Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

11-30-2020

Constraint Satisfaction Techniques for Combinatorial Problems

David E. Narvaez
den9562@rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Narvaez, David E., "Constraint Satisfaction Techniques for Combinatorial Problems" (2020). Thesis. Rochester Institute of Technology. Accessed from

This Dissertation is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

Constraint Satisfaction Techniques for Combinatorial Problems

by

David E. Narváez

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
November 30, 2020

Constraint Satisfaction Techniques for Combinatorial Problems

By

David Narváez

Committee Approval:
We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

_____

Dr. Edith Hemaspaandra — Date
Dissertation Advisor

_____

Dr. Stanisław Radziszowski — Date
Dissertation Advisor

_____

Dr. Ivona Bezáková — Date
Dissertation Committee Member

_____

Dr. Matthew Fluet — Date
Dissertation Committee Member

_____

Dr. Lane A. Hemaspaandra — Date
Dissertation Committee Member

_____

Dr. Darren Narayan — Date
Dissertation Defense Chairperson

Certified by:

_____

Dr. Pengcheng Shi — Date
Ph.D. Program Director, Computing and Information Sciences

Constraint Satisfaction Techniques for Combinatorial Problems

by

David E. Narváez

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences
Ph.D. Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

The last two decades have seen extraordinary advances in tools and techniques for constraint satisfaction. These advances have in turn created great interest in their industrial applications. As a result, tools and techniques are often tailored to meet the needs of industrial applications out of the box. We claim that in the case of abstract combinatorial problems in discrete mathematics, the standard tools and techniques require special considerations in order to be applied effectively. The main objective of this thesis is to help researchers in discrete mathematics weave through the landscape of constraint satisfaction techniques in order to pick the right tool for the job. We consider constraint satisfaction paradigms like satisfiability of Boolean formulas and answer set programming, and techniques like symmetry breaking. Our contributions range from theoretical results to practical issues regarding tool applications to combinatorial problems. We prove search-versus-decision complexity results for problems about backbones and backdoors of Boolean formulas. We consider applications of constraint satisfaction techniques to problems in graph arrowing (specifically in Ramsey and Folkman theory) and computational social choice. Our contributions show how applying constraint satisfaction techniques to abstract combinatorial problems poses additional challenges. We show how these challenges can be addressed. Additionally, we consider the issue of trusting the results of applying constraint satisfaction techniques to combinatorial problems by relying on verified computations.

Acknowledgments

The work that appears in this thesis was supported in part by grant no. DUE-1819546 of the National Science Foundation.

Contents

List of Publications

The following is a list of publications that form the base of this thesis.

The contributions in Chapter 4 appear in the following publications:

- L. A. Hemaspaandra and D. E. Narváez. The opacity of backbones. In 31st AAAI Conference on Artificial Intelligence, pages 3900-3906. AAAI Press, Feb. 2017. (Expanded version: arXiv:1606.03634)
- L. A. Hemaspaandra and D. E. Narváez. Existence versus exploitation: The opacity of backdoors and backbones under a weak assumption. In 45th International Conference on Current Trends in Theory and Practice of Computer Science, pages 247-259. Springer, 2019.

The contributions in Chapter 5 appear in the following publications:

- D. E. Narváez. Exploring the use of Shatter for AllSAT through Ramsey-type problems (Student Abstract). In 32nd AAAI Conference on Artificial Intelligence, pages 8123-8124. AAAI Press, 2018. (Expanded version: arXiv:1711.06362)
- C. Jayawardene, D. E. Narváez, and S. P. Radziszowski. Star-critical Ramsey numbers for cycles versus K_4. Discussiones Mathematicae Graph Theory, (2629), 2018. In press.
- D. E. Narváez. A QSAT benchmark based on vertex-Folkman problems (Student Abstract). In 34th AAAI Conference on Artificial Intelligence, 34(10), 13881–13882. AAAI Press, 2020.

The contributions in Chapter 6 appear in the following publications:

- Z. Fitzsimmons, E. Hemaspaandra, A. Hoover, and D. E. Narváez. Very hard electoral control problems. In COMSOC18, 2018.
- Z. Fitzsimmons, E. Hemaspaandra, A. Hoover, and D. E. Narváez. Very hard electoral control problems. In 33rd AAAI Conference on Artificial Intelligence, pages 1933-1940. AAAI Press, 2019.

The contributions in Chapter 7 appear in the following publication:

- D. E. Narváez. Formalizing CNF SAT symmetry breaking in PVS. In NASA Formal Methods Symposium, volume 11460 of Lecture Notes in Computer Science, pages 341-354. Springer, 2019.

Chapter 1

Introduction

In the original conception of computers (for instance, Babbage's Analytical Engine), these machines would work on some input to calculate the value of an expression. Problems of this kind were, and still are to some fields inside and outside of computer science, the standard definition of what a computational problem is. As a simplified example, consider the following arithmetic expression:

$$2 \times 4 + 3 \times 5 = ?$$

The language of arithmetic provides a blueprint of the steps to follow in order to calculate the value of the symbol ? that, when replaced in the above expression, will make it true. From our understanding of arithmetic, the value that ? should take is 23. On the other hand, one can ask the following question:

$$2 \; ? \; 4 \; ? \; 3 \; ? \; 5 = 23$$

That is, which operators should replace the ? symbols so that the expression is true? We know an answer from the example above, but how would one go about reaching that answer without prior knowledge? A straightforward strategy is to try every possible combination of operators in every position, but one can do better. This is indeed the nature of combinatorial problems: problems whose naïve solution strategy involves trying every possible combination over a space of potential solutions.

Combinatorial problems appear in all aspects of life, and even deceptively simple problems can hide interesting combinational underlying ques-

tions. Think back, for instance, about the simple example above: How many valid ways are there to perform the arithmetical operations with a computer that is only able to perform one operation at a time? Is one way better than the others? While the answers to these questions for the particular example we presented are both simple and inconsequential, generalizations of this question have puzzled mathematicians for decades. For example, devising new matrix multiplication schemes which reduce the number of required multiplications of floating-point numbers is a notably difficult combinatorial problem [62]. Very little is known about what the minimum number of multiplications required for matrices of order $n > 2$ is, yet obtaining minimal schemes would greatly improve the running time of many numerical applications.

As computers have advanced, mathematicians have benefited from the use of computer programs to find answers to combinatorial problems. The degree of complexity of the work done by a computer when solving a problem has ranged from straightforward-but-tedious case analysis to intricate heuristics to explore large search spaces. It is clear that the nature of combinatorial problems is such that there will always be open questions whose answers are beyond the scope of what computers can realistically achieve. Nevertheless, there is continuous research seeking to match the latest advances in computing power with the latest advances in mathematics in order to push the boundaries of what questions can be answered with the help of computers.

An important line of research in the effort of incorporating computers into research in discrete mathematics is formulating combinatorial problems as constraint satisfaction problems (CSPs). This allows mathematicians to use a common language to encode combinatorial problems. Once these encodings are obtained, one can make use of a plethora of available solvers and tools to find answers to the problem in question. Nevertheless, we argue that there is something special about combinatorial problems of the type we consider in this thesis that makes applying constraint satisfaction paradigms more difficult. The types of problems for which CSPs have been applicable can be roughly divided into two categories: industrial applications and abstract applications. While no formal definition of these has been given in the literature[1], one can informally define industrial applications as those arising

[1]It is important to mention here that some work has been done in trying to characterize instances of constraint satisfaction problems stemming from industrial applications. For example, the work by Ansótegui, Bonet, and Levy [4] proposes a structural characterization

from practical applications like circuit design and hardware checking, whereas abstract applications correspond to CSPs arising from problems in discrete mathematics. The development of constraint satisfaction techniques over the last few decades has focused mainly on industrial applications. The focus of our research, on the other hand, is on using constraint satisfaction techniques specifically for abstract combinatorial problems.

Matching the right type of constraint satisfaction paradigm to the combinatorial problem at hand can be difficult in many aspects. Different paradigms are able to encode different levels of difficulty, but some of the less powerful techniques enjoy the largest tool support in practice. The main objective of this thesis is to help researchers weave through the landscape of constraint satisfaction techniques in order to pick the right tool for the job. Our work is comprehensive in the sense that we analyze theoretical and practical aspects of applying constraint satisfaction techniques to combinatorial problems. This is important because no particular algorithm or constraint satisfaction technique will outperform the rest on every family of problems we explore [114]. Instead, we need to understand the structure and the inherent hardness of the problems and match these with the techniques and algorithms that are most appropriate.

This thesis is organized in two parts. Part I contains an overview of the theoretical foundations over which we build our contributions. A prerequisite is a basic understanding of complexity theory. We refer the reader to one of the many books on this subject (e.g., [63, 110]). Computational complexity provides a concrete definition of how "hard" a problem can be. This will help us formally differentiate the levels of hardness each paradigm supports.

Chapter 2 explains the constraint satisfaction paradigms we use in this thesis. We cover Boolean formulas in Section 2.1 and answer set programming in Section 2.2. Although these paradigms do not usually appear under the same umbrella in the literature, we bring them into one framework. Section 2.3 deals with a problem that is prevalent in encodings of abstract combinatorial problems as CSPs regardless of the paradigm: the problem of symmetries.

Chapter 3 looks at two complementary aspects of the issue of trusting the results obtained from CSP solvers: trusting the solvers themselves by

of Boolean formulas in conjunctive normal form (see Section 2.1.1) that encode instances of industrial applications by looking at the bipartite graph representing the relationships between clauses and variables.

verifying their source code (Section 3.1), and trusting the results by verifying the answers (Section 3.2).

With the fundamentals defined in Part I, Part II presents our contributions in using constraint satisfaction techniques for combinatorial problems. We start in Chapter 4, proving complexity results related to backbones and backdoors of Boolean formulas. Backbones and backdoors are hidden structures that have been linked to the hardness of instances of the satisfiability problem for Boolean formulas.

Our contributions in Chapter 5 are in graph arrowing, an important concept in extremal graph theory. We explain how to encode this property in different constraint satisfaction paradigms in Section 5.2. In Section 5.3 we address issues related to using symmetry breaking tools when enumerating unique solutions for the CSPs obtained. Section 5.4 develops a benchmark for quantified Boolean formulas based on vertex Folkman problems, an important class of problems within graph arrowing. We conclude Chapter 5 describing several results and contributions in our research that have been aided by the use of these methods.

Chapter 6 contains our contributions in computational social choice. We look at election systems whose control problems have high complexity. Encoding these as constraint satisfaction problems thus require paradigms that support such high complexity. We use answer set programming and show how advanced techniques in ASP can be used to encode these problems.

Chapter 7 applies the approach explained in Section 3.1 to the important task of breaking symmetries in Boolean formulas, as explained in Section 2.3. This chapter shows the initial steps towards obtaining a formally-verified symmetry breaking tool using the Prototype Verification System (PVS) [89].

Finally, we conclude in Chapter 8 with a recapitulation of all the contributions listed in Part II. These contributions range from very theoretical work in computational complexity to practical tools to perform (verified) computations. All of them have in common that they are intended, each from its own angle, to aid researchers make use of constraint satisfaction techniques to tackle abstract combinatorial problems.

In this thesis we have packaged our contributions in the most organized and cohesive way possible under the common academic standards. Nevertheless, the experiences accumulated during years of research in these areas are sometimes hard to structure as an academic contribution. Some of these ex-

periences are anecdotal or were obtained through ideas that ultimately failed. We thus decided to add an appendix structured in a more subjective fashion. Appendix A contains the view of the author of this thesis in relation to interactive theorem provers, one of the frontiers towards which combinatorial computing will continue expanding in the next few years.

1.1 Summary of Contributions

- We present Boolean formulas and answer set programming under the same framework of constraint satisfaction in Chapter 2.
- We prove gaps between the computational complexity of the search and decision problems of backbones of Boolean formulas (Theorem 4.3) and backdoors to Boolean formulas in conjunctive normal form (Theorem 4.14) under reasonable assumptions.
- We show how to encode graph arrowing problems as Boolean formulas and using answer set programming in Section 5.2.
- We prove that using `Shatter`, a popular tool for symmetry breaking in Boolean formulas, to generate unique graph edge-colorings can in fact increase the number of solutions (Section 5.3.1). We also present a sufficient condition under which the edge-colorings so generated do not represent every equivalence class of colorings (Theorem 5.11).
- We introduce a benchmark for quantified Boolean formulas based on vertex Folkman problems in Section 5.4.
- We show how to encode control problems of high computational complexity using the familiar "guess and check" approach for answer set programming, commonly applied only to problems in NP.
- We initiate the formalization of a symmetry breaking tool in the Prototype Verification System [89] by formalizing the proof of Theorem 2.3.

Part I

An Overview of Our Framework

Chapter 2

Constraint Satisfaction

In the broadest sense, constraint satisfaction problems are problems that are expressed as sets of constraints over variables and operations over these variables. The variables may take values from some domain D and the question is whether there exists an assignment of values in D to the variables such that the set of constraints in the problem is satisfied.

Different choices of domains and operations yield different paradigms of constraint satisfaction. In this chapter we explain the three paradigms we intend to use in our research: SAT, QSAT and ASP. Although these three paradigms do not usually appear under the same umbrella in the literature, we bring them into our framework. Our brief overview of each paradigm will discuss the basics of the theoretical fundamentals behind the paradigm and its use in practice. We will also discuss their expressive power in terms of the computational complexity they are able to encode.

For the sake of conciseness we refrain from giving detailed explanations and examples for each paradigm. Instead, we refer the reader to sources in the literature that go deep into the theoretical and practical aspects of each paradigm.

2.1 Satisfiability of Boolean Formulas

Arguably the most common paradigm in constraint satisfaction, satisfiability problems for Boolean formulas (SAT) are constraint satisfaction problems where the domain is the Boolean domain containing two values $\top$ and $\bot$ (*true*

and *false*). In theory, the operations allowed over the variables of a SAT problem are all the customary logical operations like implications ($\rightarrow$) and exclusive disjunctions ($\oplus$). In practice, most SAT solvers take as an input a Boolean formula in conjunctive normal form (CNF) which is a conjunction of disjunctions. It is well known that any Boolean formula can be turned into an equisatisfiable formula in CNF (see Section 2.1.1), thus restricting SAT solvers to only work on formulas in CNF does not limit their power. It does, however, imply that problem encodings that are not in CNF may require an additional preprocessing step before using a SAT solver.

Industrial applications of SAT have spurred great interest in this problem, leading to advances in the development of SAT solvers. Nevertheless, the popularity of SAT solvers and the wide range of SAT solvers available as off-the-shelf tools do require researchers to understand, at least to some extent, the internals of the approach each solver implements. Most solvers implement one of two approaches: conflict detection/clause learning (CDCL) or look-ahead. CDCL solvers have been extremely popular in industrial applications, yet look-ahead solvers seem to perform better in combinatorial instances that are highly structured [74].

It is easy to see what kind of problems can be expressed as Boolean satisfiability problems: the Cook-Levin [22] theorem implies that all problems in NP can be turned to SAT problems in polynomial time. What this theorem does not imply is that SAT is a convenient way to encode every problem in NP, and other paradigms of CSPs explained in this chapter and elsewhere may be more suitable for certain problems.

For a Boolean formula F, we denote by $V(F)$ the set of variables appearing in F. We assume that whenever $V(F) = \emptyset$, then either $F = \top$ or $F = \bot$, i.e., the only Boolean formulas we admit are formulas without constants except for the formulas $\top$ and $\bot$. Adopting the notation of Williams, Gomes, and Selman [113], we use the following. A partial assignment of F is a function $a_S : S \rightarrow \{\top, \bot\}$ that assigns Boolean values to the variables in a set $S \subseteq V(F)$. A Boolean formula is satisfiable if there exists and assignment $a_{V(F)}$ of the variables in F such that, when the variables are replaced with the assigned values, the formula evaluates to $\top$. A Boolean formula is unsatisfiable if no such assignment exists. For a Boolean value $v \in \{\bot, \top\}$ and a variable $x \in V(F)$, the notation $F[x/v]$ denotes the formula F after replacing every

occurrence of x by v and simplifying[1]. This extends to partial assignments, e.g., to $F[a_S]$, in the natural way.

2.1.1 The Conjunctive Normal Form

As mentioned before, the majority of SAT solvers available expect the input formula to be in conjunctive normal form (CNF). This normal form restricts the input formula to be a conjunction of disjunctions, e.g.:

$$(x \vee y) \wedge (x \vee z)$$

Each disjunction is called a (CNF) *clause*. Every Boolean formula can be converted to an *equisatisfiable* CNF formula which has a number of auxiliary variables that is at most linear in the number of variables of the original formula [107]. Equisatisfiability means that the formula resulting from the conversion will be satisfiable if and only if the original formula is satisfiable.

We define satisfiability of CNF formulas using the language of set theory. This is done by formalizing the intuition that, in order for an assignment to satisfy a CNF formula, it must set at least one literal in every clause to $\top$. One can then define a CNF formula F to be a collection of clauses, each clause being a set of literals. $F \in \text{SAT}$ if and only if there exists an assignment $a_{V(F)}$ such that for all clauses $C \in F$ there exists a literal $l \in C$ such that $a_{V(F)}$ sets l to $\top$. Under this formalization, two trivial cases arise: F is trivially in SAT if F is empty, and F is trivially not in SAT if $\emptyset \in F$. We can also formalize simplification using this notation: after assigning a variable x to $\top$ (resp., $\bot$), the formula is simplified by removing all clauses that contain the literal x (resp., $\overline{x}$) and removing the literal $\overline{x}$ (resp., x) from the remaining clauses. This formalization extends to simplification of a formula over a partial assignment in the natural way.

Example 2.1. *Consider the CNF formula* $F = (x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_5) \wedge (x_1 \vee x_2 \vee x_4 \vee x_5) \wedge (x_3 \vee \overline{x_4}) \wedge (\overline{x_1} \vee x_2 \vee x_3 \vee x_5)$. *We can express this formula in our set theory notation as* $F = \{\{x_1, \overline{x_2}, \overline{x_3}, x_5\}, \{x_1, x_2, x_4, x_5\}, \{x_3, \overline{x_4}\}, \{\overline{x_1}, x_2, x_3, x_5\}\}$. *If*

[1]Note that in our framework, Boolean formulas do not admit constant values. For example, the expression $F = (\top \vee x) \wedge (x \vee y)$ is not admissible. Thus simplification is defined as removing all constants from the expression obtained by replacing the variables with their assigned Boolean values via the standard operations. In the previous example, the Boolean formula in question is $F = x \vee y$.

we assign x_3 *to* $\bot$ *and* x_4 *to* $\top$*, we have* $F[x_3/\bot, x_4/\top] = \{\emptyset, \{\overline{x_1}, x_2, x_5\}\}$*, which is unsatisfiable because it contains the empty set.*

2.1.2 Converting from DNF to CNF

Another normal form, which is in many senses the dual of CNF, is the *disjunctive normal form* (DNF) which is a disjunction of conjunctions, e.g.:

$$(a \wedge b) \vee (a \wedge \overline{c})$$

In DNF, the conjunctions are also called (DNF) *clauses*. Many constraints are easier to express in DNF than in CNF and, in fact, we will use DNF encodings throughout this thesis. Given its prominence, we will briefly discuss in this section two transformations from DNF to CNF.

The first transformation, attributed to Tseitin [107], is based on the intuition that a DNF formula is satisfiable if there is an assignment that satisfies at least one of its clauses, and each clause is satisfied if all of the literals are set to $\top$. Let F be a formula in DNF, where we adopt a similar set-theoretical notation for its clauses and literals as in Section 2.1.1. Let $C \in F$ be a DNF clause in F, we create an auxiliary variable x_C which will be true if and only if the literals in C are all true, that is:

$$x_C \leftrightarrow \bigwedge_{l \in C} l$$

or equivalently

$$\left(x_C \rightarrow \bigwedge_{l \in C} l\right) \wedge \left(\bigwedge_{l \in C} l \rightarrow x_C\right)$$

which, applying distributivity and De Morgan's laws, is equivalent to

$$\left(\bigwedge_{l \in C} (\overline{x_C} \vee l)\right) \wedge \left(\bigvee_{l \in C} \overline{l} \vee x_C\right),$$

and this last formula is in CNF. Aggregating these CNF subformulas for every DNF clause $C \in F$, and finally adding the CNF clause $\bigvee_{C \in F} x_C$, one obtains a CNF formula F' such that F and F' are equisatisfiable.

Example 2.2. *For the formula* $F = (a \wedge b) \vee (a \wedge \overline{c})$ *above with two DNF clauses, the CNF formula resulting from the Tseitin transformation is* $F' = (\overline{x_1} \vee a) \wedge (\overline{x_1} \vee b) \wedge (\overline{a} \vee \overline{b} \vee x_1) \wedge (\overline{x_2} \vee a) \wedge (\overline{x_2} \vee \overline{c}) \wedge (\overline{a} \vee c \vee x_2) \wedge (x_1 \vee x_2)$.

A simplification of the above transformation, attributed to the work of Plaisted and Greenbaum [93] is as follows. Note that in the transformation above, any assignment $a_{V(F')}$ that satisfies F' will have to set one variable x_C to $\top$ for some clause C. Due of the implication $x_C \rightarrow \bigwedge_{l \in C} l$, all the literals $l \in C$ will have to be set to $\top$, so $a_{V(F')}$, restricted to the variables in F, satisfies F. This reasoning does not involve the condition $\bigwedge_{l \in C} l \rightarrow x_C$ and those clauses are, in fact, redundant. This optimization is important because modern SAT solvers benefit from short CNF clauses like $(\overline{x_C} \vee l)$ and not from long clauses like $\left(\bigvee_{l \in C} \overline{l} \vee x_C \right)$.

2.1.3 Encoding Cardinality Constraints in SAT

Many interesting problems in combinatorics constrain the solution to be such that some condition involving the cardinality of a set is satisfied. Cardinality constraints are thus important constraint satisfaction techniques and have been the subject of much research in the field, both from the theoretical and the practical point of view [8,101]. We discuss three basic cardinality constraints, and some ways to encode them, to illustrate the techniques used in this matter. Nevertheless, the literature regarding cardinality constraints in SAT encodings is vast and we refer the reader to other sources for more details on the state-of-the-art.

For a set L of literals, we define the $\textsc{AtLeastK}(L, k)$ constraint as the formula that guarantees that at least k of the literals in L are set to $\top$ in a satisfying assignment. As a simple example, consider the particular case $\textsc{AtLeastK}(L, 1)$, also known as $\textsc{AtLeastOne}(L)$ in the literature, which is simply the constraint $\bigvee_{l \in L} l$. This constraint can be considered a CNF clause with $\|L\|$ literals, or a DNF formula with $\|L\|$ unit clauses, where $\|L\|$ denotes the cardinality of the set L. The picture is clearer when one considers the

simplest generalization of this idea:

$$\textsc{AtLeastK}(L,k) \equiv \bigvee_{\{l_1,l_2,\ldots,l_k\}\subseteq L} (l_1 \wedge l_2 \wedge \cdots \wedge l_k)$$

which is clearly a DNF formula. The resulting formula will then require a transformation to CNF as explained in Section 2.1.2 if it was to be used as input to most of the commonly available SAT solvers.

The $\textsc{AtMostK}(L,k)$ constraint is defined analogously as the formula that guarantees that at most k of the literals in L are set to $\top$. This is to say that at least $\|L\|-k$ of the literals are set to $\bot$, so one can in fact use the idea from the previous paragraph here. Define $\overline{L}$ as the set of literals in L, but negated, i.e. $\overline{L} = \{\bar{l} \mid l \in L\}$, then $\textsc{AtMostK}(L,k) \equiv \textsc{AtLeastK}(\overline{L}, \|L\| - k)$. While this approach is convenient in the sense that it implements both $\textsc{AtLeastK}(L,k)$ and $\textsc{AtMostK}(L,k)$ under the same framework, it has the shortcoming that for small k, the DNF clauses in the $\textsc{AtMostK}(L,k)$ formula are quite large. An alternative is to note that at most k literals in L are set to $\top$ if and only if in any subset of $k+1$ literals in L, there is at least one literal set to $\bot$. This idea leads to the following encoding:

$$\textsc{AtMostK}(L,k) \equiv \bigwedge_{\{l_1,l_2,\ldots,l_{k+1}\}\subseteq L} (\overline{l_1} \vee \overline{l_2} \vee \ldots \vee \overline{l_{k+1}}).$$

Finally, the $\textsc{ExactlyK}(L,k)$ constraint guarantees that exactly k of the literals in L are set to $\top$. It is easy to implement this constraint in terms of the constraints above:

$$\textsc{ExactlyK}(L,k) \equiv \textsc{AtLeastK}(L,k) \wedge \textsc{AtMostK}(L,k).$$

In particular, we have the following:

$$\textsc{ExactlyOne}(L) \equiv \textsc{ExactlyK}(L,1) \equiv \left(\bigvee_{l\in L} l\right) \wedge \left(\bigwedge_{\{l_1,l_2\}\in L} (\overline{l_1} \vee \overline{l_2})\right).$$

2.1.4 Variations of SAT

While the SAT problem is concerned with finding a satisfying assignment for the variables of a Boolean formulas, there are several problems formulated

around SAT. The variations that are relevant for this thesis are AllSAT and #SAT. The AllSAT problem consists of listing all the satisfying assignments of a Boolean formula and the #SAT problem is to determine the number of satisfying assignments of a formula.

