)) = \alpha(l)$. By extending a literal permutation π to assignments, we are slightly abusing notation. A *symmetry* σ of a propositional formula φ over χ is a permutation over $\overline{\chi}$ that *preserves satisfaction to* φ; i.e. $\alpha \models \varphi$ iff $\pi(\alpha) \models \varphi$.

A literal permutation π (when extended to assignments) is a symmetry of a propositional formula φ over χ if the following sufficient syntactic condition is met:

- π commutes with negation: $\pi(\neg l) = \neg\pi(l)$ for all $l \in \overline{\chi}$, and
- π fixes the formula: $\pi(\varphi) = \varphi$.

It is easy to see that these two conditions guarantee that π maps assignments to assignments, preserving satisfaction to φ. Typically, only this syntactical type of symmetry is exploited, since it can be detected with relative ease – it is also the type of symmetry detected by most graph automorphism symmetry detection methods. The practical techniques presented in this chapter are no exception, though all presented theorems hold for the more general semantic notion of symmetry as well.

Symmetry breaking Symmetry *breaking* aims at eliminating symmetry, either by *statically* posting symmetry breaking constraints that invalidate symmetric assignments, or by altering the search space *dynamically* to avoid symmetric search paths. A (static) symmetry breaking formula for SAT is presented in Section 3.3. If $\mathbb{G}$ is a symmetry group, then a symmetry breaking formula ψ is *sound* if for each assignment α there exists at least one symmetry $\pi \in \mathbb{G}$ such that $\pi(\alpha)$ satisfies ψ; ψ is *complete* if for each assignment α there exists at most one symmetry $\pi \in \mathbb{G}$ such that $\pi(\alpha)$ satisfies ψ [99].

3.3 Compact CNF encodings of the lex-leader constraint

A classic approach to static symmetry breaking is to construct *lex-leader constraints.*

Definition 3.3.1 (Lex-leader constraint [24])**.** Let π be a symmetry of some formula over χ, $\preceq_\chi$ an order on χ and $\preceq_\alpha$ the lexicographic order induced by $\preceq_\chi$ on the set of χ-assignments. A formula LL_π over $\chi' \supseteq \chi$ is a *lex-leader constraint* for π if for each χ-assignment α, there exists a χ'-extension of α that satisfies LL_π iff $\alpha \preceq_\alpha \pi(\alpha)$.

In other words, each assignment whose symmetric image under π is smaller, is eliminated by LL_π. It is easy to see that the conjunction of LL_π for all π in some $\mathbb{G}' \subseteq \mathbb{G}$ is a sound (but not necessarily complete) symmetry breaking constraint for $\mathbb{G}$.

An efficient encoding of the lex-leader constraint LL_π as a conjunction of clauses is given by Aloul et al. [1], where each variable in $\mathit{Supp}(\pi)$ leads to 2 clauses of size 3 and 2 clauses of size 4. Below, we give a derivation of a more compact encoding of LL_π as a conjunction of 3 clauses of size 3 for each variable in $\mathit{Supp}(\pi)$, which is more compact and hence reduces the overhead introduced by posting it. A similar encoding is presented by Sakallah [85] but has not been experimentally evaluated before.

Theorem 3.3.2 (Compact encoding of lex-leader constraint)**.** *Let π be a symmetry, let $\mathit{Supp}(\pi) = \{x_1, \ldots, x_n\}$ be ordered such that $x_i \preceq_\chi x_j$ iff $i \leq j$ and let $\{y_0, \ldots, y_{n-1}\}$ be a set of auxiliary variables disjoint from $\mathit{Supp}(\pi)$. The following set of clauses is a lex-leader constraint for π:*

$$\begin{array}{rl|rl} y_0 & & y_j \vee \neg y_{j-1} \vee \neg x_j & 1 \leq j < n \\ \neg y_{i-1} \vee \neg x_i \vee \pi(x_i) & 1 \leq i \leq n & y_j \vee \neg y_{j-1} \vee \pi(x_j) & 1 \leq j < n \end{array}$$

Proof. Crawford et al. proposed the following lex-leader constraint [24]:

$$\forall i : (\forall j < i : x_j \Leftrightarrow \pi(x_j)) \Rightarrow \neg x_i \vee \pi(x_i) \tag{3.1}$$

Assuming $\mathbf{f} < \mathbf{t}$, this constraint expresses that the value of a variable x_i must be less than or equal to the value of $\pi(x_i)$ if, for all smaller variables x_j, x_j has the same value as $\pi(x_j)$. As such, it encodes a valid lex-leader constraint.

Aloul et al. [1] noticed that the antecedent $(\forall j < i : x_j \Leftrightarrow \pi(x_j))$ is recursively reified by introducing auxiliary variables y_j:

$$y_j \Leftrightarrow (y_{j-1} \wedge (x_j \Leftrightarrow \pi(x_j))) \tag{3.2}$$

where the base case y_0 is fixed to be true. In essence, y_j holds iff x_k and $\pi(x_k)$ have the same truth value for $1 \leq k \leq j$. Equation (3.1) then translates to:

$$y_0 \tag{3.3}$$

$$y_j \Leftrightarrow (y_{j-1} \wedge (x_j \Leftrightarrow \pi(x_j))) \qquad 1 \leq j < n \tag{3.4}$$

$$y_{i-1} \Rightarrow \neg x_i \vee \pi(x_i) \qquad 1 \leq i \leq n \tag{3.5}$$

Note that by (3.5), if y_{j-1} holds, then $\neg x_j \vee \pi(x_j)$ holds. Hence, $y_{j-1} \wedge (x_j \Leftrightarrow \pi(x_j))$ simplifies to $y_{j-1} \wedge (x_j \vee \neg\pi(x_j))$, and (3.4) simplifies to:

$$y_j \Leftrightarrow (y_{j-1} \wedge (x_j \vee \neg\pi(x_j))) \qquad 1 \leq j < n \tag{3.4'}$$

Lastly, observe that y_j only occurs negatively in formula (3.5). Thus, only one implication in the definition of y_j (3.4') is needed to enforce (3.5). Relaxing the constraints when y_j must be false leads to:

$$y_j \Leftarrow (y_{j-1} \wedge (x_j \vee \neg\pi(x_j))) \qquad 1 \leq j < n \qquad (3.4")$$

Working out the implications and applying distributivity of $\wedge$ and $\vee$ in equations $(3.3, 3.4'', 3.5)$ leads to the CNF formula in this theorem and hence, concludes the proof. □

The relaxation introduced by step (3.4") does not weaken the symmetry breaking capacity of the encoding, as it only weakens the constraints on auxiliary variables not permuted by any original symmetry. However, (3.4") still is weaker than (3.4'). In Section 3.7 we give experimental results that compare the presented relaxed encoding with an "unrelaxed" clausal encoding based on (3.4'), having an extra binary and ternary clause. It turns out that the relaxation of (3.4") is beneficial and leads to slightly less overhead.

Note that the condition y_j is satisfied in fewer assignments as j increases, so the marginal effect of posting the above constraints decreases as j increases. Still, the marginal cost is stable at three clauses and one auxiliary variable regardless of j. Because of this, the size of lex-leader constraints is often limited by putting an upper bound k on the number of auxiliary variables to be introduced [2], resulting in a shorter lex-leader constraint LL^k_π. BreakID also employs this limit on the size of its lex-leader constraints, by default posting a conservative LL^{50}_π for each generator symmetry π[2]

not part of a row interchangeability group.

3.4 Exploiting row interchangeability

An important type of symmetry is *row interchangeability*, which is present when a subset of variables can be structured as a two-dimensional matrix such that each permutation of the rows induces a symmetry. This form of symmetry is common; often it stems from an interchangeable set of objects in the original problem domain, with each row of variables expressing certain properties of one particular object. Examples are intercheangeability of pigeons or holes in a pigeonhole problem, interchangeability of nurses in a nurse scheduling problem, fleets of interchangeable trucks in a delivery system etc. Exploiting this type of

[2]Symmetry breaking formulas for symmetry generators of row interchangeability symmetry groups are not limited in size, but written out in full. See also Section 3.6.1.

matrix symmetry with adjusted symmetry breaking techniques can significantly improve SAT performance [43, 65]. In this section, we present a novel way of automatically dealing with row interchangeability in SAT.

Example 3.4.1 (Row interchangeability in graph coloring)**.** Let φ be a CNF formula expressing the graph coloring constraint that two directly connected vertices cannot have the same color. Let $\chi = \{x_{11}, \ldots, x_{nm}\}$ be the set of variables, with intended interpretation that x_{ij} holds iff vertex j has color i. Given the nature of the graph coloring problem, all colors are interchangeable, so each color permutation ρ induces a symmetry π_ρ of φ. More formally, the color interchangeability symmetry group consists of all symmetries

$$\pi_\rho : \overline{\chi} \to \overline{\chi} : x_{ij} \mapsto x_{\rho(i)j}, \neg x_{ij} \mapsto \neg x_{\rho(i)j}$$

If we structure χ as a matrix where x_{ij} occurs on row i and column j, then each permutation of rows corresponds to a permutation of colors, and hence a symmetry. ▲

Definition 3.4.2 (Row interchangeability in SAT)**.** A *variable matrix* M is a bijection $M : Ro \times Co \to \chi'$ from two sets of indices Ro and Co to a set of variables $\chi' \subseteq \chi$. We refer to $M(r, c)$ as x_{rc}. The r'th *row* of M is the sequence of variables $[x_{r1}, \ldots, x_{rm}]$, the c'th *column* is the sequence $[x_{1c}, \ldots, x_{nc}]$. A formula φ exhibits *row interchangeability* symmetry if there exists a variable matrix M such that for each permutation $\rho : Ro \to Ro$

$$\pi^M_\rho : \overline{\chi}' \to \overline{\chi}' : x_{rc} \mapsto x_{\rho(r)c}, \neg x_{rc} \mapsto \neg x_{\rho(r)c}$$

is a symmetry of φ. The row interchangeability symmetry group of a matrix M is denoted as R_M.

A useful property of row interchangeability is that it is broken completely by only a linearly sized symmetry breaking formula [43, 90]. We formalize this constraint programming (CP) result in a SAT context:

Corollary 3.4.3 (Complete symmetry breaking for row interchangeability)**.** *Let φ be a formula and R_M a row interchangeability symmetry group of φ with $Ro = \{1, \ldots, n\}$ and $Co = \{1, \ldots, m\}$. Using a total variable order $\preceq_\chi$ on χ, then the conjunction of $\preceq_\chi$-based lex-leader constraints for all symmetries $\pi^M_{(k\ k+1)}$ with $1 \leq k < n$, breaks M_R completely if, $x_{ij} \preceq_\chi x_{i'j'}$ iff $i < i'$ or $(i = i'$ and $j \leq j')$.*

Corollary 3.4.3 guarantees that if the variable order on which the lex-leader constraints are based follows the structure of the matrix, then the lex-leader constraints for the swap of each two subsequent rows form a complete symmetry breaking formula for row interchangeability. The condition that the order "matches" the variable matrix is important: the theorem no longer holds without it.

If we are able to detect that a formula exhibits row interchangeability, we can break it completely by choosing an appropriate variable order and posting the right lex-leader constraints. In practice, symmetry detection tools for SAT only present us with a set of generators for the symmetry group, which contains no information on the *structure* of this group. The challenge is to derive row interchangeability subgroups from these generators.

3.4.1 Row interchangeability detection algorithm

Given a set of generators P for a symmetry group $\mathbb{G}$ of a formula φ, the task at hand is to detect a variable matrix M that represents a row interchangeability subgroup $R_M \subseteq \mathbb{G}$. We present an algorithm that is sound, but incomplete in the sense that it does not guarantee that all row interchangeability subgroups present are detected.

The first step is to find an initial row interchangeable variable matrix M consisting of three rows. This is done by selecting two swap symmetries π_1 and π_2 that represent two row swaps sharing a row r. More formally, suppose π_1, π_2 and r satisfy the following conditions:

1. $\pi_1 = \pi_1^{-1}$ and $\pi_2 = \pi_2^{-1}$,
2. $r = Supp(\pi_1) \cap Supp(\pi_2)$,
3. $Supp(\pi_1) = r \cup \pi_1(r)$ and $Supp(\pi_2) = r \cup \pi_2(r)$, and
4. r, $\pi_1(r)$ and $\pi_2(r)$ are pairwise disjoint.

Then, r, $\pi_1(r)$ and $\pi_2(r)$ form three rows of a row interchangeable variable matrix, and π_1 and π_2 are swaps of those rows. Indeed, from 1., 2. and 3. it follows that π_1 and π_2 both swap some set of variables r with its images $\pi_1(r)$ and $\pi_2(r)$, so r forms the first row. 4. then guarantees that π_1 and π_2 map r to a different, disjoint set, forming the second and third row.

If, after inspecting all pairs of swaps in P, no initial three-rowed matrix satisfies the above conditions, the algorithm stops, in which case we do not know whether a row interchangeability subgroup exists. However, our experiments indicate that for many problems, an initial three-rowed matrix can be derived from a detected set of generator symmetries.

The second step maximally extends the initial variable matrix M with new rows. The idea is that for each symmetry $\pi \in P$ and each row r of M, $\pi(r)$ is a candidate row to add to M. This is the case if $\pi(r)$'s literals are disjoint from M's literals and swapping $\pi(r)$ with r is a syntactical symmetry of φ.

```
input  : CNF formula $\varphi$, set of generator symmetries $P$ for $\varphi$
output : Row interchangeable variable matrix $M$
1  identify two swaps $\pi_1, \pi_2 \in P$ that induce an initial variable matrix $M$ with 3
     rows;
2  repeat
3      foreach permutation $\pi$ in $P$ do
4          foreach row $r$ in $M$ do
5              if $\pi(r)$ is disjoint from $M$ and swapping $r$ and $\pi(r)$ is a symmetry
                 of $\varphi$ then
6                  add $\pi(r)$ as a new row to $M$;
7              end
8          end
9      end
10 until no extra rows are added to $M$;
11 return $M$;
```

Algorithm 1: Row interchangeability detection

Pseudocode is given in Algorithm 1. This algorithm terminates since both P and the number of rows in any row interchangeability matrix are finite. The algorithm is sound: each time a row is added, it is interchangeable with at least one previously added row and hence, by induction, with all rows in M. If k is the largest support size of a symmetry in P, then finding an initial row interchangeable matrix based on two row swap symmetries in P takes $O(|P|^2k)$ time. With an optimized implementation that avoids duplicate combinations of generators and rows, extending the initial matrix with extra interchangeable rows has a complexity of $O(|P||Ro||\varphi|k)$, with Ro the set of row indices of M. Algorithm 1 then has a complexity of $O(|P|^2k + |P||Ro||\varphi|k)$.

As mentioned before, the algorithm is not complete: it might not be possible to construct an initial matrix, or even given an initial matrix, there is no guarantee to detect all possible row extensions, as only the set of generators instead of the whole symmetry group is used to calculate a new candidate row.

It is straightforward to extend Algorithm 1 to detect multiple row interchangeability subgroups. After detecting a first row interchangeability subgroup R_M, remove any generators from P that also belong to R_M. This can be done by standard algebraic group membership tests, which are efficient for interchangeability groups [89]. Then, repeat Algorithm 1 with the reduced set of generator symmetries until no more row interchangeability subgroups are detected.

Example 3.4.4 (Example 3.4.1 continued.)**.** Let φ and the x_{ij} be as in Example 3.4.1. Suppose we have five colors and three vertices. Vertex 1 is connected to vertex 2 and 3; vertices 2 and 3 are not connected. This problem has a symmetry group $\mathbb{G}$ induced by the interchangeability of the colors and by a swap on vertex 2 and 3. A set of generators for $\mathbb{G}$ is $\{\pi_{(1\ 2)}, \pi_{(2\ 3)}, \pi_{(3\ 4)}, \pi_{(4\ 5)}, \nu_{2\ 3}\}$, where $\pi_{(i\ j)}$ is the symmetry that swaps colors i and j (as in the previous example), and ν_{23} is the symmetry obtained by swapping vertices 2 and 3, i.e.,[3]

$$\nu_{23} = (x_{12}\ x_{13})(x_{22}\ x_{23})(x_{32}\ x_{33})(x_{42}\ x_{43})(x_{52}\ x_{53}).$$

For these generators, it is obvious that the swaps $\pi_{(ij)}$ generate a row interchangeability symmetry group. However, a symmetry detection tool might return the alternative set of symmetry generators $P = \{\pi_{(1\ 2)}, \pi_{(2\ 3)}, \sigma_1, \sigma_2, \nu_{23}\}$ with

$$\sigma_1 = \pi_{(1\ 3)} \circ \pi_{(3\ 5)} = (x_{11}\ x_{31}\ x_{51})(x_{12}\ x_{32}\ x_{52})(x_{13}\ x_{33}\ x_{53})$$

$$\sigma_2 = \pi_{(3\ 4)} \circ \nu_{23} = (x_{12}\ x_{13})(x_{22}\ x_{23})(x_{31}\ x_{41})(x_{32}\ x_{43})(x_{42}\ x_{33})(x_{52}\ x_{53}).$$

The challenge is to detect the color interchangeability subgroup from the symmetry group generated by P.

The first step of Algorithm 1 searches for two swaps in P that combine to a 3-rowed variable matrix. $\pi_{(1\ 2)}$ and $\pi_{(2\ 3)}$ fit the bill, resulting in a variable matrix M with rows:

$$[x_{11}, x_{12}, x_{13}] \quad [x_{21}, x_{22}, x_{23}] \quad [x_{31}, x_{32}, x_{33}]$$

Applying σ_1 on the third row results in:

$$[x_{51}, x_{52}, x_{53}]$$

which after a syntactical check on φ is confirmed to be a new row to add to M.

Unfortunately, the missing row $[x_{41}, x_{42}, x_{42}]$ is not derivable by applying any generator in P on rows in M, so the algorithm terminates. ▲

The failure of detecting the missing row in Example 3.4.4 stems from the fact that the generators σ_1 and σ_2 are obtained by complex combinations of

[3] We omit negative literals from the cycle notation, noting that a symmetry always commutes with negation.

symmetries in the interchangeability subgroup and the symmetry ν_{23}. This inspires a small extension of Algorithm 1. As soon as the algorithm reaches a fixpoint, we call the original symmetry detection tool to search for a set of generators of the subgroup that stabilizes all but one rows of the matrix M found so far. This results in “simpler” generators that do not permute the literals of the excluded rows. Tests on CNF instances show that this simple extension, although giving no theoretical guarantees, often manages to find new generators that, when applied on the current set of rows, construct new rows. After detecting row stabilizing symmetries, Algorithm 1 resumes from line 2, aiming to extend the matrix further by applying the extended set of generators. This process ends when even the new generators can no longer derive new rows.

Example 3.4.5 (Example 3.4.4 continued.)**.** The only symmetry of the problem that stabilizes the variables $\{x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}\}$ is $\pi_{(4\ 5)}$, which has the missing row $[x_{41}, x_{42}, x_{42}]$ as image of the fifth row.

The matrix, which now contains all variables, allows one to completely break the color interchangeability. The symmetry between vertices 2 and 3 is not expressed in the matrix, but can still be broken by a regular lex-leader constraint, as described in Section 3.6. ▲

3.5 Generating binary symmetry breaking clauses

Note that state-of-the-art symmetry breaking preprocessors only post lex-leader constraints for a set of generators of a symmetry group $\mathbb{G}$, instead of posting lex-leader constraints for all symmetries in $\mathbb{G}$. The reason is obvious: $\mathbb{G}$ typically contains too many symmetries. However, the price we pay is weaker symmetry breaking constraints, straying away from the ideal of complete symmetry breaking.

An alternative we explore in this section is to post only a very short lex-leader constraint, namely LL^0_π, but do this for a large number of $\pi \in \mathbb{G}$ instead of only for generators for $\mathbb{G}$. As already mentioned in Section 3.3, the first parts of the lex-leader constraint breaks comparatively more symmetry than later parts, so in that sense, posting LL^0_π is a cost-effective way of breaking π.

LL^0_π is the lex-leader constraint with 0 auxiliary variables, which is equivalent to the binary clause $\neg x \vee \pi(x)$ with x the smallest variable in $Supp(\pi)$ according to $\preceq_\chi$. To construct as many of these binary clauses as possible without enumerating the whole symmetry group $\mathbb{G}$, we use a greedy approach that starts from the generators of $\mathbb{G}$ and exploits the freedom to choose the variable order $\preceq_\chi$ as well as the fact that one can easily compute the orbit of a literal in $\mathbb{G}$.

Theorem 3.5.1 (Binary symmetry breaking clauses)**.** *Let $\mathbb{G}$ be a non-trivial symmetry group of φ, $\preceq_\chi$ an ordering of χ, and x^* the $\preceq_\chi$-smallest variable in $Supp(\mathbb{G})$. For each $x \in Orb_{\mathbb{G}}(x^*)$, the binary clause $\neg x^* \vee x$ is entailed by LL_π for some $\pi \in \mathbb{G}$.*

Proof. If $x = x^*$, the theorem is trivially true. If $x \neq x^*$, there exists a $\pi \in \mathbb{G}$ with $\pi(x^*) = x$ since $x \in Orb_{\mathbb{G}}(x^*)$. Since x^* is the smallest variable in $Supp(\mathbb{G})$, it is also the smallest in $Supp(\pi)$. Theorem 3.3.2 shows that y_0 and $\neg y_0 \vee \neg x^* \vee \pi(x^*)$ are two clauses in LL_π. Resolution on y_0 leads to $\neg x^* \vee \pi(x^*)$, where $\pi(x^*) = x$. □

Theorem 3.5.1 allows to construct small lex-leader clauses for $\mathbb{G}$ without enumerating individual members of $\mathbb{G}$; it suffices to compute the orbit of the smallest variable in $Supp(\mathbb{G})$ to derive a set of binary symmetry breaking clauses. Theorem 3.5.1 holds for all symmetry groups, so also for any subgroup $\mathbb{G}'$ of $\mathbb{G}$. In particular, if $\mathbb{G}'$ stabilizes the smallest variable in $Supp(\mathbb{G})$, applying Theorem 3.5.1 to $\mathbb{G}'$ results in different clauses than applying it to $\mathbb{G}$, as $\mathbb{G}'$ has a different smallest variable in its support.

Example 3.5.2. Let $P = \{(a\ b)(c\ d\ ef)\}$ and $\mathbb{G} = Grp(P) = \{(a\ b)(c\ d\ e\ f), (a\ b)(c\ f\ e\ d), (c\ e)(d\ f)\}$.[4] With order $a \preceq_\chi b \preceq_\chi c \preceq_\chi d \preceq_\chi e \preceq_\chi f$, a is the $\preceq_\chi$-smallest variable of $Supp(\mathbb{G})$. Theorem 3.5.1 guarantees that $\neg a \vee b$ is a consequence of the lex-leader constraints for $\mathbb{G}$. Let $\mathbb{G}' = Stab_{\mathbb{G}}(a) = \{(c\ e)(d\ f)\}$, then c is the $\preceq_\chi$-smallest variable of $Supp(\mathbb{G}')$, hence also $\neg c \vee e$ is entailed by the lex-leader constraints for $\mathbb{G}$.

If we assume a different order $\preceq'_\chi$, different binary clauses are obtained. For instance, let c be the $\preceq'_\chi$-smallest variable of $Supp(\mathbb{G})$. Then Theorem 3.5.1 allows us to post the clauses $\neg c \vee d, \neg c \vee e$ and $\neg c \vee f$ as symmetry breaking clauses. The stabilizer subgroup $Stab_{\mathbb{G}}(c)$ is empty, so no further binary clauses can be derived for this order[5]. ▲

A *stabilizer chain* is a sequence of stabilizer subgroups starting with the full group $\mathbb{G}$ and ending with the trivial group containing only the identity, where each next subgroup in the chain stabilizes an extra element. Given a variable order $\preceq_\chi$, applying Theorem 3.5.1 to each subgroup in a stabilizer chain stabilizing literals according to $\preceq_\chi$ for a symmetry group $\mathbb{G}$, is equivalent to constructing all LL^0_π for $\pi \in \mathbb{G}$ under $\preceq_\chi$ [57]. This stabilizer chain idea was also used by

[4]We again omit negative literals in cycle notation.

[5]Lex-leader formulas based on different variable orders are incompatible, as there is no guarantee that for each symmetry class, at least one output structure satisfies the conjunction of these different lex-leader fomrulas. As these binary clauses are still lex-leader clauses, those originating from different orders can not be combined.

Puget to efficiently break all-different constraints in a constraint programming context [83].

However, as shown by Example 3.5.2, the variable order influences the number of binary symmetry clauses derivable by a stabilizer chain of $\mathbb{G}$. We present an algorithm that, given a set of generator symmetries P for symmetry group $\mathbb{G}$, decides a total order on a subset of variables, and constructs binary symmetry breaking clauses for those variables based on a simultaneously constructed sequence of subgroups stabilizing those variables. The constructed sequence of subgroups stabilizing the literals is no actual stabilizer chain, as each of the subgroups equals $Grp(P')$ for some subset $P' \subseteq P$. The advantage of this approach is simplicity of the algorithm and low computational complexity, although it would be interesting future work to compute an actual stabilizer chain using for instance the *Schreier-Sims algorithm* [89].

In detail, our algorithm starts with an empty variable order Ord and a copy Q of the given set of generators P. It iteratively chooses a suitable variable x^* as next in the variable order, constructs binary clauses based on $Grp(Q)$, and removes any permutations $\pi \in Q$ for which $x^* \in Supp(\pi)$. As a result, at each iteration, $Grp(Q)$ stabilizes all variables in Ord except the last variable x^*, allowing the construction of binary symmetry breaking clauses $\neg x^* \vee x$ for each $x \in Orb_{Grp(Q)}(x^*)$, as per Theorem 3.5.1.

A suitable next variable x^* is one that induces a high number of binary symmetry breaking clauses, but removes few symmetries from Q so that the following iterations of the algorithm still have a reasonably sized symmetry group to work with. One way to satisfy these requirements is to pick x^* such that $Orb_{Grp(Q)}(x^*)$ is maximal, and $\{\pi \in Q \mid \pi(x^*) \neq x^*)\}$ is minimal compared to other literals of x^*'s orbit.

Pseudocode is given in Algorithm 2. This algorithm terminates, as while $Q \neq \emptyset$, x^* belongs to a largest orbit of $Grp(Q)$, so $x^* \neq \pi(x^*)$ for at least one $\pi \in Q$. As a result, Q shrinks in size during each iteration, eventually becoming the empty set. The complexity of Algorithm 2 is dominated by finding the largest orbit of $Grp(Q)$, which is $O(|Q||\chi|)$, resulting in a total complexity of $O(|P|^2|\chi|)$.

In the worst case, $O(|Supp(\mathbb{G})|^2)$ binary clauses are constructed by Algorithm 2. In particular, if some subgroup $\mathbb{G}'$ of $\mathbb{G}$ represents an interchangeable set of n variables, $n(n-1)/2$ binary clauses are derived. However, in this case $\mathbb{G}'$ also represents a row interchangeability symmetry group, which is completely broken by techniques from Section 3.4. Performing row interchangeability detection and breaking before binary clause generation can avoid quadratic sets of binary clauses. Section 3.6 shows this is indeed the order by which BreakID performs its symmetry breaking.

```
input  : Set of generator symmetries $P$
output : Conjunction of binary symmetry breaking clauses $LL^{bin}$, partial
         variable order $Ord$
1  initialize $Q = P$, $Ord$ as an empty list, $LL^{bin} = \emptyset$;
2  while $Q \neq \emptyset$ do
3      $O$ is a largest orbit of $Grp(Q)$;
4      $x^*$ is a variable in $O$ for which $\{\pi \in Q \mid \pi(x^*) \neq x^*)\}$ is minimal;
5      add $x^*$ to $Ord$ as last variable;
6      foreach $x \in O$ do
7          add $\neg x^* \vee x$ to $LL^{bin}$;
8      end
9      $Q = \{\pi \in Q \mid \pi(x^*) = x^*\}$;
10 end
11 return $LL^{bin}$, $Ord$;
```

Algorithm 2: Binary symmetry breaking clause generation

3.6 Putting it all together as BreakID

This section describes how the improvements presented in the previous section combine with each other and with standard symmetry breaking techniques in the symmetry breaking preprocessor BreakID.

BreakID has been around since 2013, when a preliminary version obtained the gold medal in the hard combinatorial sat+unsat track of 2013's SAT competition [12]. This early version incorporated all of Shatter's symmetry breaking techniques and used a primitive row interchangeability detection algorithm that enumerated symmetries to detect as many row swap symmetries as possible [36]. We developed BreakID2 in 2015, using the ideas presented in the previous sections. BreakID2 entered the main track of 2015's SAT race in combination with Glucose 4.0, placing 10th, ahead of all other Glucose variants. The experiments in the next section are run with a slightly updated version – BreakID2.1– which has more usability features and reduced memory overhead. BreakID2.1 also won the gold medal in the *no limit* track of the 2016 SAT competition [55] in combination with the SAT solver COMiniSatPS [81].

For the remainder of this chapter, we use BreakID to refer to the particular implementation BreakID2.1. BreakID's source code is published online [34].

3.6.1 BreakID's high level algorithm

Preprocessing a formula φ by symmetry breaking with BreakID starts by removing duplicate clauses and duplicate literals in clauses from φ, as Saucy cannot handle duplicate edges. Then, a call to Saucy constructs an initial generator set P of the syntactical symmetry group of φ.

Thirdly, BreakID detects row interchangeability subgroups R_M of $Grp(P)$ by Algorithm 1. The program incorporates the variables of the support of all R_M in a global variable order Ord such that the conjunction of $LL_{\pi_\rho^M}$ under Ord for all subsequent row swaps π_ρ^M forms a complete symmetry breaking formula for R_M.[6] After adding the complete symmetry breaking formula of each R_M to an initial set of symmetry breaking clauses ψ, we also remove all symmetries in P that belong to some R_M, since these symmetries are broken completely already.

Next, using the pruned P, binary clauses for $Grp(P)$ are constructed by Algorithm 2, which simultaneously decides a set of variables to be smallest under Ord.[7]

Finally, Ord is supplemented with leftover variables until it is total, and limited lex-leader constraints LL_π^{50} are constructed for each π left in P. These lex-leader constraints incorporate two extra refinements also used by Shatter; one for *phase-shifted* variables and one for the *largest variable in a symmetry cycle* [1].

Algorithm 3 gives pseudocode for BreakID's high-level routine described above.

3.7 SAT experiments with BreakID

In this section, we verify the effectiveness of the proposed techniques separately, and investigate the feasibility of using BreakID in the application and hard-combinatorial track of 2014's SAT competition. We use eight benchmark sets:

- **app14**: the application track of 2014's SAT competition (300 instances)
- **app14sym**: subset of **app14** for which Saucy detected symmetry (164 instances)

[6]In case two detected row interchangeability matrices overlap, it is not always possible to choose the order on the variables so that both are broken completely. In this case, one of the row interchangeability groups will not be broken completely.

[7]A small adaptation to Algorithm 2 ensures BreakID only selects smallest variables that are not permuted by a previously detected row interchangeability group.

```
input  : CNF formula $\varphi$
output : Symmetry breaking formula $\psi$
1  remove duplicate clauses from $\varphi$ and duplicate literals from clauses in $\varphi$;
2  run **Saucy** to detect a set of symmetry generators $P$;
3  initialize $\psi$ as the empty formula and $Ord$ as an empty sequence of ordered
     variables;
4  while Algorithm 1($P$,$\varphi$) derives a new row interchangeability group $R_M$ do
5      add complete symmetry breaking clauses for $R_M$ to $\psi$;
6      add $Supp(R_M)$ to the back of $Ord$ accordingly;
7      remove $P \cap R_M$ from $P$;
8  end
9  Algorithm 2($P$) computes binary clauses $LL^{bin}$ to add to $\psi$ and partial order
     $Ord'$ to insert at the front of $Ord$;
10 add leftover variables to the middle of $Ord$;
11 foreach $\pi \in P$ do
12     add $LL^{50}_{\pi}$ to $\psi$, utilizing Shatter's optimizations;
13 end
14 return $\psi$;
```

Algorithm 3: Symmetry breaking by BreakID

- **hard14**: the hard-combinatorial track of 2014's SAT competition (300 instances)
- **hard14sym**: subset of **hard14** for which Saucy detected symmetry (159 instances)
- **pigeon**: 8 unsatisfiable pigeonhole instances
- **urquhart**: 6 unsatisfiable Urquhart instances
- **channel**: 10 unsatisfiable channel routing instances
- **color**: 10 unsatisfiable graph coloring instances

Pigeonhole and Urquhart problems are provably hard for purely resolution-based SAT solvers, in the sense that even for very small instances astronomical running time is needed to decide satisfiability of the problem [54, 96]. The employed channel routing and graph coloring instances are highly symmetric, exhibiting strong row interchangeability. They are taken from SymChaff's benchmark set [84]. The graph coloring instances were also used in 2005 and 2007's SAT competitions.

As SAT-solver, we use Glucose 4.0 [10], which is based on MiniSAT [42]. We include the symmetry breaking preprocessor Shatter [4] bundled with Saucy

3.0 [60] in our experiments. The resources available to each experiment were 10GB of memory and 3600s on an Intel® Xeon® E3-1225 cpu. The operating system was Ubuntu 14.04 with Linux kernel 3.13. Unless noted otherwise, all results *include* any preprocessing step, such as deduplicating the input CNF, symmetry detection by SAUCY and symmetry breaking clause generation by SHATTER or BREAKID. Resources to reproduce these experiments are available online [31].

3.7.1 Compact symmetry breaking clauses

We first investigate the influence of the compact lex-leader encoding presented in Section 3.3. The experiment consists of running BREAKID with the *standard* encoding used in SHATTER (four clauses for each variable in a symmetry's support), with BREAKID's default *compact* encoding (three clauses), and with an *unrelaxed* encoding that does not relax the constraints on the auxiliary variables (five clauses). To focus on the difference between the encodings, in this experiment, BREAKID does not exploit row interchangeability, does not generate binary clauses, and does not limit the size of the lex-leader formulas. The benchmark sets employed are **app14sym**, **hard14sym**, **pigeon**, **urquhart**, **channel** and **color**. The results are presented in Table 3.1.

The theoretical advantage of having a more compact encoding is not translated into a significant increase in the number of solved instances. We do observe average runtime and memory consumption correlating with the size of the encoding, being lowest for the compact encoding and highest for the unrelaxed encoding. We conclude that none of the clausal encodings strongly outperforms the others. That said, the compact encoding enjoys a small runtime and memory advantage over both other encodings.

	pigeons	**urquhart**	**channel**	**color**	**app14sym**			**hard14sym**		
	solved	solved	solved	solved	avg mem	avg time	solved	avg mem	avg time	solved
standard	4	3	2	3	334MB	597.6s	**113**	323MB	662.9s	106
unrelaxed	4	3	2	3	349MB	611.1s	**113**	336MB	708.8s	107
compact	4	3	2	3	**329MB**	**589.6s**	112	**305MB**	**638.0s**	**108**

Table 3.1: Number of solved instances for standard, unrelaxed and compact lex-leader encoding, as well as average runtime and memory consumption of Glucose (excluding BreakID's preprocessing) for solved instances.

	BreakID()	BreakID(b)		BreakID(r)		BreakID(r,b)		
	solved	(b) clauses	solved	(r) clauses	solved	(b) clauses	(r) clauses	solved
app14sym	113	37552	111	10245	**114**	190	10245	**114**
hard14sym	108	207719	105	2926	**112**	308	2926	110
pigeon	4	427	3	1627	**8**	0	1627	**8**
urquhart	3	99	**6**	0	3	99	0	**6**
channel	2	9893	2	15421	**10**	0	15421	**10**
color	3	1469	4	1481	5	656	1481	**6**

Table 3.2: Number of solved instances for BreakID configurations with and without (r)ow-interchangeability and (b)inary clauses. Also includes average number of corresponding symmetry breaking clauses introduced.