AllSAT has gained popularity in recent years due to its industrial applications like model checking [51, 82, 116] and data mining [66]. The survey by Toda and Soh [106] summarizes the state-of-the-art in techniques used for solving AllSAT problems. It is clear that #SAT can be solved by counting the models output by an AllSAT solver, yet algorithms exist that are able to count the number of models without actually generating them [105].

2.1.5 Quantified Boolean Formulas

When one extends the symbols allowed in the Boolean problem to admit the quantifiers $\exists$ and $\forall$ one obtains a quantified Boolean formula (QBF). QBFs are commonly expressed in *prenex normal form* in which quantifiers over lists of symbols alternate, i.e., formulas of the form $Q_1\mathbf{x}_1\, Q_2\mathbf{x}_2 \ldots Q_n\mathbf{x}_n.\, \phi$ where ϕ is a Boolean formula, each $\mathbf{x}_i$ is a list of variables, and $Q_i \in \{\exists, \forall\}$ with $Q_i \neq Q_{i+1}$ are the (alternating) quantifiers for each list of variables. The existential (resp. universal) variables of a QBF F are those variables that appear in a block quantified by $\exists$ (resp. $\forall$). $Q_1\mathbf{x}_1\, Q_2\mathbf{x}_2 \ldots Q_n\mathbf{x}_n$ is the *prefix* and ϕ is the *matrix*. Extending Boolean formulas with quantifiers has important implications for the complexity of problems that can be represented using this paradigm. The True Quantified Boolean Formula (TQBF) problem which asks whether a quantified Boolean formula is true is PSPACE-complete [103].

A Skolem function for an existential variable x in a QBF F in prenex normal form is a Boolean function $\mathbf{f}_x$ that takes as input the vector of all the universal variables that appear before x in the prefix of F. F is true if and only if there exists a set of Skolem functions for each existential variable in F such that, when these variables are replaced by their respective Skolem functions (themselves represented as Boolean formulas), the resulting formula is a tautology. The process of replacing existential variables by their Skolem functions is known as Skolemization. The equivalence between a QBF being true and the existence of Skolem functions that reduce the QBF to a tautology provides a parallel to the SAT problem for Boolean formulas. Namely, we say a QBF is satisfiable if there is an assignment of the existential variables mapping

them to Skolem functions such that the resulting QBF after Skolemization is a tautology. It follows that a QBF is satisfiable if and only if it is true. The problem of determining whether a QBF is satisfiable is known as QSAT and is indeed equivalent to TQBF. To illustrate the parallel between QSAT and SAT, consider a Boolean formula F and let $\mathbf{x}$ be a vector of the variables in $V(F)$ (in any order). Then the Boolean formula F is satisfiable if and only if the QBF $\exists\mathbf{x}.F$ is true. In turn, the QBF $\exists\mathbf{x}.F$ is true if and only if we can find Skolem functions for each variable $x \in V(F)$ such that $\exists\mathbf{x}.F$ after Skolemization is a tautology. Since there are no universal variables before $\mathbf{x}$ in the prefix, these Skolem functions do not take any variables as inputs, i.e., they are constant functions. If $\exists\mathbf{x}.F$ is satisfiable, then the constant Skolem functions certifying this satisfiability in turn provide a satisfying assignment for F and vice versa.

In the last few decades there has been increased interest in the development of QSAT solvers: software to determine the satisfiability of QBFs. These solvers have traditionally expected input to be in PCNF: formulas in prenex normal form whose matrix is in CNF. Nevertheless, there are disadvantages of requiring the matrix of some QBFs to be in CNF [5,68] and several formats to express Boolean formulas as circuits have been proposed [11,70].

2.2 Answer Set Programming

A (disjunctive) *answer set program* is a set of constraints expressed as rules, each rule involving *atoms*. For the purpose of this overview, it is enough to define an atom as a constant, a variable, or a predicate over atoms. We indicate that p is a k-ary predicate by writing p/k. An atom is ground if it contains no variables, and an ASP program is ground if it contains only ground atoms. A rule $\mathbf{R}$ is defined by 3 parts: the *head* of the rule $H(\mathbf{R})$, the *positive body* of the rule $B^+(\mathbf{R})$, and the *negative body* of the rule $B^-(\mathbf{R})$. Each of these parts is a set of atoms. A set of atoms A satisfies (models) a rule $\mathbf{R}$ if one of the following conditions hold[2]:

- There exists an atom $a \in H(\mathbf{R})$ such that $a \in A$.

[2]At first glance, it seems from the conditions provided as if $H(\mathbf{R})$ and $B^-(\mathbf{R})$ play the exact same role, yet there are technical circumstances under which it does make a difference if an atom appears in $H(\mathbf{R})$ or in $B^-(\mathbf{R})$ so we chose to be more general and list these two conditions separately.

- There exists an atom $a \in B^+(\mathbf{R})$ such that $a \notin A$.
- There exists an atom $a \in B^-(\mathbf{R})$ such that $a \in A$.

A set of atoms A satisfies (models) an ASP program Π if A satisfies every rule $\mathbf{R} \in \Pi$. For a rule $\mathbf{R}$, we define $\mathbf{R}^+$ by its parts $H(\mathbf{R}^+) = H(\mathbf{R})$ and $B^+(\mathbf{R}^+) = B^+(\mathbf{R})$ and $B^-(\mathbf{R}) = \emptyset$. We define the *reduct* [48] of a program Π with respect to a set of atoms A as the program

$$\Pi^A = \left\{ \mathbf{R}^+ \mid \mathbf{R} \in \Pi \text{ and for all } a \in B^-(\mathbf{R}),\, a \notin A \right\}$$

and define an *answer set* of Π as a set of atoms A that is a minimal model of Π^A in the sense that no proper subset of A is a model of Π^A as well.

While our simplified exposition of ASP is purely set-theoretical, ASP is more commonly presented in a syntactic fashion where rules are written as

$$a_1,\, a_2,\, \ldots,\, a_h \leftarrow b_1,\, b_2, \ldots,\, b_n,\, \text{not}\, b_{n+1},\, \text{not}\, b_{n+2},\, \ldots, \text{not}\, b_{n+m}.$$

where the a_i atoms are the head of the rule, the b_i atoms for $1 \leq i \leq n$ are the positive body of the rule and the b_{n+i} atoms for $1 \leq i \leq m$ are the negative body of the rule. It is also more common to find definitions of ASP that are only concerned with *normal* ASP programs, which are programs in which every rule $\mathbf{R}$ is such that $\|H(\mathbf{R})\| \leq 1$. In practice, ASP atoms allow much richer syntax like arithmetic expression and functions. Many modern input languages for ASP extend this paradigm with cardinality constraint rules of the form $\{a_1, a_2, \ldots, a_k\} = C$ meaning exactly C of the atoms in the set $\{a_1, a_2, \ldots, a_k\}$ must be in the answer set, and aggregate functions like `sum`, `count` and `average`.

Normal ASP programs are able to express NP-complete problems [78], and have become popular in doing so due to the simplicity of the encodings obtained when one exploits the input languages of popular ASP tools like `clingo` [47]. Disjunctive ASP programs, on the other hand, can express Σ_2^p-complete problems [36] but these encodings are not as accessible as those obtained from encoding NP-complete problems in normal ASP programs.[3] A

[3] Here we refrain from providing an exact bound since this depends on a number of features that may be added to the definitions above. The survey by Dantsin et. al [27] provides details on exact bounds in the complexity and expressive power of ASP programs.

$$
\begin{aligned}
\Pi_3 &= \Pi_{\text{color}} \cup \Pi_{\text{unique}} \cup \Pi_{\text{edge}} \\
\Pi_{\text{color}} &= \{\text{color}(V, r),\ \text{color}(V, w),\ \text{color}(V, b) \leftarrow \text{vertex}(V).\} \\
\Pi_{\text{unique}} &= \{\leftarrow \text{vertex}(V),\ \text{color}(V, r),\ \text{color}(V, w). \\
&\quad \leftarrow \text{vertex}(V),\ \text{color}(V, r),\ \text{color}(V, b). \\
&\quad \leftarrow \text{vertex}(V),\ \text{color}(V, w),\ \text{color}(V, b).\} \\
\Pi_{\text{edge}} &= \{\leftarrow \text{edge}(U, V),\ \text{color}(V, r),\ \text{color}(U, r). \\
&\quad \leftarrow \text{edge}(U, V),\ \text{color}(V, w),\ \text{color}(U, w). \\
&\quad \leftarrow \text{edge}(U, V),\ \text{color}(V, b),\ \text{color}(U, b).\}
\end{aligned}
$$

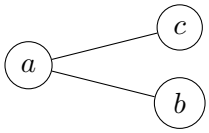
Figure 2.1: An encoding of the 3-colorability property in ASP.

more in-depth introduction to ASP can be found in Eiter, Ianni, and Krennwallner [37], including a detailed explanation of the "guess and check" approach that has become very popular for expressing NP-complete constraint problems in ASP.

2.2.1 The "Guess and Check" Approach

We illustrate the use of ASP as a constraint programming paradigm with the following example. Consider an ASP encoding of 3-colorability problem. The program Π_3 defined over predicates vertex/1, edge/2 and color/2, variables V and U, and constants r, w, and b as per Figure 2.1 encodes the problem of coloring each vertex of a graph with either red (r), white (w) or blue (b) such that two vertices connected by an edge do not have the same color. Intuitively, an explanation of the ASP program in Figure 2.1 goes like this: an ASP solver "guesses" a color for each vertex, as indicated in the program Π_{color}, and then checks that the coloring is valid according to the rules in $\Pi_{\text{unique}} \cup \Pi_{\text{edge}}$.

We can immediately observe two things about Π_3: it is not ground as it contains variables V and U and it contains no information about any specific graph. In this sense, Π_3 encodes the 3-colorability problem but does not encode an instance of the problem itself. In order to use Π_3 to solve an instance of the 3-colorability problem, one needs to provide a graph encoded in a format that is compatible with Π_3: marking every vertex with the vertex/1 predicate



(a) A graph G_{ex} on 3 vertices.

$$
\begin{aligned}
\Pi_{\text{ex}} \quad = \quad & \{ \\
& \text{vertex}(a) \leftarrow . \\
& \text{vertex}(b) \leftarrow . \\
& \text{vertex}(c) \leftarrow . \\
& \text{edge}(a, b) \leftarrow . \\
& \text{edge}(a, c) \leftarrow . \\
& \}
\end{aligned}
$$

(b) Encoding G_{ex} as an ASP program.

Figure 2.2: A graph and its encoding as an ASP program.

and every edge with the edge/2 predicate. For instance, Figure 2.2 shows a graph and an encoding of the graph in our required format.

2.2.2 The Saturation Technique

It is customary to encode a decision problem as an answer set program such that the answer sets of this program correspond to certificates for "yes" answers to the problem. Under this approach, a "no" answer is certified by the lack of an answer set. However, there are circumstances in which a "no" answer must be certified by an answer set (e.g., problems in coNP). The saturation technique (see, e.g., [37]) achieves this by designing an answer set program that has a unique answer set including a special token atom if and only if the answer to the original decision problem is "no." This answer set also contains the set S of all atoms that would be candidate certificates for "yes" answers. Rules are added so that every time the token atom is generated, all atoms in S are generated as well, thus "saturating" the model. Informally, this technique allows us to encode a "coNP check" into our program, which along with an "NP guess," allows us to encode problems in $\Sigma_2^p = \text{NP}^{\text{NP}}$, the class of problems solvable by a nondeterministic polynomial-time Turing machine with access to an NP oracle.

2.3 Symmetry Breaking for CSPs

Regardless of whether we use CSPs to determine the existence of a satisfying assignment (model) for, or to enumerate all the models of a problem encoded as a set of constraints, it is important to take into account the symmetries that exist in such an encoding. In the case of determining the satisfiability of a problem encoding, a solver may spend valuable time exploring parts of the search space searching for a model, when these parts may be symmetric to other parts that have already been explored unsuccessfully. In the case of generating all the models of a problem encoding, the number of models could be exponential in the size of the input and generating only a subset of them—namely, those which are essentially different under symmetries—may convey all the information necessary to study the entire space of models.

Symmetries in a CSP may be introduced from several sources, some of them inherent to the domain from which the problem is drawn. For instance, the encoding of a graph problem as an ASP program will inherit the symmetries (automorphisms) of the input graph. Breaking symmetries has been identified as a crucial step towards solving combinatorial problems using CSPs [21,61,65]. While the domain-specific symmetry breaking techniques that are relevant to this thesis would be better explained within the context of each problem introduced in Part II, we will explain one general-purpose syntactic symmetry breaking technique in this chapter.

The seminal work by Crawford [24, 25] addressed syntactic symmetry in CNF formulas by interpreting these as graphs and using the automorphism group of such graphs to identify syntactic symmetry. These symmetries are broken by appending symmetry breaking clauses to the input formula. Aloul et al. [2] later improved the original construction to be able to detect phase-shifts symmetries (symmetries that map a literal to its negation). Aloul et al. [3] exploited the recursive nature of the symmetry breaking predicates formulated in this technique to reduce the size of the symmetry breaking clauses added to the input formula and made their work available as the widely successful tool `Shatter`. More recently, Devriendt et al. [32] improved the symmetry breaking clauses added by `Shatter` and added another technique for symmetry breaking detection, namely row interchangeability, making these available in the `BreakID` tool. The idea of detecting syntactic symmetries through a graph representation of the input formula has also found its way into the related

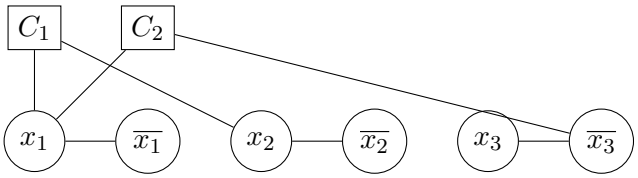


Figure 2.3: The graph corresponding to the CNF formula $(x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3})$. Vertex colors are represented by different node shapes.

paradigms of Answer Set Programming (ASP) and model expansion [33].

We lay out the fundamentals of Crawford's approach to syntactic symmetry breaking in Boolean formulas. We use the definition of CNF Boolean formulas as sets, as in Section 2.1.1.

Definition 2.3 (Graph Representation)**.** *The graph representation of a CNF formula F is a vertex-colored undirected graph G_F whose set of vertices is the union of the clauses of F in one color and the literals of F in another color. The set of edges is defined as: (a) the set of edges between a literal l and its negation $\bar{l}$ and (b) the set of edges between a clause C and the literals $l \in C$.*

Figure 2.3 shows what the graph G_F would be for a formula $F = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3})$. Recall that the automorphisms of a graph are the vertex permutations that preserve the set of edges. In our particular case, since we deal with vertex-colored graphs, we only consider color-preserving automorphisms (i.e., automorphisms that map vertices to vertices of the same color). Since literal vertices will be mapped to literal vertices, the automorphisms of G_F naturally induce a permutation of the literals of F. The main theorem behind Crawford's method for syntactic symmetry breaking is the following.

Theorem 2.4. *Given a formula F and a color-preserving automorphism ϕ of G_F, an assignment σ models a formula F if and only if the assignment $\sigma \circ \phi$ models F.*

Theorem 2.4 in itself does not break symmetries in CNF formulas. In order to use Theorem 2.4 for symmetry breaking, one can look at lexicographical orderings of assignments. Fix an ordering of the literals of F (say, $l_1, l_2, \ldots, l_n$) and let π be a permutation of these. We define the symmetry breaking predi-

cate $P(\pi)$ as follows:

$$\begin{aligned} P_1(\pi) &= l_1 \leq \pi(l_1) \\ P_i(\pi) &= \left(\bigwedge_{j=1}^{i-1} l_j \equiv \pi(l_j) \right) \rightarrow (l_i \leq \pi(l_i)) \text{ for } i > 1 \\ P(\pi) &= \bigwedge_{i=1}^{n} P_i(\pi) \end{aligned}$$

Predicate $P(\pi)$ imposes a lexicographical order between two different assignments since only one of them will satisfy $P(\pi)$. Since, by Theorem 2.4, it is true that for any satisfying assignment σ and any automorphism ϕ both σ and $\sigma \circ \phi$ satisfy F, $P(\phi)$ allows us to prefer σ over $\sigma \circ \phi$ if σ is less or equal to $\sigma \circ \phi$ in lexicographical order. If F is satisfiable, such an assignment exists because $P(\pi)$ is a total order over the satisfying assignments of F. By adding symmetry breaking predicates we are able to reduce the number of satisfying assignments to search for while preserving satisfiability.

Theorem 2.5. *Given a formula F and a color-preserving automorphism ϕ of G_F, F is satisfiable if and only if $(F) \wedge P(\phi)$ is satisfiable.*

One can in fact add symmetry breaking predicates for any number of automorphisms while preserving satisfiability: observe that the lexicographical minimum of the set of models of F is by definition less or equal to all of its permutations, thus it satisfies all possible symmetry breaking predicates.

Chapter 3

Verified Computations

Abstract combinatorial problems that remain open are such that any solution, or even significant progress towards their solution, will most likely require ground-breaking mathematical results or heavy computational methods. In fact, some will require a combination of both. While hand-written mathematical results have been traditionally accepted by researchers in combinatorics via established peer-review processes, computational methods have been under fire for a long time. Only recently have mathematicians started to become more accepting of results obtained with non-trivial help from computers. The issue with the approach of using computers to explore a large search space until a solution is found is that it is hard to argue that no solution exists when computers cannot find one. Not finding a solution may just be an indication of an error in the software used to search.

A seminal example of the dilemma regarding computer-assisted proofs is the four color theorem, which states that every map can be colored using four colors so that bordering regions do not share the same color. Proving this seemingly simple problem proved to be surprisingly difficult for purely analytical tools. In 1976, Appel and Haken [6, 7] published a proof that relied on dozens of pages of tedious manual calculations together with computer code that was hard to verify. The crux of the argument was the inexistence of a counterexample, i.e., a map that cannot be colored with just four colors. This proof was, to many researchers, insufficient because it fell out of the peer-review culture of the field. The concerns around this proof were somewhat alleviated after Robertson et al. [96] published a simpler—still computer

assisted—proof in 1995.

The controversy about the validity of previous proofs of the four color theorem was largely settled in 2005 when Gonthier [50] provided a proof of this theorem using the Coq interactive theorem prover. The fact that a proof assistant was used to produce this result shifts the concerns of validity from the proof itself to the interactive theorem prover, the latter being widely used as a general tool by many researchers and thus less likely to have defects that could impact the soundness of the proof.

Verified computations can be performed in several ways. Two ways that are relevant to our purposes are verified tools and verified computations. We take a closer look at these two approaches as researchers in combinatorial computing seeking to incorporate constraint satisfaction techniques into their toolbox will likely need them.

3.1 Verified Tools

In the process of obtaining problem encodings for combinatorial problems, we use code that automates conversions, preprocesses instances and transforms outputs. Each piece of such a toolkit introduces trust issues since they could potentially be flawed. An approach that increases the trust in these tools is to obtain them through formal methods. Theorem proving systems like Isabelle[1] and Coq[2] provide code extraction mechanisms. These are mechanisms that allows us to verify the property of the code we intend to run via theorems whose proofs can be mechanically verified, and then extract executable code from these specifications. This approach has been used to obtain verified SAT solvers [79], verified algorithms for and-inverter graphs [28] which are becoming a popular input format for QBF solvers (see Section 2.1.5), among other tools.

3.2 Verified Results

This approach implements mechanisms by which an answer is provided together with a certificate of such an answer. It is, for example, easy to see how

[1] http://isabelle.in.tum.de/
[2] https://coq.inria.fr/

this idea plays out in the case of satisfiable Boolean formulas: it is enough to provide an assignment of the variables that satisfies the formula. It is, however, harder to devise a mechanism by which we could return the answer "unsatisfiable" and provide, together with such an answer, a proof of unsatisfiability. Several approaches have been developed in recent years, spurred by the increasing use of SAT solvers in mission-critical scenarios. One way to achieve this is to provide a minimal unsatisfiable subset of the input clauses [88] (assuming an input formula in CNF). If such a set is small enough to make the unsatisfiability of the original formula evident, then this approach is satisfactory. Nevertheless, this is a strong assumption, as the minimal unsatisfiable subset of the input clauses may be the set of input clauses themselves. An alternative approach is to provide a proof of unsatisfiability. Perhaps the simplest proof of unsatisfiability one can think of is through *resolution*. Resolution is a mechanism by which one can derive new clauses by combining clauses from a database based on certain rules. Since it is known that resolution of propositional logic is complete, one can always derive the empty clause from a set of unsatisfiable CNF clauses using resolution. The downside of this simple idea is that resolution proofs can be very long, and many techniques have been developed that produce proofs that are much shorter. Of particular interest to our research is the DRAT-trim [111] format which has been successful in recent massive efforts to use SAT solvers to obtain results in combinatorics [60]. DRAT-trim has in turn inspired the development of proof formats for other constraint satisfaction paradigms [39] which overall contributes to solidify results in combinatorics obtained through these methods.

Notice that verifying a certificate of unsatisfiability for any interesting instance is not a task we can do by hand, since these certificates would be very long. Verifying these certificats thus requires another computer program. The crucial aspect about this approach is that a verifier would be much simpler than the computer program that actually produced the certificate (e.g., a QBF solver). Obtaining a verifier through the methods highlighted in Section 3.1 thus completes the trust chain. Furthermore, agreeing on a format for a certificate allows us to write one verifier that can then be used to verify results from any solver within a specific CSP paradigm. This goal has been achieved for the DRAT [59] and LRAT [26] proof formats by generating verified checkers using code extracted from interactive theorem provers.

Part II

Our Contributions

Chapter 4

Backbones and Backdoors of Boolean Formulas

Many algorithms for the Boolean satisfiability problem exploit hidden structural properties of formulas in order to find a satisfying assignment or prove that no such assignment exists. These structural properties are called hidden because they are not explicit in the input formula. A natural question that arises then is what the computational complexity associated with these hidden structures is. In this chapter we focus on two hidden structures: backbones[1] and strong backdoors [113].

The complexity of decision problems associated with backdoors and backbones has been studied by Nishimura, Ragde, and Szeider [87], Kilby, Slaney, Thiébaux, and Walsh [72], and Dilkina, Gomes, and Sabharwal [34], among others. We mention in passing that backbones also are very interesting for problems even harder than SAT, such as #SAT [108] (see Gomes, Sabharwal, and Selman [49]).

The goal of this chapter is to understand, at least in a theoretical sense, the potential obstacles to using backbones and backdoors in helping SAT solvers. Our results speak both to the importance of not viewing backdoors or backbones as magically transparent—we prove that they are in some cases rather opaque—and to the fact that the behavior we mention likely happens on quite dense sets. Further, since we tie this to whether any set is densely hard, these

[1]The term *backbone* was first used by Monasson et al. [84].

SAT-solver issues due to this chapter have now become inextricably linked to the extremely important, long-open question of how resistant to polynomial-time heuristics the sets in a complexity class can be.

The following section presents definitions and notation, and then the sections after that will cover our results and related work.

4.1 Definitions and Notation

Throughout this chapter we follow the notation of Section 2.1 for Boolean formulas. For a finite set A, $\|A\|$ denotes A's cardinality. For any string x, $|x|$ denotes the length of (number of characters of) x. For each set T of strings and each natural number n, $T^{\leq n}$ denotes the set of all strings in T whose length is less than or equal to n. In particular, $(\Sigma^*)^{\leq n}$ denotes the strings of length at most n, over the alphabet Σ.

4.2 Backbones of Satisfiable Formulas

Recall from Section 2.1 that a partial assignment of a Boolean formula F is a function $a_S : S \to \{\top, \bot\}$ that assigns Boolean values to the variables in a set $S \subseteq V(F)$.

Definition 4.1. *Let F be a Boolean formula. A set S of the variables of F is said to be a* backbone *if there is a unique partial assignment a_S such that $F[a_S]$ is satisfiable.*

A backbone S is *nontrivial* if $S \neq \emptyset$. The *size* of a backbone S is the number of variables in S. For a backbone S (for formula F), we say that a_S is the *value* of the backbone S.

For example, every satisfiable formula has the trivial backbone $S = \emptyset$. The formula $x_1 \wedge \overline{x_2}$ has four backbones, $\emptyset$, $\{x_1\}$, $\{x_2\}$, and $\{x_1, x_2\}$, with respectively the values (listing values as bit-vectors giving the assignments in the lexicographical order of the names of the variables in S) ϵ, 1, 0, and 10. The formula $x_1 \vee \overline{x_2}$ has no nontrivial backbones.

It follows from Definition 4.1 that unsatisfiable formulas do not have backbones. Note, however, that the original paper by Monasson et al. introducing the concept of backbones [84] defines a backbone as a set of variables that

are "fully constrained" by the formula, a definition that does not require the formula to be satisfiable.

Example 4.2. *Consider the formula* $F = x_1 \wedge (x_1 \leftrightarrow \overline{x_2}) \wedge (x_2 \leftrightarrow x_3) \wedge (x_2 \vee x_4 \vee x_5)$. *Any satisfying assignment of* F *must have* x_1 *set to* $\top$, *which in turn constrains* x_2 *and* x_3. *Then* $\{x_1, x_2, x_3\}$ *is a backbone of* F, *as is any subset of this backbone. It is also easy to see that* $\{x_1, x_2, x_3\}$ *is the largest backbone of this formula since the truth values of* x_4 *and* x_5 *are not entirely constrained in* F *(since* F *in effect is—once one applies the just-mentioned forced assignments—*$x_4 \vee x_5$*).*

In previous literature, especially that pertaining to statistical physics where backbones were first observed, the variables in a backbone have been called "frozen variables." [83, 99] This is because their values are "frozen", i.e., each of them is the same over all satisfying assignments.

4.2.1 Results Under a Strong Assumption

In this section we will show that even for cases when one can quickly (i.e., in polynomial time) recognize that a formula has at least one nontrivial backbone, it can be intractable to find one such backbone. And we will show that even for cases when one can quickly (i.e., in polynomial time) find a large, nontrivial backbone, it can be intractable to find the value of that backbone. In particular, we will show that if integer factoring is hard, then both the just-made claims hold. Integer factoring is widely believed to be hard; indeed, if it were in polynomial time, RSA (the Rivest-Shamir-Adleman cryptosystem) itself would fall.

In fact, integer factoring is even believed to be hard on average. And we will be inspired by that to go beyond the strength of the results mentioned above. We will argue that if any problem in NP $\cap$ coNP is frequently hard, then the bad behavior types we mention above happen "almost" as often: If the frequency of hardness of integer factoring is $d(n)$ for strings up to length n, then for some $\kappa > 0$ the frequency of hardness of our problems is $d(n^\kappa)$.

Core Results

We first look at whether there can be simple sets of formulas for which one can easily compute/obtain a nontrivial backbone, yet one cannot easily find

the value of that backbone.

Our basic result on this is stated below as Theorem 4.3. (In this section we do not assume that SAT by definition is restricted to CNF formulas.)

Theorem 4.3. *Let $k \in \{1, 2, 3, \ldots\}$. If $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$, then there exists a set $A \in \mathrm{P}$, $A \subseteq SAT$, of Boolean formulas such that:*

1. *There is a polynomial-time computable function f such that for all $F \in A$, $f(F)$ outputs a size-k backbone of F.*

2. *There does not exist any polynomial-time computable function g such that $g(F)$ computes the value of backbone $f(F)$.*

Proof. Let B be some set in $(\mathrm{NP} \cap \mathrm{coNP}) - \mathrm{P}$. The Cook-Levin theorem states that we can efficiently transform the question of whether a nondeterministic Turing machine accepts a particular string into a question about whether a certain Boolean formula is satisfiable [22, 71, 75]. The original work that did that did not require that the thus-created Boolean formula transparently revealed what machine and input had been the input to the transformation. But it was soon noted by Galil that one can ensure that the formula mapped to transparently reveals the machine and input that were the input to the transformation [43].

Galil's insight can be summarized in the following strengthened version of the standard claim regarding the so-called Cook-Karp-Levin Reduction. Let N_1, N_2, ... be a fixed, standard enumeration of clocked, polynomial-time nondeterministic Turing machines, and w.l.o.g. assume that N_i runs within time $n^i + i$ on inputs of length n, and that N_i and i are polynomially related in size and easily obtained from each other. There is a function r_{GC} such that

1. For each N_i and x: $x \in L(N_i)$ if and only if $r_{GC}(N_i, x) \in \mathrm{SAT}$.

2. There is a polynomial p such that $r_{GC}(N_i, x)$ runs within time polynomial (in particular, with p being the polynomial) in $|N_i|$ and $|x|^i + i$.

3. There is a polynomial-time function s such that for each N_i and x, $s(r_{GC}(N_i, x))$ outputs the pair (N_i, x).

We will be using two separate applications of the r function in our construction. But we need those two applications to be variable-disjoint. We

will, w.l.o.g., assume that in the output of the Galil-Cook function $r_{GC}(N_i, x)$, every variable is of the form x_j, where j itself, when viewed as a pair of integers via the standard fixed correspondence between $\mathcal{Z}^+$ and $\mathcal{Z}^+ \times \mathcal{Z}^+$, has the natural number corresponding to N_i in the standard fixed correspondence between positive integers and strings. We claim that one can implement a legal Galil-Cook r function in such a way that it has this property yet still has the property that this r function will have a polynomial-time inversion function s satisfying the behavior for s mentioned above. (For those wanting more information on how such a function $r_{GC}(N_i, x)$ can be implemented that has all the properties claimed above, you can refer to the Appendix of [57] for a detailed construction we have built that accomplishes this.)

We now can specify the set A needed. Recall we have fixed a set $B \in (\text{NP} \cap \text{coNP}) - \text{P}$. $B \in \text{NP}$ so let i be a positive integer such that N_i is a machine from the abovementioned standard enumeration such that $L(N_i) = B$. $\overline{B} \in \text{NP}$ so let j be a positive integer such that N_j is a machine from the abovementioned standard enumeration, such that $L(N_j) = \overline{B}$. Fix any positive integer k from the theorem statement. Then for the case of that fixed value k, the set A is as follows: $A = A_k = \{(z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge r_{GC}(N_i, x)) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge (r_{GC}(N_j, x))) \mid x \in \Sigma^*\}$. One must keep in mind in what follows that, as per the previous paragraph, r_{GC} never outputs literals with names involving subscripted zs or z's and the outputs of $r_{GC}(N_i, x)$ and $r_{GC}(N_j, x)$ share no variable names (since $i \neq j$).

Let us argue that A_k indeed satisfies the requirements of the A for the "k" case of the theorem.

$A_k \in$ P: Given a string y whose membership in A we are testing, we make sure y syntactically matches the form of the elements of A (i.e., elements of $A_{3,k}$). If it does, we then check that its k matches our k, and we use s to get decoded pairs (i', x') and (j'', x'') from the places in our parsing of y where we have formulas—call them F_{left} and F_{right}—that we are hoping are the outputs of the r function. That is, if our input parses as $(z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge F_{left}) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge F_{right})$, then if $s(F_{left})$ gives $(N_{i'}, x')$ our decoded pair is (i', x'), and F_{right} is handled analogously. We also check to make sure that $x' = x''$, $i = i'$, and $j = j''$. If anything mentioned so far fails, then $y \notin A$. Otherwise, we check to make sure that $r_{GC}(N_i, x') = F_{left}$ and $r_{GC}(N_j, x') = F_{right}$, and reject if either equality fails to hold. If we have reached this point, we indeed have determined in polynomial time that $y \in A$, and for each $y \in A$

we will successfully reach this point.

$A_k \subseteq$ SAT: For each x, either $x \in B$ or $x \notin B$. In the former case ($x \in B$), $r_{GC}(N_i, x) \in$ SAT and so the left disjunct of $(z_1 \wedge \cdots \wedge z_k \wedge (r_{GC}(N_i, x))) \vee (\overline{z_1} \wedge \cdots \wedge \overline{z_k} \wedge (r_{GC}(N_j, x)))$ can be made true using that satisfying assignment and setting each z_ℓ to true. In the latter case ($x \notin B$), $r_{GC}(N_j, x) \in$ SAT and so the whole formula can be made true using that satisfying assignment and setting each z_ℓ to false.

There is a polynomial-time computable function f such that for all $F \in A_k$, $f(F)$ outputs a nontrivial backbone of F: On input $F \in A_k$, f will simply output $\{z_1, z_2, \ldots, z_k\}$, which we claim is a nontrivial backbone of F. To prove our claim, notice that if the x embedded in F satisfies $x \in B$, then not only does $r_{GC}(N_i, x) \in$ SAT hold, but also $r_{GC}(N_j, x) \notin$ SAT must hold (since otherwise we would have $x \notin B \wedge x \in B$, an impossibility). So if the x embedded in F satisfies $x \in B$, then there are satisfying assignments of $(z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge r_{GC}(N_i, x)) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge r_{GC}(N_j, x))$, and every one of them has each z_ℓ set to $\top$. Similarly, if the x embedded in F satisfies $x \notin B$, then our long formula has satisfying assignments, and every one of them has each z_ℓ set to $\bot$. Thus $\{z_1, z_2, \ldots, z_k\}$ indeed is a size-k backbone.

There does not exist any polynomial-time computable function g such that $g(F)$ computes the value of backbone $f(F)$: Suppose by way of contradiction that such a polynomial-time computable function g does exist. Then we would have that $B \in$ P, by the following algorithm. Let f be the function constructed in the previous paragraph, i.e., the one that outputs $\{z_1, z_2, \ldots, z_k\}$ when $F \in A$. Given x, in polynomial time—g and f are polynomial-time computable, and although r in general is not since its running time's polynomial degree varies with its first argument and so is not uniformly polynomial, r here is used only for the first-component values N_i and N_j and under that restriction it indeed is polynomial-time computable—compute $g(f((z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge r_{GC}(N_i, x)) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge r_{GC}(N_j, x))))$. This must either tell us that the z_ℓs are $\top$ in all satisfying assignments, which tells us that it is the left disjunct that is satisfiable and thus $x \in B$, or it will tell us that the z_ℓs are $\bot$ in all satisfying assignments, from which we similarly can correctly conclude that $x \notin B$. So $B \in$ P, yet we chose B so as to satisfy $B \in (\text{NP} \cap \text{coNP}) - \text{P}$. Thus our assumption that such a g exists is contradicted. □

That ends our proof of Theorem 4.3. The following corollary follows immediately from Theorem 4.3.

Corollary 4.4. *If* $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$*, then there exists a set* $A \in \mathrm{P}$*,* $A \subseteq SAT$*, of Boolean formulas such that:*

1. *There is a polynomial-time computable function* f *such that for all* $F \in A$*,* $f(F)$ *outputs a nontrivial backbone of* F*.*

2. *There does not exist any polynomial-time computable function* g *such that* $g(F)$ *computes the value of backbone* $f(F)$*.*

Now let us turn to the question of whether, when it is obvious that there is at least one nontrivial backbone, it can be hard to efficiently produce a nontrivial backbone. The following theorem shows that, if integer factoring is hard, the answer is yes.

Theorem 4.5. *For each fixed* ϵ*,* $0 < \epsilon \leq 1$*, the following claim holds. If* $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$*, then there exists a set* $A \in \mathrm{P}$*,* $A \subseteq SAT$*, of Boolean formulas, each having at least one variable, such that:*

1. *Each formula* $F \in A$ *has a backbone whose size is at least* $(50 - \epsilon)\%$ *of* F*'s total number of variables.*

2. *There does not exist any polynomial-time computable function* g *such that, on each* $F \in A$*,* $g(F)$ *outputs a backbone whose size is at least* $(2\epsilon)\%$ *of* F*'s variables.*

Proof. We describe how to convert the construction in the proof of Theorem 4.3 into one that proves Theorem 4.5. Recall that for the "k" case of Theorem 4.3 our set A was

$$A_k = \{(z_1 \wedge \cdots \wedge z_k \wedge r_{GC}(N_i, x)) \vee (\overline{z_1} \wedge \cdots \wedge \overline{z_k} \wedge r_{GC}(N_j, x)) \mid x \in \Sigma^*\}.$$

Let us use almost the same set, except we will make two types of changes. First, in the above, replace the two occurrences of k each with the smallest positive integer m' satisfying $\frac{m'}{\|V(r_{GC}(N_i,x))\|+\|V(r_{GC}(N_j,x))\|+2m'} \geq \frac{(50-\epsilon)}{100}$. Let m henceforward denote that value, i.e., the smallest (positive integer) m' that satisfies the above equation. Second, in the right disjunct, change each $\overline{z_\ell}$ to $\overline{z'_\ell}$.

Note that if $x \in B$, then $\{z_1, z_2, \cdots, z_m\}$ is a backbone whose value is the assignment of $\top$ to each variable, and that contains at least $(50 - \epsilon)\%$ of the variables in the formula that x put into A. Similarly, if $x \notin B$, then $\{z'_1, z'_2, \cdots, z'_m\}$ is a backbone whose value is the assignment of $\bot$ to each variable, and that contains at least $(50 - \epsilon)\%$ of the variables in the formula that x put into A. It also is straightforward to see that our thus-created set A belong to P and satisfies $A \subseteq$ SAT.

So the only condition of Theorem 4.5 that we still need to show holds is the claim that, for the just-described A, there does not exist any polynomial-time computable function g such that, on each $F \in A$, $g(F)$ outputs a backbone whose size is at least $2\epsilon\%$ of F's variables. Suppose by way of contradiction that such a function g does exist. We claim that would yield a polynomial-time algorithm for B, contradicting the assumption that $B \notin$ P. Let us give such a polynomial-time algorithm. To test whether $x \in B$, in polynomial time we create the formula in A that is put there by x, and we run our postulated polynomial-time g on that formula, and thus we get a backbone, call it S, that contains at least $2\epsilon\%$ of F's variables. Note that we ourselves do not get to choose which large backbone g outputs, so we must be careful as to what we assume about the output backbone. We in particular certainly cannot assume that g happens to always output either $\{z_1, z_2, \cdots, z_m\}$ or $\{z'_1, z'_2, \cdots, z'_m\}$. But we don't need it to. Note that the two backbones just mentioned are variable-disjoint, and each contains $(50 - \epsilon)\%$ of F's variables.

Now, there are two cases. One case is that S contains at least one variable of the form z_ℓ or z'_ℓ. If it contains at least one variable of the form z_ℓ then $x \in B$. Why? If $x \in B$, then the left-hand disjunct of the formula x puts into A is satisfiable and the right-hand disjunct is not. From the form of the formula, it is clear that each z_ℓ is always $\top$ in each satisfying assignment in this case, yet that for each z'_ℓ there are satisfying assignments where z'_ℓ is $\top$ and there are satisfying assignments where z'_ℓ is $\bot$. So if $x \in B$, no z'_ℓ can belong to any backbone.

By analogous reasoning, if S contains at least one variable of the form z'_ℓ then $x \notin B$. (It follows from this and the above that S cannot possibly contain at least one variable that is a subscripted z and at least one variable that is a subscripted z', since then x would have to simultaneously belong and not belong to B.)

The final case to consider is the one in which S does not contain at least

one variable of the form z_ℓ or z'_ℓ. We argue that this case cannot happen. If this were to happen, then every variable of F other than the variables $\{z_1, z_2, \cdots, z_m, z'_1, z'_2, \cdots, z'_m\}$ must be part of the backbone, since S must involve 2% of the variables and $\{z_1, z_2, \cdots, z_m, z'_1, z'_2, \cdots, z'_m\}$ comprise $(100-2\epsilon)\%$ of the variables. But that is impossible. We know that the variables used in $r_{GC}(N_i, x)$ and $r_{GC}(N_j, x)$ are disjoint. So the variables in the one of those two that is not the one that is satisfiable can and do take on any value in some satisfying assignment, and so cannot be part of any backbone. (The only remaining worry is the case where one of $r_{GC}(N_i, x)$ or $r_{GC}(N_j, x)$ contains no variables. However, the empty formula is by convention considered illegal, in cases such as here where the formulas are not considered to be trapped into DNF or CNF.) We have thus concluded the proof of Theorem 4.5. □

Frequency of Hardness

In this section we show that if *even one problem* in NP ∩ coNP is frequently hard, then the sets in our previous sections can be made "almost" as frequently hard, in a sense of "almost" that we will make formal and specific. Note that no one currently knows for sure how frequently-hard problems in NP ∩ coNP can be. But our results are showing that, whatever that frequency is, sets of the sort we've been constructing are hard "almost" as frequently.

We now give our frequency-of-hardness version of Theorem 4.3. A claim is said to hold *for almost every* n if there exists an n_0 beyond which the claims always holds, i.e., the claim fails at most at a finite number of values of n. (In the theorems of this section, n's universe is the natural numbers, $\{0, 1, 2, \ldots\}$.) A (decision) algorithm errs with respect to B on an input x if the algorithm disagrees with B on x, i.e., if the algorithm accepts x yet $x \notin B$ or the algorithm rejects x yet $x \in B$. We will defer the proving of this section's theorems until the end of this section, where we will argue that the results in effect follow from the constructions of the previous section.

Theorem 4.6. *Let $k \in \{1, 2, 3, \ldots\}$. If h is any nondecreasing function and for some $B \in$ NP ∩ coNP it holds that each polynomial-time algorithm, viewed as a heuristic algorithm for testing membership in B, for almost every n (respectively, for infinitely many n) errs on at least $h(n)$ of the strings whose length is at most n, then there exist a $\kappa > 0$ and a set $A \in$ P, $A \subseteq SAT$, of Boolean formulas such that:*

1. *There is a polynomial-time computable function f such that for all $F \in A$, $f(F)$ outputs a size-k backbone of F.*

2. *Each polynomial-time computable function g will err (i.e., will fail to compute the value of backbone $f(F)$), for almost every n (respectively, for infinitely many n), on at least $h(n^\kappa)$ of the strings in A of length at most n.*

The precisely analogous result holds for Corollary 4.4. The analogous result also holds for Theorem 4.5:

Theorem 4.7. *For each fixed ϵ, $0 < \epsilon \leq 1$, the following claim holds. If h is any nondecreasing function and for some $B \in \mathrm{NP} \cap \mathrm{coNP}$ it holds that each polynomial-time algorithm, viewed as a heuristic algorithm for testing membership in B, for almost every n(respectively, for infinitely many n) errs on at least $h(n)$ of the strings whose length is at most n, then there exist a $\kappa > 0$ and a set $A \in \mathrm{P}$, $A \subseteq SAT$, of Boolean formulas such that:*

1. *Each formula $F \in A$ has a backbone whose size is at least $(50 - \epsilon)\%$ of F's total number of variables.*

2. *Each polynomial-time computable function g will err (i.e., will fail to compute a set of size at least $2\epsilon\%$ of F's variables that is a backbone of F), for almost every n (respectively, for infinitely many n), on at least $h(n^\kappa)$ of the strings in A of length at most n.*

What the above theorems say, looking at the contrapositives to the above results, is that if any of our above cases have polynomial-time heuristic algorithms that don't make errors too frequently, then *every single set in* NP ∩ coNP (even those related to integer factoring) has polynomial-time heuristic algorithms that don't make errors too frequently.

Let us give concrete examples that give a sense about what these theorems are saying about density transference. It follows from Theorems 4.6 and 4.7 that if there exists even one set in NP ∩ coNP such that each polynomial-time heuristic algorithm asymptotically errs exponentially often up to each length (i.e., has $2^{n^{\Omega(1)}}$ errors), then there are sets of our form that in the same sense fool each polynomial-time heuristic algorithm exponentially often. As a second example, it follows from Theorems 4.6 and 4.7 that if there exists

even one set in NP ∩ coNP such that each polynomial-time heuristic algorithm asymptotically errs quasipolynomially often up to each length (i.e., has $n^{(\log n)^{\Omega(1)}}$ errors), then there are sets of our form that in the same sense fool each polynomial-time heuristic algorithm quasipolynomially often.

Our use of B and A here reflects that of the theorems in this section. The crucial thing to note is that the mapping from strings x (as to whether they belong to B) into the string that x puts into A is (a) polynomial-time computable (and so the one string that x puts into A is at most polynomially longer than x), and (b) one-to-one.

So any collection of m instances up to a given length n that fool a particular polynomial-time algorithm for B is associated with a collection of at least m distinct instances in A all of length at most n^q (where the polynomial bound on the length of the formula that x, $|x| = n$, puts into A is that its length is n^q or less [2]). So if one had an algorithm for the "A" set such that the algorithm had at most m' errors on the strings up to length n^q, it would certainly imply an algorithm for B that up to length $n^{1/q}$ made at most m' errors. Namely, one's heuristic of that form for B would be to take x, map it to the string it put into A, and then run the heuristic for A on that string. The results of this section are the immediate consequences of this observation, applied to the constructions/results of the previous section. To make completely clear that that is the case and why it is the case, we now provide a more detailed explanation of the proof of one of this section's theorems, namely, Theorem 4.6.

Proof of Theorem 4.6. . Let A be defined as A_k in Section 4.2.1, i.e.: $A = A_k = \{(z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge (r_{GC}(N_i, x))) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge (r_{GC}(N_j, x))) \mid x \in \Sigma^*\}$ where N_i and N_j are Turing machines such that $L(N_i) = B$ and $L(N_j) = \overline{B}$. We know from our discussion in Section 4.2.1 that $A \in$ P and that, given any formula $F \in A$, $\{z_1, z_2, \ldots, z_k\}$ is a nontrivial backbone of that formula, thus the function $f(F) = \{z_1, z_2, \ldots, z_k\}$ satisfies the requirements of the first part of Theorem 4.6.

Since $r_{\textit{Galil-Cook}}$ runs in polynomial time, there exist polynomials p_i and p_j of equal degree q such that the length of the formula $r_{\textit{Galil-Cook}}(N_i, x)$ is at

[2] We have for simplicity left out any lower-order terms and the leading-term constant, but that is legal except at $n \in \{0, 1\}$—since starting with $n = 2$ we can boost q if needed—and no finite set of values, such as $\{0, 1\}$ can cause problems to our theorem, as it is about the "infinitely-often" and "almost-everywhere" cases.

most $p_i(|x|)$ and the length of the formula $r_{Galil\text{-}Cook}(N_j, x)$ is at most $p_j(|x|)$. Note that there will exist natural numbers c and N such that for all $n > N$,

$$c\,n^q \geq |(z_1 \wedge \cdots \wedge z_k \wedge (r_{GC}(N_i, x))) \vee (\overline{z_1} \wedge \cdots \wedge \overline{z_k} \wedge (r_{GC}(N_j, x)))|$$

for all strings x whose length is at most n. Let $n_B > N$ be a natural number such that every polynomial-time algorithm, viewed as a heuristic for testing membership in B, errs on at least $h(n_B)$ of the strings whose length is at most n_B. We claim that every polynomial-time algorithm, viewed as a heuristic for computing the value of $f(F) = \{z_1, z_2, \ldots, z_k\}$ for any $F \in A$, errs on at least $h(n_B)$ of the strings whose length is at most $c\,n_B^q$. Replacing $n_A = c\,n_B^q$ in our claim, we have that every polynomial-time algorithm, viewed as a heuristic for computing the value of $f(F) = \{z_1, z_2, \ldots, z_k\}$ for any $F \in A$, errs on at least $h\left(c^{-\frac{1}{q}}\, n_A^{\frac{1}{q}}\right)$ of the strings whose length is at most n_A. For any $\delta > 0$ it certainly holds that, for almost all n, $c^{-\frac{1}{q}}\, n^{\frac{1}{q}} \geq n^{\frac{1}{q}-\delta}$, and thus that, since h is nondecreasing, for almost all n, $h\left(c^{-\frac{1}{q}}\, n^{\frac{1}{q}}\right) \geq h\left(n^{\frac{1}{q}-\delta}\right)$. From the assumptions we have that almost every $n > N$ can take the role of n_B (respectively, infinitely many $n > N$ can take the role of n_B). Thus setting $\kappa = \frac{1}{q} - \delta$ proves that A satisfies the second part of Theorem 4.6.