3.7.2 Row interchangeability and binary clauses

To assess the influence of exploiting row interchangeability and binary clauses, we set up an experiment with four versions of BreakID:

- BreakID(): without both row interchangeability and binary clauses
- BreakID(r): with row interchangeability and without binary clauses
- BreakID(b): without row interchangeability and with binary clauses
- BreakID(r,b): with both row interchangeability and binary clauses

Each of these versions uses the compact encoding, and limits the lex-leader formula of each symmetry to introduce a maximum 50 auxiliary variables, irrespective of problem or symmetry size. Lex-leader formulas for swaps used to completely break row interchangeability are not limited in this regard. The results are summarized in Table 3.2.

A first observation is that the binary clause improvement shows mixed results. It performs very well on **urquhart**, allowing all instances to be solved in less than a second, but struggles with **app14sym** and **hard14sym** instances. The main reason is the huge amount of binary clauses derived, amounting over 5 million on some instances. However, activating row interchangeability fixes this problem by not allowing symmetries from row interchangeability groups to be used to construct binary clauses.

The row interchangeability improvement is more successful, improving performance on all benchmark sets except **urquhart**. Focusing on **pigeon**, full row interchangeability is detected for all instances, so each instance became polynomially solvable given the presence of symmetry breaking clauses. This is a significant improvement to the preliminary version of BreakID [36]. A similar effect is seen for **channel**, where activating row interchangeability allows deciding all instances in less than a minute. For the benchmark set as a whole, row interchangeability was detected in 54% of the instances for which Saucy could detect symmetry.

We conclude that row interchangeability exploitation is a significant improvement, while binary clauses have the potential to improve performance on certain types of problems. Furthermore, row interchangeability compensates for weaknesses of the binary clause approach, and the combination of the two yields the best overall performance.

	Glucose	Shatter		BreakID		BreakID(100s)	
	solved	pre-time	solved	pre-time	solved	pre-time	solved
pigeon	2	0.0s	3	0.1s	**8**	0.1s	**8**
urquhart	2	0.1s	2	0.2s	**6**	0.2s	**6**
channel	2	3.2s	2	9.7s	**10**	9.7s	**10**
color	3	2.9s	2	4.1s	**6**	4.1s	**6**
app14	**214**	6.3s	210	74.9s	209	14.7s	211
hard14	164	159.2s	178	181.3s	183	14.8s	**187**

Table 3.3: Number of solved instances for Glucose, Shatter, BreakID and BreakID limiting Saucy to 100 seconds. Also includes average preprocessing time in seconds.

3.7.3 Comparison to Shatter and performance on the 2014 SAT competition

This experiment compares BreakID to state-of-the-art solving configurations. We use **app14**, **hard14**, **pigeon**, **urquhart**, **channel** and **color** as benchmark sets. We effectively run all application and hard-combinatorial instances of 2014's SAT competition. The solving configurations used are (with Glucose as SAT engine):

- Glucose: Pure Glucose without symmetry breaking.
- Shatter: Shatter is run after first deduplicating the input CNF.
- BreakID: Compact encoding, row interchangeability and binary clauses activated.
- BreakID(100s): same as BreakID but Saucy is forced to stop detecting symmetry after 100 seconds of preprocessing have elapsed. While for most problem instances, Saucy derives symmetry in less than 10s, for a few large problem instances Saucy does reach the time limit. This restriction forces symmetry detection to end gracefully, lets BreakID continue, and hopefully leads to fewer timeouts.

We present the number of instances solved within resource limits, as well as the average time needed to detect symmetry and generate symmetry breaking clauses in Table 3.3. Figure 3.1 contains a cactus plot representing solving time needed on **app14** and **hard14**.

First, the two BreakID variants are the only configurations that handle **pigeon**, **urquhart** and **channel** efficiently, as Shatter constructs lex-leader constraints for the wrong set of symmetry generators, and Glucose gets lost

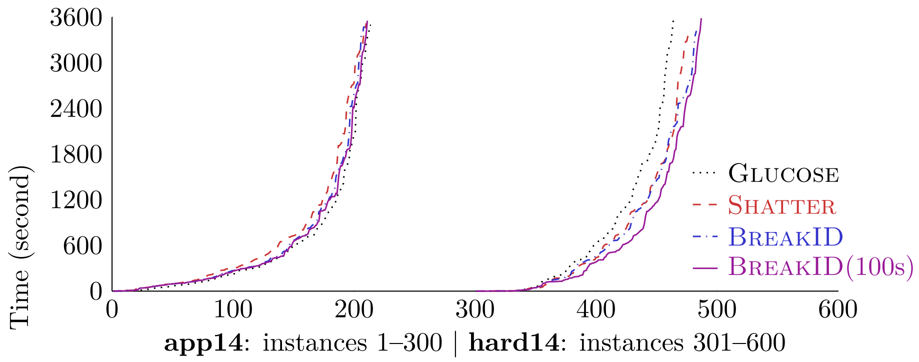

Figure 3.1: Cactus plot of runtimes of various solvers on **app14** and **hard14**.

in the symmetrical search space for non-trivial instances. A similar observation is made for **color** instances, though even BreakID remains unable to solve 4 instances.

On **hard14**, Shatter outperforms Glucose, while both BreakID approaches outperform Shatter. So for these instances, symmetry detection and breaking is worth the incurred overhead. The preprocessing time needed by Shatter is almost completely due to Saucy's symmetry detection, which exceeds 3600s for 9 instances. BreakID(100s) solves this problem by limiting the time consumed by Saucy to 100s, resulting in the best performance on **hard14**, adding 23 solved instances compared to plain Glucose. Of course, both BreakID approaches increase the preprocessing overhead by detecting row interchangeability and constructing binary clauses.

As far as **app14** is concerned, the benefit of a smaller search space does not outweigh the overhead of detecting symmetry and introducing symmetry breaking clauses.

3.8 BreakID for ASP

Answer set programming (ASP) has always benefited from progress in the satisfiability solving (SAT) community. In fact, many (if not all) modern ASP solvers [48, 6, 26] are based on conflict-driven clause learning (CDCL) [69], a technique first developed for SAT.

Thus, many techniques that improve SAT solving can be transferred to ASP, and symmetry breaking preprocessing is no exception. The ASP equivalent to the CNF preprocessor SHATTER, is SBASS [41]. SBASS' main contribution was to develop an alternative for SHATTER's first step, i.e., to transform a *(ground) ASP program* into a colored graph such that the graph automorphisms correspond to symmetries of the program. For SHATTER's last step, constructing symmetry breaking constraint, SBASS constructs the same type of symmetry breaking formula, but is able to limit the size of the constraint, similarly to BREAKID (see the end of Section 3.3).

In the following sections, we describe how BREAKID handles (ground) ASP programs. In essence, BREAKID extends and slightly modifies the automorphism graph encoding used in SBASS. The extensions serve to support a richer language: we provide support for so-called *weight rules* and *minimize statements*. On the other hand, the small modification serves to ensure more effective symmetry detection; we present several simple symmetric examples where BREAKID detects the symmetry while SBASS does not.

The symmetry breaking constraints BREAKID constructs for a CNF formula are also valid for ASP programs, so BREAKID posts the same symmetry breaking constraints as explained in Section 3.3, 3.4, 3.5 and 3.6. As a consequence, BREAKID posts stronger symmetry breaking constraints than SBASS due to its row interchangeability subgroup detection and binary clause derivation. This leads to a significant performance improvement on a number of benchmarks containing symmetry, compared to SBASS.

The ASP sections of this chapter are structured as follows. In Section 3.9 we recall preliminaries regarding answer set programming. Afterwards, in Section 3.10 we present BREAKID's symmetry detection approach for ASP. We experimentally evaluate the performance of BREAKID and SBASS this tool in Section 3.11.

3.9 ASP preliminaries

3.9.1 Answer Set Programming

A *vocabulary* χ is a set of symbols, also called *atoms*. A *literal* l is an atom or its negation. A *(ground) logic program* $\mathcal{P}$ over vocabulary χ is a set of *rules* r of form

$$h_1 \vee \cdots \vee h_l \leftarrow a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m, \tag{3.6}$$

where h_i's, a_i's, and b_i's are atoms in χ. The formula $h_1 \vee \cdots \vee h_l$ is called the *head* of r, denoted $\mathit{head}(r)$, and the formula $a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m$ the *body* of r, denoted $\mathit{body}(r)$. A program is called *normal* (resp. *positive*) if $l = 1$ (resp. $m = 0$) for all rules in $\mathcal{P}$. If $n = m = 0$, we simply write $h_1 \vee \cdots \vee h_l$. If $l = 0$, we call r a *constraint*.

The above definition defines *ground* logic programs, without quantified variables or predicate symbols. Typically, when a programmer writes an ASP program, she writes rules at the predicate level, since this is much more convenient. This "predicate level" is then *grounded* to a propositional form by instantiating all variables with appropriate constants, introducing *ground rules* of the previously defined form. As this chapter of the thesis is concerned with propositional (or ground) logic, we omit the fact that we are handling "ground ASP programs", and simply refer to them as "ASP programs".

The satisfaction relation between an ASP program and interpretations are defined as usual:

An *interpretation* I of the vocabulary χ is a subset of χ; alternatively, one can see an interpretation as an assignment of Boolean values to χ. An interpretation I is a *model* of a logic program $\mathcal{P}$ if, for all rules r in $\mathcal{P}$, whenever $\mathit{body}(r)$ is satisfied by I, so is $\mathit{head}(r)$. The *reduct* of $\mathcal{P}$ with respect to I, denoted $\mathcal{P}^I$, is a positive program that consists of rules $h_1 \vee \cdots \vee h_l \leftarrow a_1 \wedge \cdots \wedge a_n$ for all rules of the form (3.6) in $\mathcal{P}$ such that $b_i \notin I$ for all i. An interpretation I is a *stable model* or an *answer set* of $\mathcal{P}$ if it is a $\subseteq$-minimal model of $\mathcal{P}^I$ [50].

Deciding whether a logic program has a stable model is an Σ_2^P-complete task in general and an NP-complete task for normal programs; hence, logic programs can be used to encode combinatorial problems. This observation gave birth to the field of answer set programming [66, 80, 63].

Often, an ASP standard will support certain syntactical extensions. Though these do not take the form of rules, they can typically be translated to normal rules [19], and as such, form only syntactic sugar.

The following are such extensions, which we address later in this chapter:

A *cardinality atom*

$$g \leq \#\{l_1, \ldots, l_f\} \leq k$$

(with $l_1, \ldots, l_f$ being literals and $f, g, k \in \mathbb{N}$) is satisfied by I if

$$g \leq \#\{i \mid I \models l_i\} \leq k.$$

A *weight atom*

$$g \leq \mathrm{sum}\{l_1 = w_i, \ldots, l_f = w_f\} \leq k$$

(with $l_1, \ldots, l_f$ being literals and $f, g, k, w_i \in \mathbb{N}$) is satisfied if

$$g \leq \sum_{\{i \mid I \models l_i\}} w_i \leq k.$$

Sometimes, the "$g \leq$" or "$\leq k$" parts of cardinality or weight atoms are dropped. In this case, the obvious semantics applies: the atom is satisfied for any lower/upper bound. For instance, the cardinality atom

$$g \leq \#\{l_1, \ldots, l_f\}$$

is satisfied whenever

$$g \leq \#\{i \mid I \models l_i\}.$$

A *choice rule* is a rule with a cardinality atom in the head, i.e., a rule of the form

$$g \leq \#\{l_1, \ldots, l_f\} \leq k \leftarrow a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m.$$

A constraint is a rule c without head literals (so $\mathit{head}(c) = \emptyset$). I satisfies a constraint c if it does not satisfy $\mathit{body}(c)$, or equivalently, if there exists a literal $l \in \mathit{body}(c)$ such that $l \notin I$. Hence, constraints in ASP fulfill the same role as clauses in a CNF.

3.9.2 Symmetry in ASP

Let π be a permutation of χ. We extend π to literals: $\pi(\neg a) = \neg(\pi(a))$, to rules:

$$\pi(h_1 \vee \cdots \vee h_l \leftarrow a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m) =$$

$$\pi(h_1) \vee \cdots \vee \pi(h_l) \leftarrow \pi(a_1) \wedge \cdots \wedge \pi(a_n) \wedge \neg\pi(b_1) \wedge \cdots \wedge \neg\pi(b_m),$$

to logic programs: $\pi(\mathcal{P}) = \{\pi(r) \mid r \in \mathcal{P}\}$, and to interpretations: $\pi(I) = \{\pi(p) \mid p \in I\}$. A *symmetry* of a program $\mathcal{P}$ is a permutation π of χ that *preserves stable models* of $\mathcal{P}$; i.e., $\pi(I)$ is a stable model of $\mathcal{P}$ iff I is a stable model of $\mathcal{P}$.

A sufficient syntactical condition for π to be a symmetry is that π fixes $\mathcal{P}$ – $\pi(\mathcal{P}) = \mathcal{P}$. Typically, only this syntactical type of symmetry is exploited, since this type of symmetry can be detected with relative ease. The practical techniques presented in this chapter are no exception.

3.10 Colored graph encoding

To employ BreakID as a symmetry breaking preprocessor for ASP, all that is needed is to encode an ASP program as a colored graph for which the automorphism group corresponds to a symmetry group of the ASP program. The rest of a symmetry breaking preprocessor's workflow (running a graph automorphism detection tool and constructing symmetry breaking constraints) is valid in both a SAT and an ASP context.

As for encoding an ASP program as a colored graph, our approach is very close to the one introduced by sbass [41]. We discuss and justify the differences in Section 3.10.1.

We use the four colors $\{1,2,3,4\}$. Our graph encoding a program $\mathcal{P}$ consists of the following nodes:

- For each atom $p \in \chi$, two nodes, referred to as p and $\neg p$ below. Node p is colored as 1, node $\neg p$ is colored 2.
- For each rule $r \in \mathcal{P}$, two nodes, referred to as $head(r)$ and $body(r)$ below. Node $head(r)$ is colored 3 and node $body(r)$ is colored 4.

Our graph is *undirected* and the edges are as follows:

- Each node p is connected to $\neg p$.
- For each rule r of the form (3.6) in $\mathcal{P}$:
  - the node $head(r)$ is connected to $body(r)$,
  - each node h_i is connected to $head(r)$,
  - each node a_i is connected to $body(r)$,
  - each node $\neg b_i$ is connected to $body(r)$.

  The complete encoding of this type of rule is illustrated in Figure 3.2.

It can be seen that there is a one-to-one correspondence between automorphisms of this graph and syntactic symmetries of the logic program. Since atoms p are the only nodes colored in color 1, an automorphism induces a permutation of χ. The edge between p and $\neg p$ guarantee that an automorphism that maps to p to q also maps $\neg p$ to $\neg q$. Furthermore, the edges between $head(r)$, $body(r)$ and the various literals occurring in a rule capture the full structure of the rule. As such, it can be verified that automorphisms of this graph must map rules to

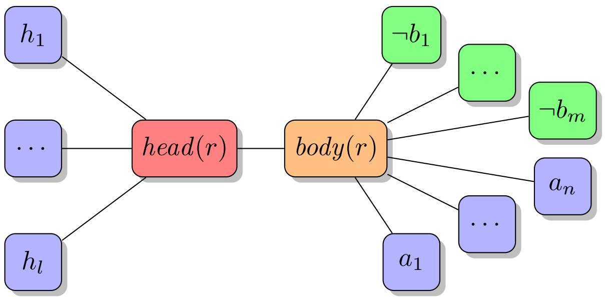

Figure 3.2: Encoding of a rule r of the form (3.6).

"syntactically symmetric" rules. Vice versa, each syntactic symmetry preserves the graph structure defined above.

Additionally, we extend this graph encoding to capture the aforementioned language extensions. As a result, BREAKID supports *exactly* the LPARSE-SMODELS intermediate format [92], extended with support for disjuctive rules. This means for instance that in cardinality rules, we assume there is no upper bound.

For this, we introduce two new colors, extending the set of colors to $\{1, \dots, 6\}$. In programs with these language extensions, integer numbers can occur, either as bounds of a cardinality or weight constraint or as weights in a weight constraint or minimize statement (see below). We assume that for each integer n that occurs in such a program, there is a unique color $c_n \notin \{1, \dots, 6\}$ available and extend our set of colors to

$$\{1, \dots, 6\} \cup \{c_n \mid n \text{ occurs as weight or bound in } \mathcal{P}\}.$$

Cardinality rules Rules of the form

$$h \leftarrow g \leq \#\{a_1, \dots, a_n, \neg b_1, \dots, \neg b_m\}$$

are encoded as follows. The head of the rule is encoded as usual. The body node of the rule is colored in c_g, the color associated with the bound g. The body node is connected to each of the a_i and $\neg b_i$ and to the head node, as usual.

Choice rules Rules of the form

$$\#\{h_1, \dots, h_l\} \leftarrow a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m$$

are encoded exactly the same as rules of form (3.6), except that $head(r)$ is colored 5. This allows to differentiate between standard rules and choice rules.

Weight rules Rules of the form

$$h \leftarrow g \leq \text{sum}\{a_1 = w_1, \dots, a_n = w_n, \neg b_1 = w_{n+1}, \dots, \neg b_m = w_{n+m}\} \tag{3.7}$$

are encoded as follows. The node *body*(r) is colored as for cardinality rules in c_g. For each occurrence of an expression $l_i = w_j$, we create one additional node, referred to as $l_i = w_j$. This node is colored in c_{w_j}, the color associated to the integer w_j and is connected to l_i. The body of this rule is connected to all the nodes $a_i = w_i$ and $\neg b_i = w_{n+i}$. *body*(r) is connected to *head*(r) and *head*(r) to h, as usual. A visualisation of this encoding can be found in Figure 3.3.

Minimize statements Minimize statements are expressions of the form

$$\text{minimize}\{a_1 = w_1, \dots, a_n = w_n, \neg b_1 = w_{n+1}, \dots, \neg b_m = w_{n+m}\} \tag{3.8}$$

They are directions to the solver that the user is only interested in models such that the term

$$\sum_{\{\text{rules of the form (3.8) that occur in } \mathcal{P}\ \}} \sum_{\{i | I \models l_i\}} w_i.$$

is minimal (among all stable models). Such a statement is encoded analogously to the body of a weight rule, except that the body node is replaced by a *minimize node* with color 6.

3.10.1 Comparison with SBASS

Our graph encoding differs from the one used by SBASS in two respects.

First of all, we added support for *minimize statements* and *weight rules.* As a result, BREAKID supports the full LPARSE-SMODELS intermediate language as documented in [92] and additionally, the rule type "8" used by GRINGO [49] to represent disjunctive rules.

Second, BREAKID uses *undirected edges* for the graph encoding whereas SBASS uses directed edges. When using directed edges, all *head*(r) nodes can be dropped by using edges from literals a_i and $\neg b_i$ to *body*(r) and from *body*(r) to h_i (the directionality thus distinguishes between head and body literals). We expect because of this that symmetry detection with BREAKID takes slightly more time than symmetry detection with SBASS, as automorphism algorithms are sensitive to the number of nodes in the input graph.

The motivation for using undirected edges in BREAKID is that we experimentally noticed that SAUCY does not always behave well with directed graphs. On

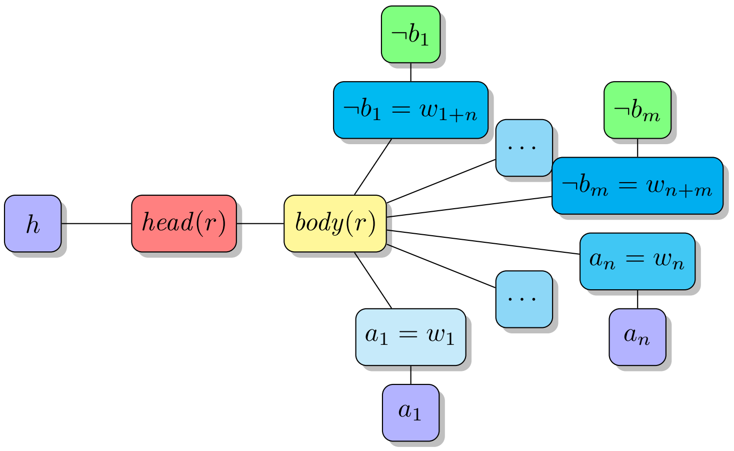

Figure 3.3: Encoding of a rule r of the form (3.7). The color of $body(r)$ depends on the bound g. The color of each $l_j = w_i$ depends on the value w_i.

very small examples already, symmetries are missed when using the directional encoding, so SBASS seems not always stable in the symmetries it detects. The following examples show this behavior, and illustrate that BREAKID is more stable than SBASS with respect to symmetry detection on different encodings.

Example 3.10.1.

$$\mathcal{P}_1 = \left\{ \begin{array}{l} 0 \leq \#\{p\} \leq 1. \\ 0 \leq \#\{q\} \leq 1. \end{array} \right\}.$$

It is clear that p and q are interchangeable in $\mathcal{P}_1$. By this we mean that the mapping $\sigma\colon \{p,q\} \rightarrow \{p,q\}\colon p \mapsto q, q \mapsto p$ is a symmetry of $\mathcal{P}_1$. BREAKID detects this, while SBASS detects (and breaks) no symmetry. ▲

Example 3.10.2. Consider the logic program

$$\mathcal{P}_2 = \left\{ \begin{array}{l} r \leftarrow p \wedge q. \\ 0 \leq \#\{p\} \leq 1. \\ 0 \leq \#\{q\} \leq 1. \end{array} \right\}.$$

It is clear that p and q are interchangeable in $\mathcal{P}_2$. In this case, both SBASS and BREAKID detect (and break) this interchangeability. ▲

Example 3.10.3. Consider the logic program

$$\mathcal{P}_3 = \left\{ \begin{array}{l} \leftarrow p \wedge q. \\ 0 \leq \#\{p\} \leq 1. \\ 0 \leq \#\{q\} \leq 1. \end{array} \right\}.$$

It is clear that p and q are interchangeable in $\mathcal{P}_3$. BREAKID detects this, while SBASS detects (and breaks) no symmetry. ▲

Example 3.10.4. Consider the logic program

$$\mathcal{P}_4 = \left\{ \begin{array}{c} p \vee q \leftarrow p \wedge q. \\ 0 \leq \#\{p\} \leq 1. \\ 0 \leq \#\{q\} \leq 1. \end{array} \right\}.$$

It is clear that p and q are interchangeable in $\mathcal{P}_4$. BREAKID detects this, while SBASS detects (and breaks) no symmetry. ▲

Example 3.10.5. Consider the logic program

$$\mathcal{P}_5 = \left\{ \begin{array}{c} p. \\ q. \end{array} \right\}.$$

It is clear that p and q are interchangeable in $\mathcal{P}_4$. BREAKID detects this, while SBASS detects no symmetry. ▲

In the last example, the fact that this symmetry is not detected does not have any practical consequences. Indeed, since these are only interchangeable facts, exploiting the symmetry will not help the solver. However, in examples such as Example 3.10.1, the difference is more important. If there are interchangeable atoms in choice rules, symmetry breaking can cut out exponentially large parts of the search space.

3.11 ASP experiments with BreakID

In this section, we experimentally compare SBASS and BREAKID, using CLASP 3.1.4 as solver and GRINGO 4.5.5 as grounder (to convert a predicate level ASP program into a propositional one). We compare the number of solved instances for a set of four symmetric decision problems, and a set of two symmetric optimization problems from 2013's ASP competition [5].

3.11.1 Setup

The four decision problems in our benchmark set are **pigeons**, **crew**, **graceful** and **200queens**.

pigeons is a set of 16 unsatisfiable pigeonhole instances where n pigeons must be placed in $n-1$ different holes. n takes values from $\{5, 6, \dots, 14, 15, 20,$

$30, 50, 70, 100\}$. The pigeons and holes are interchangeable, leading to a large symmetry group.

crew is a set of 42 unsatisfiable airline crew scheduling instances, where optimality has to be proven for a minimal crew assignment given a moderately complex flight plan. The instances are generated by hand, with the number of crew members ranging from 5 to 25. Crew members have different attributes, but depending in the instance, multiple crew members exist with the same exact attribute set, making these crew members interchangeable.

graceful consists of 60 satisfiable and unsatisfiable graceful graph instances, taken from 2013's ASP competition [5]. These instances require to label a graph's vertices and edges such that all vertices have a different label, all edges have a different label, and each edge's label is the difference of the labels of the vertices it connects. The labels used are $\{0, 1, \ldots, n\}$, with n the number of edges. Any symmetry exhibited by the input graph is present, as well as a symmetry mapping each vertex' label l to $n - l$.

200queens is a set of 4 large satisfiable N-Queens instances trying to fit n queens on an n by n chessboard so that no queen threatens another. n takes values from $\{50, 100, 150, 200\}$. The symmetries present in **200queens** are the rotational and reflective symmetries of the chessboard.

The two optimization problems are **valves** and **still**. Both problems' models and instances are taken from 2013's ASP competition [5], but manual symmetry breaking constraints were removed from the ASP specification.

valves models connected pipelines in urban hydraulic networks, and features interchangeable valves. Our instance set counts 50 instances.

still models a fixpoint connected cell configuration in Conway's game of life. The game board exhibits rotational and reflective symmetry. Our instance set counts 21 instances.

All experiments were run on on an Intel® Xeon® E3-1225 CPU with Ubuntu 14.04 Linux kernel 3.13 as operating system. Resources to reproduce these experiments are available online [31].

The decision problems had 6GB RAM and 1000s timeout as resource limits, and the results exclude grounding time, as this is the same for each solving configuration, but include any time needed to detect symmetry and construct symmetry breaking constraints. By default, BreakID limits the size of the symmetry breaking formula to 50 auxiliary variables for a given symmetry. To keep the comparison as fair as possible, sbass was given the same limit.

The optimization problems had 8GB RAM and 50000s timeout as resource limits, and the results exclude identical grounding time and negligable symmetry breaking time. No comparison with SBASS was made as SBASS does not support optimization statements.

Table 3.4 summarizes the results of the decision problems.

	CLASP	SBASS				BREAKID				
	#	#	t	V	π	#	t	V	π	δ
pigeons (16)	8	11	51.0	48814	43.5	14	12.7	95192	90	2
crew (42)	32	36	0.0	1722	7.8	41	0.0	2835	101	3.2
graceful (60)	33	28	0.7	127860	5.5	32	2.1	250614	103	1
200queens (4)	4	4	33.7	3658002	2.0	4	48.4	7277508	126	1

Table 3.4: Experimental results of (i) CLASP without symmetry breaking preprocessor, (ii) with SBASS, and (iii) with BREAKID. # represents the number of solved instances, t the average symmetry detection time in seconds, V the average number of vertices in the automorphism graph encoding, π the average number of symmetry generators detected by SAUCY, and δ the average number of detected row interchangeability symmetry groups.

Analyzing the results on **pigeons**, CLASP gets lost in symmetric parts of the search tree, solving only 8 instances (up to 12 pigeons). SBASS can only solve three more instances (up to 15 pigeons), as the derived symmetry generators do not suffice to construct strong symmetry breaking constraints. These results are consistent with the results of Drescher et al. [41]. BREAKID detects more structure, solving all but two instances (the largest being solved contains 50 pigeons). Note that BREAKID cannot solve **pigeons** instances in this benchmark set. The reason is that BREAKID, for these instances, is not able to detect the full row interchangeability groups present, but only smaller row interchangeability subgroups.

As far as symmetry preprocessing time goes, both SBASS and BREAKID spend a significant time detecting symmetry, especially on the larger instances with more than 20 pigeons. To our surprise, BREAKID requires only a quarter of the symmetry preprocessing time SBASS uses. This difference is entirely due to SAUCY needing much more time to detect automorphisms on the graph encoding of SBASS than on the graph encoding of BREAKID. This is surprising, as BREAKID's encoding graph, due to using undirected edges, has about twice the number of vertices of SBASS's encoding graph has. The most plausible hypothesis to explain the difference in symmetry detection time is simply that SAUCY is sensitive to differences in graph encodings, which gets magnified by large problem instances. Nonetheless, even for the largest instance with 100 pigeons, symmetry preprocessing by BREAKID did not reach the timeout limit.

The results on **crew** are similar to **pigeons**: BREAKID outperforms SBASS, which in turn outperforms plain Clasp. On the other hand, symmetry preprocessing time is negligible for **crew**. This is mainly due to the sizes of the ground programs remaining relatively small: less than 3000 rules for the largest ground program in the **crew** instance set.

Continuing with **graceful**, it is striking that the number of solved instances is *reduced* by symmetry breaking. Upon closer inspection, this is only the case for satisfiable instances. For unsatisfiable **graceful** instances, sbass and BreakID both solve four instances, two more than Clasp. This discrepancy is not uncommon, as static symmetry breaking formulas sometimes remove otherwise easy-to-find solutions, making a satisfiable problem harder to solve. These results are also consistent with those reported by Drescher et al. [41]. Focusing on symmetry preprocessing time, sbass is faster than BreakID. This is consistent with the results in Section 3.7, where we argued that deriving better symmetry breaking constraints incurs extra overhead.

Lastly, for **200queens**, all approaches solve all four instances easily. Again sbass is faster than BreakID, which is explained by the same reason as for **graceful**. However, BreakID's preprocessing time remains well within timeout bounds.

We conclude that on the decision problem benchmark set, BreakID outperforms sbass, especially for problems with interchangeable objects such as **pigeons** and **crew**. The price to be paid is a bit more symmetry preprocessing overhead, though the size of the symmetry breaking formula remains comparable between both approaches. This is to be expected, as BreakID compared similarly to Shatter in Section 3.7.

3.11.2 Optimization problem results

Figures 3.4 and 3.5 show the relative objective value of the best solution found after 50000s of search time for **still** and **valves**. Even though BreakID detects and breaks significant symmetry for both problems, and often infers interchangeability in **valves**, the resulting symmetry breaking constraints do not seem to vastly improve the final objective value for these two benchmark families. For **still**, the resulting objective value is virtually identical. For **valves**, the objective value for BreakID's run is improved for 11 instances, while it has worsened for 5 instances. When looking at the number of instances for which optimality was proven, both approaches were able to prove optimality for 5 **still** instances and 15 **valves** instances.

We conclude that for optimization problems **still** and **valves**, BreakID detects and breaks symmetry, but any resulting speedups are small at best.

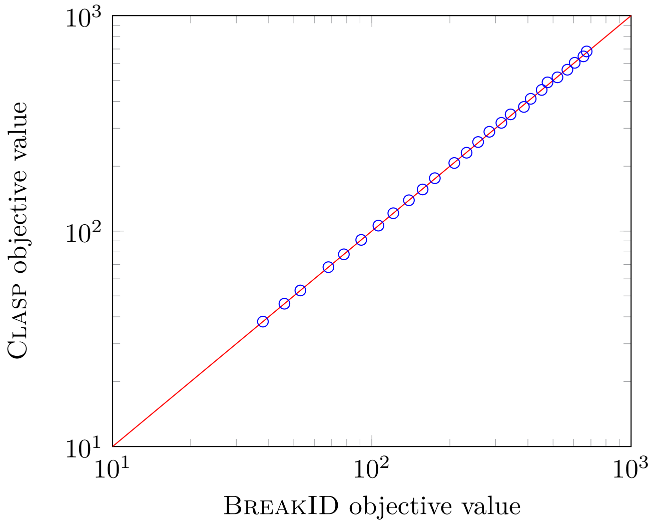

Figure 3.4: Objective value for BREAKID and CLASP after 50000s on **still**.

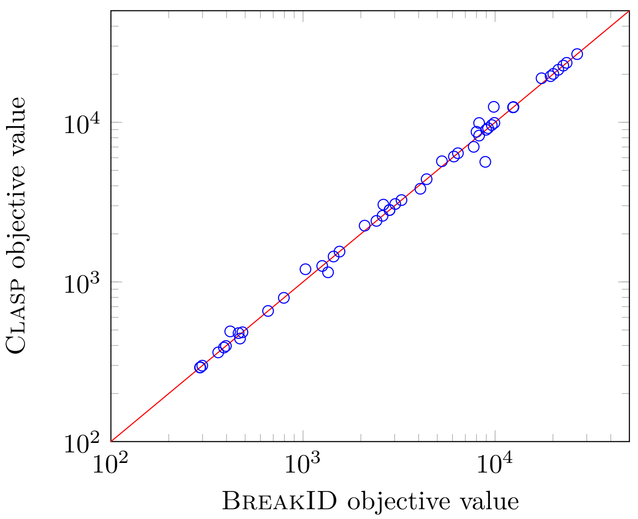

Figure 3.5: Objective value for BREAKID and CLASP after 50000s on **valves**.

3.12 Discussion

In this chapter, we presented novel improvements to state-of-the-art symmetry breaking for SAT. Common themes were to adapt the variable order and the set of generator symmetries by which to construct lex-leader constraints. BreakID implements these ideas and functions as a symmetry breaking preprocessor in the spirit of Shatter and sbass. Our experiments with BreakID show the potential for these techniques individually and combined. We observed that BreakID outperforms Shatter, and is a particularly effective preprocessor for hard-combinatorial SAT-problems. Its symmetry breaking advantages also carry over into a ground ASP context, improving upon sbass also by more reliably detecting symmetry.

The algorithms presented are effective, but also incomplete, e.g., not all row interchangeability is detected, no maximal set of binary symmetry breaking clauses is derived etc. Coupling BreakID to a computational group algebra system such as GAP [45] has the potential to alleviate these issues.

Alternatively, it might be interesting to compare different methods of graph automorphism detection, and investigate how hard it is to adjust their internal search algorithms to put out more useful symmetry generators, stabilizer chains for binary clauses, or even row interchangeability symmetry groups. Jefferson & Petrie already started this research in a constraint programming context [57].

3.12.1 Related work

We already mentioned the two symmetry breaking approaches most related to BreakID, namely Shatter [4] and sbass [41]. It is worth mentioning that other static symmetry breaking approaches than posting lex-leader constraints exist. For instance, one can use a *gray code* or *snake lex* ordering instead of a lexicographical ordering on the set of candidate solutions [79]. For different problems, the optimal symmetry breaking order differs, indicating that static symmetry breaking tools that only provide the option of one ordering (such as BreakID, Shatter and sbass do) have room to improve.

Chapter goal evaluation
We proposed two ways of inferring symmetry group structure, which, when derived, allows for clear performance benefits. Combined with technical improvements concerning symmetry detection and symmetry breaking, our resulting preprocessor BreakID improves the state-of-the-art. In this way, our research hypothesis is confirmed. However, the algorithms currently employed to infer symmetry group structure are incomplete, and might miss key properties of the symmetry group.

Chapter 4

Symmetrical Propagation and Learning

Goal of the chapter
Static symmetry breaking, though effective, fundamentally alters a problem by adding constraints that are not logical consequences of the initial set of constraints. Handling symmetry symmetry dynamically can avoid this by adding symmetrical images of logical consequences that are derived during the search process. In modern search engines, a valid formula that is a logical consequence of the theory is derived every time the solver has to backtrack. However, as potentially there exists an exponential amount of symmetrical images of one formula under a problem's symmetry group, a naive approach will be infeasible. In this chapter, we investigate whether feasible approaches exist.

This chapter is based on work presented at the IEEE 24th International Conference on Tools with Artificial Intelligence – November 2012, Athens, Greece [39].