To prove our claim, notice that, by our choice of n_B, for all inputs x of length at most n_B the length of $(z_1 \wedge \cdots \wedge z_k \wedge (r_{GC}(N_i, x))) \vee (\overline{z_1} \wedge \cdots \wedge \overline{z_k} \wedge (r_{GC}(N_j, x)))$ is at most $c\,n_B^q$. Assume, by contradiction, that there exists a polynomial-time algorithm g' that, viewed as a heuristic for computing the value of $f(F) = \{z_1, z_2, \ldots, z_k\}$ for any $F \in A$, errs on less than $h(n_B)$ of the strings of length at most $c\,n_B^q$. Consider the following polynomial-time heuristic for testing membership in B: on input x, calculate $v = g'((z_1 \wedge z_2 \wedge \cdots \wedge z_k \wedge (r_{GC}(N_i, x))) \vee (\overline{z_1} \wedge \overline{z_2} \wedge \cdots \wedge \overline{z_k} \wedge (r_{GC}(N_j, x))))$; if v sets all the z_is to $\top$ then output $x \in B$ and if v sets all the z_is to $\bot$ output $x \notin B$. (g' will not set some of the z_is to $\top$ and some to $\bot$ since that would certainly be wrong.) Based on our discussion in Section 4.2.1, this heuristic will err exactly when g' errs since, for instance, if v sets all the z_is to $\top$ but the correct value sets all the z_is to $\bot$ that would imply $x \notin B$. But g' errs in less than $h(n_B)$ inputs of length at most $c\,n_B^q$ so the polynomial-time heuristic we just constructed errs in less than $h(n_B)$ inputs x of length at most n_B, a contradiction. □

4.2.2 Results under a Weak Assumption

Our results in Section 4.2.1 rely on the assumption that P $\neq$ NP$\cap$coNP, which as noted above is likely true, since if it is false then integer factoring is in P and the RSA encryption scheme falls. Under the (less demanding) assumption that P $\neq$ NP, there are families of formulas that are easy to recognize (i.e., they can be recognized by polynomial-time algorithms) yet no polynomial-time algorithm can, given a formula from the family, decide whether the formula has a large backbone (doing so is NP-complete).

Our first result states that if P $\neq$ NP then there are families of Boolean formulas that are easy to recognize, with the property that deciding whether a formula in these families has a large backbone is NP-complete (and so is hard).

Theorem 4.8. *For any real number $0 < \beta < 1$, there is a set $A \in$ P of Boolean formulas such that the language*

$$L_A = \{F \mid F \in A \text{ and } F \text{ has a backbone } S \text{ with } \|S\| \geq \beta\|V(F)\|\}$$

is NP*-complete (and so if* P $\neq$ NP *then L_A is not in* P*).*

Proof. Fix a β from Theorem 4.8's statement. For each Boolean formula G, let

$$q(G) = \left\lceil \frac{\beta\|V(G)\|}{1-\beta} \right\rceil.$$

Define

$$A = \{(G) \wedge (\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(G)}) \mid \|V(G)\| > 0\},$$

where we define new_i is the ith (in lexicographical order) variable name that does not appear in F. Note that $\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(G)}$ is a backbone if and only if $G \in$ SAT, thus under the assumption that P $\neq$ NP and keeping in mind that for zero-variable formulas satisfiability is easy to decide, it follows that no polynomial-time algorithm can decide L_A, since the size of this backbone is $q(G) > 0$, which by our definition of q will satisfy the condition $\|S\| \geq \beta\|V(F)\|$, since $\|S\| = q(G)$ and $\|V(F)\| = \|V(G)\| + q(G)$ so the condition is claiming that $q(G) \geq \beta(\|V(G)\| + q(G))$, or equivalently, that $q(G) \geq \frac{\beta}{(1-\beta)}\|V(G)\|$, which indeed holds in light of the definition of q. Note

that SAT many-one polynomial-time reduces to L_A via the reduction $g(H)$ that equals some fixed string in L_A if H is in SAT and $H = \top$ and that equals some fixed string in $\overline{L_A}$ if $H = \bot$, and that equals

$$H \wedge (\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(H)})$$

otherwise. Since L_A is in NP,[3] we have that it is NP-complete, and since $\mathrm{P} \neq \mathrm{NP}$ was part of the theorem's hypothesis, L_A cannot be in P. □

As a corollary to the proof of Theorem 4.8, we have that if $\mathrm{P} \neq \mathrm{NP}$ then there are families of Boolean formulas that are easy to recognize, with the property that deciding whether a formula in these families has a nontrivial backbone is NP-complete (and so is hard).

Corollary 4.9. *There is a set $A \in \mathrm{P}$ of Boolean formulas such that the language*

$$L_A = \{F \mid F \in A \text{ and } F \text{ has a nontrivial backbone } S\}$$

is NP*-complete (and so if* $\mathrm{P} \neq \mathrm{NP}$ *then L_A is not in* P*).*

Proof. The set A from the proof of Theorem 4.8 is constructed in such a way that each of its potential members $G \wedge (\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(G)})$ (where G is a Boolean formula having at least one variable) either has no nontrivial backbone (indeed, no backbone) or has a backbone of size at least $\beta(\|V(G)\|)$. Thus the issue of backbones that are nontrivial but smaller than $\beta(\|V(F)\|)$, where F is $G \wedge (\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(G)})$, does not cause a problem under the construction. That is, our A (which itself is dependent on the value of β one is interested in) is such that we have ensured that $\{F \mid F \in A \text{ and } F \text{ has a nontrivial backbone } S\} = \{F \mid F \in A \text{ and } F \text{ has a backbone } S \text{ with } \|S\| \geq \beta\|V(F)\|\}$. □

We now address the potential concern that the hard instances for the decision problems we just introduced may be so infrequent that the relevance of Theorem 4.8 and Corollary 4.9 is undercut. The following theorem argues

[3] If one just looks at the definition of L_A, one might worry that L_A might have only $\mathrm{NP}^{\mathrm{NP}}$ as an obvious upper bound. However, as noted above our particular choice of A ensures that $\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(H)}$ is a backbone of $(H) \wedge (\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(H)})$ if and only if $H \in$ SAT; and that makes clear that our set is indeed in NP.

against that possibility by proving that, unless not a single NP set is frequently hard (in the sense made rigorous in the theorem's statement), there exist sets of our form that are frequently hard. (This result is making for backbones a point analogous to the ones in Section 4.2.1 but with results focused on NP rather than NP $\cap$ coNP.)

Theorem 4.10. *If h is any nondecreasing function and for some set $B \in$ NP it holds that each polynomial-time algorithm errs with respect to B, at infinitely many lengths n (resp., for almost every length n), on at least $h(n)$ of the inputs up to that length, then there will exist an $\kappa > 0$ and a set $A \in$ P of Boolean formulas satisfying the conditions of Theorem 4.8, yet being such that each polynomial-time algorithm g, at infinitely many lengths n (resp., for almost every length n), will fail to correctly determine membership in L_A for at least $h(n^\kappa)$ inputs of length at most n.*

The same claim also holds for Corollary 4.9.

Proof. We will prove the theorem's statement regarding Theorem 4.8. It is not hard to also then see that the analogous claim holds regarding Corollary 4.9.

$B \in$ NP and SAT is NP-complete. So let r_B be a polynomial-time function, transforming strings into Boolean formulas, such that (a) $r_B(x) \in \text{SAT} \Leftrightarrow x \in B$, and (b) r_B is one-to-one. (A construction of such a function is given in the Appendix of [57], and let us assume that that construction is used.) As in the proof of Theorem 4.8, if F is a Boolean formula we define $q(F) = \left\lceil \frac{\beta \|V(F)\|}{1-\beta} \right\rceil$. Without loss of generality, we assume that r_B outputs only formulas having at least one variable. Note that throughout this proof, q is applied only to outputs of r_B. Thus we have ensured that none of the logarithms in this proof have a zero as their argument.

Set

$$A = \{(r_B(x)) \wedge (\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(r_B(x))}) \mid x \in \Sigma^*\}.$$

Because r_B is computable in polynomial time, there is a polynomial b such that for every input x of length at most n, the length of $r_B(x)$ is at most $b(n)$. Fix some such polynomial b, and let k denote its degree. In order to find a bound for the length of the added "tail" $\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(r_B(x))}$ in terms of $b(n)$, notice that the length of the tail is less than some constant (that holds over all x and n, $|x| \leq n$) times $q(r_B(x)) \log q(r_B(x))$. Since $q(r_B(x)) = \left\lceil \frac{\beta \|V(F)\|}{1-\beta} \right\rceil$

and the length of $r_B(x)$ is at least a constant times the number of its variables, our assumption that $|r_B(x)| \leq b(n)$ implies the existence of a constant c such that, for all x and n, $|x| \leq n$, we have $q(r_B(x)) \leq c \cdot b(n)$. Taken together, the two previous sentences imply the existence of a constant d such that, for all x and n, $|x| \leq n$, we have that the length of $\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(r_B(x))}$ is at most $d \cdot b(n) \log(b(n))$, and so certainly is less than $d \cdot b^2(n)$. Let N be a natural number such that, for all $n \geq N$ and all x, $|x| \leq n$ implies that $|(r_B(x)) \wedge (\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(r_B(x))})| \leq n^{2k+1}$; by the previous sentence and the fact that b is of degree k, such an N will exist. Let g be a polynomial-time heuristic for L_A. Notice that $g \circ r_B$ is a polynomial-time heuristic for B, since $(r_B(x)) \wedge (\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(r_B(x))}) \in L_A \Leftrightarrow r_B(x) \in \mathrm{SAT}$ and $r_B(x) \in \mathrm{SAT} \Leftrightarrow x \in B$. Let $n_B \geq N$ be such that there is a set of strings $S_{n_B} \subseteq (\Sigma^*)^{\leq n_B}$, $\|S_{n_B}\| \geq h(n_B)$, having the property that for all $x \in S_{n_B}$, $g \circ r_B$ fails to correctly determine the membership of x in B. Consequently, there is a set of strings $T_{n_B} \subseteq (\Sigma^*)^{\leq (n_B)^{2k+1}}$, $\|T_{n_B}\| \geq h(n_B)$, such that for all $x \in T_{n_B}$, g fails to correctly determine the membership of x in L_A; in particular the set

$$T_{n_B} = \{(r_B(x)) \wedge (\mathsf{new}_1 \wedge \mathsf{new}_2 \wedge \cdots \wedge \mathsf{new}_{q(r_B(x))}) \mid x \in S_{n_B}\}$$

has this property.

Using the variable renaming $n_A = (n_B)^{2k+1}$, it is now easy to see that we have proven that every length $n_B \geq N$ at which $g \circ r_B$ (viewed as a heuristic for B) errs on at least $h(n_B)$ inputs of length up to n_B has a corresponding length n_A at which g (viewed as a heuristic for L_A) errs on at least $h((n_A)^{\frac{1}{2k+1}})$ inputs of length up to n_A. Our hypothesis guarantees the existence of infinitely many such $n_B \geq N$ (resp., almost all $n \geq N$ can take the role of n_B), each with a corresponding n_A. Setting

$$\kappa = \frac{1}{2k+1},$$

our theorem is now proven. □

4.3 Backdoors to CNF Formulas

Since CNF-SAT (the satisfiability problem restricted to CNF formulas) is well-known to be NP-complete, a polynomial-time algorithm to determine the satisfiability of CNF formulas is unlikely to exist. Nevertheless, there are several

restrictions of CNF formulas for which satisfiability can be decided in polynomial time. When a formula does not belong to any of these restrictions, it may have a set of variables that, once the formula is simplified over a partial assignment of these variables, the resulting formula belongs to one of these tractable restrictions. A formalization of this idea is the concept of backdoors.

Definition 4.11 (Subsolver [113])**.** *A polynomial-time algorithm A is a* subsolver *if, for each input formula F, A satisfies the following conditions.*

1. *A either rejects the input F (this indicates that it declines to make a statement as to whether F is satisfiable) or* determines *F (i.e., A returns a satisfying assignment if F is satisfiable and A proclaims F's unsatisfiability if F is unsatisfiable).*
2. *If F is trivially $\top$ A determines F, and if F is trivially $\bot$ A determines F.*
3. *If A determines F, then for each variable x and each value v, A determines $F[x/v]$.*

Definition 4.12 (Strong Backdoor [113])**.** *For a Boolean formula F, a nonempty subset S of its variables is a* strong backdoor *for a subsolver A if, for all partial assignments a_S, A determines $F[a_S]$.*

Many examples of subsolvers can be found in the literature (for instance, in Table 1 of [34]). The subsolver that is of particular relevance to this section is the *unit propagation subsolver*, which focuses on *unit clauses*. Unit clauses are clauses with just one literal. They play an important role in the process of finding models (i.e., satisfying assignments) because the literal in that clause must be set to $\top$ in order to find a satisfying assignment. The process of finding a model by searching for a unit clause, fixing the value of the variable in the unit clause, and simplifying the formula resulting from that assignment is known in the satisfiability literature as unit propagation. Unit propagation is an important building block in the seminal DPLL algorithm for SAT [29, 30]. Notice that the CNF formulas whose satisfiability can be decided by just applying unit propagation iteratively constitute a tractable restriction of SAT. The unit propagation subsolver attempts to decide the satisfiability of an input formula by using only unit propagation and empty clause detection. If satisfiability cannot be decided this way, the subsolver rejects the input

formula. Szeider [104] has classified the parameterized complexity of finding backdoors with respect to the unit propagation subsolver.

Example 4.13. *Consider the formula* $F = (x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_5) \wedge (x_1 \vee x_2 \vee x_4 \vee x_5) \wedge (x_3 \vee \overline{x_4}) \wedge (\overline{x_1} \vee x_2 \vee x_3 \vee x_5)$ *from Example 2.1. We will show that* $\{x_1, x_3, x_5\}$ *is a strong backdoor of* F *with respect to the unit propagation subsolver by analyzing the possible assignments of these variables. Suppose* x_1 *is assigned to* $\top$ *and notice* $F[x_1/\top] = \{\{x_3, \overline{x_4}\}, \{x_2, x_3, x_5\}\}$*. From there it is easy to see that if* x_3 *is set to* $\top$*, the resulting formula after simplification is trivially satisfiable. If* x_3 *is set to* $\bot$*, assigning* x_5 *to* $\top$ *yields the formula* $\{\{\overline{x_4}\}\}$ *after simplification and the satisfiability of this formula can be determined by the unit propagation subsolver. Assigning* x_5 *to* $\bot$ *yields a formula with two unit clauses,* $\{\{\overline{x_4}\}, \{x_2\}\}$*. The unit propagation subsolver will pick the unit clause* $\{x_2\}$*,*[4] *assign the truth value of* x_2 *and simplify, and will then pick the (sole) remaining unit clause,* $\{\overline{x_4}\}$*, and assign the truth value of* x_4 *and simplify to obtain a trivially satisfiable formula. Now suppose* x_1 *is assigned to* $\bot$ *and notice* $F[x_1/\bot] = \{\{\overline{x_2}, \overline{x_3}, x_5\}, \{x_2, x_4, x_5\}, \{x_3, \overline{x_4}\}\}$*. If we now assign* x_3 *to* $\top$*, notice* $F[x_1/\bot, x_3/\top] = \{\{\overline{x_2}, x_5\}, \{x_2, x_4, x_5\}\}$*. If we assign* x_5 *to* $\top$*,* F *simplifies to a trivially satisfiable formula. If we assign* x_5 *to* $\bot$*, the formula simplifies to* $\{\{\overline{x_2}\}, \{x_2, x_4\}\}$*. The unit propagation subsolver will pick the unit clause* $\{\overline{x_2}\}$*, assign the truth value of* x_2*, and the resulting formula after simplification will be* $\{\{x_4\}\}$ *whose satisfiability can be determined by the unit propagation subsolver. If we assign* x_3 *to* $\bot$*, notice* $F[x_1/\bot, x_3/\bot] = \{\{x_2, x_4, x_5\}, \{\overline{x_4}\}\}$*. If we now assign* x_5 *to* $\top$ *and simplify, the resulting formula will be* $\{\{\overline{x_4}\}\}$ *whose satisfiability can be determined by the unit propagation subsolver. If we assign* x_5 *to* $\bot$ *and simplify, the resulting formula will contain the unit clause* $\{\overline{x_4}\}$*. The unit propagation subsolver will then set the value of* x_4 *to* $\bot$ *and simplify, yielding the formula* $\{\{x_2\}\}$*, whose satisfiability can also be determined by the unit propagation subsolver.*

It should be clear from the case analysis above that just setting the values of x_1 *and* x_3 *is not enough for the unit propagation subsolver to always be able to determine the satisfiability of the resulting formula. In fact, a similar analysis done on every 2-element subset and every 3-element subset of* $V(F)$*—which we do not write out here—shows that* $\{x_1, x_3, x_5\}$ *is actually the smallest strong backdoor of* F *with respect to the unit propagation subsolver.*

[4]Here we assume that a clause $\{x\}$ precedes a clause $\{y\}$ in lexicographical order if x precedes y in lexicographical order.

In this section, we show that, under the assumption that P $\neq$ NP, there are easily recognizable families of formulas with strong unit propagation backdoors that are easy to find, yet the problem of determining whether these formulas are satisfiable remains hard (in fact, NP-complete). We also prove that if any NP set exists that is frequently hard (with respect to polynomial-time heuristics), then sets of our sort exist that are essentially just as frequently hard; we in effect prove an inheritance of frequency-of-hardness result, under which our sets are guaranteed to be essentially as frequently hard as any set in NP is.

Theorem 4.14. *If* P $\neq$ NP*, for each* $k \in \{1, 2, 3, \ldots\}$ *there is a set* A *of Boolean formulas such that all the following hold.*

1. $A \in$ P *and* $A \cap SAT$ *is* NP*-complete.*
2. *Each formula* G *in* A *has a strong backdoor* S *with respect to the unit propagation subsolver, with* $\|S\| \leq \|V(G)\|^{\frac{1}{k}}$*.*
3. *There is a polynomial-time algorithm that, given* $G \in A$*, finds a strong backdoor having the property stated in item 2 of this theorem.*

Proof. For $k = 1$ the theorem is trivial, so we henceforward consider just the case where $k \in \{2, 3, \ldots\}$. Consider $A \in$ P defined by

$$A = \{F \wedge (\mathsf{new}_1 \wedge \cdots \wedge \mathsf{new}_{\|V(F)\|^k - \|V(F)\|}) \mid F \text{ is a CNF formula}\},$$

where new_i is the ith (in lexicographical order) variable name that does not appear in F. The backdoor is the set of variables of F, which can be found in polynomial time by parsing. It is clear that the formula resulting from simplification after assigning values to all the variables of F only has unit clauses and potentially an empty clause, so satisfiability for this formula can be decided by the unit propagation subsolver. Finally, it is easy to see that $F \wedge (\mathsf{new}_1 \wedge \cdots \wedge \mathsf{new}_{\|V(F)\|^k - \|V(F)\|}) \in \text{SAT} \Leftrightarrow F \in \text{SAT}$ so, since the formula-part that is being appended to F can easily be polynomial-time constructed given F, under the assumption that P $\neq$ NP deciding satisfiability for the formulas in A is hard. □

We mention in passing that one can change "Boolean" to "Boolean CNF" in Theorem 4.14's statement, via adjusting appropriately the use of parentheses in the proof's definition of A to ensure that A itself is in CNF whenever F is.

The reader might worry that the hardness spoken of in the theorem occurs very infrequently (e.g., perhaps except for just one string at every double-exponentially spaced length everything is easy). That is, are we giving a worst-case result that deceptively hides a low typical-case complexity? We are not (unless all of NP has easy typical-case complexity): analogously to what we showed regarding backbones, we show that if any set in NP is frequently hard with respect to polynomial-time heuristics, then a set of our sort is almost as frequently hard with respect to polynomial-time heuristics. We will show that if any set in NP is frequently hard then a set of our type is almost-as-frequently hard. (Recall that, when n's universe is the naturals as it is in the following theorem, "for almost every n" means "for all but at most a finite number of natural numbers n.")

Theorem 4.15. *If h is any nondecreasing function and for some set $B \in \mathrm{NP}$ it holds that each polynomial-time algorithm errs with respect to B, at infinitely many lengths n (resp., for almost every length n), on at least $h(n)$ of the inputs up to that length, then there will exist an $\epsilon > 0$ and a set $A \in \mathrm{P}$ of Boolean formulas satisfying the conditions of Theorem 4.14, yet being such that each polynomial-time algorithm g, at infinitely many lengths n (resp., for almost every length n), will fail to determine membership in $A \cap SAT$ for at least $h(n^\epsilon)$ inputs of length at most n.*

Proof. For conciseness and to avoid repetition, we build this proof on top of the proof of Theorem 4.10. That proof does not rely directly or indirectly on the present theorem/proof, so there is no circularity at issue here.

We define r_B as in the proof of Theorem 4.10 (the r_B given there draws on a construction from the Appendix of [57], and due to that construction's properties outputs only conjunctive normal form formulas). For a given k, we define

$$A = \{r_B(x) \wedge (\mathsf{new}_1 \wedge \cdots \wedge \mathsf{new}_{\|V(r_B(x))\|^k - \|V(r_B(x))\|}) \mid x \in \Sigma^*\},$$

and since $r_B(x) \wedge (\mathsf{new}_1 \wedge \cdots \wedge \mathsf{new}_{\|V(r_B(x))\|^k - \|V(r_B(x))\|}) \in \mathrm{SAT} \Leftrightarrow r_B(x) \in \mathrm{SAT}$ and $r_B(x) \in \mathrm{SAT} \Leftrightarrow x \in B$, we can now proceed as in the proof of Theorem 4.10, since here too the tail's length is polynomially bounded. □

4.4 Conclusions and Related Work

The true inspiration for this work was an insightful structural complexity theory paper of Borodin and Demers [17] from the 1970s, which never appeared in any form other than as a technical report. Their paper in effect showed sufficient conditions for creating simple sets of satisfiable formulas such that it was unclear why they were satisfiable.

Borodin and Demers's work has been used only very rarely. In particular, it has been used to get characterizations regarding unambiguous computation [52], and Rothe and his collaborators have used it in various contexts to study the complexity of certificates [58, 97], see also Fenner et al. [40] and Valiant [109]. Also, one paper by Hemaspaandra, Hemaspaandra, and Menton [54] shows that some problems about the manipulation of elections have the property that if $\mathrm{P} \neq \mathrm{NP} \cap \mathrm{coNP}$ then their search versions are not polynomial-time Turing reducible to their decision problems. The key issue that 2013 paper left open is whether the type of techniques it used, descended from Borodin and Demers [17], might be relevant in other domains, or whether its results were a one-shot oddity. Our results in effect argue that the former is the case. Backbones and backdoors are topics important in both theoretical computer science and artificial intelligence. Our results show that the inspiration of the line of work initiated by Borodin and Demers [17] can be used to establish the opacity of backbones and backdoors. It is important to acknowledge that our proofs regarding Section 4.2.1 are drawing on elements of the insights of Borodin and Demers [17], although in ways unanticipated by that paper.

We make use of density transfer arguments in the context of Borodin-Demers arguments. To the best of our knowledge, the only work to previously do that is Hemaspaandra, Hemaspaandra, and Menton [54].

Chapter 5

Graph Arrowing

Some of the most studied phenomena in combinatorics deal with the properties that will necessarily hold whenever a graph is partitioned. Problems of this kind appear even in folklore, for instance in the Friendship Theorem which states that in a group of 6 or more people there are necessarily either 3 people that are friends or 3 people that are not friends. This branch of combinatorics is often referred to as recreational mathematics, yet the underlying theory has puzzled brilliant minds in discrete mathematics, most notably that of Pál Erds, for over a century. Part of the reason these problems have eluded the efforts of decades of mathematical research is because there seems to be a necessary step of reasoning by brute force over a large number of cases. The ability of performing this kind of brute force has always been beyond the human brain's capacity, and until recently has been beyond the reach of computers. The situation has changed in recent years and computational methods are being increasingly used in solving problems and settling conjectures in graph theory. This chapter is specifically devoted to problems in graph arrowing, where the set of vertices or edges of a graph are partitioned by "coloring" them. Many questions can be asked about what properties must arise from these colorings given enough vertices or edges, and we use constraint satisfaction formulations of these questions to give answers, or at least hint towards answers, for them. A summary of our contributions in this chapter is:

- In Section 5.3 we identify issues that arise from using off-the-shelf symmetry breaking tools like those mentioned in Section 2.3 when applied to model enumeration in graph arrowing problems.

- In Section 5.4 we describe a family of quantified Boolean formulas based on graph arrowing problems that has the interesting property that all formulas in this family have unique satisfying assignments to the top-level variables.
- Finally, in Section 5.5 we describe other projects where we have used SAT solvers to gain insights on problems related to graph arrowing as a complement to the theoretical analysis.

The next couple of sections provide the background definitions and a description of the general techniques used to encode graph arrowing problems using constraint satisfaction paradigms.

5.1 Definitions

We define the *order* of a graph to be the number of its vertices. Let K_n be the complete graph of order n. We define $[k] = \{1, 2, \ldots, k\}$ as the set of positive integers up to k. Let $G = (V, E)$ be a graph, where V and E are the set of vertices and edges of G, respectively. An edge k-coloring of the edges of G is a function $\sigma : E \to [k]$. For a k-coloring σ of a graph $G = (V, E)$ and $i \in [k]$ we define σ_i as the graph formed by the edges of G that have color i, i.e., $\sigma_i = (V, \{e \in E \mid \sigma(e) = i\})$. The important concept behind all definitions of arrowing is *subgraph isomorphism*:

Definition 5.1 (Subgraph Isomorphism)**.** *Given two graphs $H = (V_H, E_H)$ and $G = (V_G, E_G)$, a* subgraph isomorphism *from H to G is a function $\phi : V_H \to V_G$ that is such that, for all vertices $u, v \in V_H$, $\{u, v\} \in E_H$ implies $\{\phi(u), \phi(v)\} \in E_G$.*[1]

The concepts of graph arrowing for edge colorings are defined as follows:

[1] Not to be confused with other related concepts like *induced subgraph isomorphism* and *graph isomorphism*. In the case of induced subgraph isomorphism, the condition on ϕ is that $\{u, v\} \in E_H$ if and only if $\{\phi(u), \phi(v)\} \in E_G$. Graphs H and G are isomorphic if they have the same number of vertices and an induced subgraph isomorphism exists from H to G (or, equivalent, from G to H). We mention in passing that there are definitions of edge and vertex arrowing problems for the induced version of subgraph isomorphism, but we do not deal with those in this thesis.

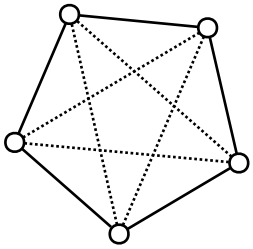


Figure 5.1: A coloring of K_5 avoiding monochromatic triangles, thus witnessing $K_5 \not\to_e (K_3, K_3)$. The two colors are represented by solid and dashed lines.

Definition 5.2 (Edge Arrowing)**.** *A graph G* edge-arrows *the list of graphs $H_1, H_2, \ldots, H_k$, written $G \to_e (H_1, H_2, \ldots, H_k)$, if for every k-coloring σ of the edges of G, there exists $i \in [k]$ such that there is a subgraph isomorphism from H_i to σ_i.*

Definition 5.3 (Edge Good Coloring)**.** *An edge k-coloring σ of G is a* $(H_1, H_2, \ldots, H_k)_e$-good *coloring of G if for every $i \in [k]$ there exists no subgraph isomorphism from H_i to σ_i. An $(H_1, H_2, \ldots, H_k)_e$-good coloring then witnesses the fact that $G \not\to_e (H_1, H_2, \ldots, H_k)$.*

Figure 5.1 shows a coloring of K_5 (the complete graph on 5 vertices) witnessing $K_5 \not\to_e (K_3, K_3)$. The following is a version of the Friendship Theorem stated above, in terms of edge arrowing:

Theorem 5.4 (Friendship Theorem)**.** *$K_6 \to_e (K_3, K_3)$.*

Analogous definitions exist for vertex colorings, defined instead over colorings $\sigma : V \to [k]$:

Definition 5.5 (Vertex Arrowing)**.** *A graph G* vertex-arrows *the list of graphs $H_1, H_2, \ldots, H_k$, written $G \to_v (H_1, H_2, \ldots, H_k)$, if for every k-coloring σ of the vertices of G, there exists $i \in [k]$ such that there is a graph isomorphism from H_i to σ_i.*

Definition 5.6 (Vertex Good Coloring)**.** *A vertex k-coloring σ of G is a* $(H_1, H_2, \ldots, H_k)_v$-good *coloring of G if for every $i \in [k]$ there exists no subgraph isomorphism from H_i to σ_i. An $(H_1, H_2, \ldots, H_k)_v$-good coloring then witnesses the fact that $G \not\to_v (H_1, H_2, \ldots, H_k)$.*

Several important concepts in combinatorics and graph theory have been defined around edge and vertex arrowing. The multicolor Ramsey number $R(G_1, G_2, \ldots, G_k)$ is the smallest natural number N such that the complete graph on N vertices edge-arrows the graphs $G_1, G_2, \ldots, G_k$. A celebrated theorem by Ramsey [95] guarantees the existence of these numbers, but finding the exact value for specific parameters remains a challenging task for computers. Similarly, the edge-Folkman number $F_e(G_1, G_2, \ldots, G_k; m)$ (respectively, vertex-Folkman number $F_v(G_1, G_2, \ldots, G_k; m)$) is the smallest N such that there exists a K_m-free graph of order N that edge-arrows (resp., vertex-arrows) the graphs $G_1, G_2, \ldots, G_k$. Folkman showed that when n is greater than the maximum order of the graphs G_i, $F(G_1, G_2, \ldots, G_k; n)$ exists [42]. Ramsey and Folkman numbers, and variations thereof, have attracted much interest in finite and extremal combinatorics. The book by Soifer [102] provides a good overview of the latest research trends in these areas and the dynamic survey by Radziszowski [94] keeps track of which finite Ramsey numbers are known.

For two graphs G and H, we denote by $\mathcal{S}(G, H)$ the set of subgraph isomorphisms from H to G. For succinctness, when $e \in E$ is an edge $e = \{u, v\}$, we write $s(e)$ for the edge $\{s(u), s(v)\}$. We define $\mathcal{C}(G; H_1, H_2, \ldots, H_k)$ as the set of $(H_1, H_2, \ldots, H_k)_e$-good colorings of G. Clearly, $\mathcal{C}(G; H_1, H_2, \ldots, H_k) = \emptyset \Leftrightarrow G \rightarrow_e (H_1, H_2, \ldots, H_k)$.

5.2 CSP Encodings for Graph Arrowing

Let us explore some of the options to tackle arrowing problems using constraint satisfaction programming. Let us first consider the case $k = 2$. To encode edge 2-colorings we exploit the fact that the Boolean domain contains two values $\top$ (*true*) and $\bot$ (*false*) and express the negation of the arrowing property in terms of Boolean formulas. Let the Boolean formula $\phi(G; H_1, H_2)$ on $\|E(G)\|$ variables x_e for $e \in E(G)$ be defined as follows.

$$\phi(G; H_1, H_2) = \left(\bigwedge_{s \in \mathcal{S}(G,H_1)} \bigvee_{e \in E(H_1)} x_{s(e)} \right) \wedge \left(\bigwedge_{s \in \mathcal{S}(G,H_2)} \bigvee_{e \in E(H_2)} \overline{x_{s(e)}} \right) \quad (5.1)$$

This formula essentially states that in every subgraph isomorphism from H_1 to G at least one of the edges involved is "colored" *true*, and in every

subgraph isomorphism from H_2 to G at least one of the edges involved is "colored" *false*. If we correspond $\{1, 2\}$ with $\{\bot, \top\}$, then a model of $\phi(G; H_1, H_2)$ can easily be mapped to an edge coloring of G that avoids monochromatic copies of H_1 and H_2 in the first and second color, respectively. It is also easy to see this mapping is one-to-one, i.e., to each model of $\phi(G; H_1, H_2)$ corresponds a coloring witnessing $G \not\to_e (H_1, H_2)$ and vice versa. Thus, $\phi(G; H_1, H_2) \in \text{SAT} \Leftrightarrow G \not\to_e (H_1, H_2)$. Furthermore, one can generate $\mathcal{C}(G; H_1, H_2)$ by using an AllSAT solver to list every model of $\phi(G; H_1, H_2)$.

When $k > 2$, a different approach is needed. Boolean formulas can still be of help in this case, but the encoding is slightly more complex. The encoding used in Codish et al. [21] has variables of the form $x_{e,i}$ for every edge $e \in E(G)$ and every color $i \in [k]$ indicating edge e is colored i. For each color $i \in [k]$, the formula

$$\psi_{\mathcal{S}}(G; H_i) = \bigwedge_{s \in \mathcal{S}(G,H_i)} \bigvee_{e \in E(H_i)} \overline{x_{e,i}}$$

encodes the constraint that at least one of the edges involved in a subgraph isomorphism from H_i to G is *not* colored i. Also, for every edge $e \in E(G)$ the constraint ExactlyOne($\{x_{e,1 \le i \le k}\}$) from Section 2.1.3 guarantees that e is colored with exactly one color. Putting these together, the formula

$$\psi(G; H_1, H_2, \ldots, H_k) = \left(\bigwedge_{i \in [k]} \psi_{\mathcal{S}}(G; H_i) \right) \wedge \left(\bigwedge_{e \in E(G)} \textsc{ExactlyOne}(\{x_{e,1 \le i \le k}\}) \right)$$

is not satisfiable if and only if $G \to_e (H_1, H_2, \ldots, H_k)$, and as before, every satisfiable assignment of this formula can be mapped to a $(H_1, H_2, \ldots, H_k)_e$-good coloring of G. Nevertheless, $\psi(G; H_1, H_2, \ldots, H_k)$ is inconvenient because it is more complicated than $\phi(G; H_1, H_2)$.

An alternative way to formulate the arrowing property using CSPs is to use answer set programming. Consider the program $\Pi(G; H_1, H_2, \ldots, H_k)$ over variables x_e for $e \in E(G)$ and the predicate color/2 is defined as

$$\Pi(G; H_1, H_2, \ldots, H_k) = \left(\bigcup_{e \in E(G)} \{\text{color}(e, i) : i \in [k]\} = 1 \right) \cup \left(\bigcup_{i \in [k]} \Pi'(G, H_i) \right)$$

where each subprogram $\Pi'(G, H)$ is defined as $\Pi'(G, H) = \bigcup_{s \in \mathcal{S}(G,H)} \mathbf{R}(G, H, s)$ and the rules $\mathbf{R}(G, H, s)$ are in turn defined by its parts as

$$\begin{aligned} H(\mathbf{R}(G,H,s)) &= \emptyset \\ B^+(\mathbf{R}(G,H,s)) &= \bigcup_{e\in E(H)} \operatorname{color}\big(x_{s(e)},i\big) \\ B^-(\mathbf{R}(G,H,s)) &= \emptyset. \end{aligned}$$

When the predicate $\operatorname{color}(e,i)$ for an edge $e \in E(G)$ is interpreted as the edge e being colored i, the models of $\Pi(G;H_1,H_2,\ldots,H_k)$ correspond to colorings of G witnessing $G \not\to_e (H_1,H_2,\ldots,H_k)$.

The main advantage of using ASP over Boolean formulas for graph arrowing in the case of $k > 2$ colors is the expressive power of the input language of ASP solvers like `clingo`. The ability to use a predicate like color/2 and cardinality constraints make for a concise encoding of the arrowing property.

5.3 Enumerating Colorings Modulo Symmetries

Finding all the assignments that satisfy the formula $\phi(F;G,H)$ (i.e., solving the AllSAT problem for this formula) can be used to generate the complete set of colorings of the edges of F witnessing $F \not\to_e (G,H)$. Generating these sets of colorings for a given triple of parameters (F,G,H) is often used as a building block towards finding exact values for Ramsey numbers under different parameters. The "gluing method" used to establish $R(4,5) = 25$ [81], which is still found embedded in more recent ideas [21], is an example of such an application. Using AllSAT solvers for this purpose offloads the combinatorial search to standard tools that are being actively developed [106], eliminating the need to craft specialized code. A shortcoming of generating these families through the method of encoding the negation of the arrowing property into a Boolean formula is that an AllSAT solver may generate many colorings that are essentially equivalent among them as it lists all possible models of the formula. As discussed in Section 2.3, the need to generate models that are distinct under some notion of equivalence is not unique to formulations of the arrowing predicate as a Boolean formula. Many symmetry breaking techniques have been developed for specific applications of SAT. A great example of a symmetry breaking technique specifically tailored for graph search is the use of *canonizing sets* for small graph searches [65]. Unfortunately, the need to

embed these techniques into the Boolean encoding of the non-arrowing property may neglect the advantage of using general-purpose AllSAT solvers. To avoid this issue, one may be interested in using off-the-shelf symmetry breaking software that is domain independent. Nevertheless, we showed that using syntactic symmetry breaking tools for the task of generating witness colorings via AllSAT solvers may have undesirable consequences if one does not take care of understanding the details of the underlying techniques.

5.3.1 Number of Satisfying Assignments

One of the main improvements of `Shatter` [3] over the original formulation of the symmetry breaking clauses [25] explained in Section 2.3 is that `Shatter` adds symmetry-breaking clauses whose number of literals is linear in the number of variables of the input formula. This is done through a relaxation on the symmetry breaking constraints. This relaxation has an undesirable effect in the number of satisfying assignments of the resulting formula. To study this effect, we summarize some of the details behind `Shatter`'s relaxation of the symmetry breaking constraints.

Using additional equality variables $e_i \equiv (x_i \leftrightarrow \pi_{\mathbf{x}}(x_i))$, one incurs in a quadratic increase on the length of the formula when adding lexicographical symmetry breaking clauses, as discussed in Section 2.3. `Shatter` avoids this by using "chaining predicates". For this, new variables $l_i \equiv (x_i \rightarrow \pi_{\mathbf{x}}(x_i))$ (that is, l_i is true if and only if x_i is "less than or equal to" $\pi_{\mathbf{x}}(x_i)$) and p_i are introduced, together with CNF clauses equivalent to $p_i \leftrightarrow (e_{i-1} \rightarrow (l_i \wedge p_{i+1}))$, with $e_0 = \top$ and $p_{n+1} = \top$. `Shatter` also replaces equality variables e_i with "greater than or equal to" variables $g_i \equiv (\pi_{\mathbf{x}}(x_i) \rightarrow x_i)$, and relaxes the if and only if conditions for the p_i variables to one way implications. The clauses added by `Shatter` are then the CNF equivalents of formulas of the form $p_i \rightarrow (g_{i-1} \rightarrow l_i \wedge p_{i+1})$, with $g_0 = \top$. It is easy to see these relaxations introduce satisfying assignments that do not satisfy the original symmetry breaking formulation. To get a better feel of how much of a "blow-up" in the number of satisfying assignments does this relaxation cause, consider the following lemmas about extensions of partial assignments of the Boolean expressions involved in these two formulations of symmetry breaking clauses. S_k in Lemma 5.7 corresponds to the original lexicographical symmetry breaking clauses with chaining, while T_k in Lemma 5.8 corresponds to the clauses added

by `Shatter`.

Lemma 5.7. *Let* $S_k = e_0 \wedge \bigwedge_{i=1}^{k} (p_i \leftrightarrow (e_{i-1} \rightarrow (l_i \wedge p_{i+1}))) \wedge p_{k+1}$. *Then a partial assignment of the variables* e_i *and* l_i *has at most one extension that satisfies* S_k.

Proof. Let $S'_k = e_0 \wedge \bigwedge_{i=1}^{k} (p_i \leftrightarrow (e_{i-1} \rightarrow (l_i \wedge p_{i+1})))$, that is, S'_k is S_k without fixing p_{k+1}. Then $S_k = S'_k \wedge p_{k+1}$. We first prove by induction on k that a partial assignment of the variables e_i and l_i has at most one extension that satisfies S'_k once p_{k+1} is assigned to a value. For $k = 0$, this is trivially true since the value of $S'_0 = e_0$ does not depend on the value of p_1. We now assume that this property holds for S'_n and prove it for S'_{n+1}. Our induction hypothesis implies that for any assignment of the variables e_i and l_i for $i \leq n$, there is at most one assignment of the variables p_i, $i \leq n$ that satisfies S'_n once p_{n+1} is assigned to a value. Suppose we assign p_{n+2} to a value, then since $p_{n+1} \leftrightarrow (e_{n-1} \rightarrow l_n \wedge p_{n+1})$ is a subformula of S'_{n+1} so p_{n+1} is uniquely determined by the assignments of e_{n-1}, l_n and p_{n+2}, which proves our claim. Since $S_k = S'_k \wedge p_{k+1}$, any satisfying assignment of S_k must fix p_{n+2} to $\top$ and the lemma follows. □

Lemma 5.8. *Let* $T_k = g_0 \wedge \bigwedge_{i=1}^{k} (p_i \rightarrow (g_{i-1} \rightarrow (l_i \wedge p_{i+1}))) \wedge p_{k+1}$. *Let* ϕ *be a partial assignment that assigns the* m *variables* $g_i, g_{i+1}, \ldots, g_{i+m-1}$ *to* $\bot$. *Then, if* ϕ *can be extended to an assignment that satisfies* T_k, *it can be extended to at least* $2^m - 1$ *assignments that satisfy* T_k.

Proof. Since ϕ assigns g_i to $\bot$, the subformula $p_{i+1} \rightarrow (g_i \rightarrow (l_{i+1} \wedge p_{i+2}))$ simplifies to $p_{i+1} \rightarrow \top$, regardless of what l_{i+1} and p_{i+2} are assigned to. Then p_i can be assigned to $\bot$ or $\top$ without falsifying T_k. Furthermore, $p_{i+1} \rightarrow (g_i \rightarrow (l_{i+1} \wedge p_{i+2}))$ does not constrain p_{i+2} if g_i is assigned to $\bot$, and $p_{i+2} \rightarrow (g_{i+1} \rightarrow (l_{i+2} \wedge p_{i+3}))$ simplifies to $p_{i+2} \rightarrow \top$ when g_{i+1} is assigned to $\bot$, so p_{i+2} can be assigned to $\bot$ or $\top$ as well. Following the same reasoning, all the variables $p_{i+1}, p_{i+2}, \ldots, p_{i+m}$ can be assigned to $\bot$ or $\top$ independently without falsifying T_k except possibly for p_{i+m} if $i + m = k + 1$, yielding the desired result. □

The combination of these lemmas means that, in the worst case, the increase in the number of satisfying assignments due to Shatter's relaxations may be exponential in the number of e_i or g_i variables. While in the original lexicographical symmetry breaking formulation the number of e_i variables is the same as the number of variables in the original formula, Shatter reduces the number of variables to consider by eliminating certain tautological subformulas from T_k, thus mitigating the effect of the increase in the number of models.

Another important detail in the analysis above is that the "blow-up" in the number of models of the formula enhanced with symmetry breaking clauses happens only at the p_i variables: for a fixed assignment of the original variables of ϕ there are many possible assignments of the p_i variables that would satisfy the formula output by Shatter. The projective model enumeration technique [46] in the clasp solver [44] is particularly useful to nullify this increase in the number of models. This technique allows for outputting models that are different modulo a subset of the variables. Thus combining projective model enumeration with the output of the formula preprocessed by Shatter restores the original goal of symmetry breaking which is reducing the number of satisfying assignments. The subset of variables that one would project to would of course be the original set of variables of the input formula.

It is important to mention that it is possible to avoid this issue altogether by using BreakID [32] which allows for using the original symmetry breaking predicates without relaxations. This tool is thus more appropriate for All-SAT applications where symmetry breaking is needed.

To illustrate the issue of increased number of models we provide a concrete example.

Example 5.9. *From finite Ramsey theory, we know that $R(C_5, C_5) = 9$ [19], where C_5 is the cycle of length 5 (see also [94] for a comprehensive survey of what is known in finite Ramsey theory). This means that $\phi(K_9; C_5, C_5) \notin SAT$, but $\phi(K_8; C_5, C_5) \in SAT$, so we are interested in finding all edge colorings of the complete graph K_8 witnessing $R(C_5, C_5) > 8$. $\phi(K_8; C_5, C_5)$ contains 28 variables (corresponding to $\binom{8}{2}$ edges in K_8) and 1344 clauses, and there are 1190 models for that formula. From this information, we know that $\|\mathcal{C}(K_8; C_5, C_5)\| = 1190$. After processing this formula with Shatter, the resulting formula with symmetry breaking clauses has 70 variables, 1499 clauses, and 824 models. On the other hand, using our own implementation of*

the chaining method without the relaxation outputs a formula with 165 variables, 1809 clauses, and 5 models. Using `nauty` *[80] to reduce any of these sets of colorings to pick just one representative from each equivalence class of colorings under isomorphism, we find that* $\|D(\mathcal{C}(K_8; C_5, C_5))\| = 4$*, so the chaining method without the relaxation outputs only one redundant coloring.*

5.3.2 Incomplete Sets of Colorings

Perhaps more concerning than the increase in the number of colorings found by preprocessing the formula $\phi(F; G, H)$ is the fact that enumerating all models of the result of preprocessing $\phi(F; G, H)$ with `Shatter` may not yield a complete set of colorings. On the other hand, there are parameters F, G, and H for which preprocessing a formula $\phi(F; G, H)$ with `Shatter` does output a Boolean formula whose models can be used to build a complete set of colorings for the given parameters. One example is Example 5.9, where we were able to generate $D(\mathcal{C}(K_8; C_5, C_5))$ from the models of $\phi(K_8; C_5, C_5)$ after preprocessing it with `Shatter`. The fact that for certain parameters this method will work and for other parameters it will not warrants an investigation. In this section, we take a closer look at this phenomenon and present a sufficient condition under which a workflow that uses `Shatter` for symmetry breaking produces incomplete sets of colorings. This condition is the presence of *free variables*, which are variables that do not appear in the CNF formula. This is possible in the context of CNF formulas in the DIMACS format because the specification[2] states that "it is not necessary that every variable appear in the instance." We study the effect of free variables in Lemma 5.10 and as an immediate consequence, we state a sufficient condition for $\phi(F; G, H)$ to produce incomplete sets of colorings in Theorem 5.11.

Lemma 5.10. *Let ϕ be a Boolean formula. Then any satisfying assignment of the formula output by preprocessing ϕ with* `Shatter` *will assign any free variables in ϕ to $\perp$.*

Proof. Let W be the set of free variables in ϕ. The graph G_ϕ generated according to the rules summarized in Section 2.3 has 2 vertices for each free variable (one for the positive literal and one for the negative literal) and an edge between these two vertices. Then the matching on $2\|W\|$ vertices (i.e.,

[2] `ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/doc/satformat.dvi`

the disjoint union of $\|W\|$ edges) is an induced subgraph of G_ϕ. From graph theory, we know that the permutations $(x, \overline{x})$ for $x \in W$ are generators of the automorphism group of G_ϕ (see, for instance, [112], Theorem 3-11). The rules applied by Shatter will turn these permutations into clauses of the form $\overline{x} \vee \overline{x}$ for $x \in W$, so each variable in W will be assigned to $\bot$. □

The above proof is based on the rules presented in Aloul, Sakallah, and Markov [3] to generate the symmetry breaking clauses. In practice, version 0.3 of Shatter[3] seems to have an additional rule that assigns each free variable to its own color. This change is effective in eliminating permutations of the type $(x_i, x_j)(\overline{x_i}, \overline{x_j})$ for $x_i, x_j \in W$ from the set of generators of the automorphism group of G_ϕ (because x_i and x_j will now have different colors), yet it does not eliminate permutations of the type $(x, \overline{x})$ which are the culprit of Lemma 5.10. It is important to highlight the fact that the addition of single-clause variables (a clause $\overline{x} \vee \overline{x}$ is equivalent to $\overline{x}$) is not a flaw in the design of the symmetry breaking clauses in Shatter but a feature, as noted in Section 3 of [1]. This is because processing permutations this way reflects the fact that free variables in the input formula can be fixed to any Boolean value, in this case $\bot$. This is advantageous when trying to determine the satisfiability of the input formula because the solver no longer has to search to set the value of that variable.

As an immediate consequence of Lemma 5.10, we have the following theorem.