4.1 Introduction

In Chapter 3, we presented a static symmetry breaking approach for propositional logic. This approach came in the form of a preprocessor for CNF formulas. Such a preprocessor is conceptually simple and easy-to-use – no modification of an often times complex search engine is required to deal with symmetry. Also, static symmetry breaking generally performs very well, as we showed in Chapters 3 and 5.

A disadvantage of static symmetry breaking is that it pushes the solver to a particular part of the search space which might not be the most efficient part for the problem and solver at hand [78]. Actually, the performance of BreakID and sbass on the **graceful** benchmark set in Section 3.11 is an example of how static symmetry breaking can influence performance negatively. Another disadvantage is that the extra constraints can be too large for a solver to handle, forcing a limit on their size and pruning power. Finally, static symmetry breaking is not always possible, for instance when symmetries are detected during search [15].

For a knowledge base system such as IDP, symmetry *breaking* has the additional disadvantage of fundamentally altering the knowledge base. It is plausible to imagine that a user is interested in the logical consequences a search engine has derived when performing, for instance, model expansion. With static symmetry breaking, these logical consequences are invalid, and hence potentially useless.

Dynamic symmetry handling deals with symmetry by interacting with the solver during search, and allows to address symmetry without *breaking* it.

In this chapter, we present *Symmetry Propagation* (SP), a dynamic symmetry handling approach that speeds up search by propagating literals symmetrical to already propagated literals. Experiments show that SP is most effective if it is extended to a form of *symmetric clause learning*, a simple and elegant dynamic symmetry handling principle. In the wake of SP, we also propose *Symmetric Explanation Learning* (SEL), a simple dynamic symmetry handling algorithm based only on the principle of symmetric clause learning. SEL experimentally outperforms SP, and performs on par with BreakID, making it, to our knowledge, the first succesful symmetric clause learning algorithm.

This chapter starts out with a brief overview of how modern SAT solvers work in Section 4.2. Section 4.3 delves into run-time properties of symmetry, providing the notion of *weak activity*. This section also contains a detailed description of how SP can be implemented in a SAT solver. An actual implementation in MiniSAT yielded the experimental results given in Section 4.4. Section 4.5 contains a description of SEL, and Section 4.6 contains an experimental evaluation of SEL, SP, BreakID and Shatter. We close this chapter with a discussion in Section 4.7.

4.2 Preliminaries

In what follows, we continue the SAT formalism established in Section 3.2.

Apart from $\mathbf{t}$ and $\mathbf{f}$ denoting standard Boolean values of "true" and "false", we also introduce the value $\mathbf{u}$ denoting the "unknown" value. A *partial assignment* α is a mapping of the variable set χ to true, false or unknown, so $\alpha\colon \chi \to \{\mathbf{t}, \mathbf{f}, \mathbf{u}\}$. A partial assignment α is *complete* if it maps every variable from χ to $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$. To remain consistent with Section 3.2, unless specifically mentioned to be partial, an assignment is presumed to be complete.

A formula ψ is a *logical consequence* of a formula φ – denoted $\varphi \models \psi$ – if for all assignments α satisfying φ, ψ holds in α. Two formulas are *logically equivalent* if each is a logical consequence of the other.

For the remainder of this chapter, we abstract a partial assignment α as a set of literals ($\alpha \subset \overline{\chi}$) such that α contains at most one literal over each variable in χ. Under this abstraction, $\alpha(x) = \mathbf{t}$ if $x \in \alpha$, $\alpha(x) = \mathbf{f}$ if $\neg x \in \alpha$, and $\alpha(x) = \mathbf{u}$ otherwise, for $x \in \chi$. If α contains exactly one literal over each variable in χ, then α is a complete assignment.

A clause c is *satisfied* under partial assignment α if α contains at least one literal from c. A clause c is a *conflict clause* (or *conflicting*) under partial assignment α if for all literals l in c, $\neg l \in \alpha$. A clause c is a *unit clause* under α if for all but one literal l in c, $\neg l \in \alpha$.

We extend the notion of a propositional symmetry π (see Section 3.2) to sets of literals: $\sigma(\alpha) = \{\sigma(l) \mid l \in \alpha\}$.

Finally, we often consider a formula φ in the context of some partial assignment α. We formalize this as the formula φ conjoined with a unit clause (l) for literal $l \in \alpha$. We denote this formula by $\varphi \downarrow \alpha$, pronounced φ "under" α. Formally, $\varphi \downarrow \alpha$ is the formula

$$\varphi \wedge \bigwedge_{l \in \alpha} l \; .$$

4.2.1 Conflict Driven Clause Learning SAT solvers

We briefly recall some of the characteristics of Conflict Driven Clause Learning SAT (CDCL) solvers [101].

A CDCL solver takes as input a formula φ over a set of Boolean variables χ. As output, it returns a complete assignment satisfying φ, or reports that none exists.

Internally, a CDCL solver keeps track of a partial assignment α – called the *current assignment* – that initially is empty. At each search step, the solver chooses a variable x for which the current assignment α does not yet contain a literal, and adds either the positive literal x or the negative literal $\neg x$ to α. The added literal is now a *choice literal*, and may result in some clauses becoming unit clauses under the *refined* current assignment. This prompts a *unit propagation* phase, where for all unknown literals l occurring in a unit clause, the current assignment is extended with l. Such literals are *propagated literals*; we refer to the unit clause that initiated l's unit propagation as l's *explanation clause*. If no more unit clauses remain under the resulting assignment, the unit propagation phase ends, and a new search step starts by deciding on a next choice literal.

During unit propagation, a clause c can become conflicting when another clause propagates the last non-false literal l of c to false. At this moment, a CDCL solver will construct a *learned clause* by investigating the explanation clauses for the unit propagations leading to the conflict clause. This learned clause c is a logical consequence of the input formula, and using c for unit propagation prevents the conflict from occurring again after a *backjump*.[1] We refer to the set of learned clauses of a CDCL solver as the *learned clause store* λ.

Formally, we characterize the *state* of a CDCL solver solving a formula φ by a quadruple $(\alpha, \gamma, \lambda, \mathcal{E})$, where

- α is the current assignment,
- γ is the set of choice literals – the set of literals $\alpha \setminus \gamma$ are known as *propagated* literals,
- λ is the learned clause store,
- $\mathcal{E}$ is a function mapping the propagated literals $l \in \alpha \setminus \gamma$ to their explanation clause $\mathcal{E}(l)$, which can be either a clause from the input formula φ or from the learned clause store λ.

During the search process, the invariant holds that the current assignment is a logical consequence of the decision literals, given the input formula. Formally:

$$\varphi \downarrow \gamma \models \varphi \downarrow \alpha.$$

Secondly, the learned clauses are logical consequences of the input formula:

$$\varphi \models c \text{ for each } c \in \lambda.$$

[1] *Backjumping* is a generalization of the more classical *backtracking* in combinatorial solvers.

4.3 Propagation via symmetry

4.3.1 Symmetrical learning

From the definition of symmetry for propositional formulas (see Section 3.2) follows:

Proposition 4.3.1. *Let φ be a propositional formula, π a symmetry of φ, and c a clause. If $\varphi \models c$, then also $\varphi \models \pi(c)$.*

Proof. If $\varphi \models c$ then $\pi(\varphi) \models \pi(c)$, as π renames the literals in formulas and clauses. Symmetries preserve satisfiability, so $\pi(\varphi)$ is equivalent to φ, hence $\varphi \models \pi(c)$. □

Since learned clauses are always logical consequences of the input formula, every time a CDCL solver learns a clause c, we may apply Proposition 4.3.1 and add $\pi(c)$ as a learned clause for every symmetry π of some symmetry group of $\mathbb{G}$. We will call this *symmetrical learning.*

Symmetrical learning can be used as a symmetry handling tool for SAT: because every learned clause prevents the solver from encountering a certain conflict, the orbit of this clause under the symmetry group will prevent the encounter of all symmetrical conflicts, resulting in a solver never visiting two symmetrical parts of the search space.

However, since the size of permutation groups can grow factorial in the number of permuted elements, this approach will in most cases add too many symmetrical clauses to the formula to be of practical use. Symmetrical learning approaches need to limit the amount of symmetrical learned clauses [56].

4.3.2 Active symmetry

Our own investigation to symmetrical learning starts with the observation that instead of learning symmetrical clauses, we can simulate the propagations they would entail. The following corollary of Proposition 4.3.1 lies at the base of this idea:

Corollary 4.3.2. *Let φ be a formula, α a partial assignment and l a literal. If π is a symmetry of $\varphi \downarrow \alpha$ and $\varphi \downarrow \alpha \models l$, then also $\varphi \downarrow \alpha \models \pi(l)$.*

Corollary 4.3.2 means that if a CDCL solver has state $(\alpha, \gamma, \lambda, \mathcal{E})$ where l can be propagated, then for every symmetry π of $\varphi \downarrow \alpha$, $\pi(l)$ can also be propagated.

To detect whether symmetries of φ are symmetries of $\varphi \downarrow \alpha$, Mears et al. introduced the notion of *activity* in their Lightweight Dynamic Symmetry Breaking algorithm [73].

Definition 4.3.3. A symmetry π is called *active* under partial assignment α if $\pi(\alpha) = \alpha$.

Which leads to the following proposition:

Proposition 4.3.4. *Let φ be a formula and α a partial assignment. If π is a symmetry of φ that is active under α, then π is also a symmetry of $\varphi \downarrow \alpha$.*

Proposition 4.3.4 states that the symmetries of φ active under partial assignment α form a subset of the symmetries of $\varphi \downarrow \alpha$. By Corollary 4.3.2, we can conclude that if a symmetry π of φ is active under partial assignment α, and a literal l is propagated, we are also allowed to propagate $\pi(l)$. Since the composition of two symmetries of $\varphi \downarrow \alpha$ is again a symmetry of $\varphi \downarrow \alpha$, we can also propagate $\pi^2(l), \pi^3(l), \ldots$ After doing so, π will again be active, so for other propagated literals l', $\pi(l')$ can again be propagated, and so on.

These propagations of literals symmetric to other propagated literals ensure a solver does not visit symmetrical branches of the search space; if l must hold (as it is propagated), then investigating $\neg l$ will lead to an inconsistent part of the search space, and by symmetry reasoning, investigating $\neg\pi(l)$ does so as well. To avoid this, it is only reasonable to propagate $\pi(l)$ – a *symmetry propagation*. In CP, several dynamic symmetry handling methods have the notion of activity at their core [53, 73].

4.3.3 Weakly active symmetry

We improve this approach by introducing the notion of *weakly active* symmetries, which generalizes activity.

Definition 4.3.5. Let φ be a formula and $(\alpha, \gamma, \lambda, \mathcal{E})$ the state of a CDCL solver. A symmetry π of φ is *weakly active* for partial assignment α and choice literals γ if $\pi(\gamma) \subseteq \alpha$.

We now show that a literal $\pi(l)$ is a logical consequence of a formula $\varphi \downarrow \alpha$ if l is a logical consequence of $\varphi \downarrow \alpha$ and π is a weakly active symmetry of φ under α.

Proposition 4.3.6. *Let φ be a formula, π a symmetry of φ, α a partial assignment, and γ a subset of α such that $\varphi \downarrow \gamma \models \varphi \downarrow \alpha$. If $\pi(\gamma) \subseteq \alpha$ then π is also a symmetry of $\varphi \downarrow \alpha$.*

Proof. To prove that π is a symmetry of $\varphi \downarrow \alpha$, we need to prove that it preserves satisfaction to $\varphi \downarrow \alpha$. We do this by showing that $\varphi \downarrow \alpha$ and $\pi(\varphi) \downarrow \pi(\alpha)$ are logically equivalent.

Because π is a symmetry of φ, $\pi(\varphi)$ and φ are logically equivalent. From the fact that $\varphi \downarrow \gamma \models \varphi \downarrow \alpha$ it then follows that $\varphi \downarrow \pi(\gamma) \models \varphi \downarrow \pi(\alpha)$, or, more generally, $\varphi \downarrow \pi^k(\gamma) \models \varphi \downarrow \pi^k(\alpha)$ for all $k \in \mathbb{N}$. Similarly, as π is a permutation of $\overline{\chi}$, and $\pi(\gamma) \subseteq \alpha$, it follows that $\pi^{k+1}(\gamma) \subseteq \pi^k(\alpha)$ for all $k \in \mathbb{N}$. Combined, this implies that $\varphi \downarrow \pi^k(\alpha) \models \varphi \downarrow \pi^{k+1}(\alpha)$ for all $k \in \mathbb{N}$.

Let n be the smallest positive natural number such that π^n is the identity.[2] It follows that

$$\varphi \downarrow \alpha \models \varphi \downarrow \pi(\alpha) \models \varphi \downarrow \pi^2(\alpha) \models \cdots \models \varphi \downarrow \pi^{n-1}(\alpha) \models \varphi \downarrow \alpha,$$

As a result, $\varphi \downarrow \alpha$, $\varphi \downarrow \pi(\alpha)$ and $\pi(\varphi) \downarrow \pi(\alpha)$ are logically equivalent. $\square$

Note that if $(\alpha, \gamma, \lambda, \mathcal{E})$ is a solver state where π is weakly active, the conditions in Proposition 4.3.6 are satisfied. Combined with Corollary 4.3.2, this implies that for every propagated literal l, $\pi(l)$ can be propagated if π is only weakly active.

Weak activity has two advantages when compared to activity. The first is that weak activity is more general than activity, which allows for more propagations. The second is that keeping track of weakly active symmetries is easier than keeping track of active symmetries, since we only need to check for every choice literal l in γ whether $\pi(l) \in \alpha$. We describe an efficient incremental approach to keep track of weakly active symmetries in Section 4.3.4.

To conclude this section, we derive from Proposition 4.3.6 a corollary describing the set of literals that forms a logical consequence from a formula given a set of weakly active symmetries. It also shows that propagating these literals turns weakly active symmetries into active ones.

Corollary 4.3.7. *Let φ be a formula, and $(\alpha, \gamma, \lambda, \mathcal{E})$ a state of a CDCL solver having φ as input. Suppose $\mathcal{P}$ is a set consisting of weakly active symmetries under α and γ. Furthermore, let $\mathbb{G}_{\mathcal{P}}$ be the group generated by $\mathcal{P}$, and β the partial assignment consisting of the union of the orbits under $\mathbb{G}_{\mathcal{P}}$ of all literals in α. Then (i) $\varphi \downarrow \alpha \models \varphi \downarrow \beta$ and (ii) all symmetries in $\mathbb{G}_{\mathcal{P}}$ are active under β.*

Proof. By Proposition 4.3.6, all $\pi \in \mathcal{P}$ are symmetries of $\varphi \downarrow \alpha$. Then (i) follows from the fact that all symmetries of $\mathbb{G}_{\mathcal{P}}$ are symmetries of $\varphi \downarrow \alpha$. Statement (ii) follows from the construction of β. $\square$

[2]This number exists for permutations with finite support.

Corollary 4.3.7 shows that potentially many symmetric literals can be propagated for weakly active symmetries if, during search, the group generated by the weakly active symmetries is big.

4.3.4 The symmetry propagation algorithm

In this section, we provide an algorithm that propagates literals symmetric to other literals in the current assignment of a CDCL solver. We refer to this algorithm as the *symmetry propagation* algorithm (SP).

As mentioned before, we characterize the state of a CDCL solver by its current partial assignment α, a set of choice literals γ, a set of learned clauses λ, and an explanation clause function $\mathcal{E}$. Symmetry is represented in the solver by a set of *input symmetries* $\mathcal{P}$, given to the solver at the beginning of the search.

At the start of the search, $\alpha = \emptyset$, so every input symmetry in $\mathcal{P}$ is weakly active. Every time a literal is added to α, all input symmetries containing that literal are notified using a watched literal scheme, allowing the symmetries to update their weak activity status accordingly. This updating is implemented by keeping a counter per symmetry π, indicating the minimum number of literals to be added to α to make π weakly active. Initially, the counter is 0, signifying the corresponding symmetry π is weakly active. The counter is increased whenever a literal l with $l \in \gamma$ and $\pi(l) \notin \alpha$ is added to α, and decreased whenever a literal l is added to α for which $\pi^{-1}(l) \in \gamma$. This way, updating the weak activity status of a symmetry is a constant time operation.

For every input symmetry π, SP also keeps track of the *first asymmetric literal* for π under partial assignment α. If such a literal exists, the first asymmetric literal is the oldest[3] literal $l \in \alpha$ for which $\pi(l) \notin \alpha$. Whenever π is weakly active, according to Proposition 4.3.6 and Corollary 4.3.2, $\pi(l)$ can be propagated. We will refer to this type of propagation as *symmetry propagation*, as opposed to unit propagation by unit clauses.

Modern CDCL solvers require that for every propagated literal l an explanation clause exists, which must contain l, and must be a unit clause when l is propagated. Fortunately, creating an explanation clause for a symmetry propagation is straightforward. A literal $\pi(l)$ will only be propagated as a symmetry propagation if l is the first asymmetric literal for some weakly active symmetry π under a solver state $(\alpha, \gamma, \lambda, \mathcal{E})$. As all false literals l' in l's explanation clause $\mathcal{E}(l)$ are older[3] than l, $\pi(\neg l') \in \alpha$. Taking into account that

[3]In CDCL solvers, literals are ordered based on their addition to the assignment timestamp. If a literal l is added before l', l is "older" than l'.

π commutes with negation, $\neg\pi(l') \in \alpha$, so $\pi(\mathcal{E}(l))$ is a unit clause under α, with $\pi(l)$ as non-false literal.

Hence, we can use $\pi(\mathcal{E}(l))$ as an explanation clause for the propagation of $\pi(l)$. By Proposition 4.3.1, we also know this clause is a logical consequence of the original formula, so we can safely add it to the learned clause store to use for future propagations.

During the propagation phase of the CDCL solver, SP alternates between unit propagation and symmetry propagation. More precisely: when no more unit propagations are possible, SP loops over the set of input symmetries in search of a weakly active symmetry π which still has a first asymmetric literal l under the current partial assignment. If such a symmetry exists, symmetry propagation of $\pi(l)$ occurs, after which unit propagation is immediately reactivated. This cycle continues until no more unit and symmetry propagations can be made, or a conflict occurs.

By switching back to unit propagation after every single symmetry propagation, SP makes sure all literals propagated by symmetry propagation could not have been propagated by unit propagation. As a consequence, all explanation clauses for symmetry propagations did not occur in the original formula or the learned clauses store, so no time is wasted constructing existing clauses. Also, the total amount of learned clauses is minimized, which is beneficial since the performance of CDCL solvers correlates with the number of clauses they have to keep track of.

Algorithm 4 is a pseudocode representation of the propagation phase of a typical CDCL solver using SP. Given a formula φ, a partial assignment α over variables of φ and a set of symmetries $\mathcal{P}$ of φ, a CDCL solver using SP propagates literals as follows:

Example 4.3.8 presents a typical run of the SP algorithm:

Example 4.3.8. Consider formula $\varphi = (\neg f \vee a) \wedge (\neg f \vee b) \wedge (\neg a \vee d) \wedge (\neg b \vee e \vee c) \wedge (\neg c \vee \neg g) \wedge (\neg c \vee g)$ and its symmetry $\pi = (a\ b)(d\ e)$. Suppose the CDCL algorithm chooses a, so $\alpha = \gamma = \{a\}$. During the unit propagation phase, we can propagate d, so $\alpha = \{a, d\}$, $\gamma = \{a\}$ and $\mathcal{E}(d) = \neg a \vee d$. Since no more unit propagation is possible, a check for symmetry propagation is made. However, since $\pi(\gamma) = \{b\} \not\subseteq \alpha$, π is not weakly active and no symmetry propagation occurs. Note that at this time, a is the first asymmetric literal for π, since a is the first literal added to α and $\pi(a) \notin \alpha$.

Since no more unit or symmetry propagation is possible, the algorithm chooses, say f, and propagates b during unit propagation. After this, $\alpha = \{a, d, f, b\}$, $\gamma = \{a, f\}$, $\mathcal{E}(b) = \neg f \vee b$, and symmetry propagation starts. π is inactive

```
data: a formula $\varphi$, a partial assignment $\alpha$, a set of symmetries $\mathcal{P}$ of $\varphi$
repeat
    execute unit propagation until fixpoint;
    foreach weakly active symmetry $\pi$ in $\mathcal{P}$ do
        if a first asymmetric literal $l$ for $\pi$ under $\alpha$ exists then
            extend $\alpha$ with $\pi(l)$;
            set $\pi(\mathcal{E}(l))$ as $\pi(l)$'s explanation clause;
            add $\pi(\mathcal{E}(l))$ to the learned clause store $\lambda$;
            break;
        end
    end
until no new literals have been propagated or a conflict has occurred;
```

Algorithm 4: propagation phase of a CDCL solver using SP

since $\pi(d) = e \notin \alpha$, but π is weakly active since $\pi(\gamma) = \{b, f\} \subseteq \alpha$. Also, the first asymmetric literal for π now is d, since $\pi(a) \in \alpha$ and $\pi(d) \notin \alpha$. As a result, $\pi(d) = e$ can be propagated during symmetry propagation, so that $\alpha = \{a, d, f, b, e\}$, and $\mathcal{E}(e) = \pi(\mathcal{E}(d)) = \neg b \vee e$.

Now, unit propagation is immediately reactivated. Since symmetry propagation did not induce new unit clauses, no further unit propagation happens, and symmetry propagation is continued. Even though π now is (weakly) active, it has no first asymmetric literal, so it can no longer propagate, ending the unit and symmetry propagation. The search now continues by choosing a new choice literal, and so on. ▲

SP offers some interesting properties. Firstly, the calculations needed to perform symmetry propagation cannot send the underlying solver in a loop, so SP preserves the completeness of the underlying solver.

Secondly, since all the literals propagated by symmetry propagation are a logical consequence of the original formula φ and the partial assignment α, and since all the corresponding explanation clauses added to φ are a logical consequence of φ, SP also preserves the soundness of the underlying solver. This also implies that SP does not prohibit the underlying solver from finding any model. SP merely avoids visiting branches of the search tree symmetrical to failed branches, and as such it leaves the underlying solver heuristic free to choose the most optimal search branch.

Thirdly, during unit and symmetry propagation, the partial assignment α only grows, while the set of decision literals does not change. Hence, the set of weakly active symmetries $\mathcal{P}$ can only increase in this phase. Also, for all weakly active symmetries $\pi, \pi' \in \mathcal{P}$, after $\pi(l)$ is propagated, $\pi'(\pi(l))$ will also be propagated – either by unit propagation or by symmetry propagation. As a consequence, all propagations allowed by Corollary 4.3.7 will take place.

4.3.5 Optimizations to SP

Having explained the main ideas of SP, this subsection discusses two optimizations to SP. The general idea is to maximize the number of symmetry propagations, steering the solver away from useless parts of the search space and deriving conflict clauses early on. In Section 4.4, we evaluate their performance.

The first optimization is based on the interaction of SP with *inverting* symmetries.

Definition 4.3.9. A literal l is *inverting* for a symmetry π if $\pi(l) = \neg l$. A symmetry π is *inverting* if at least one literal is inverting for π.

Whenever an inverting symmetry π is weakly active for partial assignment α and choices γ, and one of its inverting literals l is propagated, SP will propagate $\neg l$, resulting in a conflict and thus, a backjump in the search. However, if an inverting literal l for π would become a choice literal, π would become weakly inactive and remain so until the solver backtracks over l. By choosing choice literals in such a way that they are inverting for as few symmetries as possible, we can keep as many inverting symmetries weakly active as long as possible. In our implementation, we simply ordered the variables by the number of symmetries the corresponding literals were inverting for, and used this as the initial variable ordering by which literals were selected to become choice literals. Note that this *inverting symmetry optimization* has no effect if no inverting symmetries are present in the formula.

A second optimization concerns symmetry propagation when a symmetry π is not weakly active. In this case, it still is possible that for some propagated literal l, the clause $\pi(\mathcal{E}(l))$ is a unit clause. Since by Proposition 4.3.1 this clause is always a logical consequence of the original formula, the non-false literal $l' \in \pi(\mathcal{E}(l))$ can be propagated using $\pi(\mathcal{E}(l))$ as the explanation clause. We implemented this idea as a last check after unit propagation and (weakly active) symmetry propagation could propagate no more. More precisely, we loop over all inactive symmetries π and all propagated literals $l \in \alpha \setminus \gamma$ to check whether $\pi(\mathcal{E}(l))$ is a unit clause. If so, the appropriate propagation is made, after which unit propagation is immediately reactivated.

The inactive propagation optimization is a special case of symmetric learning, and since it does not depend on the activity status of a symmetry, it generalizes

SP. This does not make the notion of weak activity useless: if a symmetry π is weakly active, by Corollary 4.3.7, we can propagate $\pi(l)$ for each propagated literal l and skip the check of whether $\pi(\mathcal{E}(l))$ is a unit clause. Also, the inactive propagation optimization requires a solver to incorporate an explanation clause mechanism, which plain SP does not require.

4.4 Experimental evaluation of SP

To test the algorithms outlined in the previous section, we implemented SP in the CDCL solver MiniSAT [42] released on GitHub on March 27th 2011 [33]. The implementation follows the algorithm described in Section 4.3.4, with the option to use the inverting symmetry optimization or inactive propagation optimization described in Section 4.3.5. We tested the SP-implementation with different combinations of these options, of which we will present two here. The first version has both optimizations deactivated. We refer to this regular version by MiniSAT+SP^{reg}. The second version has both optimizations activated. We refer to this optimized version by MiniSAT+SP^{opt}. We refer to both versions by MiniSAT+SP. The source code of MiniSAT+SP is available on GitHub as a branch of Niklas Sörensson's MiniSAT solver [33].

We compare the performance of our solvers with Shatter [2] and plain MiniSAT as reference.

As benchmarks, we used SAT theories modeled in the standard DIMACS CNF format. Since this format does not contain information about symmetries, we detect symmetry by using Shatter's builtin symmetry detection algorithm based on the graph automorphism detection tool Saucy. We use the symmetry generators returned by Saucy as input symmetries for all symmetry breaking algorithms.

We employ two benchmark sets. The first benchmark set was constructed by running Shatter on the problems of the SAT 2011 competition benchmark set [86]. If, within a time limit of 1000 seconds, Shatter derived that a certain problem exhibited symmetry, we included this problem in the first benchmark set. In the end, the first benchmark set contains 96 problems, of which the results are summarized in Figure 4.1.

The second benchmark set consists of classical SAT symmetry breaking problems: problems of wire routing in the channels of field-programmable integrated circuits (**chnl** and **fpga**) [77], pigeonhole problems (**holes**) and Urquhart's problems (**Urq**) [96]. We use a shuffled version of the pigeonhole problem, because we experienced that using the initial variable ordering resulted in very fast solving

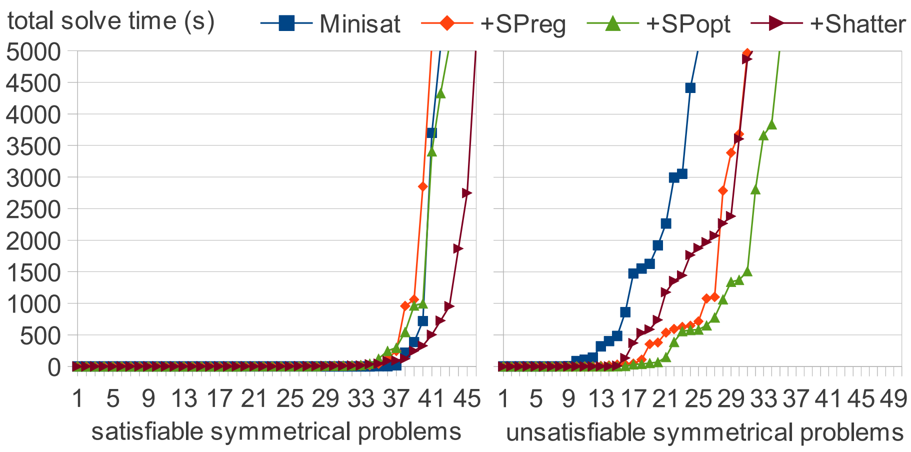

Figure 4.1: The performance of MINISAT, MINISAT+SP$^{\text{reg}}$, MINISAT+SP$^{\text{opt}}$ and MINISAT+SHATTER on 96 symmetry exhibiting problems of the SAT 2011 competition. For each algorithm, the problems are ordered on solve time.

times for both MINISAT+SP and MINISAT+SHATTER, which might not be representative for the expected performance of both algorithms. The results of the second benchmark set are described in Table 4.1.

Two problem families from the first benchmark set (**x** and **battleship**) gave particularly interesting results, so we also included their information in Table 4.1. Note that all problems in Table 4.1 contain no inverting symmetries, except for **Urq** and **x**, which contain only inverting symmetries.

The total solve time given to each algorithm on each problem was 5000 seconds, including the time needed to detect symmetry for all algorithms except MINISAT. The symmetry detection time was typically very low: less than 4 seconds for all problems in Table 4.1, and less than 100 seconds for all but 12 problems of the first benchmark set.

When no answer was given in the desired time limit, a “-” is shown in Table 4.1. The problems were solved using an Intel® Core® i7-2600 processor and 16 GiB of memory, with Ubuntu 10.04 64 bit as operating system. Resources to reproduce these experiments are available online [31, 33]. For all problems of the SAT 2011 competition, CNF files are available on the corresponding site [86].

The running times in Figure 4.1 on problems of the first benchmark set show some interesting patterns. Firstly, on unsatisfiable symmetrical problems, adding

Table 4.1: Performance comparison of MiniSAT, MiniSAT+SP$^{\text{reg}}$, MiniSAT+SP$^{\text{opt}}$ and MiniSAT+Shatter. A “*” indicates inverting symmetries. The last two columns show the amount of symmetry propagations as a percentage of the total amount of unit and symmetry propagations.

	Input	Solve Time (s)				Decisions				Sym. prop. ratio	
Problem name	syms	MiniSAT	+SP$^{\text{reg}}$	+SP$^{\text{opt}}$	+Shatter	MiniSAT	+SP$^{\text{reg}}$	+SP$^{\text{opt}}$	+Shatter	+SP$^{\text{reg}}$	+SP$^{\text{opt}}$
fpga10_8_sat	22	**0.0**	**0.0**	**0.0**	**0.0**	405	352	**320**	503	6.7%	**12.0%**
fpga10_9_sat	23	**0.0**	**0.0**	**0.0**	**0.0**	408	348	**316**	607	6.2%	**9.4%**
fpga12_11_sat	29	**0.0**	**0.0**	**0.0**	**0.0**	467	**363**	524	474	4.8%	**8.1%**
fpga12_8_sat	24	**0.0**	**0.0**	**0.0**	**0.0**	**296**	324	472	602	7.0%	**8.8%**
fpga12_9_sat	25	**0.0**	**0.0**	**0.0**	**0.0**	451	**378**	492	713	5.9%	**7.8%**
fpga13_10_sat	28	**0.0**	**0.0**	**0.0**	**0.0**	**278**	477	396	591	5.0%	**6.1%**
fpga13_12_sat	32	**0.0**	**0.0**	**0.0**	**0.0**	**316**	334	365	1112	2.5%	**6.1%**
fpga13_9_sat	26	**0.0**	**0.0**	**0.0**	**0.0**	**330**	858	696	601	2.4%	**4.9%**
fpga10_11_uns_rcr	38	71.5	**0.1**	**0.1**	0.2	5985130	17093	**6888**	15152	2.1%	**4.3%**
fpga10_12_uns_rcr	41	209.8	**0.1**	**0.1**	0.2	12881772	12154	**5530**	14446	2.5%	**3.5%**
fpga10_13_uns_rcr	42	955.2	0.2	**0.1**	0.2	42096627	25671	**4938**	16921	1.6%	**3.6%**
fpga10_15_uns_rcr	46	1841.0	**0.2**	**0.2**	**0.2**	60680481	19039	**9183**	13249	2.3%	**3.2%**
fpga10_20_uns_rcr	57	1271.8	0.4	**0.2**	0.4	29871337	21404	**4498**	16306	1.8%	**4.6%**
fpga11_13_uns_rcr	44	-	**0.3**	**0.3**	0.9	-	25790	**19236**	56623	2.8%	**4.2%**
fpga11_14_uns_rcr	46	-	0.6	**0.2**	0.8	-	51389	**10843**	48327	1.7%	**3.3%**
fpga11_15_uns_rcr	49	-	0.3	**0.2**	0.6	-	29983	**9836**	32194	1.9%	**3.4%**
fpga11_20_uns_rcr	59	-	0.8	**0.3**	1.3	-	45832	**9746**	54054	1.8%	**3.9%**
chnl10_11-uns	39	167.2	**0.0**	**0.0**	**0.0**	12437023	**46**	**46**	1473	**28.5%**	**28.5%**
chnl10_12-uns	41	85.0	**0.0**	**0.0**	**0.0**	6050506	**46**	**46**	1682	**25.9%**	**25.9%**
chnl10_13-uns	43	94.8	**0.0**	**0.0**	**0.0**	6006811	**46**	**46**	1766	**23.7%**	**23.7%**
chnl11_12-uns	43	3353.1	**0.0**	**0.0**	**0.0**	127407652	**56**	**56**	2017	**29.0%**	**29.0%**
chnl11_13-uns	45	3868.6	**0.0**	**0.0**	**0.0**	126818865	**56**	**56**	2264	**26.4%**	**26.4%**
chnl11_20-uns	59	3405.1	**0.1**	**0.1**	**0.1**	58302347	**56**	**56**	3954	**16.5%**	**16.5%**
hole010_shuffled-uns	19	114.3	0.3	**0.1**	**0.1**	12675295	33740	**10949**	17342	1.7%	**4.2%**
hole011_shuffled-uns	21	3320.3	0.5	**0.3**	1.4	193488347	55047	**18662**	142711	1.8%	**3.3%**
hole012_shuffled-uns	23	-	4.1	**0.3**	10.5	-	313497	**30992**	729083	2.0%	**3.3%**
hole013_shuffled-uns	25	-	31.0	**0.8**	105.2	-	1532124	**67184**	4531023	1.8%	**3.3%**
hole014_shuffled-uns	27	-	311.2	**13.3**	2821.2	-	9836194	**765810**	67375809	1.6%	**3.5%**
hole015_shuffled-uns	29	-	715.7	**201.5**	-	-	17048418	**7299563**	-	1.4%	**2.7%**
hole016_shuffled-uns	31	-	-	**122.7**	-	-	-	**3884224**	-	-	**2.6%**
hole017_shuffled-uns	33	-	-	**2863.2**	-	-	-	**54961134**	-	-	**2.1%**
hole018_shuffled-uns	35	-	-	**994.5**	-	-	-	**18031344**	-	-	**2.1%**
Urq3_5-uns	29*	139.7	**0.0**	**0.0**	0.1	73481538	6124	**33**	91985	7.3%	**35.1%**
Urq4_5-uns	43*	-	**0.0**	**0.0**	39.0	-	1200	**43**	20323094	4.4%	**40.4%**
Urq5_5-uns	72*	-	7.0	**0.2**	3810.3	-	2963855	**72**	1428323031	2.4%	**42.0%**
Urq6_5-uns	109*	-	-	**0.6**	-	-	-	**139**	-	-	**27.3%**
Urq7_5-uns	143*	-	-	**1.3**	-	-	-	**145**	-	-	**37.5%**
Urq8_5-uns	200*	-	-	**3.3**	-	-	-	**205**	-	-	**39.9%**
x1_40.shuffled-uns	40*	141.4	**0.0**	**0.0**	3.1	72930235	29191	**100**	1686967	0.9%	**7.6%**
x1_80.shuffled-uns	80*	-	29.8	**0.1**	1972.4	-	11256887	**84**	680826737	0.6%	**17.4%**
battleship-07-13-sat	12	**0.0**	0.4	0.4	0.4	414	**237**	517	606	0.4%	**1.0%**
battleship-08-15-sat	14	**0.0**	**0.0**	**0.0**	**0.0**	277	**268**	429	1078	1.3%	**2.1%**
battleship-09-17-sat	15	**0.0**	0.1	0.1	0.1	**1395**	1773	5148	4203	1.1%	**1.6%**
battleship-10-17-sat	12	3.9	**1.4**	2.2	5.4	368958	**142969**	143459	211317	1.1%	**1.2%**
battleship-10-18-sat	13	**0.0**	**0.0**	0.1	0.1	**622**	659	6360	6348	0.6%	**1.4%**
battleship-10-19-sat	15	**0.0**	0.1	0.1	0.1	648	759	**462**	3293	**0.3%**	**0.3%**
battleship-12-23-sat	18	**0.0**	0.1	0.1	0.1	1252	3049	**1034**	4429	**0.8%**	0.8%
battleship-14-26-sat	17	718.2	1060.2	546.1	**14.3**	29589334	37732810	17825596	**735680**	1.0%	**1.1%**
battleship-15-29-sat	22	386.0	**16.5**	296.6	88.1	21961391	**744001**	10058234	3373200	**1.1%**	1.1%
battleship-24-57-sat	41	16.5	**2.8**	21.9	34.3	1706712	**223013**	779295	3102966	1.0%	**1.2%**
battleship-05-08-uns	8	**0.0**	**0.0**	**0.0**	**0.0**	9281	1406	463	**447**	1.9%	**2.1%**
battleship-06-09-uns	7	0.1	**0.0**	**0.0**	**0.0**	45893	7947	2358	**1105**	1.0%	**1.6%**
battleship-07-12-uns	10	485.1	17.3	2.0	**1.4**	61922751	3040267	295311	**141734**	1.4%	**1.5%**
battleship-10-10-uns	8	1.6	1.1	0.2	**0.0**	199396	107931	13673	**5017**	0.0%	**0.1%**
battleship-12-12-uns	10	402.3	45.6	**0.7**	1.3	29000352	2828621	**45112**	119304	0.1%	**0.3%**
battleship-14-14-uns	8	-	-	1372.2	**736.6**	-	-	61447105	**18278571**	-	**0.0%**
battleship-15-15-uns	10	-	-	**149.0**	-	-	-	**7252565**	-	-	**0.0%**
battleship-16-16-uns	17	-	-	**32.9**	-	-	-	**1476236**	-	-	**0.2%**

symmetry information and exploiting it significantly improves performance compared to MiniSAT. We also learn that for unsatisfiable symmetrical problems, MiniSAT+Shatter and MiniSAT+SP^{reg} solve about the same number of problems, with MiniSAT+SP^{reg} being a bit faster on average. The best performing algorithm on the unsatisfiable instances clearly is MiniSAT+SP^{opt}.

The left part of Figure 4.1 sketches another picture. Firstly, all algorithms are able to solve almost all satisfiable symmetric problems. This can be explained by the fact that when a problem is both satisfiable and symmetric, it often contains many (symmetric) solutions, which as a result are less hard to find. Secondly, MiniSAT+SP^{reg} and MiniSAT+SP^{opt} both perform similarly to MiniSAT, while MiniSAT+Shatter on the other hand is able to deliver improved performance. An explanation can be found in the difference between SP and Shatter: the symmetry breaking constraints generated by Shatter will always exclude some part of the search space, while SP cannot guarantee that symmetry propagation (and corresponding search space reduction) will always happen. When symmetries are inactive or have no first asymmetric literal, they will not propagate. In this sense, Shatter is a more complete symmetry breaking tool, still breaking significant symmetry when solving relatively easy satisfiable symmetrical problems.

The results presented in Table 4.1 concerning the classical SAT symmetry breaking problems confirm the above observations: the satisfiable instances of **fpga** are easily solved, MiniSAT+SP^{reg} has performance similar to MiniSAT+Shatter, and MiniSAT+SP^{opt} performs best. Note that MiniSAT is able to solve the satisfiable instances, but has great trouble with the unsatisfiable ones.

Since MiniSAT+SP^{opt} uses two different optimizations at the same time, the question remains whether both optimizations have complementary strengths. Since the inverting symmetry optimization has no effect when no inverting symmetries are present in the problem, the performance difference between MiniSAT+SP^{reg} and MiniSAT+SP^{opt} on the unsatisfiable **fpga**, **holes** and **battleship** problems (which contain no inverting symmetries) is due to the inactive propagation optimization. Further testing with only the inactive propagation optimization activated, showed no significant performance gain on **Urq** and **x** (which contain only inverting symmetries) compared to MiniSAT+SP^{reg}. As a result, we can conclude that the performance gain of MiniSAT+SP^{opt} on **Urq** and **x** is due to the inverting symmetry optimization.

4.5 Symmetric explanation learning

The performance of SP, and in particular its optimized variant SP^{opt}, is encouraging.

Note that SP^{opt} can be seen as a symmetrical learning approach, where, for some symmetry π and some propagated literal l, $\pi(\mathcal{E}(l))$ is added to the learned clause store if $\pi(\mathcal{E}(l))$ is a unit or conflict clause under the current assignment. If π is active, SP^{opt} checks whether $\pi(\mathcal{E}(l))$ is unit or conflict for all literals l in the current assignment. If π is not active, $\pi(\mathcal{E}(l))$ is checked for only recently propagated literals l.

This asynchronicity seems arbitrary. Symmetry might be handled even more effectively if we have the guarantee that $\pi(\mathcal{E}(l))$ is added to the learned clause store whenever it is a unit or conflict clause, regardless of whether symmetry π is weakly active or not, and regardless of whether literal l is propagated recently or early on.

For this, we propose the *Symmetric Explanation Learning* algorithm (SEL), a symmetric learning variant with theoretical guarantees.

4.5.1 Details on SEL

SEL's core idea is simple: given a solver state $(\alpha, \gamma, \lambda, \mathcal{E})$, whenever $\pi(\mathcal{E}(l))$ is a unit or conflict clause for some propagated literal l and some input symmetry π, add it to the learned clause store, and perform the necessary propagation or conflict resolution.

We give pseudocode for SEL's behavior during a CDCL solver's propagation phase in Algorithm 5. The idea is to keep a set of clauses $\mathcal{SC}$, called the *symmetrical clause store*, containing clauses symmetrical to explanations of currently propagated literals. The symmetrical clause store $\mathcal{SC}$ expands when more propagations l are made, by adding any not yet satisfied symmetrical clause $\pi(\mathcal{E}(l))$ to $\mathcal{SC}$. $\mathcal{SC}$ shrinks when the solver backjumps over literal l, removing l from the current assignment and removing any $\pi(\mathcal{E}(l))$ from $\mathcal{SC}$.

If, after regular unit propagation reached a fixpoint, a clause $\pi(\mathcal{E}(l))$ from $\mathcal{SC}$ becomes a unit or conflict clause, it is upgraded to a full learned clause, allowing the CDCL solver to avoid a part of the search space symmetrical to the one closed by $\mathcal{E}(l)$. By requiring that unit propagation is at fixpoint, we obtain the guarantee that $\pi(\mathcal{E}(l))$ did not yet belong to the learned clause store. After a symmetrical clause is upgraded, either conflict resolution is initiated or unit clause propagation is reprised.

```
data: a formula $\varphi$, a set of symmetries $\mathcal{P}$ of $\varphi$, a partial assignment $\alpha$, a set of
      learned clauses $\lambda$, an explanation function $\mathcal{E}$, a set of symmetrical
      explanation clauses $\mathcal{SC}$
repeat
    foreach unit clause $c$ in $\varphi$ or $\lambda$ do
        let $l$ be the unassigned literal in $c$;
        add $l$ to $\alpha$;
        foreach symmetry $\pi$ in $\mathcal{P}$ do
            if $\pi(\mathcal{E}(l))$ is not yet satisfied by $\alpha$ then
                add $\pi(\mathcal{E}(l))$ to $\mathcal{SC}$;
            end
        end
    end
    foreach unit clause $c$ in $\mathcal{SC}$ do
        add $c$ to $\lambda$;
        break;
    end
until no new literals have been propagated or a conflict has occurred;
```
Algorithm 5: propagation phase of a CDCL solver using SEL

Example 4.5.1 presents a unit propagation phase with the SEL technique. It also gives a situation where SEL derives propagations (and thus search space reductions) not detected by $\mathrm{SP}^{\mathrm{reg}}$ and $\mathrm{SP}^{\mathrm{opt}}$.

Example 4.5.1. Let a CDCL solver have as input some unsatisfiable formula φ, a learned clause store $\lambda = \{(a \vee b), (\neg c \vee d \vee e)\}$. As φ is unsatisfiable, $\pi = (a\ c)(b\ d)$ induces a symmetry. We will assume no clauses in φ will matter for the following exposition, e.g., because they all contain literals over variables not in $\{a, b, c, d, e\}$.

Suppose the CDCL algorithm chooses $\neg a$, so the current assignment α equals the literal set $\{\neg a\}$, which is also the current set of decisions γ. During unit propagation, the CDCL algorithm propagates b, so $\alpha = \{\neg a, b\}$, $\gamma = \{\neg a\}$ and $\mathcal{E}(b) = a \vee b$. By Algorithm 5, SEL adds $\pi(\mathcal{E}(b)) = (c \vee d)$ to $\mathcal{SC}$, so $\mathcal{SC} = \{c \vee d\}$. No further unit propagation is possible, and $c \vee d$ is not unit or conflicting, so the solver enters a new decision phase.

We let the solver choose $\neg d$, so $\alpha = \{\neg a, b, \neg d\}$, $\gamma = \{\neg a, \neg d\}$. Still, no unit propagation on clauses from φ or from the empty learned clause store λ is possible. However, $c \vee d$ is unit, so SEL adds $c \vee d$ to λ. Now unit propagation is reinitiated, leading to the propagation of c with reason $c \vee d$, and e with reason $\neg c \vee d \vee e$.

Note that π is weakly inactive, and that the propagation triggered by the symmetric explanation clause $c \vee d$ did not follow some other propagation, but rather a decision by the solver. Hence, both $\mathrm{SP}^{\mathrm{reg}}$ and $\mathrm{SP}^{\mathrm{opt}}$ would not have derived the valid propagation of c with reason $c \vee d$. ▲

4.5.2 Properties of SEL

A first useful property of SEL is that in a given solver state, its propagations are a superset of the ones derived by SP, both in the regular incarnation $\mathrm{SP}^{\mathrm{reg}}$ as the optimized version $\mathrm{SP}^{\mathrm{opt}}$. The argument is simple: every propagation or conflict by these algorithms is backed by a reason clause $\pi(\mathcal{E}(l))$, and SEL guarantees to track *all* these clauses, ensuring the same propagations are made. Combined with Example 4.5.1 where SEL derives a propagation that both $\mathrm{SP}^{\mathrm{reg}}$ and $\mathrm{SP}^{\mathrm{opt}}$ would miss, we argue that SEL is a strictly stronger symmetry handling method that SP. From this, it also follows that SEL will perform all propagations promised by Corollary 4.3.7.

Secondly, SEL does not rely on a weak activity notion for symmetries, as opposed to $\mathrm{SP}^{\mathrm{opt}}$. As such, we expect its implementation to be less complex.

Thirdly, SEL's time and memory overhead seem acceptable for small sets of input symmetries $\mathcal{P}$. Assuming a two-watched literal implementation for checking the symmetrical clause store $\mathcal{SC}$ on conflict or unit clauses, the computationally most intensive step for SEL is filling $\mathcal{SC}$ with symmetrical explanation clauses during unit propagation. Worst case, for each propagated literal l, SEL loops over $\mathcal{P}$ to construct $\pi(\mathcal{E}(l))$, with $\pi \in \mathcal{P}$. Assuming k to be the size of the largest clause in φ or λ, this incurs a polynomial $O(|\mathcal{P}|k)$ time overhead each propagation. As for memory overhead, SEL must maintain a symmetrical clause store containing $O(|\mathcal{P}||\alpha|)$ clauses, with α the solver's current assignment.

Of course, as with any symmetrical learning approach, SEL might flood the learned clause store with many symmetrical clauses. In effect, as only symmetrical explanation clauses are added to the learned clause store if they propagate (or are conflicting), an upper bound on the number symmetrical clauses added is the number of propagations performed by the solver, which can be huge. Aggressive learned clause store cleaning strategies might be required to maintain efficiency.

Lastly, a symmetrical learning approach such as SEL (or SP) is compatible with current state-of-the-art formula preprocessing techniques, as long as they adhere to the property that the preprocessed formula $pre(\varphi)$ is a logical consequence of φ – $\varphi \models pre(\varphi)$.

In this case, any model of φ is also a model of $pre(\varphi)$, and any symmetrical learned clause derived by SEL will satisfy those models of $pre(\varphi)$ that satisfy φ, since SEL, being a symmetrical learning approach, only derives logical consequences of φ. Then, SEL remains sound in combination with such preprocessing techniques, as, if φ was satisfiable, a SEL-implementing SAT solver will find $pre(\varphi)$ to still be satisfiable. Of course, a symmetry *breaking* preprocessing technique does not have the property $\varphi \models pre(\varphi)$, so it cannot be easily combined with SEL (or SP).

4.5.3 Implementing SEL

An implementation of SEL should take care to include efficient data structures and avoid superfluous method calls. E.g., checking whether a clause in $\mathcal{SC}$ is unit under an extension of the current assignment is efficiently done by a *two-watched literal scheme* [76]. Also, when constructing symmetrical explanation clauses $\pi(\mathcal{E}(l))$ to add to $\mathcal{SC}$, it is useless to calculate those for which l is not in π's support. If so, $\pi(l) = l$, and any symmetrical clause $\pi(\mathcal{E}(l))$ will be satisfied under the current assignment α (as $l \in \alpha$ after l's propagation).

Also, we would ideally like SEL to "completely" handle any row interchangeability, e.g., those detected by BreakID. Our current experiments with SEL and SP$^{\text{opt}}$ indicate that, for a given row interchangeability group R_M with rows $Ro = \{1, \ldots, k\}$, each row swap $\pi_{(i\ j)}$ with $i, j \in Ro, i < j$ is a useful input symmetry. Leaving out any of these $k(k-1)/2$ generators of R_M significantly lowers performance on problems with much row interchangeability. Providing a theoretical argument for this experimental observation remains future work.

4.6 Experimental evaluation of SEL

To verify the effectiveness of SEL, we implemented this dynamic symmetry handling technique in Glucose 4.0 [10]. The source code of our implementation is available online [32].

We use a large benchmark set consisting of three components:

- **collected**: a set of highly symmetric instances, the majority being unsatisfiable. They were collected incrementally for the purpose of this thesis, and amongst others, comprise pigeonhole, urquhart, graph coloring, channel routing and number theory instances. (209 instances)

- **hard14**: the hard-combinatorial track of 2014's SAT competition. (300 instances)
- **app14**: the application track of 2014's SAT competition. (300 instances)

Five solver configurations are compared:

- Glucose: pure Glucose 4.0 without symmetry breaking.
- Shatter: Shatter is used as a symmetry breaking preprocessor, after first deduplicating the input CNF, using Glucose 4.0 as SAT solver.
- BreakID: BreakID's optimal configuration as symmetry breaking preprocessor, exploiting the compact encoding, row interchangeability, and binary clauses. Saucy is forced to stop detecting symmetry after 100 seconds of preprocessing have elapsed, and Glucose 4.0 is used as SAT solver.
- SP^{opt}: SP's optimal configuration, employing both the inverting symmetry optimization and the weakly inactive propagation optimization. As SP is implemented in MiniSAT, this is the SAT solver used for this configuration. The input symmetry generators are those returned by BreakID's symmetry detection, with any detected row interchangeability group represented by all its row swaps (see Section 4.5.3).
- SEL: the implementation of SEL in Glucose 4.0, using the input symmetry generators returned by BreakID's symmetry detection, with any detected row interchangeability group represented by all its row swaps (see Section 4.5.3).

The resources available to each experiment were 16GB of memory and 3500s on an Intel® Xeon® E3-1225 cpu. The operating system was Ubuntu 14.04 with Linux kernel 3.13. Unless noted otherwise, all results *include* any preprocessing step, such as deduplicating the input CNF, symmetry detection by Saucy and symmetry breaking clause generation by Shatter or BreakID. Resources to reproduce these experiments are available online [31].

The summarized results are presented in Table 4.2.

As **collected** contains instances with a lot of (row interchangeability) symmetry, Glucose performs worst, Shatter only slightly better, while the three symmetry breaking approaches using BreakID as symmetry detection tool perform similarly, as they all are provided with row interchangeability information. It is worth noting that BreakID, SP^{opt} and SEL all are able

	Glucose	Shatter	BreakID	SP^{opt}	SEL
collected (209)	109	115	154	155	**156**
hard14 (300)	163	177	**183**	165	**183**
app14 (300)	**213**	210	212	179	211
total (809)	485	502	549	499	**550**

Table 4.2: Number of solved instances for Glucose, Shatter, BreakID, SP^{opt} and SEL for three different benchmark sets. A time limit of 3500s and a memory limit of 16GB was imposed.

to solve all pigeonhole instances in **collected**, justifying the choice of input symmetries representing row interchangeability groups (see Section 4.5.3).

On **hard14**, BreakID and SEL give the best performance, with SP^{opt} and Glucose performing worst, and Shatter in between. The most striking feature is SP^{opt}'s bad performance, where its symmetry exploitation routines hardly justify the incurred overhead of keeping track of weakly active symmetries.

This behavior is even more apparent in **app14**, where all configurations perform similarly, except for SP^{opt} whose performance is down the drains. One explanation for SP^{opt}'s weak performance on **app14** and **hard14** is MiniSAT as base solver. Another is that SP^{opt}'s activity-tracking overhead is simply too high.

As for SEL, its symmetrical clause tracking overhead is low enough to keep up with Glucose on **app14**, and to perform competitively to BreakID over all benchmark sets. As such, we consider it a succesful implementation of symmetrical learning.

One remark is that both SP^{opt} and SEL require a lot more memory resources than the other solver configuration, reaching a memory-out for a total of 24 and 22 instances, respectively. This might be due to a large number of symmetry generators residing in memory for certain instances. This is an area of improvement for future implementations.

Finally, we present the performance of the solver configurations on the graph coloring problem instances present in **collected**. The data to generate these graph coloring CNF instances were taken from Michael Trick's Operations Research Page [95]. Table 4.3 lists the results.

The picture painted here is strongly in favor of SEL, with static symmetry breaking by BreakID and Shatter performing poorly.

	Glucose	Shatter	BreakID	SP$^{\text{opt}}$	SEL
graph coloring (57)	10	21	26	33	**37**

Table 4.3: Number of solved instances for Glucose, Shatter, BreakID, SP$^{\text{opt}}$ and SEL for the graph coloring problem instances present in **collected**. A time limit of 3500s and a memory limit of 16GB was imposed.

4.7 Discussion

In this chapter, we presented the notion of weak activity, a dynamic property of symmetries that allows the derivation of propagations symmetric to already performed propagations. Weak activity is a generalization of activity, a notion employed by dynamic symmetry handling techniques in the field of constraint programming [73].

Based on the notion of weak activity, we presented Symmetry Propagation (SP), a novel approach to dynamically handle symmetries in SAT problems. We implemented both a regular and optimized version of SP in MiniSAT. Our experiments indicate that SP outperforms Shatter on unsatisfiable symmetrical benchmarks, while the satisfiable symmetrical benchmarks remained relatively easy to solve. However, SP in its optimized version performs badly on application instances.

Based on SP ideas, we also presented Symmetric Explanation Learning (SEL), a second dynamic symmetry handling technique for SAT. SEL is a relatively simple form of symmetrical learning, that nonetheless performs strictly more symmetry propagations than SP. The performance of a first implementation is on par with BreakID, making it the first symmetrical learning scheme to be a viable alternative to static symmetry breaking.

4.7.1 Related work

Benhamou et al. proposed a general dynamic symmetry handling approach for SAT [15], based on Corollary 4.3.2: every time a SAT solver with formula φ and partial assignment α backjumps from a certain choice literal l, a local symmetry group $\mathbb{G}$ of $\varphi \downarrow \alpha$ is computed, and the orbit of $\neg l$ under $\mathbb{G}$ is propagated. The drawback of this technique is that repeated computation of symmetries of $\varphi \downarrow \alpha$ can be very expensive.

SP avoids this overhead by only considering the symmetry group of $\varphi \downarrow \alpha$ generated by weakly active input symmetries, which, as shown in Section 4.3.4, can be implemented without recomputing a symmetry group. Similarly, SEL does not require recomputation of the symmetry group. Also, SP and SEL check for symmetry propagations during every propagation phase, not only when the solver backtracks.

Other dynamic symmetry handling approaches for SAT are based on Proposition 4.3.1. The simplest one is by Heule et al. [56], where for a learned clause, all its symmetric images under the input symmetry group are constructed and retained as extra learned clauses. They only present experiments on graph coloring problems, and cannot improve on Shatter. We hypothesize that this approach simply adds too many symmetrical learned clauses.

A more restrictive form of symmetrical clause learning is the Symmetrical Learning Scheme [16], which only constructs symmetric images of learned clauses for a set of input symmetries $\mathcal{P}$ instead of for the whole symmetry group $\mathbb{G}$. A disadvantage of the Symmetrical Learning Scheme is that not all symmetrical learned clauses are guaranteed to contribute to the search by propagating a literal. This might result in lots of useless clauses being added to the solver, without breaking a significant amount of symmetry. It also is possible that some of the clauses added already belong to the set of learned clauses. Thirdly, it does not allow to derive compositional images of a learned clause c, e.g., $\pi(\pi(c))$ for $\pi \in \mathcal{P}$. Both SP and SEL improve on these three points.

A completely different dynamic symmetry handling approach in SAT is SymChaff, a structure-aware SAT solver [84]. SymChaff exploits matrices of row-interchangeable variables (see Section 3.4). It handles symmetry by keeping track of the subsets of rows that are still interchangeable under the current assignment, and by adjusting the search heuristic to branch over subsets of variables that must have the same truth value simultaneously. As such, it can handle row interchangeability very efficiently, but it can only be used with this type of symmetry.

A last dynamic symmetry handling approach for SAT is one based on an alternative learning scheme for graph coloring problems [87]. The idea is that after every conflict, a special-purpose clause is learned based on *Zykov contractions*, an alternative way of solving graph coloring problems. Though effective, it is not clear how to extend this approach to non-graph coloring problems and other forms of symmetry.

4.7.2 Future work

In Chapter 3 we provided completeness results for symmetry breaking constraints for row interchangeability symmetry groups. These complete symmetry breaking constraints, in a sense, ensure that a SAT solver is no longer hindered by the row interchangeability symmetry group. It would be useful to formalize the analogous notion of “complete dynamic symmetry handling”, and to provide (in)completeness guarantees for symmetrical learning approaches. A starting point could be the last remark of Section 4.5.3.

Also, as the notion of weak activity is a generalization of a concept from constraint programming, transferring our results back to constraint programming could improve existing dynamic symmetry handling algorithms.

Finally, it might be worthwhile to combine static symmetry breaking and symmetric learning. E.g., one could statically break graph coloring symmetry by finding some large clique of size k, coloring it with k different colors, and then break the remaining interchangeability symmetry between unused colors dynamically.

Chapter goal evaluation
Using the notion of weak activity, we devised Symmetry Propagation (SP), a dynamic symmetry handling approach based on performing propagations symmetrical to those already executed. Experiments show that this approach is most effective if also symmetry propagations for weakly *in*active symmetries are performed. This led to SEL, a second dynamic symmetry handling algorithm and an instantiation of symmetrical learning. Experiments suggest that SEL performs competitively with the state-of-the-art BreakID, making it the first symmetrical learning scheme to be a viable alternative to static symmetry breaking.

Chapter 5

Symmetry in First-Order Logic

Goal of the chapter
Though symmetry breaking preprocessors on a propositional level are convenient, a ground, propositional representation of a combinatorial problem is often magnitudes larger than a more high-level, predicate-based representation. Also, on a higher level, more information on the structure of a symmetry group is still present, which might be inferred with more precision. We investigate what symmetry on a first-order level is, how to detect it, and how to exploit special features of certain symmetry groups.

This chapter is based on work presented at the 32nd International Conference on Logic Programming – October 2016, New York City, USA [38].

5.1 Introduction

The goal of this chapter is to introduce a formal framework for exploiting symmetry arising in first-order logic (FO) theories and model expansion problems. We firstly provide the necessary formalism (syntax and semantics) of FO, and then discuss different, increasingly more refined, notions of symmetry in FO. Next, we show how to detect and exploit these types of symmetry, and provide an experiment based on an implementation in the knowledge base system IDP. Finally, we relate our work to existing literature on symmetry in combinatorial solvers.

5.2 Preliminaries

We briefly provide a syntax and semantics for first-order logic. A *vocabulary* Σ is a set of *predicate symbols* P/n of arity $n \geq 0$ and *function symbols* f/n of arity $n \geq 0$. A 0-arity function symbol is a *constant symbol.* Often, we will simply refer to a *symbol* $S/n \in \Sigma$, which represents an n-ary predicate or function symbol. We assume a set of *variable symbols* x, y, z, etc. Slightly deviating from the standard presentation of first-order logic, we consider both variable symbols and constant symbols to be 0-arity function symbols, as this simplifies our representation.

5.2.1 Syntax of FO

Given a vocabulary Σ, a *term* is inductively defined as

- a 0-arity function symbol, either from Σ or a variable symbol
- a function symbol application $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and $f/n \in \Sigma$, with $n > 0$.

Given a vocabulary Σ, an *atom* is defined as

- an equality $t_1 = t_2$ where t_1, t_2 are terms
- a predicate symbol application $P(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms and $P/n \in \Sigma$

Given a vocabulary Σ, a *formula* is inductively defined as

- an atom a
- a negation $\neg\varphi$ where φ is a formula
- a conjunction $\varphi \land \varphi'$ where φ, φ' are formulas
- a disjunction $\varphi \lor \varphi'$ where φ, φ' are formulas

- a universal quantification $\forall x\colon \varphi$ where φ is a formula and x a variable symbol
- an existential quantification $\exists x\colon \varphi$ where φ is a formula and x a variable symbol

The formulas $\varphi \Rightarrow \varphi'$ and $\varphi \Leftrightarrow \varphi'$ are shorthand for $\neg\varphi \lor \varphi'$ and $(\varphi \Rightarrow \varphi') \land (\varphi' \Rightarrow \varphi)$. The connectives have a priority of application defined as $=< \neg < \land < \lor < \Rightarrow < \Leftrightarrow$. Brackets $(\ldots)$ are used to locally change the order of connectives or limit the scope of quantifiers.

A quantifier $\exists x\colon \varphi$ or $\forall x\colon \varphi$ *scopes* a variable x within φ. We say x is *bound* by the quantifier. A *sentence* over vocabulary Σ is a formula φ where all variable symbols not bound by a quantifier belong to Σ. Without loss of generality, we assume variables are *renamed apart*, i.e., each variable symbol occurs in the scope of at most one quantifier. A *theory* over a vocabulary Σ is a set of sentences over Σ.

5.2.2 Semantics of FO

A *structure* over a vocabulary Σ represents a "possible world" or "state of affairs" in which sentences over Σ have a "meaning", or more formally, a semantics. A structure I consists of a *domain* D of objects of interest and an *interpretation* S^I for each symbol S in Σ. An interpretation P^I for a predicate symbol $P/n \in \Sigma$ is a an n-ary relation over D (so $P^I \subseteq D^n$). An interpretation f^I for a function symbol $f/n \in \Sigma$ is an n-ary function over D (so $f^I\colon D^n \to D$). Given Σ-structure I, we denote with $I[x:d]$ the $\Sigma \cup \{x\}$-structure with the same domain and interpretations as I, except for the variable symbol x which is interpreted by the 0-arity function mapping to domain element d.

Given a Σ-structure I, a term t containing only symbols from Σ evaluates to a domain element t^I as follows:

- if t is a 0-arity function symbol f, then $t^I = f^I()$
- if t is a function symbol application $f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ then $t^I = f^I(t_1^I, \ldots, t_n^I)$

Given a Σ-structure I, a Σ-sentence φ evaluates to the Boolean value φ^I as follows:

- $(t_1 = t_2)^I$ is true iff $t_1^I = t_2^I$

- $(P(t_1,\dots,t_n))^I$ is true iff $(t_1^I,\dots,t_n^I) \in P^I$
- $(\neg\varphi)^I$ is true iff φ^I is false
- $(\varphi \wedge \varphi')^I$ is true iff φ^I and φ'^I both are true
- $(\varphi \vee \varphi')^I$ is false iff φ^I and φ'^I both are false
- $(\forall x\colon \varphi)^I$ is true iff $\varphi^{I[x:d]}$ is true for all d in the domain of I
- $(\exists x\colon \varphi)^I$ is false iff $\varphi^{I[x:d]}$ is false for all d in the domain of I

As such, every Σ-sentence φ evaluates to true or false under a given Σ-structure. A Σ-theory $\mathcal{T}$ is *s*atisfied by a Σ-structure I if for each sentence $\varphi \in \mathcal{T}$, φ^I is true. If I satisfies $\mathcal{T}$, I is a *model* for $\mathcal{T}$, denoted by $I \models \mathcal{T}$.

5.2.3 Model Expansion (MX)

Many combinatorial problems can be conveniently modeled as a *model expansion* problem $MX(\mathcal{T}, I_{in})$, where $\mathcal{T}$ is a Σ-theory and I_{in} is a Σ_{in}-structure with $\Sigma_{in} \subseteq \Sigma$. We refer to Σ_{in} as the *input vocabulary*, and $\Sigma_{out} = \Sigma \setminus \Sigma_{in}$ as the *output vocabulary*.

If Σ-structure I and Σ'-structure I' have the same domain D and $\Sigma \cap \Sigma' = \emptyset$, $I \sqcup I'$ is the *merged* $\Sigma \cup \Sigma'$-structure over D that interprets all symbols in Σ according to I and all symbols in Σ' according to I'. A *solution* to a model expansion problem $MX(\mathcal{T}, I_{in})$ with output vocabulary Σ_{out} is a Σ_{out}-structure I_{out} (sharing I_{in}'s domain) such that $I_{in} \sqcup I_{out} \models \mathcal{T}$; the merged structure $I_{in} \sqcup I_{out}$ is a model to $\mathcal{T}$ that *expands* I_{in}.

5.2.4 Graph Coloring running example

Throughout the rest of this chapter, we assume a fixed domain D and use Γ_D to refer to the set of all structures with domain D.

As a running example, we use a simple graph coloring problem.

Example 5.2.1. Let Σ_{gc} be the vocabulary consisting of predicate symbols $V/1$, $C/1$, $Edge/2$ and a function symbol $Color/1$. A valid colored graph is expressed by the theory $\mathcal{T}_{gc}$:

$$\begin{aligned}
&\forall x_1\ y_1\colon Edge(x_1,y_1) \Rightarrow (Color(x_1) \neq Color(y_1))\\
&\forall x_2\ y_2\colon Edge(x_2,y_2) \Rightarrow V(x_2) \wedge V(y_2)\\
&\forall x_3\colon C(Color(x_3))
\end{aligned}$$

Let $\Sigma_{gcin} = \Sigma_{gc} \setminus \{Color/1\}$. Input data containing vertices, colors and a graph is expressed as a Σ_{gcin}-structure I_{gcin} with domain $D = \{t, u, v, w, r, g, b\}$, and interpretations

$$V^{I_{gcin}} = \{t, u, v, w\} \quad Edge^{I_{gcin}} = \{(t, u), (u, v), (v, w), (w, t)\} \quad C^{I_{gcin}} = \{r, g, b\}.$$

The model expansion problem $MX(\mathcal{T}_{gc}, I_{gcin})$ now consists of finding a $\Sigma_{gcout} = \{Color/1\}$-structure I_{gcout} such that $I_{gcin} \sqcup I_{gcout} \models \mathcal{T}_{gc}$. We let I_{gcout} contain the interpretation

$$Color^{I_{gcout}} = t \mapsto r, u \mapsto g, v \mapsto b, w \mapsto g, r \mapsto r, g \mapsto g, b \mapsto b$$

which represents a valid coloring of the input graph. Indeed, $I_{gc} = I_{gcin} \sqcup I_{gcout} \models \mathcal{T}_{gc}$. ▲

5.3 Symmetries

In this section, we define how symmetry manifests itself in FO. We initially ignore any input structure, focusing only on the symmetry of a theory in Section 5.3.1. Using these notions, we identify symmetry for the model expansion formalism in Section 5.3.2, zooming in on a particular class of (model expansion) symmetry in Section 5.3.3. Finally, we argue that not all notions of symmetry are captured by our definitions in Section 5.3.4.

5.3.1 Symmetry of a theory

If the problem at hand is to find models of a theory over some domain, then the candidate solutions for this problem are the set of possible structures with that domain over the vocabulary of the theory. In this context, a symmetry is a transformation of structures that preserves satisfiability:

Definition 5.3.1 (Symmetry for a theory)**.** A mapping $\sigma\colon \Gamma_D \to \Gamma_D$ is a *structure transformation.* A structure transformation σ is a *symmetry* for Σ-theory $\mathcal{T}$ if for all Σ-structures $I \in \Gamma_D$, $I \models \mathcal{T}$ iff $\sigma(I) \models \mathcal{T}$.

From this definition, it follows that the set of symmetries of a theory form an algebraic group under composition ($\circ$). In this thesis, we study symmetry in order to improve the algorithms that find models of theories, or more general, that check the satisfiability of theories. For this, we will also have to detect the symmetry group present in a theory. However, detecting all symmetries of a theory is computationally at least as hard as deciding whether the theory is satisfiable (if not, all structure transformations are symmetries).

Instead, researchers typically focus on *syntactical* symmetries; those that can be detected by means of a syntactical analysis of the problem statement. This work is no exception: we restrict our notion of symmetry to one induced by a permutation of domain elements. As we show in Section 5.5, there exist efficient techniques to detect symmetry arising from domain permutations.

Definition 5.3.2 (Domain permutation)**.** A bijection $\pi \colon D \to D$ is a *domain permutation.* A domain permutation *induces a structure transformation* σ_π: for each predicate symbol P/n, $(\pi(d_1), \ldots, \pi(d_n)) \in P^{\sigma_\pi(I)}$ iff $(d_1, \ldots, d_n) \in P^I$, and for each function symbol f/n, $f^{\sigma_\pi(I)}(\pi(d_1), \ldots, \pi(d_n)) = \pi(d)$ iff $f^I(d_1, \ldots, d_n) = d$.

Proposition 5.3.3. *Any structure transformation induced by a domain permutation is a symmetry for any theory.*

We call this type of symmetry induced by only a domain permutation *global domain symmetry.* Again, the global domain symmetries of a theory form an algebraic group under composition.

Example 5.3.4 (Example 5.2.1 continued)**.** The domain permutation $(v\ r)$ induces a global domain symmetry $\sigma_{(v\ r)}$ of $\mathcal{T}_{gc}$. $\sigma_{(v\ r)}(I_{gc})$ gives

$$\begin{aligned}
&D = \{t, u, v, w, r, g, b\} \qquad V^{\sigma_{(v\ r)}(I_{gc})} = \{t, u, r, w\} \qquad C^{\sigma_{(v\ r)}(I_{gc})} = \{v, g, b\}\\
&Edge^{\sigma_{(v\ r)}(I_{gc})} = \{(t, u), (u, r), (r, w), (w, t)\}\\
&Color^{\sigma_{(v\ r)}(I_{gc})} = t \mapsto v, u \mapsto g, r \mapsto b, w \mapsto g, v \mapsto v, g \mapsto g, b \mapsto b
\end{aligned}$$

which is still a model of $\mathcal{T}_{gc}$, though r now acts as a vertex and v as a color. ▲

Finite model generators such as KODKOD [94], SEM [100], MACE [70] or PARADOX [23] focus on the task of generating a model with a given domain for a given theory. Since every domain permutation induces a global domain symmetry, these systems have mechanisms to cope with global domain symmetry.