Theorem 5.11. *Let F, G and H be graphs and suppose an edge $e \in E(F)$ does not participate in any subgraph isomorphism from G to F or from H to F (i.e., $e \notin \mathrm{Im}(\sigma)$ for any $\sigma \in \mathcal{S}(F, G) \cup \mathcal{S}(F, H)$). Then, if $\mathcal{C}(F; G, H)$ is not empty, the set of colorings obtained from the models of $\phi(F; G, H)$ after preprocessing it with* Shatter *is incomplete.*

Proof. Let $\phi'(F; G, H)$ be the formula output by preprocessing $\phi(F; G, H)$ with Shatter. Because e does not participate in any subgraph isomorphism from G to F, or from H to F, the variable x_e is free in $\phi(F; G, H)$. By Lemma 5.10 it will be assigned to $\bot$ in any model of $\phi'(F; G, H)$. Let m be a model of the formula $\phi'(F; G, H)$ and let m' be the assignment obtained from m by assigning x_e to $\top$. Because e is free in $\phi(F; G, H)$, the restriction of m' to the variables of $\phi(F; G, H)$ (i.e., the restriction of m' to the original

[3] http://www.aloul.net/Tools/shatter/Shatter_Linux_v03.tar.gz

variables) is a model of $\phi(F; G, H)$, but m' itself is not a model $\phi'(F; G, H)$ thus the coloring corresponding to m' will not be represented in the colorings obtained from the models of $\phi'(F; G, H)$. □

An obvious modification one can do to avoid this issue is to restrict the set of variables in Equation 5.1 to x_e for edges $e \in E(F)$ that are actually involved in some subgraph isomorphism from G to F or from H to F. Unfortunately, this does not guarantee the resulting formula will output a complete set of colorings. There are cases where the models of formulas encoding the non-arrowing property do not generate complete sets of colorings even when the formula does not have free variables. This indicates Theorem 5.11 is not a necessary condition for this phenomenon to occur. Here we provide an example of such a formula, which is related to Example 5.9:

Example 5.12. *Let K_{ex} be the graph obtained from K_8 by selecting one of its vertices and removing all but 2 edges incident to it (alternatively, this graph can be obtained by adding a vertex to K_7 and connecting it to two of the original vertices). The resulting graph is illustrated in Figure 5.2. We are interested in obtaining all 2-colorings avoiding monochromatic C_5 in any of the colors. It is easy to see that $\phi(K_{ex}; C_5, C_5)$ has no free variables: because the only vertex of degree less than 6 has degree 2, it participates in at least one cycle of length 5. Nevertheless, the models of the formula obtained from preprocessing $\phi(K_{ex}, C_5, C_5)$ with* `Shatter` *represent 64 of the 90 possible isomorphism classes in $\mathcal{C}(K_{ex}; C_5, C_5)$.*

5.4 A QSAT Benchmark Based on Vertex-Folkman Graphs

Recent years have seen great interest in the problem of determining the satisfiability of quantified Boolean formulas (QSAT). The study of QSAT algorithms has been spurred by applications to many areas of computer science, including model checking [31], games [77], and security [117], among others. However, it is still lagging behind the study of SAT solving algorithms which has been at the spotlight of constraint satisfaction research for much longer. In order to bring some parity between these two tightly related areas of research, much effort has been put into "porting" SAT techniques and algorithms to the QSAT

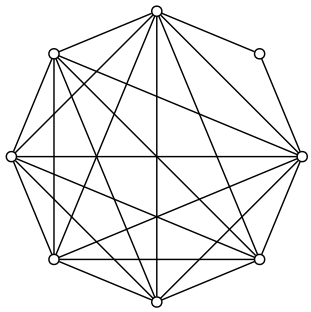


Figure 5.2: An illustration of the graph K_{ex} in Example 5.12. The models of the formula output by `Shatter` on input $\phi(K_{ex}; C_5, C_5)$ do not correspond to a complete set of colorings witnessing $K_{ex} \not\to (C_5, C_5)$.

setting, for instance by using SAT solvers as black-boxes for QSAT solving [67] or composing SAT solvers to build a QSAT solver [13]. Another area in which QSAT research needs to catch up to its SAT counterpart is in the availability of benchmarks that are well defined and well understood. We contribute to the latter by initiating the study of a family of QSAT instances based on problems in extremal graph theory. We argue that this family of formulas is interesting for QSAT research because it is both conceptually simple and parametrized in a way that allows for a fine-grained diversity in the level of difficulty of its instances. Additionally, when coupled with symmetry breaking, the formulas in this family exhibit backbones (unique satisfying assignments) at the top-level existential variables. This benchmark is thus suitable for addressing questions regarding the connection between the existence of backbones and the hardness of QBFs.

We define the set of vertex-Folkman graphs $\mathcal{F}_v^N(a_1, a_2, \ldots, a_r; m)$, where N and m are natural numbers and $(a_c)_{1 \leq c \leq r}$ is a sequence of natural numbers such that $a_c \geq 2$, as the set of all graphs G of order N that are K_m-free and are such that $G \to_v (K_{a_1}, K_{a_2}, \ldots, K_{a_r})$.

Consider the problem of determining whether $\mathcal{F}_v^N(a_1, a_2, \ldots, a_r; m)$ is non-empty. It is possible to encode this problem as a QBF using $\binom{N}{2}$ variables $x_{i,j}$ and rN variables $y_{i,c}$ such that the formula is satisfiable if and only if $\mathcal{F}_v^N(a_1, a_2, \ldots, a_r; m)$ is non-empty. The variable $x_{i,j}$ is set to true if and only if there exists an edge between vertices i and j, while the variable $y_{i,c}$

is set to true if and only if vertex i has color c. The intuition behind our encoding, which we present next, is that we are looking for a graph G that avoids K_m as a subgraph and is such that for all valid (in a sense explained below) assignments to the variables $y_{i,c}$ there exists a $c \in \{1, 2, \ldots, r\}$ and a subset $T \subseteq \{1, 2, \ldots, N\}$ of cardinality a_c such that all $x_{i,j}$ for $i, j \in T$, $i \neq j$, and all $y_{i,c}$ for $i \in T$ are true. To avoid K_m as a subgraph, the formula

$$A_m^N = \bigwedge_{\substack{S \subseteq \{1,2,\ldots,N\}, \\ \|S\|=m}} \bigvee_{\substack{i,j \in S, \\ i \neq j}} \overline{x_{i,j}}$$

guarantees that for no subset $S \subseteq \{1, 2, \ldots, N\}$ of cardinality m are the variables $x_{i,j}$ all true for $i, j \in S$ with $i \neq j$. In order to consider only valid r-colorings, we use the constraint ExactlyOne($\{y_{i,1 \leq c \leq r}\}$) from Section 2.1.3 to guarantee that exactly one of $y_{i,1}, y_{i,2}, \ldots, y_{i,r}$ is true for each vertex i. The formula

$$C_{n,c}^N = \bigvee_{\substack{T \subseteq \{1,2,\ldots,N\}, \\ \|T\|=n}} \left(\left(\bigwedge_{\substack{i,j \in T \\ i \neq j}} x_{i,j} \right) \wedge \left(\bigwedge_{i \in T} y_{i,c} \right) \right)$$

is true if there is a subset $T \subseteq \{1, 2, \ldots, N\}$ of cardinality n such that all the edges between these vertices exist in G (i.e., $x_{i,j}$ is true for all $i, j \in T$, $i \neq j$) and all the vertices in T are of the same color c. Putting these together, the formula

$$\begin{aligned} F^N(a_1, \ldots, a_r; m) \quad = \quad & \exists (x_{i,j})_{1 \leq i < j \leq N}. \\ & \forall (y_{i,c})_{1 \leq i \leq N,\, 1 \leq c \leq r}. \\ & A_m^N \wedge \\ & \left(\textsc{ExactlyOne}(\{y_{i,1 \leq c \leq r}\}) \rightarrow \bigvee_{c=1}^{r} C_{a_c,c}^N \right) \end{aligned} \tag{5.2}$$

is true if and only if $\mathcal{F}^N(a_1, a_2, \ldots, a_r; m)$ is non-empty.

For the rest of the chapter, we set $a = \max\{a_1, a_2, \ldots, a_r\}$. The particular case we are concerned with is the case $N = m + a$ where $m = 1 + \sum_{c=1}^{r} (a_c - 1)$. uczak, Ruciski, and Urbaski [76] proved that the only vertex-Folkman graph

for these parameters is $K_{m+a} - C_{2a+1}$, where C_n is the cycle on n vertices. We include an adapted, self-contained version of this theorem for the sake of completeness:

Theorem 5.13 (uczak, Ruciski, and Urbaski [76], Theorem 3)**.** *For every non-decreasing sequence of natural numbers $a_1, a_2, \ldots, a_r$, $a_c \geq 2$, and $m = 1 + \sum_{c=1}^{r}(a_c - 1)$ the graph $K_{a_r+m} - C_{2a_r+1}$ is the unique graph in $\mathcal{F}^{a_r+m}(a_1, a_2, \ldots, a_r; m)$.*

From Theorem 5.13 we can derive several properties of Formula (5.2) for these parameters. By counting the number of cycles of length $2a+1$ in K_{m+a}, we can see that $F^{m+a}(a_1, a_2, \ldots, a_r; m)$ has exactly $\binom{m+a}{2a+1}\frac{(2a)!}{2}$ satisfying assignments of the $x_{i,j}$ variables, so in particular it is satisfiable. It is also evident that all graphs represented by these satisfying assignments are pairwise isomorphic. This can be exploited to create a family of Boolean formulas with a unique satisfying assignment for the top-level existential variables. In the following section we explore how symmetry breaking can be used for this purpose.

5.4.1 Symmetry Breaking

Coupling $F^{m+a}(a_1, a_2, \ldots, a_k; m)$ with perfect symmetry breaking at the domain level would guarantee that the formulas of this family have a unique solution: the only admitted graph from the class of graphs of order $m+a$ that are isomorphic to $K_{m+a} - C_{2a+1}$. Good progress has been made in developing perfect symmetry breaking techniques, yet these typically only work for graphs of small order. Heule [61] provides symmetry breaking clauses called *isolators* that can be added to graph-search problems in CNF for graphs of order up to 8, and Itzhakov and Codish [65] provide permutations called *canonizing sets* that can be turned into predicates that can be added to graph-search problems for graphs of order up to 10. We can obtain a family of QBF benchmarks based on Theorem 5.13, each having a unique solution by using partial symmetry breaking. The key is to notice that we only need perfect symmetry breaking within the class of graphs that are isomorphic to $K_{m+a} - C_{2a+1}$. It is then enough to consider a subset $\mathcal{I}$ of the isomorphism classes of graphs of order $m+a$ which includes the equivalence class of graphs isomorphic to $K_{m+a} - C_{2a+1}$ and create symmetry breaking constraints that

are specific to that target subset. These could then be appended to A_m^{m+a} in the form $\textsc{BelongsTo}(\mathcal{I}) \rightarrow \textsc{SymmBreaking}(\mathcal{I})$, where the predicates $\textsc{BelongsTo}(\mathcal{I})$ is a predicate that determines whether a graph belongs to one of the isomorphism classes in $\mathcal{I}$, and $\textsc{SymmBreaking}(\mathcal{I})$ is a predicate that breaks symmetries among graphs in $\mathcal{I}$. The partial symmetry breaking strategy we explore in this paper is based on the observation that it is sufficient to consider the subset $\mathcal{I}_{m+a}^{\leq 2a+1} \subset \mathcal{I}_{m+a}$ of isomorphism classes of graphs of order $m+a$ with at least $\binom{m+a}{2} - (2a+1)$ edges. In this case, $\textsc{BelongsTo}(\mathcal{I}_{m+a}^{\leq 2a+1}) \equiv \textsc{AtMostK}((\overline{x_{u,v}})_{u,v<a+m,\, u\neq v}; 2a+1)$ which is true whenever at most $2a+1$ of the literals $\overline{x_{u,v}}$ are true (i.e., whenever the graph is missing at most $2a+1$ edges), and $\textsc{SymmBreaking}(\mathcal{I}_{m+a}^{\leq 2a+1})$ can be built from enumerating all non-isomorphic graphs with at least $\binom{m+a}{2} - (2a+1)$ edges through the `nauty` [80] package and then building a sum-of-products predicate from the edge variables involved.

5.4.2 Clausal Encoding

Clausal encodings of Boolean formulas restrict the syntax of the formulas by requiring their variables to be grouped into clauses (typically conjunctions or disjunctions of variables). The most common normal forms are conjunctive normal form (CNF) which are conjunctions of disjunctions, and disjunctive normal form (DNF) which are disjunctions of conjunctions. In order to encode the formula $F^{m+a}(a_1, a_2, \ldots, a_k; m)$ in the QDIMACS format, the matrix of the formula must be transformed to CNF. Notice the subformula A_m^N is already in CNF. On the other hand the constraint $\textsc{ExactlyOne}(\{y_{i,1\leq c\leq r}\})$ is also in CNF and thus the subformula $\textsc{ExactlyOne}(\{y_{i,1\leq c\leq r}\}) \rightarrow \bigvee_{c=1}^{k} C_{a_i,i}^N$ is in DNF and a transformation to CNF is needed. To avoid a combinatorial explosion, the particular form of our formula allows us to use the Plaisted-Greenbaum transformation [93] (see Secton 2.1.2). This will introduce a new level of quantification for the auxiliary variables. Thus the instances are technically not instances of 2QSAT (that is, QSAT with two quantifiers) but of 3QSAT instead, yet they essentially encode an $\exists\forall$ problem.

We incorporated symmetry breaking as described in Section 2.3 thus generating Boolean formulas with exactly one statisfying assignment for the top-level existential variables. To implement a partial symmetry breaking tech-

nique based on breaking symmetries only on graphs with at least $\binom{m+a}{2}-2a-1$ edges we need an encoding of the AtMostK predicate that is suitable for our benchmark (see Section 2.1.3). Since this predicate will be used as a conditional in the expression BelongsTo($\mathcal{I}_{m+a}^{\leq 2a+1}$) $\rightarrow$ SymmBreaking($\mathcal{I}_{m+a}^{\leq 2a+1}$) and the predicate SymmBreaking will be built as a sum of products, it will be useful to encode the AtMostK predicate as a CNF formula so that the expression BelongsTo($\mathcal{I}_{m+a}^{\leq 2a+1}$) $\rightarrow$ SymmBreaking($\mathcal{I}_{m+a}^{\leq 2a+1}$) will be in DNF allowing us to again use a DNF to CNF transformation. Several encodings of the AtMostK predicate into CNF have been published in the literature, and the PySAT project [64] provides implementations for a number of them. For simplicity, we chose to use the default which is the sequential counter [101], but it would be a good topic for future work to determine if there is any advantage in using a different encoding for this part of the formula. Care then needs to be taken as this encoding introduces auxiliary variables which need to be placed at the appropriate levels of quantification.

5.4.3 Circuit Encoding

The disadvantages of requiring the matrix of some QBFs to be in CNF have been well documented [5, 68] and several formats to express Boolean formulas as circuits have been proposed. In this section, we explore encoding the formula $F^{m+a}(a_1, a_2, \ldots, a_k; m)$ as a circuit in the QCIR-14 format [70]. It is in principle straightforward to generate circuits from the definition of $F^{m+a}(a_1, \ldots, a_k; m)$, even incorporating perfect symmetry breaking predicates in CNF.

As in Section 5.4.2, we would like to generate instances with parameters that yield larger graph searches while still exploiting symmetry breaking to guarantee these instances have unique solutions. In order to do so we again turn to partial symmetry breaking using the ideas discussed in Section 2.3, namely, breaking symmetries only among graphs with at least $\binom{m+a}{2} - 2a - 1$ edges. Notice, however, that in contrast with the clausal encoding case where in order to encode the AtMostK predicate one typically needs to devise encodings of circuits into conjunctive or disjunctive clauses, one can directly encode circuits into the QCIR-14 format. Thus in order to obtain partial symmetry breaking based on the number of edges of the graph we use `QuAbS` [53] C API to encode a cardinality network based on Batcher's odd-even

merge sorting networks [8, 10]. Such a network requires $\mathcal{O}(n \log^2 n)$ gates for n inputs and in our case the number of inputs is $\binom{m+a}{2}$. A better encoding which we leave for future work would be based on k-cardinality networks [8] since $2a+1$ will typically be much smaller than $\binom{m+a}{2}$.

Regarding SymmBreaking($\mathcal{I}_{m+a}^{\leq 2a+1}$), the straightforward encoding from a sum of product to a circuit format will yield $\|\mathcal{I}_{m+a}^{\leq 2a+1}\|$ OR gates with $\binom{m+a}{2}$ inputs each. Such an encoding would then depend on $m+a$, yet we can do slightly better than that by noticing that for natural numbers N and k such that $N \geq 2k$ every graph on N vertices with at most k edges is isomorphic to a graph on N vertices with at most k edges in the subgraph defined by the first $2k$ vertices (assuming an ordering over the set of edges). Thus, whenever $m+a \geq 2(2a+1)$ we can "pin" the $2a+1$ missing edges to the subgraph defined by the first $4a+2$ vertices. This can be implemented with an AND gate connected to all the literals $x_{u,v}$ with $\max(u,v) > 4a+2$ and $\|\mathcal{I}_{4a+2}^{\leq 2a+1}\| = \|\mathcal{I}_{m+a}^{\leq 2a+1}\|$ OR gates with $\binom{4a+2}{2}$ inputs each. This shows that whenever $m+a > 4a+2$, the number of OR gates needed and the number of inputs per OR gate depend only on $a = \max(a_1, a_2, \ldots, a_k)$. Even further simplification of the resulting circuit could be achieved through methods that simplify sum-of-product circuits (for instance, techniques generalizing Karnaugh maps) but we leave these considerations for future work.

5.4.4 Case Studies

We implemented software to generate instances from our benchmark in QDIMACS[4] and QCIR-14 [70] formats, thus covering the majority of QSAT solvers available. We also implemented model enumerators for instances of this benchmark. These enumerators verify the correctness of the instances generated since they enumerate the unique solution modulo symmetries. The software and sample running times of some solvers on these instances are available online[5].

[4] http://www.qbflib.org/qdimacs.html
[5] https://doi.org/10.5281/zenodo.3548977

n \ v	6	7	8	9	10	11	12	13	14	15	16	17	18
3	15	9	3										
4	22	30	22	8									
5	44	63	81	73	52	19	6						
6	72	133	198	259	236	192	138	81	22	5			
7	120	302	490	666	868	972	653	463	368	241	127	27	5

Table 5.1: Number of $(C_n, K_4; v)_e$-good colorings for $3 \leq n \leq 7$.

5.5 Insight for Ramsey-type Problems via SAT

The previous sections presented contributions in which the encoding of graph arrowing problems using constraint satisfactions paradigms sits at the heart of the contribution. In this section, we highlight two other applications in which said encodings help by providing insights towards obtaining analytical results.

5.5.1 The Number of $(C_n, K_4)_e$-good Colorings

Let $(C_n, K_4; v)_e$ be the set of all $(C_n, K_4)_e$-good colorings of K_v, where C_n is the cycle on n vertices. Recall that the Ramsey number $R(C_n, K_4)$ is the minimum number N such that $K_N \to_e (C_n, K_4)$, so for any $m \geq R(C_n, K_4)$, $(C_n, K_4; m)_e$ is empty. The *Ramsey-critical* colorings for $R(C_n, K_4)$ are the colorings in $(C_n, K_4; R(C_n, K_4)-1)_e$, since these colorings indeed witness that $R(C_n, K_4)$ is the minimum order required to guarantee the arrowing property. Table 5.1 shows the number of $(C_n, K_4; v)_e$ good colorings for small values of n. This dataset was generated by exploiting the fact that all $(C_n, K_4; v+1)_e$ good colorings can be obtained from all the $(C_n, K_4; v)_e$ good colorings by adding one vertex and connecting it to every vertex in the original coloring, then coloring the new edges avoiding C_n in the first color and K_4 in the second color. The initial set $(C_n, K_4; v)_e$ for $v = 6$ was generated by enumerating all models of $\phi(K_6; C_n, K_4)$ and then keeping one representative from each isomorphism class using `nauty` [80].

We used the numbers in Table 5.1 to complement the following analytical result in Jayawardene, Narváez, and Radziszowski [69]:

Lemma 5.14 ([69]). *The set of $R(C_n, K_4)$-critical graphs consists of:*

- *Three critical graphs for $n = 3$.*

- *Eight critical graphs for* $n = 4$.
- *Six critical graphs for* $n = 5$.
- *Five critical graphs for* $n \geq 6$.

We can see how Lemma 5.14, whose proof in [69] is of analytical nature, provides a complete description of the critical cases for $R(C_n, K_4)$, while Table 5.1 provides additional information about how the number of colorings behave around the critical value. This kind of insight is important in Ramsey-type problems because it is often difficult (if not impossible) to detect patterns in the first few values for the parameters, so verification through computational methods is warranted.

5.5.2 The Vertex-Folkman Number $F_v(K_3, K_3; J_4)$

Let J_4 be the complete graph K_4 minus one edge. The Folkman number $F_v(K_3, K_3; J_4)$ is thus the smallest number N such that there exists a J_4-free graph G of order N such that $G \to_v (K_3, K_3)$. Xu, Liang and Radziszowski [115] proved that this number is in fact well defined, based on a purely analytical construction by Neetil and Rödl [86] which does not provide any concrete bound on the actual value of $F_v(K_3, K_3; J_4)$. Nevertheless, a simple randomized construction together with the translation of the vertex arrowing problem as presented in Section 5.2 gives a much better bound. In order to explain this construction, we first define J_4-free saturated graphs. A J_4-free graph $G = (V, E)$ is saturated if for any two vertices $u, v \in V$ such that $\{u, v\} \notin E$, the graph $(V, E \cup \{\{u, v\}\})$ contains J_4 as a subgraph. We generated random J_4-free saturated graphs of different orders and formulated the corresponding Boolean formulas encoding $G \not\to (K_3, K_3)$. The unsatisfiable instances thus provide witnesses of graphs G that are J_4-free and are such that $G \to (K_3, K_3)$. The smallest order for which our simple experiment provided a witness was 135, thus establishing a new bound $F_v(K_3, K_3; J_4) \leq 135$. As future work in this research direction, we will study these graphs in order to better understand how to use these examples to improve the upper bound analytically or refine our initial experiments to generate smaller examples.

Chapter 6

Computational Social Choice

Social choice theory studies among other things the decision making of multiple agents. This has of course been studied for centuries in the context of the larger field of political sciences. Nevertheless, decision making by multiple agents is not necessarily tied to human agents. In the age of artificial intelligence, it is increasingly common to run into applications where these agents are in fact ranking algorithms, expert systems and such. In large populations, e.g., city elections, the goal is typically to simplify the decision making process and rules like majority (or even dictatorship, the simplest one of all rules) are used. In contrast, the goal of smaller populations, e.g., committee elections, may be to take a decision that better represents the agents. What "better" represents the agents is obviously tied to a quantitative or qualitative measure of "good." Common measures are majority, fairness, and envy, among others. In order to achieve these goals, different mechanisms to aggregate preferences have been proposed and developed, with various levels of complexity. This gives raise to interesting combinatorial problems that stem from different scenarios in social choice. Computational social choice then studies the algorithmic aspects of these combinatorial problems.

In this chapter we provide the details of our contributions to computational social choice. Our main contribution is using answer set programming to formulate control problems for election systems where the winner problem is NP-hard. These election systems with hard winner problems have even harder control problems, and encoding and solving these problems using ASP is non-trivial.

6.1 Background and Definitions

The main concept from computational social choice that we will deal with in this chapter is that of *election systems*. Election systems consist of a set of candidates C, a set of voters V and a rule that determines the winner of the election based on the preferences of the voters. The preferences are provided as orders over the candidates in C, thus the order $>_v$ will stand for the preference of voter $v \in V$ over the candidates in C. Election systems are a source of interesting combinatorial problems since it is not always straightforward to determine a winner given some rule. For instance, a natural way to determine a winner is to choose a candidate that, once all preferences are aggregated, does not lose against any other candidate pairwise. This candidate is known as the *weak Condorcet* winner of an election and may not exist in some elections. A classical example of this issue is the following election, known as the Condorcet paradox:

$$a >_1 b >_1 c$$
$$c >_2 a >_2 b$$
$$b >_3 c >_3 a$$

In this example, candidate a is preferred over b by the majority of voters, candidate b is preferred over c by the majority of voters, but then candidate c is preferred over a by the majority of voters.

Given an election system and the set of preferences from the voters, the *winner problem* is the computational problem of determining a winner of the election, if one exists. Other problems are defined on top of the winner problem, and seek to model real-world situations where some adversary agent (not necessarily part of the election) is interested in affecting the result of the election. *Control problems* are decision problems of determining, given an election system and the voter preferences, if there are voters or candidates that can be added or deleted such that the result of the election can be altered with respect to a prescribed candidate. In the *constructive* version, the adversary agent seeks to guarantee that the prescribed candidate wins the election. In the *destructive* version, the objective of the adversary agent is to guarantee that the prescribed candidate does not win the election. By combining the

different parameters of control problems, one obtains different variations, e.g., constructive control by adding candidates (CCAC) or destructive control by deleting voters (DCDV).

Determining the winner of an election and finding a control action in an election are important aspects of computational social choice [18]. Computational complexity has provided a framework to classify how resistant are election systems to attacks like control [9].

6.2 Election Systems with Very Hard Control Problems

Of special interest to our research are election systems whose winner problems are harder than the hardest problems in NP. This is because these are the only elections systems that could possibly have control problems harder than the hardest problems in NP as well. Perhaps the easiest way to see this is true is to consider the contrapositive statement: Suppose we are in the constructive scenario and that the winner problem for an election system $\mathcal{E}$ is in P, then one can guess a control action and verify in polynomial time if the winner of the election is the desired candidate.

We focus on three popular election systems that have winner problems harder than the hardest problems in NP: the Kemeny, Dodgson and Young election systems. These three election systems have winner problems that are complete for the class Θ_2^p of decision problems that can be solved in nondeterministic polynomial time with parallel access to an NP oracle.