However, a global domain symmetry σ_π is a rather restrictive concept as it applies π on every argument of every tuple in every interpretation of a structure. A larger class of transformations can be described when π is only applied locally. For example, one could apply π only on the interpretation of some symbols, or even more fine-grained, only on some of the arguments in the tuples of an interpretation. Given a predicate or function symbol S/n, we use $S|i$ with $1 \leq i \leq n$ to denote the i^{th} *argument position* of S; if S is a function symbol, we use $S|0$ for the output argument of S. Note that variables, being treated as function symbols, also form argument positions.

Definition 5.3.5 (Structure transformation induced by A, π)**.** Let π be a domain permutation and A a set of argument positions. The structure transformation σ_π^A *induced by* A, π is defined by

$$(\tau_{P|1}(d_1), \ldots, \tau_{P|n}(d_n)) \in P^{\sigma_\pi^A(I)} \text{ iff } (d_1, \ldots, d_n) \in P^I$$

$$f^{\sigma_\pi^A(I)}(\tau_{f|1}(d_1), \ldots, \tau_{f|n}(d_n)) = \tau_{f|0}(d_0) \text{ iff } f^I(d_1, \ldots, d_n) = d_0$$

where $\tau_{S|i}(d) = \pi(d)$ if $S|i \in A$ and $\tau_{S|i}(d) = d$ otherwise.

Thus, for each domain tuple in the interpretation of a symbol S, the structure transformation induced by A, π only applies π to domain elements that occur at an index i corresponding to an argument position $S|i \in A$. Note that if A contains argument positions over symbols S not interpreted by I (e.g., variable symbols), those argument positions are simply ignored by σ_π^A.

Definition 5.3.6 (Local domain symmetry)**.** Let $\mathcal{T}$ be a theory. A *local domain symmetry* for $\mathcal{T}$ is a structure transformation induced by a set of argument positions A and a domain permutation π, that also is a symmetry for $\mathcal{T}$.

A global domain symmetry σ_π for a Σ-theory is a local domain symmetry σ_π^A where A includes all argument positions of all symbols in Σ. As such, local domain symmetry is a generalization of global domain symmetry, and allows us to detect and exploit more symmetry. The set of local domain symmetries for $\mathcal{T}$ with the same set of argument positions A forms an algebraic group under composition.

However, not all A, π-induced structure transformations are symmetries. Below, we propose a syntactic criterion to identify a set of argument positions A that guarantees that σ_π^A is a symmetry for a given theory. Intuitively, the criterion can be formulated as follows: whenever a term $f(\ldots)$ occurs as the i'th argument in a predicate or function symbol S, then $f|0 \in A$ if and only if $S|i \in A$.

Definition 5.3.7. Let $\mathcal{T}$ be a theory. Assume $f|0$ and $S|i$ are argument positions with S either a predicate or a function symbol. We call $f|0$ and $S|i$ *directly connected by* $\mathcal{T}$ if one of the following holds:

- an expression $S(t_1, \ldots, t_{i-1}, f(\bar{t}'), t_{i+1}, \ldots, t_n)$ occurs in $\mathcal{T}$, or
- $i = 0$ and an expression $S(\bar{t}) = f(\bar{t}')$ occurs in $\mathcal{T}$.

Note that if two argument positions are connected, at least one of them is the output position of a function symbol (hence the $f|0$). A set A of argument positions is *connectively closed under* $\mathcal{T}$ if for each $S|i \in A$, each argument position directly connected to $S|i$ by $\mathcal{T}$, is also in A.

Example 5.3.8 (Example 5.2.1 continued)**.** According to the first formula in $\mathcal{T}_{gc}$, $x_1|0$ is directly connected to $Edge|1$ and $Color|1$, while $y_1|0$ is directly connected to $Edge|2$ and $Color|1$. Analyzing all formulas, we find the following two sets are connectively closed under $\mathcal{T}_{gc}$: $A = \{C|1, Color|0\}$ and $B = \{V|1$, $Edge|1$, $Edge|2$, $Color|1$, $x_1|0$, $y_1|0$, $x_2|0$, $y_2|0$, $x_3|0\}$.

Applying the induced structure transformation $\sigma^A_{(v\ r)}$ on I_{gc} gives

$$\begin{array}{l} D = \{t, u, v, w, r, g, b\} \quad V^{\sigma^A_{(v\ r)}(I_{gc})} = \{t, u, v, w\} \quad C^{\sigma^A_{(v\ r)}(I_{gc})} = \{v, g, b\} \\ Edge^{\sigma^A_{(v\ r)}(I_{gc})} = \{(t, u), (u, v), (v, w), (w, t)\} \\ Color^{\sigma^A_{(v\ r)}(I_{gc})} = t \mapsto v, u \mapsto g, v \mapsto b, w \mapsto g, r \mapsto v, g \mapsto g, b \mapsto b \end{array}$$

which is also a model of $\mathcal{T}_{gc}$ (here, domain element v serves both as a vertex and a color). ▲

We can now formally give a syntactic condition on when a set of argument positions and a domain permutation induces a local domain symmetry:

Theorem 5.3.9 (Local domain symmetry condition)**.** *Let Σ be a vocabulary, $\mathcal{T}$ a theory over Σ, π a domain permutation and A a set of argument positions. If A is connectively closed under $\mathcal{T}$, then σ^A_π is a local domain symmetry of $\mathcal{T}$.*

We refer to Appendix A.1 for a proof.

This theorem is useful when detecting symmetry for model expansion problems with an empty input structure, but it will also prove useful for non-empty input structures.

Example 5.3.10 (Example 5.3.8 continued)**.** The argument position set A induces local domain symmetries that correspond to permuting the colors of a graph coloring problem, while B induces symmetry on the vertices and $A \cup B$ induces global domain symmetries. ▲

5.3.2 Symmetry for model expansion

Recall that a model expansion problem $MX(\mathcal{T}, I_{in})$ consists of finding structures I_{out} such that $I_{in} \sqcup I_{out} \models \mathcal{T}$. Intuitively, a model expansion problem does not consider all models to a theory to be a solution, but only those that "contain" the input structure I_{in}. We adjust the general notion of symmetry for a theory (see Definition 5.3.1) to take this into account:

Definition 5.3.11 (Symmetry for MX)**.** Let $MX(\mathcal{T}, I_{in})$ be a model expansion problem with output vocabulary Σ_{out}, and let $\Gamma^{\Sigma_{out}}_D$ be the set of Σ_{out}-structures with domain D. A structure transformation $\sigma\colon \Gamma_D^{\Sigma_{out}} \to \Gamma_D^{\Sigma_{out}}$ is a symmetry of $MX(\mathcal{T}, I_{in})$ if for each $I_{out} \in \Gamma_D^{\Sigma_{out}}$, $I_{in} \sqcup I_{out} \models \mathcal{T}$ iff $I_{in} \sqcup \sigma(I_{out}) \models \mathcal{T}$.

Analogous to Definition 5.3.5, a domain permutation π and argument position set A induce a structure transformation σ_π^A on $\Gamma_D^{\Sigma_{out}}$. We call σ_π^A a *local domain symmetry of* $MX(\mathcal{T}, I_{in})$ if σ_π^A is a symmetry of $MX(\mathcal{T}, I_{in})$.

Example 5.3.12 (Example 5.2.1 continued)**.** Let A be the argument position set $\{V|1, Edge|1, Edge|2, Color|1, x_1|0, y_1|0, x_2|0, y_2|0, x_3|0\}$. Observe that A is connectively closed under $\mathcal{T}_{gc}$ and that the induced structure transformation $\sigma^A_{(t\ u\ v\ w)}$ is a local domain symmetry of $MX(\mathcal{T}_{gc}, I_{gcin})$. However, connectively closedness under the theory is neither a sufficient nor a necessary condition for an A, π-induced structure transformation to be a symmetry of a model expansion problem.

For instance, argument position set $B = \{Edge|1, Edge|2, Color|1, x_1|0, y_1|0, x_3|0\}$ is not connectively closed under $\mathcal{T}_{gc}$, though $\sigma^B_{(t\ u\ v\ w)}$ is still a symmetry of $MX(\mathcal{T}_{gc}, I_{gcin})$.

Moreover, since A is connectively closed, $\sigma^A_{(v\ r)}$ is a local domain symmetry of $\mathcal{T}_{gc}$, but it is not a symmetry of $MX(\mathcal{T}_{gc}, I_{gcin})$. Indeed,

$$Color^{\sigma^A_{(v\ r)}(I_{gcout})} = t \mapsto v, u \mapsto g, v \mapsto b, w \mapsto g, r \mapsto v, g \mapsto g, b \mapsto b$$

maps t and r to node v, which is not consistent with $\forall x_3\colon C(Color(x_3))$ and C^I_{gcin}. ▲

The above example shows that for model expansion, local domain symmetries are useful, but Theorem 5.3.9 does not suffice to identify them. Below, we give a sufficient condition for A, π-induced structure transformations to be local domain symmetries of a model expansion problem. For this, we require the notion of a *decomposition*.

Definition 5.3.13. Let $MX(\mathcal{T}, I_{in})$ be a model expansion problem with input vocabulary Σ_{in}. Also, let $\mathcal{T}^*$ be equal to $\mathcal{T}$ with each occurrence of a symbol $S \in \Sigma_{in}$ replaced by a unique new copy S_i, let Σ^*_{in} be the vocabulary containing all copy symbols S_i, and let I^*_{in} be the Σ^*_{in}-structure where for each copy S_i, $S_i^{I^*_{in}} = S^I_{in}$. We call $MX(\mathcal{T}^*, I^*_{in})$ the *decomposition* of $MX(\mathcal{T}, I_{in})$.

It is clear that a model expansion problem and its decomposition have the same solutions, as they have the same output vocabulary and as each occurrence of a copy S_i in $\mathcal{T}^*$ imposes the same constraints on models for $\mathcal{T}^*$ as S did for $\mathcal{T}$ (since $S_i^{I^*_{in}} = S^I_{in}$).

Example 5.3.14 (Example 5.3.12 continued)**.** Let $MX(\mathcal{T}^*_{gc}, I^*_{gcin})$ be the decomposition of $MX(\mathcal{T}_{gc}, I_{gcin})$. $\mathcal{T}^*_{gc}$ consists of

$$\begin{aligned}
&\forall x_1\ y_1\colon Edge_1(x_1, y_1) \Rightarrow (Color(x_1) \neq Color(y_1))\\
&\forall x_2\ y_2\colon Edge_2(x_2, y_2) \Rightarrow V_1(x_2) \land V_2(y_2)\\
&\forall x_3\colon C_1(Color(x_3))
\end{aligned}$$

▲

Given this decomposition notion, we finally give a sufficient condition on local domain symmetry for model expansion:

Theorem 5.3.15 (Local domain symmetry condition for MX)**.** *Let $MX(\mathcal{T}, I_{in})$ be a model expansion problem with decomposition $MX(\mathcal{T}^*, I^*_{in})$. If A is connectively closed under $\mathcal{T}^*$ and $\sigma^A_\pi(I^*_{in}) = I^*_{in}$ then σ^A_π is a symmetry for $MX(\mathcal{T}, I_{in})$.*

We refer to Appendix A.2 for a proof.

Example 5.3.16 (Example 5.3.14 continued)**.** Argument position set $A = \{Edge_1|1, Edge_1|2, Color|1, x_1|0, y_1|0, x_3|0\}$ is connectively closed under $\mathcal{T}^*_{gc}$, and $\sigma^A_{(t\ u\ v\ w)}(I^*_{gcin}) = I^*_{gcin}$. Thus, $\sigma^A_{(t\ u\ v\ w)}$ is a symmetry of $MX(\mathcal{T}_{gc}, I_{gcin})$, representing cyclicity of the input graph. However, $\sigma^A_{(t\ u)}$ is not a symmetry of $MX(\mathcal{T}_{gc}, I_{gcin})$, as the input interpretation of $Edge_1$ is not preserved by swapping t and u. ▲

Note that if $\Sigma_{in} = \emptyset$, the conditions of Theorem 5.3.15 simplify into the conditions of Theorem 5.3.9. Also, for σ^A_π satisfying Theorem 5.3.15, A typically contains argument positions over both Σ_{out} and the decomposed Σ^*_{in} (as well as over variables). Lastly, the requirement that A is connectively closed under $\mathcal{T}^*$ is weaker than being closed under $\mathcal{T}$. For example, let theory $\mathcal{T}$ consist of the sentence $P(f) \lor P(g)$, with only P interpreted by the input structure. The only connectively closed set under $\mathcal{T}$ is $\{P|1, f|0, g|0\}$. However, under the corresponding decomposition $P_1(f) \lor P_2(g)$, there are three connectively closed sets: $\{P_1|1, f|0\}$, $\{P_2|1, g|0\}$ and their union.

5.3.3 Subdomain interchangeability

Local domain symmetries for a theory $\mathcal{T}$ can be identified by computing argument position sets A that are connectively closed. Then, as mentioned in Section 5.3.1, any permutation π of the domain D gives rise to a local domain symmetry σ^A_π. For model expansion, σ^A_π must preserve the input structure, so not all π are guaranteed to induce symmetry. However, given a set of argument positions A, a subdomain $\delta \subseteq D$ might exist for which any permutation of δ induces a local domain symmetry of the model expansion problem.

Definition 5.3.17 (A-interchangeable subdomain)**.** Let $MX(\mathcal{T}, I_{in})$ be a model expansion problem, A a set of argument positions and δ a subset of the domain. δ is an *A-interchangeable subdomain* if for every permutation π over δ, the structure transformation σ^A_π induced by A, π is a local domain symmetry for $MX(\mathcal{T}, I_{in})$. The *subdomain interchangeability group* $\mathbb{G}^A_\delta$ is the group of all local domain symmetries induced by an A-interchangeable subdomain δ.

Example 5.3.18 (Example 5.2.1 continued)**.** Given $MX(\mathcal{T}_{gc}, I_{gcin})$, $\{r, g, b\}$ and $\{t, u, v, w\}$ are A-interchangeable subdomains for $A = \{C|1,\ Color|0\}$. For $B = \{V|1, Edge|1, Edge|2, Color|1, x_1|0, y_1|0, x_2|0, y_2|0, x_3|0\}$, $\{r, g, b\}$ is a B-interchangeable subdomain. However, as $\sigma^B_{(t\ u)}$ does not preserve the interpretation of $Edge$, $\{t, u, v,\ w\}$ is not a B-interchangeable subdomain. ▲

Many problems, when modeled as a model expansion problem, exhibit subdomain interchangeability. For instance, a set of nurses in a scheduling problem, a set of colors in a graph coloring problem, or a set of trucks in a planning problem often are interchangeable subdomains.

Subdomain interchangeability groups contain a number of symmetries factorial in the size of the interchangeable subdomain, leading to an exponential slowdown of many combinatorial search algorithms. However, as we argue in Section 5.4, many subdomain interchangeability groups can be completely broken with a number of constraints linear in the size of the subdomain.

5.3.4 More symmetry

Even though local domain symmetry is a useful form of symmetry, it does not capture all symmetry properties that might be present in a model expansion problem.

Example 5.3.19 (Example 5.2.1 continued)**.** The graph coloring problem $MX(\mathcal{T}_{gc}, I_{gcin})$ asks to color a circular directed graph of 4 vertices $\{t, u, v, w\}$. Note that given any satisfying coloring for this graph, swapping the colors of t and v (or u and w) keeps $\mathcal{T}_{gc}$ satisfied. This is a clear symmetry property of the graph coloring instance, but it cannot be captured using the notion of local domain symmetry as defined in this chapter. For instance, if we take the argument position set $A = \{Edge_1|1, Edge_1|2, Color|1\}$ representing symmetry in the vertices, then the induced structure transformation $\sigma^A_{(t\ v)}$ is not a symmetry of $MX(\mathcal{T}_{gc}, I_{gcin})$ since it does not preserve the interpretation of $Edge_1$.

One way to fix this is based on the observation that argument positions $Edge|1$ and $Edge|2$ are indistinguishable in $\mathcal{T}_{gc}$: in each sentence of $\mathcal{T}_{gc}$, one could swap any quantifier over $Edge|1$ with one over $Edge|2$, ending up with a sentence equivalent to the original one. In more bold words, argument positions $Edge|1$ and $Edge|2$ *themselves* are symmetric. This symmetry property can be captured by generalizing the notion of an A, π-induced structure transformation to allow for swaps or permutations of argument positions in addition to swaps or permutations of domain elements.

For instance, we could define $\sigma_A^{(Edge_1|1\ Edge_1|2)(t\ v)}$ to first map $Edge_1^{I^*_{gcin}}$ to $\{(e, d) \mid (d, e) \in Edge_1^{I^*_{gcin}}\}$ before applying $\sigma_A^{(t\ v)}$. Note that $\sigma_A^{(Edge_1|1\ Edge_1|2)(t\ v)}$ would preserve $Edge_1^{I^*_{gcin}}$ and I^*_{gcin} in general, while also preserving satisfaction to $\mathcal{T}^*_{gc}$, making it a symmetry of $MX(\mathcal{T}_{gc}, I_{gc})$. ▲

Similarly, problems with spatial properties often have rotational or reflectional symmetry, which is not covered by the presented notion of local domain symmetry. One such example is the N-Queens problem, which is experimentally investigated in Section 5.6.

5.4 Symmetry breaking

It follows from Definition 5.3.11 that a symmetry group $\mathbb{G}$ of a model expansion problem $MX(\mathcal{T}_{gc}, I_{gcin})$ partitions the set of output structures $\Gamma_D^{\Sigma_{out}}$ into *symmetry classes*, such that two output structures I_{out} and I'_{out} belong to the same symmetry class if there exists a symmetry $\sigma \in \mathbb{G}$ such that $\sigma(I_{out}) = I'_{out}$. When solving a model expansion problem $MX(\mathcal{T}, I_{in})$ with symmetry group $\mathbb{G}$, it suffices to only check for one output structure I_{out} in each symmetry class whether $I_{in} \sqcup I_{out} \models \mathcal{T}$.

Hence, a standard approach of dealing with symmetry extends the theory $\mathcal{T}$ with *symmetry breaking formulas* (sbf) that falsify as many output structures in a symmetry class as possible, while guaranteeing at least one output structure in each symmetry class satisfies the sbf. This way, a solver will avoid visiting symmetrical parts of the search space, as the sbf guides the search towards only the satisfying output structures. More formally, a sbf φ is *sound* for a symmetry group $\mathbb{G}$ of a model expansion problem if for each output structure I_{out}, there exists *at least* one $\sigma \in \mathbb{G}$ such that $\sigma(I_{out})$ satisfies φ; it is *complete* if there exists *at most* one such $\sigma \in \mathbb{G}$ [99].

Often, symmetry breaking is done by defining a lexicographical order over the set of output structures [79]. For a given symmetry, so-called *lex-leader constraints* then encode that an output structure's symmetrical image cannot be strictly smaller under the defined lexicographical order, hence removing "large" output structures, but retaining "small" ones under the lexicographical order. As long as the chosen lexicographical order is fixed, the conjunction of lex-leader constraints for any set of symmetries is a sound sbf.

We construct a lexicographical order $\preceq_\Gamma$ over a set of output structures $\Gamma_D^{\Sigma_{out}}$ from an order $\preceq_D$ over the domain D and an order $\preceq_{\Sigma_{out}}$ over the vocabulary Σ_{out}. $\preceq_\Gamma$ itself is constructed from a lexicographical order $\preceq_D^\Sigma$ on the possible interpretations in Γ for a symbol $S \in \Sigma_{out}$. We take $S^{I_{out}} \prec_D^\Sigma S^{I'_{out}}$ to hold iff there exists some domain element tuple $\bar{d}$ such that $\bar{d} \notin S^{I_{out}}$, $\bar{d} \in S^{I'_{out}}$, and for all $\bar{d}' \prec_D \bar{d}$ it holds that $\bar{d}' \in S^{I_{out}} \Leftrightarrow \bar{d}' \in S^{I'_{out}}$. Then, $I_{out} \prec_\Gamma I'_{out}$ iff there exists some symbol $S \in \Sigma_{out}$ where $S^{I_{out}} \prec_D^\Sigma S^{I'_{out}}$, and for all $S' \prec_{\Sigma_{out}} S$ it holds that $S'^{I_{out}} = S'^{I'_{out}}$. In essence, the structure order is built on an interpretation order, which is built on a tuple order.

Assuming $\Sigma_{out} = \{S_1, \ldots, S_n\}$ and $S_i \preceq_{\Sigma_{out}} S_j$ iff $i < j$, the lex-leader constraint for a symmetry σ of a model expansion problem $MX(\mathcal{T}, I_{in})$ can be encoded as the propositional formula:

$$\begin{aligned}
& S_1^{I_{out}} \preceq_D^\Sigma S_1^{\sigma(I_{out})} \wedge \\
S_1^{I_{out}} = S_1^{\sigma(I_{out})} \Rightarrow {} & S_2^{I_{out}} \preceq_D^\Sigma S_2^{\sigma(I_{out})} \wedge \\
S_1^{I_{out}} = S_1^{\sigma(I_{out})} \wedge S_2^{I_{out}} = S_2^{\sigma(I_{out})} \Rightarrow {} & S_3^{I_{out}} \preceq_D^\Sigma S_3^{\sigma(I_{out})} \wedge \\
& \ldots \wedge \\
S_1^{I_{out}} = S_1^{\sigma(I_{out})} \wedge \ldots \wedge S_{n-1}^{I_{out}} = S_{n-1}^{\sigma(I_{out})} \Rightarrow {} & S_n^{I_{out}} \preceq_D^\Sigma S_n^{\sigma(I_{out})}
\end{aligned}$$

which informally states that an interpretation to S should be lexicographically smaller than its symmetric $\sigma(S)$, as long as the symmetry is not broken by an interpretation to a "smaller" symbol S'.

The propositions $S_i^{I_{out}} = S_i^{\sigma(I_{out})}$ and $S_i^{I_{out}} \preceq_D^\Sigma S_i^{\sigma(I_{out})}$ are then further encoded in terms of auxiliary propositional variables denoting whether tuples of domain elements belong to a symbol's interpretation. We have provided such an encoding in Section 3.3.

Using subformula chaining techniques similar to those used in Section 3.3, the lex-leader constraint for any symmetry can be encoded as a propositional formula in conjunctive normal form (CNF) of size $O(|\Sigma_{out}||D|^{k+1})$ with k the highest arity of a symbol in Σ_{out}.

For the remainder of this section, we leave the order over Σ_{out} implicit, but explicitly state the order $\preceq_D$ over domain D, as this turns out to be important. Given a model expansion problem with symmetry σ and a lexicographical order over $\Gamma_D^{\Sigma_{out}}$ with $\preceq_D$ as D-order, we use $LL^{\preceq_D}(\sigma)$ to refer to the logical formula encoding the lex-leader constraint for σ.

Example 5.4.1 (Example 5.2.1 continued)**.** Let $A = \{C|1, \mathit{Color}|0\}$, then $\sigma^A_{(r\ g)}$ is a local domain symmetry for $MX(\mathcal{T}_{gc}, I_{gcin})$. As *Color* is the only uninterpreted vocabulary symbol, the set of output structures $\Gamma_D^{\Sigma_{gcout}}$ corresponds to the set of interpretations for *Color* under domain D. Let D be ordered: $t \prec_D u \prec_D v \prec_D w \prec_D r \prec_D g \prec_D b$. $\prec_D$ induces an order $\prec_D^{\Sigma}$ over the interpretations of the only output symbol *Color*, which in turn induces an order $\prec_\Gamma$ over the set of output structures $\Gamma_D^{\Sigma_{gcout}}$.

Then, local domain symmetry $\sigma^A_{(r\ g)}$ is broken by the lex-leader constraint $LL^{\preceq_D}(\sigma^A_{(r\ g)})$, which states that each interpretation to *Color* must be lexicographically smaller than its symmetrical interpretation. Formally, $LL^{\preceq_D}(\sigma^A_{(r\ g)})$ enforces $\mathit{Color}^{I_{gcout}} \preceq_D^{\Sigma} \mathit{Color}^{\sigma^A_{(r\ g)}(I_{gcout})}$.

In detail, $\mathit{Color}^{I_{gcout}} \prec_D^{\Sigma} \mathit{Color}^{I'_{gcout}}$ iff there exists $(d, d') \in D \times D$ such that for all $(e, e') \prec_D (d, d') \in D \times D$ holds $\mathit{Color}^{I_{gcout}}(e) = e' \Leftrightarrow \mathit{Color}^{I'_{gcout}}(e) = e'$ and $\mathit{Color}^{I_{gcout}}(d) \neq d'$ and $\mathit{Color}^{I'_{gcout}}(d) = d'$.

Thus, to enforce $\mathit{Color}^{I_{gcout}} \preceq_D^{\Sigma} \mathit{Color}^{\sigma^A_{(r\ g)}(I_{gcout})}$, we state for all $(d, d') \in D \times D$ that if for all $(e, e') \prec_D (d, d') \in D \times D$ it holds that $\mathit{Color}^{I_{gcout}}(e) = e' \Leftrightarrow \mathit{Color}^{\sigma^A_{(r\ g)}(I_{gcout})}(e) = e'$, then $\mathit{Color}^{I_{gcout}}(d) \neq d'$ or $\mathit{Color}^{\sigma^A_{(r\ g)}(I_{gcout})}(d) = d'$.

By Definition 5.3.5, $\mathit{Color}^{\sigma^A_{(r\ g)}(I_{gcout})}(d) = (r\ g)(d')$ iff $\mathit{Color}^{I_{gcout}}(d) = d'$. Thus, $\mathit{Color}^{I_{gcout}} \preceq_D^{\Sigma} \mathit{Color}^{\sigma^A_{(r\ g)}(I_{gcout})}$ holds if, e.g for domain element pair (t, r), constraints enforce that

$$\mathit{Color}^{I_{gcout}}(t) \neq r \vee \mathit{Color}^{I_{gcout}}(t) = (r\ g)^{-1}(r) = g$$

if for all $(e, e') \prec_D (t, r)$, $\mathit{Color}^{I_{gcout}}(e) = e' \Leftrightarrow \mathit{Color}^{I_{gcout}}(e) = (r\ g)^{-1}(e')$. Under our current ordering, these constraints cut away Σ_{out}-structures that color t with r.

Using auxiliary propositional variables $C_{dd'}$ denoting whether $\mathit{Color}^{I_{gcout}}(d) = d'$, auxiliary chaining variables $E_{dd'}$ denoting whether $\mathit{Color}^{I_{gcout}}(e) = e' \Leftrightarrow \mathit{Color}^{I_{gcout}}(e) = (r\ g)^{-1}(e')$ for all $(e, e') \prec_D (d, d')$, and employing symmetry breaking constraint optimizations mentioned in Section 3.3, the following is the full propositional symmetry breaking constraint $LL^{\preceq_D}(\sigma^A_{(r\ g)})$:

$$E_{tr}$$

$$E_{tr} \Rightarrow \neg C_{tr} \vee C_{tg}$$

$$E_{tr} \wedge (C_{tr} \vee \neg C_{tg}) \Rightarrow E_{ur}$$

$$E_{ur} \Rightarrow \neg C_{ur} \vee C_{ug}$$

$$E_{ur} \wedge (C_{ur} \vee \neg C_{ug}) \Rightarrow E_{vr}$$

$$E_{vr} \Rightarrow \neg C_{vr} \vee C_{vg}$$

$$E_{vr} \wedge (C_{vr} \vee \neg C_{vg}) \Rightarrow E_{wr}$$

$$E_{wr} \Rightarrow \neg C_{wr} \vee C_{wg}$$

$$E_{wr} \wedge (C_{wr} \vee \neg C_{wg}) \Rightarrow E_{rr}$$

$$E_{rr} \Rightarrow \neg C_{rr} \vee C_{rg}$$

$$E_{rr} \wedge (C_{rr} \vee \neg C_{rg}) \Rightarrow E_{gr}$$

$$E_{gr} \Rightarrow \neg C_{gr} \vee C_{gg}$$

$$E_{gr} \wedge (C_{gr} \vee \neg C_{gg}) \Rightarrow E_{br}$$

$$E_{br} \Rightarrow \neg C_{br} \vee C_{bg}$$

Informally, $LL^{\preceq_D}(\sigma^A_{(r\ g)})$ enforces that for each vertex v, if all vertices $v' \prec_D v$ are not colored by r or by g, then v cannot be colored with r. ▲

Note that lex-leader constraints are constructed for individual symmetries. In general, to obtain a complete symmetry breaking constraint for a symmetry group $\mathbb{G}$, one needs to post $LL^{\preceq_D}(\sigma)$ for each $\sigma \in \mathbb{G}$. As symmetry groups can contain a factorial amount of symmetries this is infeasible, e.g., in the case of subdomain interchangeability. Instead, the standard approach is *partial symmetry breaking*, where $LL^{\preceq_D}(\sigma)$ is posted for a minimal set of generators σ of $\mathbb{G}$ [4]. Partial symmetry breaking is feasible, but does not guarantee that $\mathbb{G}$ is broken completely, leaving symmetrical parts of the search space open to a search engine.

For instance, for a subdomain interchangeability group $\mathbb{G}_\delta^A$, a minimal set of generator symmetries is $\{\sigma^A_{(d\ s(d))} \mid d, s(d) \in \delta\}$, where $s(d)$ is the successor of d in δ according to $\preceq_D$. Other minimal generator sets exist as well, e.g., $\{\sigma^A_{(d_0\ d)} \mid d \in \delta, d \neq d_0\}$ for a fixed $d_0 \in \delta$. However, the choice of the generator set influences the power of the symmetry breaking formula. For subdomain interchangeability groups $\mathbb{G}$, choosing the right generator set can guarantee that the lex-leader constraints used in partial symmetry breaking are actually complete for $\mathbb{G}$:

Theorem 5.4.2. *Let $MX(\mathcal{T}, I_{in})$ be a model expansion problem, δ an A-interchangeable subdomain, $\preceq_D$ a total order on domain D and $s(d)$ the successor of d in δ according to $\preceq_D$. If A contains at most one argument position $S|i$ for each symbol $S \in \Sigma_{out}$, then the conjunction of lex-leader constraints*

$$\{LL^{\preceq_D}(\sigma^A_{(d\ s(d))}) \mid d, s(d) \in \delta\}$$

is a complete symmetry breaking constraint for the subdomain interchangeability group $\mathbb{G}_\delta^A$.

We refer to Appendix A.3 for a proof.

Intuitively, Theorem 5.4.2 states that local domain interchangeability is completely broken by a linear number of lex-leader constraints if the set of argument positions contains at most one argument position for each output symbol. These lex-leader constraints $LL^{\preceq_D}(\sigma^A_{(d\ s(d))})$ are based on swaps $(d\ s(d))$ of two consecutive domain elements over the chosen domain ordering.

It is worth remarking that Theorem 5.4.2 is in fact the first-order conversion of Corollary 3.4.3, which stated a completeness result for propositional row interchangeability symmetry. The requirement that for each symbol at most one argument position can belong to the argument position set then guarantees that at *ground* level, the propositional variables of concern can be structured as a matrix with interchangeable rows.

A strongly related result is that when constructing a relation $R \subseteq D_1 \times \ldots \times D_n$ for which exactly one dimension D_i contains interchangeable values, an efficient lex-leader constraint exists that completely breaks the resulting symmetry [90]. Theorem 5.4.2 can be seen as a conversion of this result to a model expansion context with local domain interchangeability.

Example 5.4.3 (Example 5.2.1 continued)**.** Given the graph coloring problem $MX(\mathcal{T}_{gc}, I_{gcin})$, let $A = \{C|1, Color|0\}$ and $r \prec_D g \prec_D b$. $\mathbb{G}^A_{\{r,g,b\}}$ is a subdomain interchangeability group of $MX(\mathcal{T}_{gc}, I_{gcin})$. Since A contains only one argument position for the symbol $Color$, it is completely broken by

$$LL^{\preceq_D}(\sigma^A_{(r\ g)}) \wedge LL^{\preceq_D}(\sigma^A_{(g\ b)})$$

▲

Example 5.4.4 (Ramsey number interchangeability)**.** In combinatorial mathematics, *Ramsey's theorem* states that one will find monochromatic cliques in any edge coloring of a sufficiently large complete graph. Since the graphs are complete, all vertices are interchangeable. A sensible FO model expansion representation of a Ramsey number problem could involve the function symbol $edgecolor/2$, mapping pairs of vertices to colors. The vertices would then form an interchangeable subdomain, with the corresponding set of argument positions A containing both $edgecolor|1$ and $edgecolor|2$. As A contains two argument positions for the symbol *edgecolor*, this subdomain interchangeability group cannot be completely broken by Theorem 5.4.2. ▲

The finite model generation system SEM also breaks this type of symmetry completely, by way of *dynamically* avoiding symmetrical decisions during search. [100] The more recent model generator KODKOD [94] breaks symmetry statically by posting lex-leader constraints from [4] for global domain symmetry. Although [94] do not mention any completeness result, experiments with a pigeonhole encoding in KODKOD indicate that it uses the right set of generator symmetries to completely break all pigeon and hole interchangeability symmetry.

5.5 Symmetry detection

In this section, we give two local domain symmetry detection algorithms for model expansion problems. The first detects generators of a local domain symmetry group, the second derives interchangeable subdomains. Both approaches work on a first-order level, avoiding the need to *ground* the model expansion problem to a propositional counterpart. Both algorithms are based on Theorem 5.3.15, which conditions argument position set A to be connectively closed under decomposition theory $\mathcal{T}^*$. To find such A, one simply constructs a partition of $\mathcal{T}^*$'s argument positions where connected argument positions belong to the same partition component. Using a disjoint-set data structure[1], the computational cost to find A is linear in the size of $\mathcal{T}$. In the following subsections, we assume a set of argument positions A satisfying the connectedness condition is available, leaving only the concern of finding an appropriate domain permutation π (Section 5.5.1) or interchangeable subdomain δ (Section 5.5.2).

[1] `en.wikipedia.org/wiki/Disjoint-set_data_structure`

5.5.1 Local domain symmetry detection

Our approach follows other symmetry detection techniques [1, 41] by converting the symmetry detection problem to a *graph automorphism* detection problem. An *automorphism* of a graph is a permutation τ of its vertices such that each vertex pair (v, u) forms an edge iff $(\tau(v), \tau(u))$ forms an edge. If the graph is colored, then each vertex v must have the same color as $\tau(v)$.

This existing work encodes a propositional theory into a graph, which we call the *detection graph*. If the detection graph is well-constructed, its automorphism group corresponds to a symmetry group of the propositional theory. Tools such as SAUCY [60], NAUTY [71] and BLISS [58] then are employed to derive generators for the detection graph's automorphism group, which in turn are converted to symmetry generators for the propositional theory.

Our approach differs by not encoding a propositional theory into the detection graph, but an input structure and a set of argument positions, as these are all we need to detect local domain symmetry. Formally, given a structure I and an argument position set A, we construct an undirected colored graph whose automorphisms correspond to domain permutations π such that $\sigma_\pi^A(I) = I$ – satisfying the second condition of Theorem 5.3.15.