In the Kemeny election system, a winner of the election is a candidate that wins in a preference $>_K$ that minimizes the Kendall's τ distance

$$\sum_{a,b \in C, a >_K b} \|\{v \in V \mid b >_v a\}\|$$

from the set of voters. Hemaspaandra, Spakowski, and Vogel showed that determining the winner in this election system is Θ_2^p-complete [56].

In the Dodgson election system a winner is a candidate who can become a Condorcet winner (one that beats every other candidate) with the fewest number of swaps between adjacent candidates in the votes. In the Young election system the winner is the candidate who can become a weak Condorcet winner

by deleting the fewest voters. The winner problems for Dodgson and Young election systems were proven to be Θ_2^p-complete by Hemaspaandra, Hemaspaandra, and Rothe [55] and Rothe, Spakowski, and Vogel [98], respectively.

In [41] we prove that several variants of constructive control problems related to Kemeny, Young, and Dodgson elections are Σ_2^p-complete. $\Sigma_2^p = \mathrm{NP}^{\mathrm{NP}}$ is the class of problems solvable by a nondeterministic polynomial-time Turing machine with access to an NP oracle. Table 6.1, taken from [41], summarizes the results in that paper.

Table 6.1: Summary of Σ_2^p-completeness results for control from [41]. Kemeny$'$ refers to a natural variant of Kemeny defined in [35] and $(*)$ refers to the variant of control where the chair can delete only certain candidates.

	Adding	Deleting
Voters	Young Kemeny$'$	Young
Candidates	Kemeny Dodgson	Kemeny $(*)$ Dodgson

6.3 ASP Encodings

In recent years, ASP has been touted as a convenient paradigm to express social choice problems as CSPs. Starting with Konczak [73], several encodings have been provided over the years tackling different problems in voting theory. The work by Charwat and Pfandler [20] provides several ASP programs encoding winner problems for different election systems. These programs exploit the expressiveness of the input language of `gringo` [47] which allows computing some winner problems whose complexity is beyond NP, like those introduced in Section 6.2, via the use of aggregates. Their approach is heavily based on the *guess and check* approach mentioned in Section 2.2. For instance, for the winner problem in Kemeny elections, the *guess* part consists of guessing a preference that minimizes Kendall's τ distance with respect to the voters' preferences. In the *check* part, the guessed preference is discarded if there exists another preference with smaller distance to the voters' preferences. The interplay between these two parts (i.e., iteratively guessing a preference and

checking its minimality) naturally arrives at a minimum preference. The winner of that minimal preference is then the winner of the election.

We address the question of whether ASP would still be practical to express computational problems regarding election control problems that are not in NP. This is important because, although disjunctive ASP programs can express problems whose complexity is higher than NP in the polynomial hierarchy, doing so requires techniques that are more complicated than the standard *guess and check* approach described above, namely the saturation technique explained in Section 2.2.2.

We propose an approach that allows for expressing the control problem in a *guess and check* fashion where both the check and the guess parts are able to express problems in NP. The combined encoding returns an affirmative answer if and only if the guess program has an answer set for which the check program does not have an answer set. Figures 6.1 and 6.2 show the guess and check ASP programs for the control action of adding unregistered candidates to the election. These encodings utilize a syntax very similar to that of `gringo` [47], the input language for `clingo`. The parameters for this encoding include the limit L of unregistered candidates that can be added, and a set of unregistered candidates marked by the predicate ucand/1. Figure 6.1 encodes guessing a preference gpref/2 that has the property that the preferred candidate is ranked first (due to the last constraint), and the Kendall τ distance is recorded in the predicate gkt/1. Figure 6.2 then encodes guessing, given the preference gpref/2 guessed earlier, a preference cpref/2 that has strictly smaller Kendall τ distance with respect to the set of voters than that recorded in gkt/1 and is such that the preferred candidate is not ranked first. For our example, the answer to the problem of whether a control by adding candidates exists is *yes* if the guess program can guess a set of candidates to add and a preference over the resulting set of candidates for which the check program *cannot* guess a preference with a smaller distance where the preferred candidate does not rank first.

Our implementation builds on the ideas by Eiter and Polleres [38] combined with normalization of programs with aggregates [16]. The work of Eiter and Polleres [38] addresses the issue of having to craft cumbersome ASP encodings of Σ_2^p problems in the following way: They provide a template "meta-interpreter" and a transformation of ASP programs that can, together, be used to integrate an ASP program that guesses a solution to the Σ_2^p problem with

$\{\text{candidate}(C) : \text{ucandidate}(C)\}\ L \leftarrow \text{limit}(L).$
% A preference gpref/2 and a "worse" rank gwrankC/3
% are guessed as in Democratix [20]
$\text{gkt}(K) \leftarrow K = \#sum\{N, C_1, C_2 : \text{gwrankC}(C_1, C_2, N)\}.$
$\leftarrow \text{preferredCand}(C), not\ \text{gpref}(1, C).$

Figure 6.1: Relevant parts of the encoding of the guess part of Kemeny-CCAC.

% A new preference cpref/2 and corresponding rank cwrankC/3
% are guessed as in Figure 6.1
$\leftarrow \text{gkt}(K), K\ \#sum\{N, C_1, C_2 : \text{cwrankC}(C_2, C_1, N)\}.$
$\leftarrow \text{preferredCand}(X), \text{cpref}(1, X).$

Figure 6.2: Relevant parts of the encoding of the check part of Kemeny-CCAC.

an ASP program that checks whether the guessed solution is incorrect. The combination of these two then amounts to a "guess and check" encoding of a Σ_2^p problem. While this is a significant step towards simplifying the encoding of Σ_2^p problems as ASP programs, for our specific application it has the drawback that it does not support ASP programs extended with aggregates (e.g., `#count`). This is problematic for our ASP encodings since not using aggregates would lead to more complex, less intuitive programs.

To work around this issue, we need a way to transform ASP programs with aggregates into equivalent ASP programs that do not employ aggregates. The `lp2normal` tool [15] provides such a transformation. We combine `lp2normal` with meta-interpretation to express control problems with elections while harnessing the full expressive power of modern ASP input languages. Figure 6.3 illustrates the interaction between the several parts of this approach. We note that the transformation to eliminate aggregates is only needed for the check program, since the integration between the guess and the check programs only transforms the latter.

We implement the approach of Eiter and Polleres combined with normalization of certain aggregates like `#count` and `#sum` in order to provide an intuitive approach to solve problems in the second level of the polynomial hier-

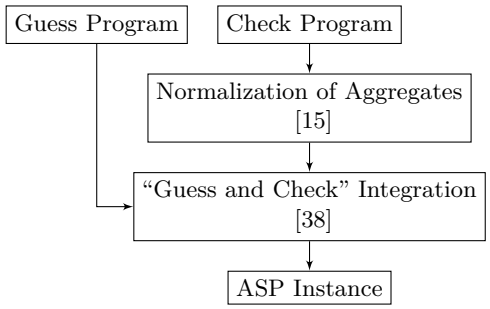


Figure 6.3: Illustration of our "guess and check" approach for problems in Σ_2^p.

archy. Our implementation is on top of the development version of the very popular `clingo` system for ASP.

6.4 Similar Encoding Approaches

The framework of *stable-unstable* semantics by Bogaerts, Janhunen and Tasharrofi [14] provides a similar "guess and check" strategy to solving problems in Σ_2^p. They mention that an advantage of the stable-unstable semantics is that they can be easily extended to represent problems at any level of the polynomial hierarchy. In their implementation both the guess and the check program are normalized, essentially discarding aggregates altogether. It is an interesting direction for future work to determine if not exploiting aggregates as implemented in modern ASP solvers like `clasp` [45] could hurt the performance of solvers employed in the task of solving very hard control problems in election systems.

Chapter 7

A Verified Symmetry Breaking Tool for CNF SAT

The contributions presented so far in this thesis, particularly those of Chapter 5, highlight the importance of the SAT problem (see Section 2.1) in combinatorial computing. The applications SAT solvers have found in theory and practice have brought along interesting consequences. On the one hand, it has spurred the development of an entire ecosystem of related software, from preprocessors to software that verifies proofs of unsatisfiability. On the other hand, it has increased the complexity of SAT solvers themselves as they are used to tackle larger instances with intricate structures. The fact that SAT solvers are becoming more complex while at the same time being used in more mission-critical scenarios has sparked interest in the verification of SAT solvers [12, 79, 100]. Nevertheless, applications usually depend not only on the SAT solver employed to find satisfying assignments (models) of the generated instances but also on preprocessing and postprocessing tools. For instance, most SAT solvers available accept the input formula in DIMACS format[1], which assumes the input is in conjunctive normal form (CNF), so depending on the application, a CNF transformation may be needed as a preprocessing step, introducing a new trust issue.

The particular type of preprocessing tool we focus on in this chapter is syntactic symmetry breaking tools, as those discussed in Section 2.3. The

[1] https://www.satcompetition.org/2009/format-benchmarks2009.html

brief overview provided in Section 2.3, although by no means thorough, is enough to make the argument that by using today's tools in syntactic symmetry breaking we are leveraging decades of work. At the heart of this work is the concept introduced by Crawford [24, 25], still present in modern tools: the representation of a Boolean formula as a graph whose automorphisms correspond to syntactic symmetries in the input formula. Thus formalizing this idea would be a step towards formalizing state-of-the-art tools.

We initiated the formalization in the Prototype Verification System [89] (PVS) of the graph construction by Crawford [24,25] and the verification of the fundamental property linking the automorphisms of this graph with symmetric assignments of the variables. Our formalization is based on the `graphs` theory from NASA's PVS library[2].

7.1 Formalizing Crawford's Symmetry Breaking

We begin our formalization by defining a new datatype for literals with constructors for positive and negative literals. The type `CNFClause` is then defined as an alias for a finite set of literals, and the type `CNFFormula` is in turn defined as a finite set of clauses.

```
lit: DATATYPE
BEGIN
 poslit(n: nat): poslit?
 neglit(n: nat): neglit?
END lit

CNFClause: TYPE = finite_set[lit]

CNFFormula: TYPE = finite_set[CNFClause]
```

We define some natural operations over literals and assignments, namely the `neg` function which returns the negation of a literal, and the `litval` function which takes a literal l and an assignment a and returns the Boolean value assigned to l under a. We omit the definition of these for the sake of space. We define the `models` predicate indicating an assignment models a formula

[2] `https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html`

and the SAT predicate indicating a satisfying assignment exists for the input formula.

```
models(F: CNFFormula, assignment: [nat -> bool]): bool =
FORALL (c: (F)): EXISTS (l: (c)): litval(assignment, l) = True

SAT(F: CNFFormula): bool =
EXISTS (assignment: [nat -> bool]): models(F, assignment)
```

Next, we define the vertex datatype, the type of the vertices of our graph. A vertex can be a literal vertex or a clause vertex so our datatype has a constructor for each.

```
vertex: DATATYPE
BEGIN
 litvertex (l: lit): litvertex?
 clausevertex (C: CNFClause): clausevertex?
END vertex
```

We are now prepared to define the functions that will extract the vertices that are used to build the graph of a formula. These are essentially formula_lits which collects all the literals that appear in a formula, and the functions lit_vertices and clause_vertices which map a set of literals to a set of literal vertices, and CNF formulas to sets of clause vertices, respectively.

```
formula_lits(F: CNFFormula): finite_set[lit] =
lits_for_vars(formula_vars(F))

lit_vertices(L: finite_set[lit]): finite_set[vertex] =
image(litvertex, L)

clause_vertices(F: CNFFormula): finite_set[vertex] =
image(clausevertex, F)
```

We also define functions to generate the edges of the graph G_F. We take a similar approach as with the vertices of G_F, defining functions that output different sets of edges. The function connect_lits takes a set of natural numbers (variables, in our domain) and generates a set of edges connecting

the positive and negative literals of those variables. The `connect_clauses` function connects the clauses of a CNF formula to the literals they contain. From Definition 2.3 it follows that the set of edges of G_F is the union of the output of these functions.

```
connect_lits(V: finite_set[nat]): finite_set[doubleton[vertex]] =
image(
  (LAMBDA (v: nat): dbl(litvertex(poslit(v)), litvertex(neglit(v)))), V)

connect_clause_internal(C: CNFClause, D: CNFClause):
finite_set[doubleton[vertex]] =
image((LAMBDA (l: lit): dbl(clausevertex(C), litvertex(l))), D)

connect_clause(C: CNFClause): finite_set[doubleton[vertex]] =
connect_clause_internal(C , C)

connect_clauses(F: CNFFormula): finite_set[doubleton[vertex]] =
IUnion[(F),doubleton[vertex]](connect_clause)
```

With the definitions above we are finally ready to define what the graph of a formula F is and prove a *type-correctness condition (TCC)* generated by PVS. The TCC generated asks us to prove that if an edge is in the union of the sets `connect_lits(formula_vars(F))` and `connect_clauses(F)`, then both of the members of the edge are in the union of `lit_vertices(formula_lits(F))` and `clause_vertices(F)`.

```
formula_graph(F: CNFFormula): graph[vertex] =
 (# vert := union(lit_vertices(formula_lits(F)), clause_vertices(F)),
 edges:= union(connect_lits(formula_vars(F)), connect_clauses(F)) #)
```

A fundamental property of this graph is that a literal l is a member of a clause C of a formula F if and only if the edge $\{C, l\}$ is in the set of edges of the graph. We prove this property as a separate lemma.

```
graph_clause_lit: LEMMA
FORALL (F: CNFFormula, C: CNFClause, l: lit):
 (member(C, F) AND member(l, C)) IFF
 member(dbl(clausevertex(C), litvertex(l)), edges(formula_graph(F)))
```

The next concept we need to formalize is that of automorphisms. In particular, for our application we need to define color-preserving automorphisms. Notice that our definition of a vertex does not explicitly allow for us to specify a color, but the separate constructors naturally induce a coloring on the vertex datatype. We leverage the induced coloring of vertices by using the `ord` function in PVS [90]. An overload of this function which assigns a natural number to each constructor of an abstract datatype is automatically generated by PVS when a new abstract datatype is defined. We start by defining the concept of a (color-preserving) vertex permutation, which is a nonempty type dependent on a graph. One would naturally think this should be defined as a bijection from the set of vertices of the graph to itself, but this generates some incompatibilities with the way graphs are defined in the library. Consider the following natural definition of a vertex permutation, which is admissible under PVS' syntax, and a theorem stating a fairly obvious property regarding a doubleton of permuted vertices and an edge of G.

```
vertex_permutation(G: graph[vertex]): TYPE+ =
{π: [(vert(G)) -> (vert(G))] | bijective?(π) AND
 FORALL (v: (vert(G))): ord(v) = ord(π(v))} CONTAINING id

bad: THEOREM
FORALL (G: graph[vertex], π: vertex_permutation(G),
        x: (vert(G)), y: (vert(G))):
 nonempty?(edges(G)) AND
 choose(edges(G)) = dbl(π(x), π(y)) IMPLIES
  member(π(x), choose(edges(G)))
```

Theorem `bad` seems to follow immediately from the `dbl_in` lemma which is part of the `graphs.doubleton` theory in PVS:

```
dbl_in: LEMMA D = dbl(x,y) IMPLIES D(x) AND D(y)
```

Nevertheless, once we skolemize all the quantified variables, we see what PVS really thinks of the assumption `choose(edges(G!1)) = dbl(π(x),π(y))`. Because of the way NASA's graph library declares the set of edges of a graph, `choose(edges(G))` is of type `doubleton[vertex]`, yet defining π as `[(vert(G)) -> (vert(G))]` makes the expression `dbl(π!1(x!1), π!1(y!1))`

to be of type `doubleton[(vert(G))]`. Because of this, the `dbl_in` lemma does not apply to the antecedent -2.

```
{-1}  nonempty?(edges(G!1))
{-2}  choose(edges(G!1)) =
       extend[vertex, (vert(G!1)), bool, FALSE](dbl(π!1(x!1), π!1(y!1)))
  |-------
{1}   member(π!1(x!1), choose(edges(G!1)))

Rule? (forward-chain "dbl_in")
No forward match for dbl_in.
No change on: (FORWARD-CHAIN "dbl_in")
```

The fix in this particular example is to make the right coercion explicit by assuming that `choose(edges(G!1)) = dbl[vertex](π(x),π(y))`. Unfortunately, for our larger use of vertex permutations, having many of these simple fixes amounts to very cluttered statements. Instead, we define them as bijective functions from the set of vertices to itself such that vertices of the graph are mapped to vertices of the graph and the color of the vertex is preserved (in the sense explained before, based on the `ord` function).

```
vertex_permutation(G: graph[vertex]): TYPE+ =
{π: [vertex -> vertex] | bijective?(π) AND
  (FORALL (v: vertex): member(v, vert(G))
    IMPLIES member(π(v), vert(G))) AND
  FORALL (v: vertex): ord(v) = ord(π(v))}
CONTAINING id
```

We now define an operation `permute_edges` on graphs which generates a new set of edges using a vertex permutation of the graph. It will then allow us to define automorphisms as the nonempty type of vertex permutations for which this operation preserves the set of edges.

```
permute_edges(G: graph[vertex], π: vertex_permutation(G)):
finite_set[doubleton[vertex]] =
{ e: doubleton[vertex] |
  EXISTS (x: vertex, y: vertex):
    member(x, vert(G)) AND member(y, vert(G)) AND x /= y AND
```

```
    member(dbl(x, y), edges(G)) AND e = dbl(π(x), π(y)) }

automorphism(G: graph[vertex]): TYPE+ =
{ π: vertex_permutation(G) | permute_edges(G, π) = edges(G) }
CONTAINING id
```

A fundamental property that follows from the structure of G_F and the fact that automorphisms preserve the set of edges is the following:

Lemma 7.1. *If ϕ is an automorphism of G_F, then for any literal l in F, ϕ maps $\bar{l}$ to $\overline{\phi(l)}$.*

Proof. $\phi(l)$ is a literal from the formula, so $\{\phi(l), \overline{\phi(l)}\}$ is an edge of G_F and there must exist vertices x, y in G_F such that $\{\phi(l), \overline{\phi(l)}\} = \{\phi(x), \phi(y)\}$. One of x or y, say x, must be l because ϕ is injective, and that forces y to be $\bar{l}$ since ϕ is color preserving and the only edges connecting literals to literals connect every literal to its negation. □

We show the formalization of Lemma 7.1 below, where we can see how the abuse of notation throughout our informal proof of the lemma (using ϕ as both a permutation of the vertices of G_F as well as a permutation of literals) is made explicit and causes some clutter. Nevertheless, it is worth noting it could be worse: since `ϕ` is of type `[vertex->vertex]`, the expression `phi(litvertex(l))` could in principle be a `clausevertex`, in which case we would not be able to use the `l` accessor. Instead of flagging this as an issue, PVS will formulate a type-correctness condition (TCC) asking to prove that `phi(litvertex(l))` is a `litvertex`, which is easy to show from the fact that `phi` is color-preserving. We include the generated TCC below. An important issue to point out is that, despite the fact that Lemma 7.1 is fairly straightforward, proving the `mapped_lit_neg` lemma required laborious case analysis.

```
mapped_lit(F: CNFFormula, l: (formula_lits(F)),
           ϕ: automorphism(formula_graph(F))) : lit =
l(ϕ(litvertex(l)))

mapped_lit_TCC1: OBLIGATION
  FORALL (F: CNFFormula, l: (formula_lits(F)),
```

```
          φ : automorphism(formula_graph(F))):
    litvertex?( φ (litvertex(l)));

mapped_lit_neg: LEMMA
FORALL (F: CNFFormula, l: lit):
 member(l, formula_lits(F)) IMPLIES
  (FORALL ( φ : automorphism(formula_graph(F))):
   mapped_lit(F, l, φ ) = neg(mapped_lit(F, neg(l), φ )))
```

The last piece we need to formalize before stating and proving Theorem 2.4 is the notion of permuting an assignment. The crucial property of this operation is that, if l is a literal of a formula F, the value of $\phi(l)$ under an assignment σ is the value of l under the assignment $\sigma \circ \phi$. We have intentionally given a provocatively simple description of this property which seems to go without saying. In reality, the description is making use of some notation overloading that we cannot obviate when formalizing this property in PVS: ϕ is a vertex permutation, so a composition of ϕ with σ is impossible—not even if we consider σ as an assignment of literals, which is another overloading we employ in our description. Because of all the overloading that is needed to express this property, we state it as a lemma whose proof actually requires rather heavy case analysis.

```
permute_assignment(F: CNFFormula, a: [nat->bool],
                   φ : automorphism(formula_graph(F))): [nat->bool] =
LAMBDA (n: nat): litval(a, mapped_lit(F, poslit(n), φ ))

permute_assignment_preserves: LEMMA
FORALL (F: CNFFormula, a: [nat->bool], l: lit,
        φ : automorphism(formula_graph(F))):
 member(l, formula_lits(F)) IMPLIES
  litval(a, mapped_lit(F, l, φ )) =
  litval(permute_assignment(F, φ , a), l)
```

We are finally ready to state and prove Theorem 2.4. We can in fact state a weaker version of this theorem: that if σ models F and ϕ is an automorphism of G_F, then $a \circ \phi$ models F. We can then use that theorem and the fact that ϕ^{-1} is an automorphism of G_F as well to prove Theorem 2.4. As mentioned before, the proof relies heavily on the `graph_clause_lit` lemma. It

also uses the `permute_assignment_preserves` lemma and the fact that, since an automorphism is injective and maps the set of vertices of G_F to itself, its inverse also maps the set of vertices of G_F to itself. Notice that Theorem 2.4 can be easily stated if we allow for an abuse of notation and consider ϕ both a permutation of the vertices of G_F *and* a permutation of literals. In fact, a closer look at the composition $\sigma \circ \phi$ reveals that we are both recasting σ as an assignment of literals and ϕ as a permutation of literals. This will prove somewhat problematic since PVS will require us to make these casts explicit, thus increasing the complexity of the proofs.

```
Crawford_imp: THEOREM
FORALL (F: CNFFormula, a: [nat->bool],
        ϕ: automorphism(formula_graph(F))):
models(F, a) IMPLIES models(F, permute_assignment(F, ϕ, a))

Crawford: THEOREM
FORALL (F: CNFFormula, a: [nat->bool],
        ϕ: automorphism(formula_graph(F))):
models(F, a) IFF models(F, permute_assignment(F, ϕ, a))
```

We now focus on defining a total order over the assignments of a formula. This proved to be tricky given our choice of defining assignments as functions from the natural numbers to the Boolean domain. We define a relation `assignment_leq` between two assignments σ_1 and σ_2 which depends on a list of variables. This relation essentially says that σ_1 is lexicographical less or equal to σ_2 over the given list of variables. Notice that the list of variables induces an order over the variables. There are a number of ways to define this relation, we chose to define it in a recursive fashion which will facilitate proofs by induction.

```
assignment_leq (vars: list[nat])(a1: [nat->bool], a2: [nat->bool]):
RECURSIVE [bool] =
CASES vars OF
  null: TRUE,
  cons(v, rest_vars):
 IF a1(v) THEN
   IF a2(v) THEN assignment_leq(rest_vars)(a1, a2) ELSE FALSE ENDIF
 ELSE
```

```
   IF a2(v) THEN TRUE ELSE assignment_leq(rest_vars)(a1, a2) ENDIF
 ENDIF
ENDCASES
MEASURE vars by <<
```

On the other hand, it will sometimes be helpful to work with an explicit definition of what this recursive function means. This will be particularly helpful later on to prove that there is a lexicographical minimum model for any satisfiable formula. In order to work with this alternative formulation, we define a predicate over natural numbers `assignments_differ` which depends on two assignments σ_1 and σ_2 stating that these two assignments differ on a given number. We then prove an equivalence between the recursive definition of `assignment_leq` and the property that if the two assignments σ_1 and σ_2 differ over a list of variables and σ_1 is less or equal to σ_2, then σ_1 assigns *False* to the first variable where they both differ.

```
assignments_differ(a1, a2: [nat->bool])(n: nat): bool =
a1(n) /= a2(n)

assignment_leq_alt: LEMMA
FORALL (vars: list[nat], a1: [nat->bool], a2: [nat->bool]):
assignment_leq(vars)(a1, a2) IFF
  (some(assignments_differ(a1, a2))(vars) IMPLIES
  NOT a1(nth(vars, find_first(vars, assignments_differ(a1, a2)))))
```

Notice that `assignment_leq` as defined is not a total order over assignments. In order to use this definition as a total order, we need to consider a restricted version `assignment_leq_restricted` which also depends on a list of variables but is defined over functions that assign specifically those variables in the list. This definition uses the `extend` function in PVS to extend the assignments over a list of variables to assignments of natural numbers in order to define this restricted predicate in terms of `assignment_leq`.

```
assignment_leq_restricted(D: list[nat])
(a1, a2: [(set_as_list.list2set(D))->bool]): bool =
assignment_leq(D)(
  extend[nat,(set_as_list.list2set(D)),bool,TRUE](a1),
  extend[nat,(set_as_list.list2set(D)),bool,TRUE](a2))
```

```
assignment_leq_iff_restricted: LEMMA
FORALL (D: list[nat], S: finite_set[nat], a1, a2: [nat->bool]):
set_as_list.list2set(D) = S IMPLIES
  (assignment_leq(D)(a1, a2) IFF
    assignment_leq_restricted(D)(
      restrict[nat,(S),bool](a1),
      restrict[nat,(S),bool](a2)))

assignment_leq_restricted_total_order: LEMMA
FORALL (D: list[nat], S: set[nat]):
set_as_list.list2set(D) = S IMPLIES
  total_order?[[(S)->bool]](assignment_leq_restricted(reverse(D)))
```

We mention in passing that the `assignment_leq_restricted_total_order` lemma was the most challenging lemma to prove in the entire specification.