Definition 5.5.1 (Domain permutation graph)**.** Let I be a Σ-structure with domain D and A a set of argument positions. The *domain permutation graph* $DPG(I, A)$ for I and A is an undirected colored graph with labeled vertices V, edges E and color function c that satisfies the following requirements:

V is partitioned into three subsets:

- DE (domain element vertices)
- AP (argument position vertices)
- IT (interpretation tuple vertices)

DE contains a vertex labeled d for each $d \in D$. AP contains $k + 1$ vertices labeled $\{d.i \mid i \in [0..k]\}$ for each $d \in D$, with k the maximum arity of symbols in Σ. IT contains a vertex labeled $S(\bar{d})$ for each tuple $\bar{d} \in S^I$ with $S^I \in I$.

E consists only of edges between DE and AP, and between AP and IT. An AP vertex labeled $d.i$ is connected to a DE vertex e iff $d = e$. An IT vertex labeled $S(\ldots, d_i, \ldots)$ is connected to an AP vertex $e.j$ iff $d = e$, $i = j$ and $S|i \in A$.

Vertices from different partitions have different colors. All DE vertices have the same color. Two AP vertices labeled $d.i$ and $e.j$ have the same color iff $i = j$.

Two IT vertices labeled $S(d_1,\ldots,d_n)$ and $R(e_1,\ldots,e_n)$ have the same color iff $S = R$ and $d_i = e_i$ for all i such that $S|i \notin A$.

The intuition behind the domain permutation graph $DPG(I,A)$ is that a permutation of its DE vertices corresponds to a domain permutation π, a permutation of its IT vertices corresponds to a permutation of domain element tuples in interpretations in I, and the AP vertices and vertex coloring serve to link DE and IT in such a way that Definition 5.3.5 is preserved for automorphisms.

Theorem 5.5.2. *Let I be a Σ-structure with domain D and A a set of argument positions. There exists a bijection between the automorphism group of the domain permutation graph $DPG(I,A)$ and the group of domain permutations π such that $\sigma^A_\pi(I) = I$. This bijection maps an automorphism τ to domain permutation π iff $\tau(d) = \pi(d)$ for all DE vertices (equated with domain elements) d.*

We refer to Appendix A.4 for a proof.

Example 5.5.3 (Example 5.3.16 continued)**.** Using argument position set $A = \{Edge_1|1, Edge_1|2, Color|1, x_1|0, y_1|0, x_3|0\}$ (which is connectively closed under $\mathcal{T}^*_{gc}$) and input structure I^*_{gcin}, the domain permutation graph $DPG(I^*_{gcin}, A)$ is illustrated in Figure 5.1. The automorphism group of $DPG(I^*_{gcin}, A)$ corresponds to the group of induced structure transformations σ^A_π such that $\sigma^A_\pi(I^*_{gcin}) = I^*_{gcin}$. As a result, its automorphism group corresponds to a local domain symmetry group of $MX(\mathcal{T}_{gc}, I_{gcin})$. E.g., $\sigma^A_{(t\ u\ v\ w)}$ corresponds to an automorphism that permutes the four left-most groups of five vertices, and $\sigma^A_{(b\ g)}$ to an automorphism that swaps the two right-most groups of four vertices. ▲

Let k be the largest arity of a symbol in I for a domain permutation graph $DPG(I,A)$. The size of DE is $|D|$, the size of AP is $(k+1)|D|$, and the size of IT is $|I|$, which is $O(|D|^k)$. Thus, the total number of nodes is $O(k|D| + |D|^k)$. There are $(k+1)|D|$ edges between DE and AP, and, if all argument positions over some symbol S/k occur in A, then there are $O(k|I|) = O(k|D|^k)$ edges between AP and IT. Thus, the total number of edges is $O(k|D|^k)$.

Note that the size of $DPG(I,A)$ does not depend on the size of the theory of the model expansion problem. This is a major advantage compared to automorphism-based symmetry detection on ground theories, as the detection graph grows linearly with the ground theory [41], which is typically much larger than the input structure.

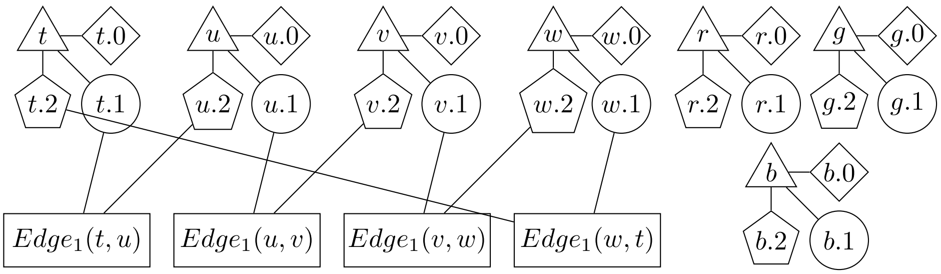

Figure 5.1: Domain permutation graph $DPG(I^*_{gcin}, A)$ with $A = \{Edge_1|1, Edge_1|2, Color|1, x_1|0, y_1|0, x_3|0\}$. Each shape denotes a unique color, so vertices with the same shape have the same color.

5.5.2 Subdomain interchangeability detection

While Section 5.5.1 detects local domain symmetry generators individually, we have no information on the structure of the symmetry group represented by the generators. To optimally construct symmetry breaking constraints for symmetry groups, we would like to detect subdomain interchangeability symmetry groups as a whole. Then, by Theorem 5.4.2, we might often be able to break subdomain interchangeability groups completely with a set of lex-leader constraints linear in $|D|$.

Given a model expansion problem $MX(\mathcal{T}, I_{in})$ with decomposition $MX(\mathcal{T}^*, I^*_{in})$ and a set of argument positions A connectively closed under $\mathcal{T}^*$, the task at hand is to find a subdomain $\delta \subseteq D$ such that for each permutation π over δ, $\sigma^A_\pi(I^*_{in}) = I^*_{in}$. If so, Theorem 5.3.15 guarantees σ^A_π to be a symmetry of $MX(\mathcal{T}, I_{in})$, which makes δ an A-interchangeable subdomain.

The actual algorithm finds a partition Δ of D, such that each $\delta \in \Delta$ is A-interchangeable. The idea is based on the fact that the permutation group of a set is generated by swaps of two elements of the set. As such, if we know for each pair $d_1, d_2 \in D$ whether $\sigma^A_{(d_1\ d_2)}(I^*_{in}) = I^*_{in}$, it is straightforward to construct the partition Δ. The resulting symmetry detection algorithm is simple: for each pair of domain elements $d_1, d_2 \in D$, check whether $\sigma^A_{(d_1\ d_2)}(I^*_{in}) = I^*_{in}$. When using a disjoint-set data structure[1] to keep track of the partition Δ, the complexity of this algorithm is $O(|D|^2|I^*_{in}|)$. This algorithm is optimized by exploiting transitivity, counting the ocurrences of domain elements in the input structure, or using interpretations of unary symbols to partition the domain. However, this does not improve the worst-case complexity.

Example 5.5.4 (Example 5.3.16 continued)**.** Given argument position set $A = \{C_1|1, Color|0\}$ (which is connectively closed under $\mathcal{T}^*_{gc}$), we detect A-interchangeable domains by checking whether the (only) input symbol C_1 has the same interpretation in $\sigma^A_{(d_1\ d_2)}(I^*_{in})$ as in I^*_{in} for combinations of $d_1, d_2 \in \{t, u, v, w, r, g, b\}$. For $(d_1, d_2) \in \{(t, u), (u, v), (v, w), (r, g), (g, b)\}$ this is indeed the case. For $(d_1, d_2) = (w, r)$ this is not the case, so the A-interchangeable sets are $\{t, u, v, w\}$ and $\{r, g, b\}$. ▲

An interesting property that is useful to avoid superfluous symmetry detection is the following:

Theorem 5.5.5. *Let $MX(\mathcal{T}, I_{in})$ be a model expansion problem with decomposition $MX(\mathcal{T}^*, I^*_{in})$ and decomposed input vocabulary Σ^*_{in}. Let A be an argument position set connectively closed under $\mathcal{T}^*$. If A contains at most one argument position for each decomposed symbol in Σ^*_{in}, the permutations π such that $I^*_{in} = \sigma^\pi_A(I^*_{in})$ always stem from interchangeable subdomains $\delta \supseteq Supp(\pi)$.*

We refer to Appendix A.5 for a proof.

In other words, If A contains at most one argument position for each decomposed symbol in Σ^*_{in}, the only local domain symmetry we can detect by both methods presented in the previous sections, is due to local domain interchangeability. Hence, in this case, it is sufficient to only detect local domain interchangeability with the technique from this section, and skip the more general symmetry detection method based on automorphism graphs presented in Section 5.5.1.

5.6 Experiments

Based on the theory and algorithms presented in this chapter, we implemented symmetry exploitation in the model expansion inference of the IDP system [25]. IDP is a *knowledge base system* where knowledge about a problem can be modeled in FO(·), a rich extension of first-order logic [29]. Our implementation is incorporated in IDP version 3.6.0 and up, which is published online.[2] Our implementation makes use of Saucy version 3 to solve the graph automorphism component of symmetry detection, and constructs symmetry breaking formulas based on the lex-leader encoding of Section 3.3. Similarly to BreakID, for any symmetry generator π returned by graph automorphism detection, our implementation limits the size of a symmetry breaking formula for π to 50 (see Section 3.3). We do not limit the size of symmetry breaking formulas for subdomain interchangeability generators. Finally, our implementation also

[2] `dtai.cs.kuleuven.be/software/idp/try`

performs two equivalence preserving transformations on the theory and input structure, as to maximize the number of symmetries detected. The first is pushing quantifiers as deeply as possible in formulas, and calculating the interpretations of fixed output vocabulary symbols.

We compare this implementation with the ASP system Clasp [47] version 3.1.4, using version 4.5.4 of the ASP grounder gringo to generate ground answer set programs. For Clasp, the symmetry breaking preprocessor sbass has been developed [41]. sbass takes a ground answer set program, encodes it to a detection graph, uses Saucy to solve the automorphism detection problem, converts Saucy's output to permutations of propositional atoms that induce symmetries, and constructs symmetry breaking constraints following [4]. As BreakID also is a ground ASP symmetry breaking preprocessor, we include it in this experiment.

Our experiment uses five different system configurations: IDP and Clasp refer to both systems without symmetry breaking, and IDPsym refers to IDP extended with the techniques described in this chapter. sbass refers to Clasp coupled with the eponymous symmetry breaking preprocessor, as does BreakID.

This experiment can only broadly compare the IDP and Clasp configurations, as both systems use similar but ultimately different techniques to solve the model expansion problem. Our main interest is to investigate the types of symmetry detected, the overhead needed to detect those, and the relative speedup gained when activating symmetry algorithms for both systems.

We expect that IDPsym, compared to sbass and BreakID, has less symmetry detection overhead, as IDPsym detects symmetry on the first-order level instead of on the ground level. E.g., the structure information present in a set of connectively closed argument positions can be derived with a syntactical check on the first-order theory, but this information is lost after grounding. As a result, we expect IDPsym's detection graph to be smaller, or even non-existent.[3]

Also, we expect a larger relative speedup for IDPsym than for sbass on problems with a lot of subdomain interchangeability, as IDPsym detects and often completely breaks this type of symmetry. Recall that BreakID's row interchangeability detection is a form of subdomain interchangeability detection, so BreakID's performance could approach IDPsym's in this regard. However, BreakID's row interchangeability detection is approximative, and might miss some members of an interchangeable subdomain.

Finally, as mentioned in Section 5.3.4, IDPsym's detected symmetry group

[3]IDPsym does not construct the detection graph if the only generators it will detect are due to subdomain interchangeability, as per Theorem 5.5.5.

might be smaller than BreakID's or sbass's, as not all symmetry properties of a problem can be captured by our notion of local domain symmetry.

Our benchmark set consists of four problem families also used in the experiments in Section 3.11: **pigeons**, **crew**, **graceful** and **200queens**.

pigeons is a set of 16 unsatisfiable pigeonhole instances where n pigeons must be placed in $n-1$ different holes. n takes values from $\{5, 6, \ldots, 14, 15, 20, 30, 50, 70, 100\}$. The pigeons and holes are interchangeable, leading to a large symmetry group.

crew is a set of 42 unsatisfiable airline crew scheduling instances, where optimality has to be proven for a minimal crew assignment given a moderately complex flight plan. The instances are generated by hand, with the number of crew members ranging from 5 to 25. Crew members have different attributes, but depending in the instance, multiple crew members exist with the same exact attribute set, making these crew members interchangeable.

graceful consists of 60 satisfiable and unsatisfiable graceful graph instances, taken from 2013's ASP competition [5]. These instances require to label a graph's vertices and edges such that all vertices have a different label, all edges have a different label, and each edge's label is the difference of the labels of the vertices it connects. The labels used are $\{0, 1, \ldots, n\}$, with n the number of edges. Any symmetry exhibited by the input graph is present, as well as a symmetry mapping each vertex' label l to $n-l$.

200queens is a set of 4 large satisfiable N-Queens instances trying to fit n queens on an n by n chessboard so that no queen threatens another. n takes values from $\{50, 100, 150, 200\}$. The symmetries present in **200queens** are the rotational and reflective symmetries of the chessboard.

The available resources were 6GB RAM and 1000s timeout on an Intel® Core® i5-3570 CPU with Ubuntu 14.04 Linux kernel 3.13 as operating system. Resources to reproduce these experiments are available online [31].

Table 5.1 summarizes the results. Note that Table 5.1 is an extension of Table 3.4 with IDP-based results.

	CLASP	SBASS				BREAKID					IDP	IDPSYM				
	#	#	t	V	π	#	t	V	π	δ	#	#	t	V	π	δ
pigeons (16)	8	11	51.0	48814	43.5	14	12.7	95192	90	2	8	**16**	0.0	0	0	2
crew (42)	32	36	0.0	1722	7.8	**41**	0.0	2835	101	3.2	28	39	0.0	0	0	4.0
graceful (60)	**33**	28	0.7	127860	5.5	32	2.1	250614	103	1	23	28	0.5	15201	5.4	0.6
200queens (4)	4	4	33.7	3658002	2.0	4	48.4	7277508	126	1	4	4	4.8	0	0	0

Table 5.1: Experimental results of CLASP- and IDP-based solvers with and without symmetry breaking. # represents the number of solved instances, t the average symmetry detection time in seconds, V the average number of vertices in the detection graph, π the average number of symmetry generators detected by SAUCY, and δ the average number of detected row (for BREAKID) or domain (for IDPSYM) interchangeability symmetry groups.

When analyzing the results on **pigeons**, it is clear that plain Clasp and IDP get lost in symmetric parts of the search tree, solving only 8 instances (up to 12 pigeons). sbass can only solve three more instances (up to 15 pigeons), as the derived symmetry generators do not suffice to construct strong symmetry breaking constraints. These results are consistent with [41]. BreakID performs better due to its row interchangeability detection, but for the larger instances it cannot detect the full interchangeability group. IDPsym detects the full pigeon and hole interchangeable subdomains, and its complete symmetry breaking constraints allow all 16 instances to be solved (up to 100 pigeons). As far as symmetry detection time goes, unlike sbass and BreakID, IDPsym has negligible detection overhead as it infers it does not need to construct an automorphism detection graph, and as the input structure is trivial to analyze.

The results on **crew** are similar to **pigeons** but less outspoken. The reason is that even though there are more subdomain interchangeability groups, the subdomains are a lot smaller, incurring less symmetry overhead. As a result, IDPsym enjoys a small advantage over sbass but is at a small disadvantage compared to BreakID. Nonetheless, IDPsym improves IDP's performance more than BreakID or sbass does Clasp's. Concerning symmetry detection, IDPsym has to analyze a more complex input structure before deriving any subdomain interchangeability groups, which contrasts with the trivially interchangeable pigeons and holes in **pigeons**. Nonetheless, IDPsym solves this task in the blink of an eye, as do sbass and BreakID.

Continuing with **graceful**, IDPsym is the only approach that actually improves performance through symmetry breaking. This is probably due to Clasp simply having a stronger search engine than IDP, hence leaving less room for improvement. However, zooming in on unsatisfiable **graceful** instances, sbass and BreakID solve four compared to Clasp's two, and IDPsym solves four compared to IDP's one. For unsatisfiable instances, these static symmetry breaking approaches remain effective. This discrepancy between satisfiable and unsatisfiable instances is not uncommon, as static symmetry breaking reduces the search space by removing possibly easy-to-find solutions. These results are also consistent with those reported by [41].

Looking at the number of symmetry generators detected, BreakID's large number of symmetry generators is due to auxiliary variables introduced during grounding, which sbass seems to ignore. It is more interesting to remark that sbass and IDPsym detect about the same number of symmetry generators for **graceful**, indicating that they detect the same symmetry groups. Note that both IDPsym and BreakID detect some subdomain interchangeability groups, apparently present in the input graph. As far as the size of the automorphism detection graph goes, IDPsym's is an order of magnitude smaller than the ones constructed by sbass and BreakID. We conclude that for **graceful**, IDPsym detects the same symmetry group as SBASS and BREAKID, though with a smaller automorphism detection graph.

Lastly, for **200queens**, IDPSYM cannot detect the geometric symmetries of the chessboard, as this type of symmetry does not fit the definition of local domain symmetry. For instance, a square (i, j) on the chess board is diagonally reflected to square (j, i), while square (i, k) is reflected to (k, i). Domain element i is mapped to both j and k at position 0, violating the local domain symmetry requirement that it stems from a domain *permutation*. SBASS and BREAKID can detect this type of symmetry, as they detect permutations of propositional variables instead of domain elements. However, note the significant overhead incurred, as the detection graphs employed are huge.

We conclude that our approach has very low symmetry detection overhead, due to a smaller or non-existent detection graph. Moreover, by completely breaking subdomain interchangeability, we significantly increase the number of solved instances. However, not all symmetry present in the problem set is detected by our approach.

5.7 Discussion

We presented the notion of local domain symmetry for model expansion problems, which manifests itself on the first-order level. We gave a completeness result on the strength of symmetry breaking constraints for a special case of local domain symmetry, and we posted syntactical conditions to efficiently detect symmetry from a model expansion specification. Our experiment highlights the strengths and weaknesses of our approach. We have a very low symmetry detection overhead and we give symmetry breaking completeness guarantees for local domain interchangeability that are effective in practice. However, we cannot detect some forms of symmetry.

It is worth mentioning that local domain symmetry is not limited to pure classical logic; it is straightforward to extend our work to cardinalities, types or arithmetic. Similarly, logic programs under stable or well-founded semantics have symmetry properties induced by permutations of the domain (or *Herbrand universe*) and sets of argument positions. Our work easily transfers to these domains. In fact, our implementation in IDP already supports such extensions.

Investigating which types of symmetry fall outside our formalism, and inventing ways to detect and exploit these types of symmetry is natural future work. An interesting idea is that not only permutations of the domain lead to symmetry, but that permutations on (argument positions of) symbols in the vocabulary do so as well.

Our detection algorithms can also be used in combination with other types of symmetry exploitation than avoiding symmetrical overhead during search. Chapter 6 follows down this road.

5.7.1 Related work

For systems employing predicate logic as a modeling language, much work concerning symmetry has been done in the context of theorem proving and finite model generation systems [100, 9, 23, 94]. These systems differ from model expansion systems by not taking any input interpretations into account. Hence, by Theorem 5.5.5, all detected symmetry is due to subdomain interchangeability. We formalized the ideas present in these systems, and extended them to take input interpretations into account, allowing better symmetry detection.

Other non-ground symmetry detection methods can be found in constraint programming (CP). One approach is to detect symmetry on small instances of a problem, fit these symmetries to builtin but common symmetry types, and use theorem proving technology to check whether the instance symmetries are also symmetries of the problem specification [74]. Though original, it cannot detect forms of symmetry outside of the builtin types, and requires a costly symmetry checking step.

A more elegant approach is to detect symmetry of a conjunction of global constraints by finding the intersection of the symmetry group of each individual constraint [97]. This approach has the drawback of not being able to take instance specific information into account. Furthermore, there is no CP system implementing this approach, as the set of global constraints supported by current CP systems is large, with sometimes exotic symmetry groups attached to them. In this regard, FO is a very simple modeling language, easily allowing symmetry detection.

Chapter goal evaluation
Indeed, inferring symmetry on the first-order level is feasible, and provides efficiency advantages in both detecting symmetry as well as breaking it. The only encountered drawback is that not all forms of symmetry are incorporated in our formalism yet.

Chapter 6

Automated Local Search via Symmetry

Goal of the chapter
Given a combinatorial optimization problem, local search algorithms *move* locally from one (suboptimal) solution to another, hoping to improve a given objective function by such moves. These algorithms typically guarantee that after performing a move, the new solution still satisfies a set of *hard* constraints. As symmetries are transformations of solutions that preserve satisfiability, there is a clear but uninvestigated link between local search algorithms and symmetry. Moreover, very few automated local search approaches exist, while symmetry can be detected automatically. Symmetry detection algorithms might be employed to generate the necessary information for a local search algorithm.

This chapter is based on work presented at the Fourteenth International Workshop on Constraint Modelling and Reformulation – September 2015, Cork, Ireland [40].

6.1 Introduction

Combinatorial *optimization* problems are a generalization of combinatorial problems, where not only a satisfying, feasible solution has to be found, but where this solution should also optimize some given objective function.

The approaches used to solve a combinatorial optimization problem can be divided in two categories: *complete* and *incomplete* approaches. *Complete* approaches guarantee that, after halting, the full search space has been investigated, and an optimal solution (or the fact that none exists) will be returned. Examples of these complete approaches are SAT, ASP, MP, CP, or even a straightforward brute force search algorithm. Amongst many others, IDP and CLASP use complete techniques to tackle combinatorial optimization problems.

However, for problems with huge search spaces with many (suboptimal) solutions, complete approaches often do not halt within reasonable time. Instead, *incomplete* approaches have been devised, which do not guarantee that an optimal solution will ever be returned. One incomplete technique is implementing *metaheuristics*. Roughly, metaheuristics take an initial, suboptimal solution, and iteratively adjusting this solution to improve the optimization objective, until a stop-criterion is met and a hopefully optimal solution is achieved. Examples of metaheuristic approaches are *local search*, *genetic algorithms*, *hyperheuristics*, *swarm-based optimization* etc. [51]

In this chapter, we investigate how we can bring both fields of research closer together by providing an automated way of transforming a combinatorial optimization problem specification in the language FO($\cdot$) (used by IDP) to input for a local search algorithm. The method is based on symmetry properties of combinatorial optimization problems that hopefully map suboptimal solutions to better solutions. Our approach is not limited by the actual specification language; the only requirement for it to function is to be able to derive or state symmetry properties of the combinatorial optimization problem.

This chapter is organized as follows. In Section 6.2, we give a formalization of a combinatorial optimization problem as a *model optimization* problem, and how symmetry can be used to derive input neighborhoods for local search algorithms. Section 6.3 then formalizes local search, and establishes the link between local search algorithms and symmetry of a model optimization problem. Section 6.4 reports on the nature of local search neighborhoods derived by an implementation in the IDP system on multiple constraint optimization specifications. We also give performance experiments, showing that our implementation outperforms IDP's complete technology on certain benchmark sets.

6.2 Preliminaries

For the remainder of this chapter, we rely on the formalism established in Section 5.2 and 5.3.

6.2.1 Model optimization (MO)

A combinatorial optimization problem can be abstracted as a *model optimization* problem. The the IDP system tackles this kind of problem when executing its model optimization inference.

A *model optimization* (MO) problem $MO(\mathcal{T}, I_{in}, o)$ is a generalization of a model expansion problem, where $\mathcal{T}$ is a Σ-theory, I_{in} is a Σ_{in}-structure, and o is a Σ-term representing an objective function (typically to some numerical domain, say the integer numbers $\mathbb{Z}$).

A *feasible solution* to a model optimization problem $MO(\mathcal{T}, I_{in}, o)$ is a Σ_{out}-structure I_{out} (sharing I_{in}'s domain) such that $I_{in} \sqcup I_{out} \models \mathcal{T}$. An *(optimal) solution* to a model optimization problem $MO(\mathcal{T}, I_{in}, o)$ is a feasible solution I_{out} such that $o^{I_{in} \sqcup I_{out}} \leq o^{I_{in} \sqcup I'_{out}}$ for all feasible solutions I'_{out}. In other words, a model optimization problem aims to find an output structure that, when merged with the input structure, forms a model to the theory, and it is optimal with respect to other such output structures.

Without loss of generality, we assume models with a smaller objective function value to be better, so minimal objective values are optimal.

As a running example, we use the Traveling Salesman Problem (TSP), a classical combinatorial optimization problem.

Example 6.2.1 (TSP). Let Σ_{tsp} be the vocabulary consisting of predicate symbols $City/1$, and function symbols $d/2$ (representing distance), $nxt/1$ (representing a cyclic order) and $p/1$ (representing a permutation of cities). We assume $City/1$, $d/2$ and $nxt/1$ are part of the input vocabulary. In particular, $nxt/1$'s interpretation represents a cyclic order over cities, mapping each city to a *next* one.

A valid TSP tour is expressed by the theory $\mathcal{T}_{tsp}$ over Σ_{tsp}:

$$\begin{aligned}&\forall x\ y\colon City(x) \land City(y) \land x \neq y \Rightarrow p(x) \neq p(y)\\&\forall z\colon City(p(z))\end{aligned}$$

The first formula states that $p/1$ maps two different cities to a different object, hence $p/1$ is injective for cities. The second formula states that $p/1$ is a surjective to the set of cities. Together, $\mathcal{T}_{tsp}$ states $p/1$ is a permutation of cities, which is a valid abstraction of a TSP tour.

Next, the total length of the TSP tour is represented by the term o_{tsp} over Σ_{tsp}:

$$o_{tsp} = \sum_{\{x|City(x)\}} d(p(x), p(nxt(x)))$$

Note that we assume that $d/2$ maps pairs of cities to some numeric domain (e.g., $\mathbb{N}$), and that o_{tsp} can make use of some builtin summation function.[1]

Let $\Sigma_{tspin} = \Sigma_{tsp} \setminus \{p/1\}$. Input data containing cities, a cyclic order, and a distance matrix are expressed as a Σ_{tspin}-structure I_{tspin} with domain $D = \{a, b, c, 0, 1, 2, 3\}$, and interpretations

$$\begin{aligned}
&City^{I_{tspin}} = \{a, b, c\}\\
&nxt^{I_{tspin}} = a \mapsto b, b \mapsto c, c \mapsto a, \ldots \mapsto 0 \text{ otherwise}\\
&d^{I_{tspin}} = (a, b) \mapsto 3, (b, a) \mapsto 2, (b, c) \mapsto 2, (c, b) \mapsto 1, (a, c) \mapsto 3,\\
&(c, a) \mapsto 1, \ldots \mapsto 0 \text{ otherwise.}
\end{aligned}$$

Model optimization problem $MO(\mathcal{T}_{tsp}, I_{tspin}, o_{tsp})$ then represents a simple TSP instance, with as optimal solution the output structure I_{tspout}

$$p^{I_{tspout}} = a \mapsto a, b \mapsto c, c \mapsto b, \ldots \mapsto a \text{ otherwise.}$$

representing the optimal tour $a \to c \to b \to a$. Note that $I_{tspin} \sqcup I_{tspout} \models \mathcal{T}_{tsp}$. Also, the objective function value $o_{tsp}^{I_{tspin} \sqcup I_{tspout}}$ for this tour is $3 + 1 + 2 = 6$. ▲

6.2.2 Symmetry for model optimization

A *symmetry* for a model optimization problem is a permutation of the output structures that preserves both satisfaction to the theory, as well as the value of the objective function (for feasible solutions):

Definition 6.2.2 (Symmetry for MO). Let $MO(\mathcal{T}, I_{in}, o)$ be a model optimization problem with output vocabulary Σ_{out}, and let $\Gamma_D^{\Sigma_{out}}$ be the set of Σ_{out}-structures with domain D. A structure transformation $\sigma\colon \Gamma_D^{\Sigma_{out}} \to \Gamma_D^{\Sigma_{out}}$ is a symmetry of $MO(\mathcal{T}, I_{in}, o)$ if for each $I_{out} \in \Gamma_D^{\Sigma_{out}}$: (i) $I_{in} \sqcup I_{out} \models \mathcal{T}$ iff $I_{in} \sqcup \sigma(I_{out}) \models \mathcal{T}$, and (ii) if $I_{in} \sqcup I_{out} \models \mathcal{T}$ then $o^{I_{in} \sqcup I_{out}} = o^{I_{in} \sqcup \sigma(I_{out})}$.

A permutation of the output structures that only preserves satisfaction to the theory, but potentially not the value of the objective function is called a *quasisymmetry*:

Definition 6.2.3 (quasisymmetry for MO). Let $MO(\mathcal{T}, I_{in}, o)$ be a model optimization problem with output vocabulary Σ_{out}, and let $\Gamma_D^{\Sigma_{out}}$ be the set of

[1]Strictly speaking, this requires extensions to the first-order logic formalism we employ in this thesis. The input language of IDP supports such extensions. For brevity and clarity's sake, we make abstraction of this issue in this text, so we will not provide a formal description of these extensions.

Σ_{out}-structures with domain D. A structure transformation $\sigma\colon \Gamma_D^{\Sigma_{out}} \to \Gamma_D^{\Sigma_{out}}$ is a *quasisymmetry* of $MO(\mathcal{T}, I_{in}, o)$ if for each $I_{out} \in \Gamma_D^{\Sigma_{out}}$, $I_{in} \sqcup I_{out} \models \mathcal{T}$ iff $I_{in} \sqcup \sigma(I_{out}) \models \mathcal{T}$.

So each symmetry for a model optimization problem is also a quasisymmetry, and each symmetry for a model expansion problem $MX(\mathcal{T}, I_{in})$ is a quasisymmetry for the extended $MO(\mathcal{T}, I_{in}, o)$. A quasisymmetry that is not a symmetry is called a *strict quasisymmetry.*

It is easy to extend the notions of local domain symmetry and interchangeable subdomains presented in Section 5.3 to the model optimization context by requiring these types of symmetry to also preserve the objective function (for feasible solutions). In the case they might not, we can still refer to them as *local domain quasisymmetries* and *quasi-interchangeable subdomains.* Similarly, a quasisymmetry group $\mathbb{G}$ partitions the set of feasible solutions into *quasisymmetry classes* such that two feasible solutions I_{out} and I'_{out} belong to the same quasisymmetry class if there exists a quasisymmetry $\sigma \in \mathbb{G}$ such that $\sigma(I_{out}) = I'_{out}$.

Example 6.2.4 (Chromatic number problem)**.** The chromatic number problem (CNP) is the optimization generalization of the graph coloring problem, aiming to find the lowest amount of colors with which a graph can be colored so that no two adjacent vertices have the same color. As was the case for the graph coloring problem (see Example 5.3.18), all colors form an interchangeable subdomain – preserving the objective function. ▲

Example 6.2.5 (Example 6.2.1 continued)**.** $\delta = \{a, b, c\}$ forms a quasi-interchangeable subdomain for both argument position set $A = \{City|1, p|1\}$ as well as $B = \{City|1, p|0\}$. This represent a form of interchangeability of the cities in a tour, though it does not preserve the length of a tour. ▲

As with model expansion problems, it is useful to break symmetry for an optimization problem when trying to solve it with a complete solver. However, in the context of local search – the rest of this chapter – we are particularly interested in quasisymmetries.

6.3 Local search and neighborhoods

6.3.1 Local search

Local search algorithms use the concept of a *neighborhood* to perform a heuristic walk through the *feasible solution space* of a combinatorial optimization problem.

We formalize this notion in the context of model optimization.

Firstly, the *feasible solution space* of a model optimization problem is its set of feasible solutions:

Definition 6.3.1 (Solution space)**.** The *feasible solution space Sol_{MO}* of a model optimization problem $MO(\mathcal{T}, I_{in}, o)$ is the set of output structures $I_{out} \in \Gamma_D^{\Sigma_{out}}$ such that $I_{in} \sqcup I_{out} \models \mathcal{T}$.

Then, we formalize a neighborhood as a mapping from the feasible solution space to the powerset of the feasible solution space:

Definition 6.3.2 (Neighborhood)**.** A *neighborhood N* for a model optimization problem $MO(\mathcal{T}, I_{in}, o)$ is a mapping $N\colon Sol_{MO} \to \mathcal{P}(Sol_{MO})$. $N(I_{out})$ is referred to as the set of *neighbors* under N of a feasible solution $I_{out} \in Sol_{MO}$.

Local search approaches such as those based on *simulated annealing* or *tabu search* require as input a neighborhood N and some initial feasible solution I_{out}. Given these, a typical local search algorithm explores the feasible solution space by enumerating the neighbors of I_{out} under N. When some neighbor $I'_{out} \in N(I_{out})$ satisfies an acceptance criterion (typically based on the objective function evaluation in I'_{out}), it is accepted and becomes the starting point of a new iteration of the algorithm. Search continues by exploring the neighbors of I'_{out}, until a new neighbor is accepted, leading to another iteration, and so on. This loop ends when some stop criterion is met, and the feasible solution with the best objective value encountered during the search is returned.

In a sense, local search *moves* from I_{out} to $I'_{out} \in N(I_{out})$, with the execution of one iteration of the algorithm being called a *move*.

The above notion of local search is in a way restrictive, since it does not capture optimization approaches such as *genetic programming* or *swarm-based optimization*. Also, sometimes a local search algorithm is allowed to ignore some hard constraints, in which case a neighborhood to a feasible solution may contain infeasible solutions. We encounter some examples in Section 6.4.6 and 6.4.7. Nonetheless, many implementations of metaheuristic methods such as tabu search, simulated annealing, variable neighborhood search or greedy optimization, can be characterized by moving from feasible solution to feasible solution using the above notion of neighborhoods.

Example 6.3.3 (Example 6.2.1 continued)**.** For the TSP problem, a typical neighborhood is the so-called 2-*opt* neighborhood [64]. This neighborhood maps each TSP tour to a set of new tours by removing a pair of edges, say between cities c_1, c_2 and c_3, c_4, and reconnecting the resulting two TSP subpaths by introducing an edge between c_1, c_4 and an edge between c_2, c_3. ▲

6.3.2 Symmetries induce a neighborhood

Note that the 2-opt neighborhood of Example 6.3.3 is based on a particular set of permutations of cities in the TSP tour. It is no coincidence that these permutations also induce quasisymmetries for TSP. We formalize this connection:

Definition 6.3.4. Given a set of quasisymmetries $\mathcal{P}$ for a model optimization problem $MO(\mathcal{T}, I_{in}, o)$, the *quasisymmetry-induced neighborhood* $N_{\mathcal{P}}$ maps each feasible solution $I_{out} \in Sol_{MO}$ to its image under $\mathcal{P}$. More formally, $N_{\mathcal{P}}\colon I_{out} \mapsto \{\sigma(I_{out}) \mid \sigma \in \mathcal{P}\}$.

By the definition of quasisymmetry (Definition 6.2.3), any feasible solution has only feasible solutions as quasisymmetry-induced neighbors, which ensures Definition 6.3.4 for a quasisymmetry-induced neighborhood is a sound neighborhood definition.

Note however that Definition 6.3.4 requires some *set* $\mathcal{P}$ of quasisymmetries as input. Since $\mathcal{P}$ generates a group $\mathbb{G}_{\mathcal{P}}$ under composition, the number of possible quasisymmetry sets to form neighborhoods with often is astronomical. In general, we have no definitive answer on what sets of quasisymmetries one should use to construct neighborhoods, but it seems plausible to use a small set $\mathcal{P}$ that generates the detected quasisymmetry group $\mathbb{G}_{\mathcal{P}}$. This way, each move possible under the maximal neighborhood $N_{\mathbb{G}_{\mathcal{P}}}$ can be simulated by a series of moves under $N_{\mathcal{P}}$, while $N_{\mathcal{P}}$ still maps a feasible solution to a relatively small set of neighbors.

Example 6.3.5 (Example 6.2.5 continued)**.** Recall the presence of the subdomain interchangeability group $\mathbb{G}^{A}_{\{a,b,c\}}$ with $A = \{perm|1\}$ or $A = \{perm|0\}$ in the TSP model optimization specification. A set of generators for this group is $\mathcal{P} = \{\sigma^{A}_{(a\ b)}, \sigma^{A}_{(a\ c)}, \sigma^{A}_{(b\ c)}\}$. The quasisymmetry-induced neighborhood $N_{\mathcal{P}}$ maps a feasible solution to the feasible solutions obtained by all possible city swaps. ▲

Note that a quasisymmetry-induced neighborhood $N_{\mathcal{P}}$ only contains neighbors $N(I_{out})$ that belong to the same quasisymmetry class under $Grp(\mathcal{P})$ as I_{out}. As such, a local search approach employing only the neighborhood $N_{\mathcal{P}}$ is not capable of crossing a quasisymmetry class border and cannot guarantee *connectedness of the local search space.* However, this is a frequent issue in the local search community; many proposed neighborhoods do not guarantee connectedness of the local search space either. [51]

6.3.3 Quasisymmetry-induced neighborhood detection in IDP

Using the notion of a quasisymmetry-induced neighborhood, we can automatically compile a model optimization problem specification to input for a local search algorithm.

However, we put forward that symmetries that preserve the objective function are bad candidates for neighborhood generation [2]. Such symmetries cannot transform a feasible solution into a reasonably different one, for the purpose of traversing the feasible solution space. Of course, sometimes a move in a local search algorithm transforms the current feasible solution into one with the same objective value, but having a neighborhood that *only* leads to such moves seems like a waste of resources.

Instead, we focus on *strict* quasisymmetries; quasisymmetries that are not symmetries of the model optimization problem.

We can detect local domain quasisymmetries by simply ignoring the objective function, and check afterwards whether they are strict quasisymmetries by taking the objective function into account.[3] As such, IDP's symmetry detection techniques presented in Chapter 5 can be reused for the detection of quasisymmetry, by ignoring the objective function, or strict quasisymmetry, by checking whether a quasisymmetry preserves the objective function.

This leads to a simple workflow to generate input for a local search algorithm solving a model optimization problem $MO(\mathcal{T}, I_{in}, o)$:

1. Generate an initial feasible solution I_{out}.
2. Detect a quasisymmetry group $\mathbb{G}$ of $MO(\mathcal{T}, I_{in}, o)$ that contains (mainly) strict quasisymmetries.
3. Estimate a set $\mathcal{P}$ generating $\mathbb{G}$.
4. Use $\mathcal{P}$ to construct a quasisymmetry-induced neighborhood $N_{\mathcal{P}}$

The only question that remains is what set of quasisymmetries should be used to induce neighborhoods, as was mentioned at the end of Section 6.3.2. IDP has two (quasi)symmetry detection methods, where one returns a set of generators $\mathcal{P}'$ for some local domain quasisymmetry group (see Section 5.5.1), and the other returns a set of interchangeable subdomains δ over some argument position set

[2]Though such symmetries should be exploited to reduce the search time of a complete solving algorithm.

[3]"taking the objective function into account" requires to specify the notion of connectedness (see Definition 5.3.7) for terms instead of theories. This is straightforward.

A (see Section 5.5.2). The former quasisymmetry group already is conveniently represented by the small set of generators $\mathcal{P}'$. For the latter, we restrict the set of neighborhood-inducing quasisymmetries to the set $\{\sigma^A_{(d\ d')} \mid d, d' \in \delta\}$. In words, we employ all local domain quasisymmetries induced by swaps of two A-interchangeable domain elements. This set contains $O(|\delta|^2)$ quasisymmetries, and generates the subdomain quasi-interchangeability group.

Arguably, it is also possible to construct a set of symmetries $\{\sigma^A_{(d_i\ d_{i+1})} \mid d \in \delta\}$, where d_{i+1} is the subsequent domain element in δ according to some chosen total order on the domain elements. Even though this choice would lead to smaller neighborhoods, it also skews any local search algorithm according to the chosen order, putting a possibly unwarranted bias on the direction of the search over the solution space. For this reason, we stick with the quadratic set of symmetries $\{\sigma^A_{(d\ d')} \mid d, d' \in \delta\}$ to induce a neighborhood for an interchangeable subdomain δ.

6.3.4 A simple local search implementation

Given that systems such as IDP can automatically construct quasisymmetry-induced neighborhoods, we also devised a very simple *hill-climbing* local search algorithm that utilizes these constructed neighborhoods.

In its default complete search configuration, IDP solves a model optimization problem by repeated model expansion calls, iteratively searching for a new feasible solution that improves the best-so-far. If none can be found, the current best-so-far has been proven optimal with regard to the objective function. Also, IDP currently only supports integer domains as objective function codomain.

To add local search on the constructed neighborhoods into the mix, every time an improvement to the best-so-far is found, the system performs a hill-climbing local optimization step. In the ideal case, the hill-climbing algorithm improves the current feasible solution, after which complete search is again requested to find a better feasible solution or to prove that none exists, closing a complete search $\leftrightarrow$ local search loop. Figure 6.1 visualizes this local search workflow.

The local search hill-climbing phase is characterized by some model $I_{in} \sqcup I_{out}$ and a quasisymmetry-induced neighborhood $N_{\mathcal{P}}$. Given the detected quasisymmetry generator set $\mathcal{P}'$ and local domain quasi-interchangeability groups $\mathbb{G}^{A_i}_{\delta_i}$, $\mathcal{P}$ consists of the quasisymmetry generators $\bigcup_i \{\sigma^{A_i}_{(d\ d')} \mid d, d' \in \delta_i\} \cup \mathcal{P}'$.

When selecting a neighbor from $N_{\mathcal{P}}$, the quasisymmetry generators $\sigma \in \mathcal{P}$ are enumerated in a pseudorandom order, and the first σ for which the objective function improves – $o^{I_{in} \sqcup \sigma(I_{out})} < o^{I_{in} \sqcup I_{out}}$ – leads to the selected neighbor

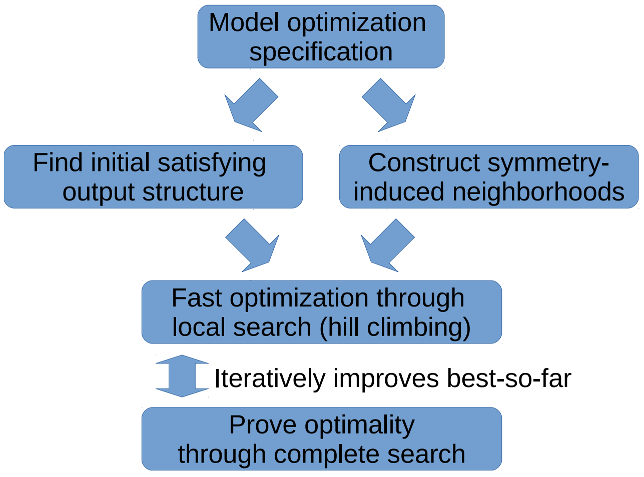
Model optimization specification
Find initial satisfying output structure
Construct symmetry-induced neighborhoods
Fast optimization through local search (hill climbing)
Iteratively improves best-so-far
Prove optimality through complete search

Figure 6.1: Experimental local search workflow in a system such as IDP

$I_{in} \sqcup \sigma(I_{out})$. When no such generator σ exists, the hill-climbing phase ends, and control is passed to the complete search phase (see Figure 6.1).

6.4 Experiments

We implemented the neighborhood detection scheme described in the previous section in IDP, as well as the hill climbing local search workflow.

In this section we experimentally investigate this implementation for seven combinatorial optimization problems. These are typical *operations research* problems, often not easily solved by an artificial intelligence system such as IDP.

We focus mainly on a qualitative evaluation of the type of neighborhoods automatically constructed, comparing them to local search neighborhoods proposed in literature. However, we also give quantitative results comparing IDP's standard complete search with the hill climbing workflow, referred to as IDP-hill. This serves as an illustration of the feasibility of combining local and complete search in an automated way.

This quantitative comparison was done by running both IDP's standard complete search and IDP's hill climbing workflow for 200 seconds. Once this timeout was reached, the values of the objective function were compared, with a lower value denoting a more optimal solution. The available machine had 8GB RAM and an Intel® Xeon® E3-1225 CPU with Ubuntu 14.04 Linux kernel 3.13 as operating system. All timing results include grounding and symmetry detection time expended by IDP. The experiments were run with an early version of IDP's symmetry detection algorithm, which could only detect subdomain quasi-interchangeability. Resources to reproduce these experiments are available online [31].

6.4.1 Traveling Salesman Problem

Let us first investigate the TSP problem, since this was our running example.

As mentioned in Example 6.3.5, two quasi-interchangeable subdomains represent interchangeability of the cities; one with argument position set $\{City|1, p|1\}$ and one with argument position set $\{City|1, p|0\}$. Though both induce valid neighborhoods, both induce an identical neighborhood.[4] This is a drawback of our approach.

However, there exist other reasonable FO($\cdot$) specifications of TSP other than the one in Example 6.2.1.

Example 6.4.1. Let Σ_{tsp} be the vocabulary consisting of predicate symbols $City/1$, $Following/2$ and $Reachable/1$, function symbol $d/2$, and constant symbol $start/0$.

A valid TSP tour is expressed by the following theory over Σ_{tsp}:

$$\begin{aligned}
&\forall x\ y\colon Following(x,y) \Rightarrow City(x) \land City(y)\\
&\forall x\colon City(x) \Rightarrow \exists 1 y\colon Following(x,y) \land \exists 1 z\colon Following(z,x)\\
&\{\ \forall x\colon Reachable(x) \leftarrow\\
&\quad City(x) \land (x = start \lor (\exists y\colon Reachable(y) \land Following(y,x))).\ \}\\
&\forall x\colon City(x) \Rightarrow Reachable(x)\\
&City(start)
\end{aligned}$$

The formula between curly brackets is an *inductive definition* [29], constraining the *Reachable* predicate to only be true for cities reachable from a city *start* using the *Following* relation. By next stating that all cities must be reachable, we effectively posted a subtour elimination constraint.

[4]The reason for this is that they employ the same city swaps $\{(a\ b), (b\ c), (c\ a)\}$ on the input or output of p. As p represents a bijection on the cities, they alter p in a similar way, hence, the identical neighborhood.

The objective function o_{tsp}:

$$\sum_{\{x,y \ | \ Following(x,y)\}} d(x,y)$$

The output vocabulary then is $\Sigma_{tspout} = \{Following/2, Reachable/1, start/0\}$.

In this case, the input interpretation of $City/1$ will form an interchangeable subdomain for argument position set $\{Reachable|1, Following|1, Following|2, start|0\}$. This again represents the interchangeability of cities, but in this case there is only one quasi-interchangeable subdomain doing so instead of two. ▲

In this alternative TSP specification, IDP-hill indeed constructs a city-swapping neighborhood-based on the quasi-interchangeable cities.

This alternative specification experiment shows that our proposed neighborhood construction method exhibits robustness: different specifications of the same problem still lead to comparable neighborhoods. This of course only holds as long as symmetry detection[5] properties of different specifications are similar.

In literature, many local search neighborhoods have been proposed for TSP, the most famous one being the *k-opt* neighborhood [64]. We briefly touched the 2-*opt* neighborhood in Example 6.3.3, where a path represented by a selection of edges has as neighbors those paths containing exactly two different edges. The more general k-opt neighborhood allows the reconfiguration of any k edges instead of only two.

In both our specifications the quasisymmetry-induced neighborhood was not a k-opt neighborhood, meaning our automated neighborhood detection algorithm is not yet powerful enough to derive a state-of-the-art neighborhood.[6]

The quantitative evaluation of the hill climber workflow on TSP is presented in Figure 6.2. It is made with a model optimization specification akin to the one given in Example 6.4.1. The results are positive: IDP-hill beats standard IDP hands-down. As much as this is an encouraging result, it is also a testament to how unsatisfactorily IDP solves TSP in its standard configuration. As such, it would be interesting future work to compare the performance of a k-opt local search neighborhood for TSP to our automatically generated one.

[5] As symmetry is a semantic notion, it is an intrinsic property of the problem at hand, not its specification. However, *detecting* symmetry from a problem specification does depend on the particular specification at hand.

[6] "state-of-the-art" is used in a broad sense, as the k-opt neighborhoods have been improved upon.

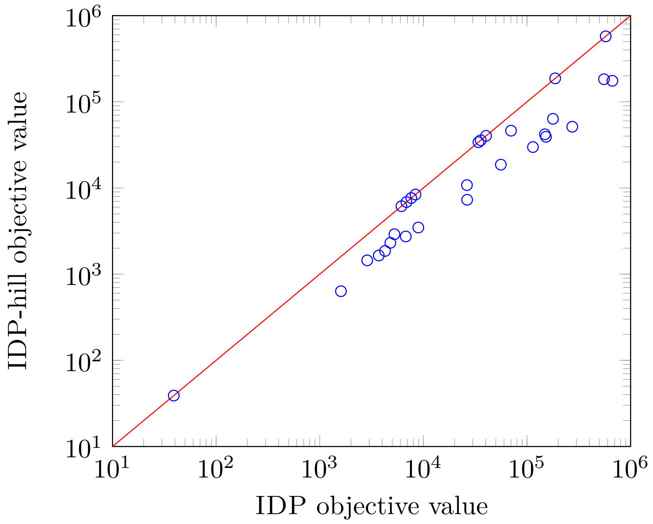

Figure 6.2: Objective value for TSP obtained by IDP's automated hill climbing implementation relative to standard IDP. Lower is better.

6.4.2 Traveling Tournament Problem

The Traveling Tournament Problem (TTP) is a combinatorial optimization problem that combines features from the traveling salesman problem and the tournament scheduling problem. The task at hand is to schedule a round-robin tournament where each team has to play against each other team twice, once at their home location, and once at the other's. A round in the tournament consists of all teams playing against an opponent at the same day. The objective function measures the total sum of all distances traveled by all teams, assuming each team moves from playing location to playing location in between rounds. Some extra constraints require that teams do not stay at home or away from home for too long, and the same teams cannot play each other in subsequent rounds of the tournament.

As with the TSP, several well-performing local search neighborhoods have been proposed [46, 7]. One neighborhood swaps the opponents of two teams for a given round, another swaps the opponents of a team in two given rounds. Both these moves require some repair mechanism called a *recover chain* to ensure the hard constraints of the TTP remain satisfied after performing the swaps.

IDP-hill constructs only one quasisymmetry-induced neighborhood, based on the quasi-interchangeability of teams. Again, this simple team-swapping

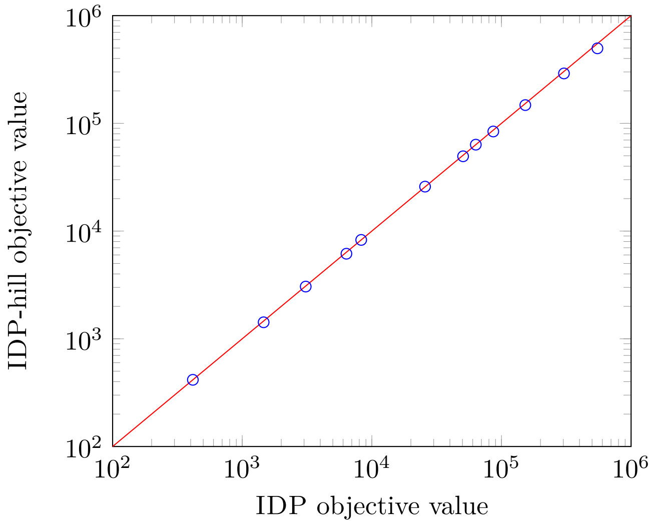

Figure 6.3: Objective value for TTP obtained by IDP's automated hill climbing implementation relative to standard IDP. Lower is better.

neighborhood does not correspond to the most important neighborhoods for TTP proposed in literature, and might perform worse.

A performance comparison of IDP-hill with standard IDP is given in Figure 6.3. Though it is not clear from Figure 6.3, the number of instances for which IDP-hill performs strictly better than IDP is eight, and the number of instances for which IDP-hill performs strictly worse than IDP is one.

The observed behavior of the combined search method was that IDP-hill's complete search had a hard time detecting an (objective improving) model, after which IDP-hill's local search slightly optimized the objective value, quickly passing control back to the complete search component of IDP-hill. This small improvement can be explained either by the greedy nature of our hill-climbing local search phase, or by the possibility that IDP-hill's detected neighborhood is not powerful enough. It is worth noting that relaxing the extra constraints restricting which teams play against each other each round allows IDP-hill to derive the quasi-interchangeability of the rounds themselves, leading to an additional quasisymmetry-induced neighborhood. A similar observation was made in Section 6.4.2.

6.4.3 Knapsack Problem

The knapsack problem consists of filling up some knapsack with objects, such that the volume of the objects fits the knapsack, but the weight of the objects in the knapsack is minimized.

A neighborhood-based local search approach is not often employed to tackle the classical knapsack problem.

Also, since swapping any two objects in and out of the knapsack might violate the volume constraint, there is no a priori local domain (quasi)symmetry present in the knapsack problem. Hence, we expect IDP-hill's neighborhood detection algorithm to return empty handed.

However, IDP's quasisymmetry detection is sufficiently fine-grained to identify that for some instances, some objects had the same volume but a different weight. These objects could safely be swapped in and out of the knapsack, leading to an unexpected neighborhood where, for a given knapsack, small variations on the filling of the knapsack could be explored. In practice, this meant that the local search algorithm automatically filled the knapsack with the objects with smallest weight relative to other objects of the same volume.

The quantitative evaluation of the hill climber workflow on the knapsack problem is presented in Figure 6.4. The results give a small edge to the simple hill climbing implementation, since for some instances it was able to quickly prioritize objects with lower weight but equal volume to belong to the knapsack. These results are similar to those for TTP, as the automatically constructed neighborhoods allow for a quick objective function improvement after finding a new model.

6.4.4 Shortest Path Problem

This problem consists of finding the shortest path between a start and end vertex in a weighted graph. Its specification in $\mathrm{FO}(\cdot)$ is very similar to the TSP specification of the previous subsection, and centers on finding a minimal interpretation to some $Following/2$ predicate. The shortest path problem resides in complexity class $\mathbf{P}$. The most widely known algorithm to solve the shortest path problem is Dijkstra's algorithm; typically other approaches than local search are used to tackle this problem.

IDP-hill's neighborhood detection mechanism detects that all two cities except the start and end city are interchangeable. The resulting quasisymmetry group leads to a large induced neighborhood.

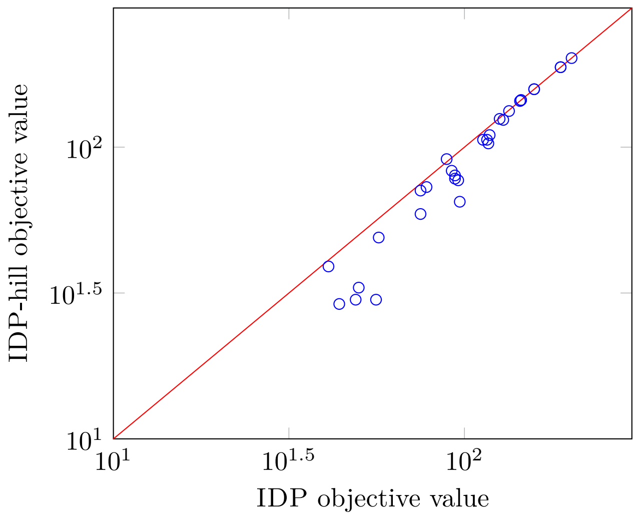

Figure 6.4: Objective value for the knapsack problem obtained by IDP's automated hill climbing implementation relative to standard IDP. Lower is better.

The quantitative evaluation of the hill climber workflow on the shortest path problem is presented in Figure 6.5. The results are similar to those for TSP: the simple hill climbing implementation beats standard IDP hands-down.

6.4.5 Assignment Problem

For the assignment problem, a bijection between a set of agents and a set of tasks must be found that minimizes the sum of the costs of assigning a certain task to a certain agent. It is a specialization of the *max-flow* problem, and is typically solved by the Hungarian algorithm or the simplex algorithm [61]. The assignment problem resides in complexity class **P**.

IDP detects that both the set of agents as well as tasks form two quasi-interchangeable subdomains, but not regular interchangeable subdomains, leading to two quasisymmetry-induced neighborhoods.

The quantitative evaluation of the hill climber workflow on the assignment problem is presented in Figure 6.6. As with the shortest path problem, we observe that IDP-hill is able to exploit the assignment problem's tractability significantly better than IDP's complete search.

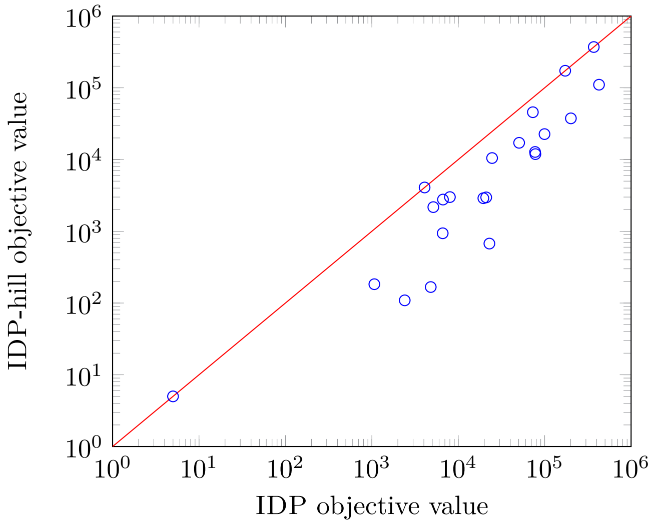

Figure 6.5: Objective value for the shortest path problem obtained by IDP's automated hill climbing implementation relative to standard IDP. Lower is better.

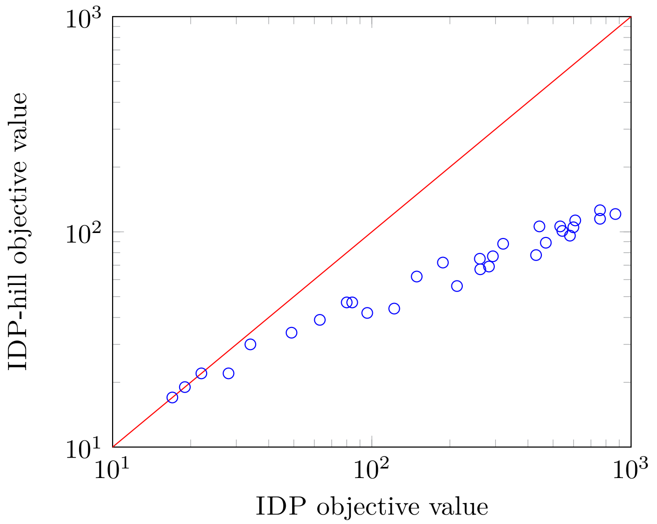

Figure 6.6: Objective value for the assignment problem obtained by IDP's automated hill climbing implementation relative to standard IDP. Lower is better.

6.4.6 Max Clique

The task of a max clique problem is to identify the largest clique in a graph. We modeled this problem in such a way that each feasible solution represents a set of vertices forming a clique in the input graph, while the objective function counts the number of vertices not part of the clique.[7]

In literature, an effective local search neighborhood is based on sequences of vertex addition and removal from a candidate clique [59].

However, in a model optimization context, the only domain elements in this problem are the vertices of the graph, and for typical graphs, vertices are not interchangeable. And even if the graph itself contains some symmetry properties, these will preserve the objective function, since a simple permutation of vertices has no influence on the size of any detected clique. Hence, IDP-hill does not construct a quasisymmetry-induced local search neighborhood for this problem. This makes sense, since there is no obvious way of transforming a clique in the graph in one move to some other clique in the graph.

This can potentially be adressed by relaxing the clique constraint, making any set of vertices a feasible solution, regardless of whether it forms a clique in the input graph. The lesson we take from this problem is that our neighborhood detection scheme would be aided by a form of constraint relaxation to allow for a larger set of potential neighborhoods.

As no neighborhoods are constructed, a quantitative evaluation of the hill climbing workflow would be rather useless: both approaches would simply rely on IDP's complete search mechanism.

6.4.7 Chromatic Number Problem

The chromatic number problem aims at establishing a graph's *chromatic number*, being the smallest number of colors needed to color the vertices of the graph such that all pairs of connected vertices have a different color.

In literature, a plethora of local search neighborhoods for graph coloring have been proposed [44]. A recurring theme is to move to alternative candidate solutions by recoloring (chains of) individual vertices.

As mentioned in Example 6.2.4, for the chromatic number problem, all colors form an interchangeable subdomain. As a result, this quasisymmetry group on its own is useless to induce local search neighborhoods with, as it preserves

[7]Hence minimizing this number maximizes the clique size.

the objective function – the number of colors needed to color the graph. IDP-hill detects this information, and constructs no quasisymmetry-induced neighborhood from color interchangeability.

However, as adjusting colors in a colored graph in general does not preserve the graph coloring constraints, IDP-hill detects no other quasisymmetry to induce neighborhoods with. To detect the neighborhoods proposed by literature, we would again need to relax some of the problem constraints.

As IDP-hill does not construct any neighborhoods, we again skip its quantitative evaluation.

6.4.8 Experimental conclusions

Testing the quasisymmetry-based automated neighborhood detection algorithm of IDP on the above problems yielded interesting insights. Firstly, the approach is robust in changes to the specification as long as the quasisymmetry properties are not disturbed. Secondly, IDP-hill, even in its current simple form, is able to improve IDP's performance on several problems. Thirdly, the automated local search implementation was able to construct a neighborhood where a human would not have expected one. This was due to unexpected symmetry properties of input data in problem instances. And finally, to detect more quasisymmetry-induced neighborhoods, it might be needed to relax certain constraints.

6.5 Discussion

6.5.1 Related Work

In the previous sections we described how a model optimization system such as IDP can exploit symmetry properties of combinatorial optimization problems to derive local search neighborhoods for those problems.

The constraint programming system COMET allows a user to easily specify neighborhood-based local search algorithms in a constraint-centered way [75, 98]. However, COMET does not provide any automatic neighborhood detection algorithm.

Stochastic SAT solvers such as WALKSAT [88] allow a propositional logic problem to be solved by local search. In WALKSAT, every assignment to the Boolean variables is a feasible solution, and neighborhoods are defined in terms of "flips" on the truth value of a variable. In our view, this is an extreme approach, since all original constraints might be violated by any move. Note that in a context where all constraints are relaxed, any permutation on the feasible solution space is a quasisymmetry suitable for quasisymmetry-induced neighborhoods. However, these neighborhoods are not the kind a human programmer would devise for most combinatorial optimization problems.

To our knowledge, the only system that allows automatic derivation of local search neighborhoods from an input specification is LocalSolver [17]. LocalSolver allows a user to write down constraints in a mathematical modeling language centered on Boolean variables, and uses flips on those variables as well as satisfiability preserving moves based on "ejection chains applied to the hypergraph induced by boolean variables and constraints". We conjecture that these feasibility preserving moves can be seen as symmetries of the propositional problem, but further study is needed on this topic.

One weakness of LocalSolver is that the initial feasible solution is found by a basic randomized greedy algorithm, and that it is not designed for solving hardly-constrainted optimization problems. This is opposed to the local search approach described in this chapter, which can always fall back on the solving capabilities of a state-of-the-art complete search engine.

To put our work in a broader perspective, it is worth mentioning that much research has been done on the relationship between symmetry *breaking* and local search (e.g. [82]). An important result is that adding symmetry breaking constraints has a negative impact on the efficiency of local search algorithms. In our work, we do not break any symmetry, but exploit quasisymmetries to traverse the local search space. Moreover, breaking quasisymmetry this would risk removing an optimal solution from the search space. In this light, symmetry breaking for optimization problems and exploiting quasisymmetry properties to construct local search neighborhoods are two orthogonal uses of (quasi)symmetry.

Neighborhood inducing quasisymmetries share the fact that they do not preserve objective function values with *dominance relations* used in *dominance breaking*. Dominance breaking is a generalization of symmetry breaking for complete search algorithms where extra constraints remove *dominated* solutions with a worse objective value from the search space [22]. These dominance relations seem to also induce neighborhoods in the same way quasisymmetries do, and they can be detected automatically [72]. Currently, we do not know if dominance relations are fundamentally different to quasisymmetries.

Finally, in recent years the metaheuristic community is actively investigating how to formalize and automate local search algorithms [91]. Our work can be seen as an effort in that direction, opening up unexplored research avenues by linking local search neighborhoods to quasisymmetry properties of problems.

6.5.2 Conclusion and future work

In this chapter, we propose a link between local search neighborhoods and (quasi)symmetry. To our knowledge, this is the first time such a link is established. We used this link to design an automated local search workflow for model optimization problems.

We conducted an experimental investigation of the type of quasisymmetry-induced neighborhoods detected by our implementation in IDP, as well as an evaluation of the performance of this hybrid technique. These first experiments are promising, and show the potential of this approach.

As future work, it remains to be seen how well quasisymmetry-induced neighborhood search algorithms perform on larger, more complex problems, with potentially more constraints breaking any quasisymmetry. Allowing the relaxation of certain constraints might prove crucial in detecting sufficiently large neighborhoods.

Second, to gauge the performance of quasisymmetry-induced neighborhoods, it would be useful to run experiments comparing these to state-of-the-art neighborhoods proposed in literature.

Third, it would be interesting to couple the detected neighborhoods and input feasible solutions to a highly optimized local search engine, and experimentally verify their performance. The challenge here lies in being able to quickly move from one feasible solution to another, and to incrementally update the value of an objective function. The IDP system is currently unoptimized in this regard.

Fourth, for some problems, there exist complex neighborhoods involving smart perturbation and repair steps. Whether or not these can be linked to an easily detectable form of quasisymmetry remains unstudied.

Lastly, the link with techniques for dominance breaking is definitely worth investigating.

Chapter goal evaluation
The link between local search and symmetries proved useful: a new avenue for automating local search has opened up. Initial experiments are promising, though it remains to be seen whether all important neighborhood types can be conveniently linked to some type of symmetry.

Chapter 7

Conclusion

In this thesis, we set out to explore symmetry properties in model expansion problems for both predicate and propositional logic. A crucial motivation was that systems solving model expansion problems should not be hindered by symmetry, so that a wider range of problems can be solved with less human input. This required the study of both symmetry detection and symmetry exploitation, and forced us to develop integrated techniques only relying on input specifications containing no specific symmetry information.

In Chapter 3, we developed BreakID, a new symmetry breaking preprocessor for propositional logic. Its core idea is to investigate structural properties of the symmetry group of a problem, and to adjust any generated symmetry breaking formulas accordingly. Besides this, we also added usability features and technical optimizations, allowing symmetry detection and subsequent breaking for a broader range of propositional formulas, both in a SAT and an ASP context. Experimental results show that BreakID improves on Shatter and sbass, the previous state-of-the-art preprocessors for static symmetry breaking for propositional logic.

Though effective and easy-to-use, the static symmetry breaking approaches from Chapter 3 and 5 have the disadvantage that they fundamentally alter the input specification. Instead, Chapter 4 investigates two new *dynamic symmetry handling* algorithms, based on symmetrically deriving logical consequences. The first, SP, focuses on propagating literals symmetrical to already propagated literals. For this, we introduce the notion of *weak activity*, a generalization of the constraint programming notion of *activity*. Experiments show that this approach is effective, but can be improved by also performing symmetrical propagations not based on the weak activity status of a symmetry. This led to the second approach, SEL, which derives symmetrical explanation clauses, and as such, is a form of *symmetrical learning.* The experimental performance of a first implementation of SEL is on par with BreakID, making it the first symmetrical learning scheme to be a viable alternative to static symmetry breaking.

Working on a propositional level impedes the algorithms detecting symmetry and deriving symmetry group information, as this is more readily available at the predicate level. For this, we studied how symmetry manifests itself in first-order logic in Chapter 5. We proposed the notion of *local domain symmetry*, a form of symmetry in predicate logic theories that captures a broad range of symmetry groups occurring in practical problems. Based on theoretical properties of local domain symmetry, we designed efficient ways of both detecting and breaking it. Our implementation in IDP outperforms both BreakID and sbass in symmetry detection time, and in symmetry breaking power for problems with large row interchangeability groups. Besides, our approach is one of the few automated approaches that detects symmetry at the predicate level. The modest price we paid was that some forms of symmetry are not captured by the notion of local domain symmetry, though future generalizations of local domain symmetry might eliminate this drawback.

As our main interest lies in improving model expansion systems to require less human input, Chapter 6 links a local search solving algorithm with an input specification containing no information on neighborhoods – a crucial concept for local search. The central idea is that a *quasisymmetry*, a symmetry of the "hard" constraints of an optimization problem, but not of its "soft" objective function leads to a sensible neighborhood for a local search algorithm solving the optimization problem. These quasisymmetries can be automatically detected with techniques from Chapter 5, leading to a novel automated local search approach. Our experiments indicate that this approach is feasible, though we doubt it can already beat a human programmer devising a local search algorithm.

Together, these four chapters provide new insights in how symmetry can be detected and exploited without human interaction. These ideas, rooted in propositional and predicate logic, are useful for anyone designing automated combinatorial problem solving systems, be they affiliated with constraint programming, operations research, Boolean satisfiability solving, answer set programming or related fields.

Appendix A

Appendix

A.1 Proof for Theorem 5.3.9

Let Σ be a vocabulary, $\mathcal{T}$ a theory over Σ, π a domain permutation and A a set of argument positions. If A is connectively closed under $\mathcal{T}$, then σ_π^A is a local domain symmetry of $\mathcal{T}$.

Proof. To prove this theorem, we prove the following consecutive claims for each Σ-structure I. Without loss of generalization, we assume I interprets the neccessary variables.

1. For each term $f(\bar{t})$ that occurs in $\mathcal{T}$, it holds that $f(\bar{t})^{\sigma_\pi^A(I)} = \begin{cases} \pi(f(\bar{t})^I) & \text{if } f|0 \in A \\ f(\bar{t})^I & \text{otherwise} \end{cases}$

2. For each atom a of the form $P(t_1, \ldots, t_n)$ or of the form $t_1 = t_2$ that occurs in $\mathcal{T}$, it holds that $a^{\sigma_\pi^A(I)} = a^I$.

3. For each formula φ that occurs in $\mathcal{T}$, $\varphi^{\sigma_\pi^A(I)} = \varphi^I$.

The first claim is proven by induction on the subterm relation. The induction step follows from the fact that A is connectively closed. The second claim follows from the first, also using the fact that A is connectively closed. Consider for instance the case of atom $f(\bar{t}) = g(\bar{t}')$ occurring in $\mathcal{T}$, with $f|0 \in A$. Then $g|0 \in A$ (since A is connectively closed), so $f(\bar{t})^{\sigma_\pi^A(I)} = g(\bar{t}')^{\sigma_\pi^A(I)}$ iff

$\pi(f(\bar{t})^I) = \pi(g(\bar{t}')^I)$ iff $f(\bar{t})^I = g(\bar{t}')^I$ (since π is a permutation). The other cases are analogous.

The last claim follows by induction on the subformula relation since the value of a first-order formula is entirely determined by the value of the atoms occuring in it. Consider for instance the case of formula $\exists x \colon \varphi$ occurring in $\mathcal{T}$ with $x|0 \in A$. $(\exists x \colon \varphi)^I$ holds iff there exists a $d \in D$ such that $\varphi^{I[x:d]}$ holds. By the induction hypothesis, $\varphi^{I[x:d]} = \varphi^{\sigma_\pi^A(I[x:d])} = \varphi^{\sigma_\pi^A(I)[x:\pi(d)]}$ (since $x|0 \in A$). $(\exists x \colon \varphi)^{\sigma_\pi^A(I)}$ holds iff there exists a $d' \in D$ such that $\varphi^{\sigma_\pi^A(I)[x:d']}$ holds. Without loss of generalization, let $d' = \pi(d)$, then $(\exists x \colon \varphi)^I = (\exists x \colon \varphi)^{\sigma_\pi^A(I)}$. The other cases are analogous. □

A.2 Proof for Theorem 5.3.15

Let $MX(\mathcal{T}, I_{in})$ be a model expansion problem with decomposition $MX(\mathcal{T}^*, I_{in}^*)$. If A is connectively closed under $\mathcal{T}^*$ and $\sigma_\pi^A(I_{in}^*) = I_{in}^*$ then σ_π^A is a symmetry for $MX(\mathcal{T}, I_{in})$.

Proof. Since $MX(\mathcal{T}, I_{in})$ and $MX(\mathcal{T}^*, I_{in}^*)$ have the same set of solutions, it suffices to prove that σ_π^A is a symmetry of $MX(\mathcal{T}^*, I_{in}^*)$. Firstly, due to the connectively closed condition, σ_π^A is a symmetry of $\mathcal{T}^*$, so $I_{in}^* \sqcup I_{out} \models \mathcal{T}^*$ iff $\sigma_\pi^A(I_{in}^* \sqcup I_{out}) \models \mathcal{T}^*$. Secondly, since $\sigma_\pi^A(I_{in}^*) = I_{in}^*$, $I_{in}^* \sqcup I_{out} \models \mathcal{T}^*$ iff $I_{in}^* \sqcup \sigma_\pi^A(I_{out}) \models \mathcal{T}^*$, so σ_π^A is a symmetry for $MX(\mathcal{T}^*, I_{in}^*)$. □

A.3 Proof for Theorem 5.4.2

Let $MX(\mathcal{T}, I_{in})$ be a model expansion problem, δ an A-interchangeable subdomain, $\preceq_D$ a total order on domain D and $s(d)$ the successor of d in δ according to $\preceq_D$. If A contains at most one argument position $S|i$ for each symbol $S \in \Sigma_{out}$, then the conjunction of lex-leader constraints

$$\{LL^{\preceq_D}(\sigma^A_{(d\ s(d))}) \mid d \in \delta\}$$

is a complete symmetry breaking constraint for the subdomain interchangeability group $\mathbb{G}_\delta^A$.

Proof. To prove this theorem, we (1) convert the task of finding a solution to a model expansion problem to a constraint programming problem, where an assignment over a set of Boolean variables V has to be found. Further, we show that (2) a subset of these Boolean variables can be organized as a matrix M_δ, where each permutation over the rows of M_δ corresponds to a permutation over δ. The interchangeability group $\mathbb{G}^A_\delta$ then corresponds to a row interchangeability symmetry group induced by permuting M_δ's rows. Using a result from constraint programming, such row interchangeability symmetry groups are broken completely by posting a lex-leader constraint (based on the appropriate row ordering) for each symmetry induced by the swap of two consecutive rows [43, 36]. This corresponds to posting $\{LL^{\preceq_D}(\sigma^A_{(d\ s(d))}) \mid d \in \delta\}$, ending the proof.

(1) Given a vocabulary Σ_{out} and a domain D, finding a Σ_{out}-structure consists of deciding for each $\bar{d} \in D^n$ whether $\bar{d} \in S^{I_{out}}$ for each symbol $S/n \in \Sigma_{out}$. Hence, a model expansion problem $MX(\mathcal{T}, I_{in})$ can be seen as finding an assignment to a set of Boolean variables $V = \{S(\bar{d}) \mid S/n \in \Sigma_{out}, \bar{d} \in D^n\}$ such that $I_{in} \sqcup I_{out} \models \mathcal{T}$. A local domain symmetry σ^A_π for $MX(\mathcal{T}, I_{in})$ now corresponds to a *variable symmetry* [43] mapping

$$S(d_1, \dots, d_n) \quad \text{to} \quad S(\tau_{S|1}(d_1), \dots, \tau_{S|n}(d_n))$$

where $\tau_{S|i}(d) = \pi(d)$ if $S|i \in A$ and $\tau_{S|i}(d) = d$ otherwise.

(2) The variables in V that are not fixed by some $\sigma^A_\pi \in \mathbb{G}^A_\delta$ are those $S(\dots, d_{j-1}, \delta_i, d_{j+1}, \dots)$ where $\delta_i \in \delta$ is the jth domain element of an S-tuple with $S|j \in A$. We can partition this subset into "rows" $R_{\delta_i} = \{S(\dots, d_{j-1}, \delta_i, d_{j+1}, \dots) \mid S|j \in A, d_k \in D\}$ where δ_i is fixed. It is clear that $\sigma^A_\pi(R_{\delta_i}) = R_{\pi(\delta_i)}$, so a permutation of the set of rows corresponds to a symmetry of $\mathbb{G}^A_\delta$. Since A contains at most one argument position for each $S \in \Sigma_{out}$, these rows are pairwise disjoint, and under some column organization form the requested matrix M_δ. □

A.4 Proof for Theorem 5.5.2

Let I be a Σ-structure with domain D and A a set of argument positions. There exists a bijection between the automorphism group of the domain permutation graph $DPG(I, A)$ and the group of domain permutations π such that $\sigma^A_\pi(I) = I$. This bijection maps an automorphism τ to domain permutation π iff $\tau(d) = \pi(d)$ for all DE vertices (equated with domain elements) d.

Proof. We prove the bijection by showing that all induced structure transformations σ^A_π with $\sigma^A_\pi(I_{in}) = I_{in}$ correspond to an automorphism of $DPG(I_{in}, A)$ ($\Rightarrow$) and vice versa ($\Leftarrow$).

First, some preliminaries. For a given symbol S, let each tuple $(d_1, \ldots, d_n) \in S^I_{in}$ be split as two tuples $d^+_A d^-_A$ such that $d^+_A = \{d_i \mid S|i \in A\}$ and $d^-_A = \{d_i \mid S|i \notin A\}$. Let π naturally extend to tuples: $\pi((d_1, \ldots, d_n)) = (\pi(d_1), \ldots, \pi(d_n))$. The symmetrical interpretation $\sigma^A_\pi(I_{in})$ can then be described as $\{\pi(d^+_A)d^-_A \mid d^+_A d^-_A \in S^I_{in}\}$, so $\sigma^A_\pi(I_{in}) = I_{in}$ iff for all symbols S, $d^+_A d^-_A \in S^I_{in}$ iff $\pi(d^+_A)d^-_A \in S^I_{in}$. Also, without loss of generalization, let an IT vertex's label be $S(d^+_A d^-_A)$ for symbol S. Lastly, $(v, w) \in E$ denotes that graph E has an (undirected) edge between vertices v and w.

($\Rightarrow$) If $\sigma^A_\pi(I_{in}) = I_{in}$, σ^A_π corresponds to a permutation α of the vertices of $DPG(I_{in}, A)$: $\alpha(d) = \pi(d)$ (for DE vertices), $\alpha(d.i) = \pi(d).i$ (for AP vertices), $\alpha(S(d^+_A d^-_A)) = S(\pi(d^+_A)d^-_A)$ (for IT vertices). We show that α is an automorphism of $DPG(I_{in}, A)$.

By the definition of $DPG(I_{in}, A)$, α preserves the colors. To show that α preserves the edges, we need to show that $(v, w) \in DPG(I_{in}, A)$ iff $(v, w) \in \alpha(DPG(I_{in}, A))$. Firstly, remark that α maps each vertex in a layer to another vertex in that layer, so we only need to check whether the edges between (1) DE-AP and (2) AP-IT are conserved.

(1) The following statements are equivalent

$$\begin{aligned}
&(\alpha(d), \alpha(e.i)) \in \alpha(DPG(I_{in}, A)) \\
(d, e.i) &\in DPG(I_{in}, A) \text{ (α is a permutation of vertices)} \\
d &= e \text{ (definition of domain permutation graph)} \\
\pi(d) &= \pi(e) \text{ (π is a permutation)} \\
(\pi(d), \pi(e).i) &\in DPG(I_{in}, A) \text{ (definition of domain permutation graph)} \\
(\alpha(d), \alpha(e.i)) &\in DPG(I_{in}, A) \text{ (definition of α)}
\end{aligned}$$

(2) Similarly, the following statements are equivalent

$$(\alpha(d.i), \alpha(S(d_A^+ d_A^-)) \in \alpha(DPG(I_{in}, A))$$

$$(d.i, S(d_A^+ d_A^-)) \in DPG(I_{in}, A) \text{ (α is a permutation of vertices)}$$

$$d_i \in d_A^+ \text{ (definition of domain permutation graph)}$$

$$\pi(d_i) \in \pi(d_A^+) \text{ (π is a permutation)}$$

$$(\pi(d).i, S(\pi(d_A^+)d_A^-)) \in DPG(I_{in}, A) \text{ (definition of domain permutation graph)}$$

$$(\alpha(d.i), \alpha(S(d_A^+ d_A^-)) \in DPG(I_{in}, A) \text{ (definition of α)}$$

($\Leftarrow$) We must show that an automorphism α of $DPG(I_{in}, A)$ corresponds to an A, π-induced structure transformation σ_π^A such that $\sigma_\pi^A(I_{in}) = I_{in}$.

Notice that, since α is an automorphism of a three-layered graph with different colors for each layer DE, AP and IT, we can write it as a composition of three permutations $\alpha_{DE} \circ \alpha_{AP} \circ \alpha_{IT}$. As there exists a bijection between DE and the domain D of I_{in}, we assume $\alpha_{DE} \backsimeq \pi$, with π a permutation of D.

We now show that (1) $\alpha(d.i) = \pi(d).i$ and (2) $\alpha(S(d_A^+ d_A^-)) = S(\pi(d_A^+)d_A^-)$. From this, it follows that α represents a structure transformation σ_π^A mapping tuples $d_A^+ d_A^-$ to $\pi(d_A^+)d_A^-$, and hence, $\sigma_\pi^A(I_{in}) = I_{in}$.

(1) Since $d.i$ and $e.j$ have the same color iff $i = j$, $\alpha(d.i) = e.i$ for some domain element e. As each vertex $d.i$ is connected to exactly one vertex d, $\alpha(d.i) = \pi(d).i$.

(2) Since $S(d_A^+ d_A^-)$ and $R(e_A^+ e_A^-)$ have the same color iff $S = R$ and $d_A^- = e_A^-$, $\alpha(S(d_A^+ d_A^-)) = S(e_A^+ d_A^-)$ for some tuple domain elements e. All that is left to show is that $e_A^+ = \pi(d_A^+)$. For this, note that $S(d_A^+ d_A^-)$ is connected only to $d.i$ for each d on index i in d_A^+. As α is an automorphism that maps $d.i$ to $\pi(d).i$, $\alpha(S(d_A^+ d_A^-)$ must be connected only to all $\pi(d).i$. The only vertex doing so (taking colors into account) is $S(\pi(d_A^+)d_A^-)$. □

A.5 Proof for Theorem 5.5.5

Let $MX(\mathcal{T}, I_{in})$ be a model expansion problem with decomposition $MX(\mathcal{T}^*, I_{in}^*)$ and decomposed input vocabulary Σ_{in}^*. Let A be an argument position set connectively closed under $\mathcal{T}^*$. If A contains at most one argument position for each decomposed symbol in Σ^*_{in}, the permutations π such that $I^*_{in} = \sigma^\pi_A(I^*_{in})$ always arise from interchangeable subdomains $\delta \supseteq Supp(\pi)$.

Proof. We show that for each permutation π such that $I^*_{in} = \sigma^\pi_A(I^*_{in})$, each permutation $\pi_d = (d\ \pi(d))$ with $d \in D$ also induces a local domain symmetry that preserves the decomposed input structure – $I^*_{in} = \sigma^{\pi_d}_A(I^*_{in})$. If so, $Supp(\pi)$ forms an interchangeable subdomain of the domain D, as interchangeable subdomains are generated by local domain symmetries induced by swaps of elements of the interchangeable subdomain. Any interchangeable subdomain is a subset of one, e.g., of itself.

Without loss of generality, we assume the argument position set A concerns at most the first argument of a symbol $S^* \in \Sigma^*_{in}$. Then, each local domain symmetry σ^π_A maps domain element tuples $(d, \dots, d') \in {S^*}^{I^*_{in}}$ to domain element tuples $(\pi(d), \dots, d') \in {S^*}^{\sigma^\pi_A(I^*_{in})}$. $I^*_{in} = \sigma^\pi_A(I^*_{in})$ holds iff ${S^*}^{I^*_{in}} = {S^*}^{\sigma^\pi_A(I^*_{in})}$ for each $S^* \in \Sigma^*_{in}$, meaning $(d, \dots, d') \in {S^*}^{I^*_{in}}$ iff $(\pi(d), \dots, d') \in {S^*}^{I^*_{in}}$. As a result, $(d\ \pi(d))$ would also preserve each interpretation ${S^*}^{I^*_{in}}$, so $I^*_{in} = \sigma^{(d\ \pi(d))}_A(I^*_{in})$ holds as well. $\square$

Bibliography

[1] Aloul, F., Ramani, A., Markov, I., and Sakallah, K. Solving difficult SAT instances in the presence of symmetry. In *Design Automation Conference, 2002. Proceedings. 39th* (2002), pp. 731–736.

[2] Aloul, F. A., Markov, I. L., and Sakallah, K. A. Shatter: Efficient symmetry-breaking for Boolean satisfiability. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003* (2003), ACM, pp. 836–839.

[3] Aloul, F. A., Ramani, A., Markov, I. L., and Sakallah, K. A. Dynamic symmetry-breaking for Boolean satisfiability. *Annals of Mathematics and Artificial Intelligence 57*, 1 (Sept. 2009), 59–73.

[4] Aloul, F. A., Sakallah, K. A., and Markov, I. L. Efficient symmetry breaking for Boolean satisfiability. *IEEE Transactions on Computers 55*, 5 (2006), 549–558.

[5] Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L. K., Wallner, J. P., and Xiao, G. The fourth Answer Set Programming competition: Preliminary report. In Cabalar and Son [21], pp. 42–53.

[6] Alviano, M., Dodaro, C., Faber, W., Leone, N., and Ricca, F. WASP: A native ASP solver based on constraint learning. In Cabalar and Son [21], pp. 54–66.

[7] Anagnostopoulos, A., Michel, L., Van Hentenryck, P., and Vergados, Y. A simulated annealing approach to the traveling tournament problem. *Journal of Scheduling 9*, 2 (4 2006), 177–193.

[8] Apt, K. R. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[9] Audemard, G., and Benhamou, B. Reasoning by symmetry and function ordering in finite model generation. In *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings* (2002), A. Voronkov, Ed., vol. 2392 of *Lecture Notes in Computer Science*, Springer, pp. 226–240.

[10] Audemard, G., and Simon, L. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI* (2009), C. Boutilier, Ed., pp. 399–404.

[11] Babai, L. *Handbook of Combinatorics.* Elsevier, 1995, ch. Automorphism groups, isomorphism, reconstruction, pp. 1447–1540.

[12] Balint, A., Belov, A., Heule, M. J., and Järvisalo, M. The 2013 international SAT competition. `satcompetition.org/2013`, 2013.

[13] Barrett, C. W., Sebastiani, R., Seshia, S. A., and Tinelli, C. Satisfiability modulo theories. In Biere et al. [18], pp. 825–885.

[14] Benhamou, B. Dynamic and static symmetry breaking in answer set programming. In *Logic for Programming, Artificial Intelligence, and Reasoning: 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings* (Berlin, Heidelberg, 2013), K. McMillan, A. Middeldorp, and A. Voronkov, Eds., Springer Berlin Heidelberg, pp. 112–126.

[15] Benhamou, B., Nabhani, T., Ostrowski, R., and Saïdi, M. R. Dynamic symmetry breaking in the satisfiability problem. In *Proceedings of the 16th international conference on Logic for Programming, Artificial intelligence, and Reasoning* (Dakar, Senegal, April 25 - may 1, 2010), LPAR-16.

[16] Benhamou, B., Nabhani, T., Ostrowski, R., and Saïdi, M. R. Enhancing clause learning by symmetry in SAT solvers. In *ICTAI (1)* (2010), IEEE Computer Society, pp. 329–335.

[17] Benoist, T., Estellon, B., Gardi, F., Megel, R., and Nouioua, K. LocalSolver 1.x: A black-box local-search solver for 0-1 programming. *4OR 9*, 3 (2011), 299–316.

[18] Biere, A., Heule, M., van Maaren, H., and Walsh, T., Eds. *Handbook of Satisfiability* (2009), vol. 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press.

[19] Bomanson, J., and Janhunen, T. Normalizing cardinality rules using merging and sorting constructions. In Cabalar and Son [21], pp. 187–199.

[20] Bonet, M., Pitassi, T., and Raz, R. Lower bounds for cutting planes proofs with small coefficients. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1995), STOC '95, ACM, pp. 575–584.

[21] Cabalar, P., and Son, T. C., Eds. *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings* (2013), vol. 8148 of *LNCS*, Springer.

[22] Chu, G., and Stuckey, P. J. A generic method for identifying and exploiting dominance relations. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming* (Berlin, Heidelberg, 2012), CP'12, Springer-Verlag, pp. 6–22.

[23] Claessen, K., and Sörensson, N. New Techniques that Improve MACE-style Model Finding. In *Workshop on Model Computation (MODEL)* (2003).

[24] Crawford, J. M., Ginsberg, M. L., Luks, E. M., and Roy, A. Symmetry-Breaking Predicates for Search Problems. In *Principles of Knowledge Representation and Reasoning* (1996), Morgan Kaufmann, pp. 148–159.

[25] De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., and Denecker, M. Predicate logic as a modelling language: The IDP system. *CoRR abs/1401.6312v2* (2016).

[26] De Cat, B., Bogaerts, B., Devriendt, J., and Denecker, M. Model expansion in the presence of function symbols using constraint programming. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013* (2013), IEEE Computer Society, pp. 1068–1075.

[27] Déharbe, D., Fontaine, P., Merz, S., and Woltzenlogel Paleo, B. *Exploiting Symmetry in SMT Problems.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 222–236.

[28] Denecker, M., Lierler, Y., Truszczyński, M., and Vennekens, J. A Tarskian informal semantics for answer set programming. In *ICLP (Technical Communications)* (2012), A. Dovier and V. S. Costa, Eds., vol. 17 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 277–289.

[29] Denecker, M., and Ternovska, E. A logic of nonmonotone inductive definitions. *ACM Trans. Comput. Log. 9*, 2 (Apr. 2008), 14:1–14:52.

[30] Denecker, M., and Vennekens, J. Building a knowledge base system for an integration of logic programming and classical logic. In *ICLP* (2008), M. García de la Banda and E. Pontelli, Eds., vol. 5366 of *LNCS*, Springer, pp. 71–76.

[31] Devriendt, J. Binaries, benchmark results and problem specifications for "exploiting symmetry in model expansion for predicate and propositional logic". `bitbucket.org/krr/thesis_jo_experiments`.

[32] Devriendt, J. An implementation of Symmetric Explanation Learning in Glucose on Bitbucket. `bitbucket.org/krr/glucose-syrup-spfs`.

[33] Devriendt, J. An implementation of Symmetry Propagation in Minisat on Github. `github.com/JoD/minisat-SPFS`.

[34] Devriendt, J., and Bogaerts, B. BreakID, a symmetry breaking preprocessor for SAT solvers. `bitbucket.org/krr/breakid`, 2015.

[35] Devriendt, J., and Bogaerts, B. BreakID: Static symmetry breaking for ASP (system description). *CoRR abs/1608.08447* (2016).

[36] Devriendt, J., Bogaerts, B., and Bruynooghe, M. BreakIDGlucose: On the importance of row symmetry in SAT. In *Proceedings of the Fourth International Workshop on the Cross-Fertilization Between CSP and SAT (CSPSAT)* (2014).

[37] Devriendt, J., Bogaerts, B., Bruynooghe, M., and Denecker, M. Improved static symmetry breaking for SAT. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings* (2016), N. Creignou and D. L. Berre, Eds., vol. 9710 of *Lecture Notes in Computer Science*, Springer, pp. 104–122.

[38] Devriendt, J., Bogaerts, B., Bruynooghe, M., and Denecker, M. On local domain symmetry for model expansion. *Theory and Practice of Logic Programming 16*, 5-6 (009 2016), 636–652.

[39] Devriendt, J., Bogaerts, B., De Cat, B., Denecker, M., and Mears, C. 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* (2012), IEEE Computer Society, pp. 49–56.

[40] Devriendt, J., De Causmaecker, P., and Denecker, M. Transforming constraint programs to input for local search. In *The Fourteenth International Workshop on Constraint Modelling and Reformulation, Constraint Modelling and Reformulation, Cork, Ireland, 31 August 2015* (Aug. 2015), pp. 1–16.

[41] Drescher, C., Tifrea, O., and Walsh, T. Symmetry-breaking answer set solving. *AI Communications 24*, 2 (2011), 177–194.

[42] Eén, N., and Sörensson, N. An extensible SAT-solver. In *SAT* (2003), E. Giunchiglia and A. Tacchella, Eds., vol. 2919 of *LNCS*, Springer, pp. 502–518.

[43] Flener, P., Frisch, A. M., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., and Walsh, T. Breaking row and column symmetries in matrix models. In *Principles and Practice of Constraint Programming - CP 2002*, P. Hentenryck, Ed., vol. 2470 of *LNCS*. Springer Berlin Heidelberg, 2002, pp. 462–477.

[44] Galinier, P., and Hertz, A. A survey of local search methods for graph coloring. *Computers & Operations Research 33*, 9 (2006), 2547 – 2562. Part Special Issue: Anniversary Focused Issue of Computers & Operations Research on Tabu Search.

[45] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.7.9*, 2015.

[46] Gaspero, L. D., and Schaerf, A. A composite-neighborhood tabu search approach to the traveling tournament problem. *Journal of Heuristics 13*, 2 (2007), 189–207.

[47] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. *Clingo* = ASP + control: Preliminary report. In *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)* (2014), M. Leuschel and T. Schrijvers, Eds., vol. 14(4-5). Online Supplement.

[48] Gebser, M., Kaufmann, B., and Schaub, T. Conflict-driven answer set solving: From theory to practice. *Artif. Intell. 187* (2012), 52–89.

[49] Gebser, M., Schaub, T., and Thiele, S. GrinGo: A new grounder for Answer Set Programming. In *LPNMR* (2007), C. Baral, G. Brewka, and J. S. Schlipf, Eds., vol. 4483 of *Lecture Notes in Computer Science*, Springer, pp. 266–271.

[50] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *ICLP/SLP* (1988), R. A. Kowalski and K. A. Bowen, Eds., MIT Press, pp. 1070–1080.

[51] Gendreau, M., and Potvin, J.-Y. *Handbook of Metaheuristics*, 2nd ed. Springer Publishing Company, Incorporated, 2010.

[52] Gent, I. P., Petrie, K. E., and Puget, J.-F. Symmetry in constraint programming. *Handbook of Constraint Programming 10* (2006), 329–376.

[53] Gent, I. P., and Smith, B. M. Symmetry breaking in constraint programming. In *Proceedings of ECAI-2000* (2000), IOS Press, pp. 599–603.

[54] Haken, A. The intractability of resolution. *Theoretical Computer Science 39* (1985), 297 – 308. Third Conference on Foundations of Software Technology and Theoretical Computer Science.

[55] Heule, M., Järvisalo, M., and Balyo, T. The 2016 international SAT competition. `http://baldur.iti.kit.edu/sat-competition-2016`, 2016.

[56] Heule, M., Keur, A., Maaren, H. V., Stevens, C., and Voortman, M. Cnf symmetry breaking options in conflict driven sat solving, 2005.

[57] Jefferson, C., and Petrie, K. E. Automatic generation of constraints for partial symmetry breaking. In *Principles and Practice of Constraint Programming – CP 2011: 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings* (Berlin, Heidelberg, 2011), J. Lee, Ed., Springer Berlin Heidelberg, pp. 729–743.

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

[59] Katayama, K., Hamamoto, A., and Narihisa, H. An effective local search for the maximum clique problem. *Inf. Process. Lett. 95*, 5 (Sept. 2005), 503–511.

[60] Katebi, H., Sakallah, K. A., and Markov, I. L. Symmetry and satisfiability: An update. In *SAT* (2010), O. Strichman and S. Szeider, Eds., vol. 6175 of *LNCS*, Springer, pp. 113–127.

[61] Kuhn, H. W. *The Hungarian Method for the Assignment Problem.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 29–47.

[62] Lee, J. H. M., and Li, J. Increasing symmetry breaking by preserving target symmetries. In *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings* (Berlin, Heidelberg, 2012), M. Milano, Ed., Springer Berlin Heidelberg, pp. 422–438.

[63] Lifschitz, V. Answer set planning. In *Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999* (1999), D. De Schreye, Ed., MIT Press, pp. 23–37.

[64] Lin, S., and Kernighan, B. W. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research 21*, 2 (1973), 498–516.

[65] Lynce, I., and Marques-Silva, J. Breaking symmetries in sat matrix models. In *Theory and Applications of Satisfiability Testing – SAT 2007: 10th International Conference, Lisbon, Portugal, May 28-31, 2007. Proceedings* (Berlin, Heidelberg, 2007), J. Marques-Silva and K. A. Sakallah, Eds., Springer Berlin Heidelberg, pp. 22–27.

[66] Marek, V., and Truszczyński, M. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, K. R. Apt, V. Marek, M. Truszczyński, and D. S. Warren, Eds. Springer-Verlag, 1999, pp. 375–398.

[67] Margot, F. *Symmetry in Integer Linear Programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 647–686.

[68] Marques Silva, J. P., Lynce, I., and Malik, S. Conflict-driven clause learning SAT solvers. In Biere et al. [18], pp. 131–153.

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

[70] McCune, W. Mace4 reference manual and guide. *CoRR cs.SC/0310055* (2003).

[71] McKay, B. D., and Piperno, A. Practical graph isomorphism, II. *Journal of Symbolic Computation 60*, 0 (2014), 94 – 112.

[72] Mears, C., and García de la Banda, M. Towards automatic dominance breaking for constraint optimization problems. In *IJCAI* (2015), Q. Yang and M. Wooldridge, Eds., AAAI Press, pp. 360–366.

[73] Mears, C., García de la Banda, M., Demoen, B., and Wallace, M. Lightweight dynamic symmetry breaking. *Constraints* (2013), 1–48.

[74] Mears, C., García de la Banda, M. J., Wallace, M., and Demoen, B. A novel approach for detecting symmetries in CSP models. In *CPAIOR* (2008), L. Perron and M. A. Trick, Eds., vol. 5015 of *LNCS*, Springer, pp. 158–172.

[75] Michel, L., and Van Hentenryck, P. The Comet programming language and system. In *CP* (2005), P. van Beek, Ed., vol. 3709 of *LNCS*, Springer, pp. 881–881.

[76] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an efficient SAT solver. In *DAC'01* (2001), ACM, pp. 530–535.

[77] Nam, G.-J., Aloul, F., Sakallah, K. A., and Rutenbar, R. A. A comparative study of two Boolean formulations of FPGA detailed routing constraints. *IEEE Trans. Comput. 53* (June 2004), 688–696.

[78] Narodytska, N., and Walsh, T. Dynamic versus static value symmetry breaking. In *11th International Workshop on Symmetry in Constraint Satisfaction Problems, SymCon'11 at CP'11* (2011).

[79] Narodytska, N., and Walsh, T. Breaking symmetry with different orderings. In *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings* (Berlin, Heidelberg, 2013), Springer Berlin Heidelberg, pp. 545–561.

[80] Niemelä, I. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell. 25*, 3-4 (1999), 241–273.

[81] Oh, C. *Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL.* PhD thesis, New York University, 2015. PhD thesis.

[82] Prestwich, S., and Roli, A. Symmetry breaking and local search spaces. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, R. Barták and M. Milano, Eds., vol. 3524 of *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 2005, pp. 273–287.

[83] Puget, J.-F. Breaking symmetries in all-different problems. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence, IJCAI 2005* (2005), pp. 272–277.

[84] Sabharwal, A. SymChaff: Exploiting symmetry in a structure-aware satisfiability solver. *Constraints 14*, 4 (2009), 478–505.

[85] Sakallah, K. A. *Symmetry and Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, Feb. 2009, ch. 10, pp. 289–338.

[86] The international SAT Competitions web page. `www.satcompetition.org`.

[87] Schaafsma, B., Heule, M., and van Maaren, H. Dynamic symmetry breaking by simulating Zykov contraction. In *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings* (2009), O. Kullmann, Ed., vol. 5584 of *Lecture Notes in Computer Science*, Springer, pp. 223–236.

[88] Selman, B., Kautz, H., and Cohen, B. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* (1995), pp. 521–532.

[89] Seress, Á. *Permutation Group Algorithms.* Cambridge University Press, 2003. Cambridge Books Online.

[90] Shlyakhter, I. Generating effective symmetry-breaking predicates for search problems. *Discrete Appl. Math. 155*, 12 (June 2007), 1539–1548.

[91] Swan, J., Adriaensen, S., Bishr, M., Burke, E. K., Clark, J. A., De Causmaecker, P., Durillo, J., Hammond, K., Hart, E., Johnson, C. G., Kocsis, Z. A., Kovitz, B., Krawiec, K., Martin, S., Merelo, J. J., Minku, L. L., Özcan, E., Pappa, G. L., Pesch, E., García-Sánchez, P., Schaerf, A., Sim, K., Smith, J. E., Stützle, T., Voss, S., Wagner, S., and Yao, X. A research agenda for metaheuristic standardization. In *Proceedings of the XI Metaheuristics International Conference, June 7-10, 2015, Agadir, Morocco* (2015).

[92] Syrjänen, T. Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`, 2000.

[93] Theise, E. S., and Minoux, M. *Mathematical Programming: Theory and Algorithms.* JSTOR, 1990.

[94] Torlak, E., and Jackson, D. Kodkod: A relational model finder. In *TACAS* (2007), O. Grumberg and M. Huth, Eds., vol. 4424 of *LNCS*, Springer, pp. 632–647.

[95] Trick, M. Network resources for coloring a graph. `http://mat.gsia.cmu.edu/COLOR/color.html`, 1994.

[96] Urquhart, A. Hard examples for resolution. *J. ACM 34*, 1 (Jan. 1987), 209–219.

[97] Van Hentenryck, P., Flener, P., Pearson, J., and Ägren, M. Compositional derivation of symmetries for constraint satisfaction. In *Abstraction, Reformulation and Approximation*, J.-D. Zucker and L. Saitta, Eds., vol. 3607 of *LNCS*. Springer Berlin Heidelberg, 2005, pp. 234–247.

[98] Van Hentenryck, P., and Michel, L. *Constraint-Based Local Search*. MIT Press, 2005.

[99] Walsh, T. Symmetry breaking constraints: Recent results. *CoRR abs/1204.3348* (2012).

[100] Zhang, J., and Zhang, H. SEM: A system for enumerating models. In *Department of Philosophy University of Wisconsin-Madison Mathematics and Computer Science* (1995), pp. 298–303.

[101] Zhang, L., and Madigan, C. F. Efficient conflict driven learning in a Boolean satisfiability solver. In *In ICCAD* (2001), pp. 279–285.

Curriculum Vitae

Jo Devriendt was born in Bruges, Belgium, on February 3, 1988. After completing secondary school at the Sint-Lodewijkscollege in Bruges, he started a Bachelor in Physics in 2006 at the KU Leuven, campus Kortrijk. After two years, he transferred to Leuven, switching to a Bachelor in Informatics. In 2010 he finished this degree, followed by the Master in Engineering Science: Computer Science in 2012. His master thesis was supervised by Prof. Dr. Marc Denecker, and was titled “Breaking symmetry for model generation in first order logic”.

After graduating, Jo joined the DTAI (“Declarative Languages and Artificial Intelligence”) research group to pursue a PhD under supervision of Prof. dr. Marc Denecker.

List of Publications

A complete and up-to-date list of publications can be found at `www.cs.kuleuven.be/publicaties/lirias/mypubs.php?unum=U0059879`

Journal Articles

- J. Devriendt, B. Bogaerts, M. Bruynooghe, M. Denecker. "On local domain symmetry for model expansion" In: Theory and Practice of Logic Programming, volume 16, issue 5-6, pages 636-652, 2016.

Peer-reviewed Articles at Conferences

- J. Devriendt, B. Bogaerts, M. Bruynooghe, M. Denecker. "Improved static symmetry breaking for SAT". In: Lecture Notes in Computer Science, volume 9710, issue 1, pages 104-122, 2016.
- J. Jansen, B. Bogaerts, J. Devriendt, G. Janssens, M. Denecker. "Relevance for SAT(ID)". In: International Joint Conference on Artificial Intelligence, 9-15th of July 2016, pages 596-603.
- B. De Cat, B. Bogaerts, M. Denecker, J. Devriendt. "Model expansion in the presence of function symbols using constraint programming". In: International Conference on Tools For Aritificial Intelligence, Washington D.C., USA, 4-6 November 2013, pages 1068-1075.
- J. Devriendt, B. Bogaerts, C. Mears, B. De Cat, M. Denecker. "Symmetry propagation: Improved dynamic symmetry breaking in SAT". In: International Conference on Tools with Artificial Intelligence, Athens, Greece, 7-9 November 2012, pages 49-56.

- J. Jansen, I. Dasseville, J. Devriendt, G. Janssens. “Experimental evaluation of a state-of-the-art grounder”. In: Principles and Practice of Declarative Programming, 8-10 September 2014, pages 249-259.

Peer-reviewed Articles at Workshops

- J. Devriendt, B. Bogaerts. “BreakID: Static symmetry breaking for ASP (system description)”. In: Workshop on answer set programming and other computing paradigms, New York City, USA, 16 October 2016, pages 25-39.
- J. Devriendt, P. De Causmaecker, M. Denecker. “Transforming constraint programs to input for local search”. In: Constraint Modelling and Reformulation, Cork, Ireland, 31 August 2015, pages 1-16.
- J. Devriendt, B. Bogaerts, M. Bruynooghe. “BreakIDGlucose: On the importance of row symmetry in SAT”. In: International Workshop on the Cross-Fertilization Between CSP and SAT, Vienna, Austria, 18 July 2014, Proceedings, pages 1-17.
- P. Van Hertum, J. Vennekens, B. Bogaerts, J. Devriendt, M. Denecker. “The effects of buying a new car: An extension of the IDP Knowledge Base System” (technical communication). In: International Conference on Logic Programming, Istanbul, Turkey, 24-29 August 2013, pages 1-4.

Informal and Other Publications

- S. Pham, J. Devriendt, M. Bruynooghe, P. De Causmaecker. “A MIP Backend for the IDP System”. In: arXiv.org, 1609.007, 2014.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
DECLARATIVE LANGUAGES AND ARTIFICIAL INTELLIGENCE
Celestijnenlaan 200A box 2402
B-3001 Leuven
jo.devriendt@cs.kuleuven.be
http://www.dtai.cs.kuleuven.be