We now need a way to encode the symmetry breaking predicate $P(\pi)$ described in Section 2.3 as a CNF formula. The standard way to do this is to use auxiliary variables $e_i \equiv (l_i = \pi(l_i))$ which would lead to an encoding of the predicates that is quadratic in the number of variables in the formula. While implementing this in a procedural fashion is straightforward, implementing it in a functional fashion in a way that lends itself to proving properties about the encoding is more involved. We opted for an alternative approach: expressing the $P(\pi)$ predicate entirely in terms of the variables of F without employing auxiliary variables. A way to do this is to notice that predicates of the type $\left(\bigwedge_{j=1}^{i-1}(l_{i-1} \equiv \phi(l_{i-1}))\right) \rightarrow (l_i \rightarrow \pi(l_i))$ are equivalent to $\left(\bigvee_{j=1}^{i-1}((\overline{l_i} \wedge \pi(l_i)) \vee (l_i \wedge \overline{\pi(l_i)}))\right) \vee (l_i \rightarrow \pi(l_i))$ and $\bigvee_{j=1}^{i-1}((\overline{l_i} \wedge \pi(l_i)) \vee (l_i \wedge \overline{\pi(l_i)}))$ is in *disjunctive normal form* (DNF). Then we can employ a brute-force transformation from DNF to CNF which, although adding a number of clauses that is exponential in the number of variables in the formula, is theoretically simple enough to fit our purposes. We implement this transformation in a recursive function `add_symbreaking_predicates_dnf_helper` but we omit its definition here for the sake of clarity. We do, however, present the lemma formalizing the main property of this function, namely, that an assignment models the formula output by the `add_symbreaking_predicates_dnf_helper` if and only

if there is a literal i in the list of variables for which a assigns different values to l_i and $\phi(l_i)$.

```
add_symbreaking_predicates_dnf_helper_neq: LEMMA
FORALL (F: CNFFormula, vars: finite_set[nat],
        ϕ : automorphism(formula_graph(F)), a: [nat->bool]):
models(add_symbreaking_predicates_dnf_helper(F, vars, ϕ ), a) IFF
(EXISTS (v: (vars)):
  litval(a, poslit(v)) /= litval(a, mapped_lit(F, poslit(v), ϕ )))
```

In order to build the predicate $P(\pi)$ we process the list of variables as follows: Let the head of the list be the variable v, and let l_v be the positive literal of v. We take the output of `add_symbreaking_predicates_dnf_helper` applied to the rest of the list and add $\overline{l_v}$ and $\phi(l_v)$ to each clause, then repeat the same procedure for the rest of the list.

```
add_symbreaking_predicates_helper(F: CNFFormula, vars: list[nat],
  ϕ : automorphism(formula_graph(F))): RECURSIVE CNFFormula =
CASES vars OF
  null: emptyset[CNFClause],
  cons(v, rest_vars):
  union(
    add_lit_to_clauses(
      add_lit_to_clauses(
      add_symbreaking_predicates_dnf_helper(F, list2set(rest_vars), ϕ ),
      mapped_lit(F, poslit(v), ϕ )),
    neglit(v)),
    add_symbreaking_predicates_helper(F, rest_vars, ϕ ))
ENDCASES
MEASURE vars BY <<
```

Notice the procedure above imposes a lexicographical order of the variables in the list using the *reverse* of the order induced by the list, as stated in the following lemma.

```
models_add_symbreaking_predicates_helper_leq: THEOREM
FORALL (F: CNFFormula, ϕ : automorphism(formula_graph(F)),
        a: [nat->bool], vars: list[nat]):
```

```
models(add_symbreaking_predicates_helper(F, vars, ϕ), a) IFF
assignment_leq(reverse(vars))(a, permute_assignment(F, ϕ, a))
```

We define another function that takes a list of automorphisms of the formula graph of F and adds symmetry breaking predicates for all of them.We also prove a simple lemma stating a model σ of the formula output by such function will be less or equal to $\sigma \circ \phi$ for every automorphism ϕ in the list.

```
add_symbreaking_predicates(F: CNFFormula,
  ϕs: list[automorphism(formula_graph(F))]): CNFFormula =
reduce(F,
 LAMBDA (ϕ: automorphism(formula_graph(F)), G: CNFFormula):
  union(
   add_symbreaking_predicates_helper(F, set2list(formula_vars(F)), ϕ),
   G))(ϕs)

add_symbreaking_predicates_models: LEMMA
FORALL (F: CNFFormula, ϕs: list[automorphism(formula_graph(F))],
        a: [nat->bool]):
models(add_symbreaking_predicates(F, phis), a) IFF
(models(F, a) AND
  FORALL (ϕ: automorphism(formula_graph(F))):
  member(ϕ, ϕs) IMPLIES
    assignment_leq(reverse(set2list(formula_vars(F))))(
      a, permute_assignment(F, ϕ, a)))
```

Finally, we prove (a list version of) Theorem 2.5. The fact that F is satisfiable if the formula together with the symmetry breaking predicates is satisfiable follows easily from the fact that F is a subset of that formula (see the base case of the `reduce` function call above). The interesting part of this proof is providing a model that will satisfy all the symmetry breaking predicates given that F is satisfiable. We provide the extension of the lexicographical minimum of the set of satisfying assignments of the variables of the formula as a witness. In order to do this, PVS will ask us to prove (a) that the set of satisfying assignments of the variables of F is non-empty, (b) that `assignment_leq_restricted` is in fact a total order over assignments of the variables of F, and (c) that the lexicographical minimum satisfies all the symmetry breaking predicates appended to F. Part (a) is true because the

restriction of the witness to the satisfiability of F belongs to the set, part (b) is true by lemma `assignment_leq_restricted_total_order`, and part (c) follows from Theorem 2.4.

```
symbreaking: THEOREM
FORALL (F: CNFFormula, phis: list[automorphism(formula_graph(F))]):
SAT(F) IFF SAT(add_symbreaking_predicates(F, phis))
```

This concludes our formalization of symmetry breaking for Boolean formulas in CNF. The complete development of our formalization, which consists of 61 formulas at the time of this writing, is available online[3].

7.2 Conclusions and Future Work

In this chapter, we presented the formalization in PVS of the syntactic symmetry breaking technique for Boolean formulas in conjunctive normal form (CNF) introduced by Crawford [24,25]. We discussed the main components of our formalization and the challenges we faced formulating and proving them.

There are several directions for future work that could stem from this work. One relatively simple improvement would be to specify and prove properties for an encoding of the symmetry breaking predicate $P(\pi)$ that uses auxiliary variables and adds a number of clauses that is polynomial in the number of variables of the formula. Our choice of representing assignments as functions from natural numbers to the Boolean domain would allow for using auxiliary variables without affecting the proof of `symbreaking`. This would bring our formalization closer to the actual implementations available in tools like `Shatter` [3].

Related to the extension discussed above, an interesting direction for future work would be to obtain executable code out of this formalization and compare it to the performance of tools like `Shatter` [3] and `BreakID` [32]. PVSio[4] [85] could in principle help in this task, yet one big roadblock in using it for our purpose is that basic functions related to finite sets (like adding elements to the set and calculating set unions) are, to the best of our knowledge, not supported by default. Adding support for these would be a major improvement towards

[3] doi:10.5281/zenodo.2597138.

[4] `https://shemesh.larc.nasa.gov/people/cam/PVSio/`

being able to run our specification and compare it to the CNF symmetry breaking tools available.

Chapter 8

Conclusions

Research in combinatorial computing can benefit from recent advances in constraint satisfaction paradigms. Nevertheless, throughout this thesis we have shown that in the particular case of abstract combinatorial problems, applying constraint satisfaction techniques is often not easy. In order to successfully apply these techniques one needs to take care of several aspects that range from theoretical to practical concerns, including the verification of the results.

Our contributions started from the theoretical end of this range. In Chapter 4 we looked at issues regarding using backdoors of Boolean formulas and backbones to Boolean formulas in conjunctive normal form to aid SAT solvers. We showed that, under very plausible assumptions, there is a complexity gap between the search and decision problems for these hidden structures of Boolean formulas.

In Chapter 5 we presented our contributions to using constraint satisfaction techniques for graph arrowing problems. We presented some options to encode these problems as Boolean formulas and as answer set programs. We then analyzed some properties of the Boolean formulas encoding the arrowing property when paired with symmetry breaking tools. We also described a benchmark for quantified Boolean formulas based on vertex Folkman problems. Finally, we showed how applying constraint satisfaction techniques to these problems can help in gaining insight to theoretical results.

Chapter 6 presented a contribution in computational social choice. In this field there are control problems for election systems whose computational complexity is higher than NP. The "guess and check" approach of answer set

programming for these control problems is thus hard to achieve with standard encoding techniques. We showed how to restore the ability to address these problems through a familiar guess-and-check approach by using program transformations to combine two answer set programs targeting NP-complete problems.

Chapter 7 addressed the issue of trusting the results obtained through constraint satisfaction techniques. We initiated the formalization of a seminal syntactic symmetry breaking technique by Crawford. We used the Prototype Verification System [89] for this formalization.

Several of the ideas presented in this thesis have the potential of being further developed and extended. The guess-and-check approach for high-complexity problems shown in Chapter 6 can be applied to many other combinatorial problems outside of computational social choice. Also, the development started in Chapter 7 towards a formalized symmetry breaking tool remains to be completed. It should also be extended to answer set programs as in the work of Devriendt et al. [33].

Bibliography

[1] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *Proceedings of the 39th annual Design Automation Conference*, pages 731–736. ACM, 2002.

[2] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult instances of Boolean satisfiability in the presence of symmetry. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 22(9):1117–1137, 2003.

[3] Fadi A Aloul, Karem A Sakallah, and Igor L Markov. Efficient symmetry breaking for boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, 2006.

[4] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. On the structure of industrial SAT instances. In Ian P. Gent, editor, *15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 127–141. Springer, 2009.

[5] Carlos Ansótegui, Carla P. Gomes, and Bart Selman. The Achilles' heel of QBF. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 275–281. AAAI Press / The MIT Press, 2005.

[6] K. Appel and W. Haken. Every planar map is four colorable. Part I: Discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 09 1977.

[7] K. Appel, W. Haken, and J. Koch. Every planar map is four colorable. Part II: Reducibility. *Illinois Journal of Mathematics*, 21(3):491–567, 09 1977.

[8] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

[9] John J. Bartholdi III, Craig A. Tovey, and Michael A. Trick. How hard is it to control an election? *Mathematical and Computer Modelling*, 16(8-9):27–40, 1992.

[10] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[11] Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[12] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *Journal of Automated Reasoning*, 61(1-4):333–365, 2018.

[13] Bart Bogaerts, Tomi Janhunen, and Shahab Tasharrofi. Solving QBF instances with nested SAT solvers. In Adnan Darwiche, editor, *AAAI Workshop: Beyond NP*, volume WS-16-05 of *AAAI Workshops*. AAAI Press, 2016.

[14] Bart Bogaerts, Tomi Janhunen, and Shahab Tasharrofi. Stable-unstable semantics: Beyond NP with normal logic programs. *TPLP*, 16(5-6):570–586, 2016.

[15] Jori Bomanson, Martin Gebser, and Tomi Janhunen. Improving the normalization of weight rules in answer set programs. In *14th European Conference On Logics In Artificial Intelligence (JELIA-14)*, pages 166–180, September 2014.

[16] Jori Bomanson and Tomi Janhunen. Normalizing cardinality rules using merging and sorting constructions. In Pedro Cabalar and Tran Cao Son, editors, *LPNMR*, volume 8148 of *Lecture Notes in Computer Science*, pages 187–199. Springer, 2013.

[17] Allan Borodin and Alan Demers. Some comments on functional self-reducibility and the NP hierarchy. Technical Report TR 76-284, Department of Computer Science, Cornell University, Ithaca, NY, July 1976.

[18] Felix Brandt, Vincent Conitzer, Ulle Endriss, Jérôme Lang, and Ariel D. Procaccia, editors. *Handbook of Computational Social Choice*. Cambridge University Press, 2016.

[19] Gary Chartrand and Seymour Schuster. On the existence of specified cycles in complementary graphs. *Bulletin of the American Mathematical Society*, 77:995–998, 1971.

[20] Günther Charwat and Andreas Pfandler. Democratix: A declarative approach to winner determination. In Toby Walsh, editor, *Proceedings of the 4th international conference on Algorithmic decision theory*, volume 9346 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2015.

[21] Michael Codish, Michael Frank, Avraham Itzhakov, and Alice Miller. Computing the Ramsey number $R(4, 3, 3)$ using abstraction and symmetry breaking. *Constraints*, 21(3):375–393, 2016.

[22] Stephen A Cook. The complexity of theorem-proving procedures. In *3rd Annual ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[23] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, March 1988.

[24] James Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. In *AAAI Workshop on Tractable Reasoning*, pages 17–22, 1992.

[25] James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *International Conference on the Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, 1996.

[26] Luís Cruz-Filipe, Marijn Heule, Warren Hunt, Matt Kaufmann, and Peter Schneider-Kamp. Efficient certified RAT verification. In *International Conference on Automated Deduction*, pages 220–236. Springer, 2017.

[27] Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, September 2001.

[28] Jared Davis and Sol Swords. Verified AIG algorithms in ACL2. *Electronic Proceedings in Theoretical Computer Science*, pages 95–110, April 2013.

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

[30] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, July 1960.

[31] Nachum Dershowitz, Ziyad Hanna, and Jacob Katz. Bounded model checking with QBF. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 408–414. Springer, 2005.

[32] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In Nadia Creignou and Daniel Le Berre, editors, *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 104–122. Springer, 2016.

[33] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. On local domain symmetry for model expansion. *Theory and Practice of Logic Programming*, 16(5-6):636–652, 2016.

[34] Bistra N. Dilkina, Carla P. Gomes, and Ashish Sabharwal. Tradeoffs in the complexity of backdoor detection. In Christian Bessiere, editor, *13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2007.

[35] Cynthia Dwork, Ravi Kumar, Moni Naor, and Dandapani Sivakumar. Rank aggregation methods for the web. In *Proceedings of the 10th international conference on World Wide Web*, pages 613–622, 2001.

[36] Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15(3):289–323, Sep 1995.

[37] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. *Answer Set Programming: A Primer*. Springer, 2009.

[38] Thomas Eiter and Axel Polleres. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *Theory and Practice of Logic Programming*, 6(1-2):23–60, January 2006.

[39] Jan Elffers, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Justifying all difference using pseudo-Boolean reasoning. In *AAAI-20*, 2020. To appear.

[40] Stephen A. Fenner, Lance Fortnow, Ashish V. Naik, and John D. Rogers. Inverting onto functions. *Information and Computation*, 186(1):90–103, 2003.

[41] Zack Fitzsimmons, Edith Hemaspaandra, Alexander Hoover, and David E. Narváez. Very hard electoral control problems. In *33rd AAAI Conference on Artificial Intelligence*, volume 33, pages 1933–1940. AAAI Press, 2019.

[42] Jon Folkman. Graphs with monochromatic complete subgraphs in every edge coloring. *SIAM Journal on Applied Mathematics*, 18(1):19–24, 1970.

[43] Zvi Galil. On some direct encodings of nondeterministic Turing machines operating in polynomial time into P-complete problems. *SIGACT News*, 6(1):19–24, January 1974.

[44] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Answer set solving in practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(3):1–238, 2012.

[45] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Proceedings of the 9th international conference on logic programming and nonmonotonic reasoning*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

[46] M. Gebser, B. Kaufmann, and T. Schaub. Solution enumeration for projected Boolean search problems. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 71–86. Springer, 2009.

[47] Martin Gebser, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Sven Thiele. On the input language of ASP grounder gringo. In Esra Erdem, Fangzhen Lin, and Torsten Schaub, editors, *10th International Conference Logic Programming and Nonmonotonic Reasoning (LPNMR 2009)*, volume 5753 of *Lecture Notes in Computer Science*, pages 502–508. Springer, 2009.

[48] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3-4):365–385, 1991.

[49] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 633–654. IOS Press, 2009.

[50] Georges Gonthier. Formal proof–the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[51] Orna Grumberg, Assaf Schuster, and Avi Yadgar. Memory efficient all-solutions sat solver and its application for reachability analysis. In

Alan J. Hu and Andrew K. Martin, editors, *FMCAD*, volume 3312 of *Lecture Notes in Computer Science*, pages 275–289. Springer, 2004.

[52] Juris Hartmanis and Lane A Hemachandra. Complexity classes without machines: On complete languages for UP. *Theoretical Computer Science*, 58(1-3):129–142, 1988.

[53] Jesko Hecking-Harbusch and Leander Tentrup. Solving QBF by abstraction. In Andrea Orlandini and Martin Zimmermann, editors, *GandALF*, volume 277 of *EPTCS*, pages 88–102, 2018.

[54] Edith Hemaspaandra, Lane Hemaspaandra, and Curtis Menton. Search versus decision for election manipulation problems. pages 377–388. Leibniz International Proceedings in Informatics (LIPIcs), 2013.

[55] Edith Hemaspaandra, Lane A. Hemaspaandra, and Jörg Rothe. Exact analysis of Dodgson elections: Lewis Carroll's 1876 voting system is complete for parallel access to NP. *Journal of the ACM*, 44(6):806–825, November 1997.

[56] Edith Hemaspaandra, Holger Spakowski, and Jörg Vogel. The complexity of Kemeny elections. *Theor. Comput. Sci.*, 349(3):382–391, 2005.

[57] Lane A. Hemaspaandra and David E. Narváez. The opacity of backbones. Technical Report arXiv:1606.03634 [cs.AI], Computing Research Repository, arXiv.org/corr/, June 2016. Revised, January 2019.

[58] Lane A. Hemaspaandra, Jörg Rothe, and Gerd Wechsung. Easy sets and hard certificate schemes. July 1997.

[59] Marijn Heule, Warren Hunt, Matt Kaufmann, and Nathan Wetzler. Efficient, verified checking of propositional proofs. In *International Conference on Interactive Theorem Proving*, pages 269–284. Springer, 2017.

[60] Marijn Heule and Oliver Kullmann. The science of brute force. *Communications of the ACM*, 60(8):70–79, 2017.

[61] Marijn J. H. Heule. Optimal symmetry breaking for graph problems. *Mathematics in Computer Science*, 13(4):533–548, 2019.

[62] Marijn J. H. Heule, Manuel Kauers, and Martina Seidl. Local search for fast matrix multiplication. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, pages 155–163, Cham, 2019. Springer International Publishing.

[63] Steven Homer and Alan L. Selman. *Computability and Complexity Theory*. Texts in Computer Science. Springer, New York, second edition, 2011.

[64] Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *SAT*, pages 428–437, 2018.

[65] Avraham Itzhakov and Michael Codish. Breaking symmetries in graph search with canonizing sets. *Constraints*, 21(3):357–374, 2016.

[66] Said Jabbour, Lakhdar Sais, and Yakoub Salhi. Boolean satisfiability for sequence mining. In *Proceedings of the 22nd ACM International Conference on Information & Knowledge Management*, CIKM '13, pages 649–658, New York, NY, USA, 2013. ACM.

[67] Mikoláš Janota and Joao Marques-Silva. *Abstraction-Based Algorithm for 2QBF*. Springer, 2011.

[68] Mikolás Janota and João Marques-Silva. An Achilles' heel of term-resolution. In Eugénio C. Oliveira, João Gama, Zita A. Vale, and Henrique Lopes Cardoso, editors, *EPIA*, volume 10423 of *Lecture Notes in Computer Science*, pages 670–680. Springer, 2017.

[69] Chula Jayawardene, David E. Narváez, and Stanisław P. Radziszowski. Star-critical Ramsey numbers for cycles versus K_4. *Discussiones Mathematicae Graph Theory*, (2629), 2018. In press.

[70] Charles Jordan, Will Klieber, and Martina Seidl. Non-CNF QBF solving with QCIR. In Adnan Darwiche, editor, *AAAI Workshop: Beyond NP*, volume WS-16-05 of *AAAI Workshops*. AAAI Press, 2016.

[71] Richard M. Karp. *Reducibility among Combinatorial Problems*. Springer, 1972.

[72] Philip Kilby, John Slaney, Sylvie Thiébaux, and Toby Walsh. Backbones and backdoors in satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, page 1368, 2005.

[73] Kathrin Konczak. Voting theory in answer set programming. In *20th Workshop on Logic Programming, Vienna, Austria, February 22–24, 2006*, pages 45–53, 2006.

[74] Oliver Kullmann. Green-tao numbers and SAT. In Ofer Strichman and Stefan Szeider, editors, *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 352–362. Springer, 2010.

[75] Lenoid Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1975. March 1975 translation into English of Russian article originally published in 1973.

[76] Tomasz Łuczak, Andrzej Ruciński, and Sebastian Urbański. On minimal Folkman graphs. *Discrete Mathematics*, 236(1-3):245–262, 2001.

[77] P. Madhusudan, Wonhong Nam, and Rajeev Alur. Symbolic computational techniques for solving games. *Electronic Notes in Theoretical Computer Science*, 89(4):578–592, 2003.

[78] Wiktor Marek, Arcot Rajasekar, and Mirosław Truszczyński. Complexity of computing with extended propositional logic programs. *Annals of Mathematics and Artificial Intelligence*, 15(3-4):357–378, 1995.

[79] Filip Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theoretical Computer Science*, 411(50):4333–4356, 2010.

[80] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *Journal of Symbolic Computation*, 60:94–112, 2014.

[81] Brendan D. McKay and Stanisław P. Radziszowski. $r(4,5) = 25$. *Journal of Graph Theory*, 19(3):309–322, 1995.

[82] Kenneth L. McMillan. Applying SAT methods in unbounded symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002.

[83] Michael Molloy and Ricardo Restrepo. Frozen variables in random Boolean constraint satisfaction problems. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1306–1318. SIAM, Philadelphia, PA, 2012.

[84] Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400(6740):133–137, 1999.

[85] César Muñoz. Rapid prototyping in PVS. Contractor Report NASA/CR-2003-212418, NASA, Langley Research Center, Hampton VA 23681-2199, USA, May 2003.

[86] Jaroslav Nešetřil and Vojtěch Rödl. Simple proof of the existence of restricted Ramsey graphs by means of a partite construction. *Combinatorica*, 1(2):199–202, 1981.

[87] N. Nishimura, P. Ragde, and S. Szeider. Detecting backdoor sets with respect to Horn and binary clauses. pages 96–103, May 2004.

[88] Yoonna Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. Amuse: a minimally-unsatisfiable subformula extractor. In *Proceedings. 41st Design Automation Conference, 2004.*, pages 518–523, July 2004.

[89] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[90] Sam Owre and Natarajan Shankar. Abstract datatypes in PVS. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993. Extensively revised June 1997; Also available as NASA Contractor Report CR-97-206264.

[91] Lawrence C Paulson. Organizing numerical theories using axiomatic type classes. *Journal of Automated Reasoning*, 33(1):29–49, 2004.

[92] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *International Conference on*

Mathematical Foundations of Programming Semantics, pages 209–228. Springer, 1989.

[93] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[94] Stanisław P. Radziszowski. Small Ramsey numbers. *Electronic Journal of Combinatorics, Dynamic Surveys*, DS1, 1994. Revision #15, 2017.

[95] F. P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, s2-30(1):264–286, 1930.

[96] Neil Robertson, Daniel Sanders, Paul Seymour, and Robin Thomas. A new proof of the four-colour theorem. *Electronic Research Announcements of the American Mathematical Society*, 2(1):17–25, 1996.

[97] Jörg Rothe. Complexity of certificates, heuristics, and counting types, with applications to cryptography and circuit theory. Habilitation thesis, Friedrich-Schiller-Universität Jena, Institut für Informatik, Jena, Germany, June 1999.

[98] Jörg Rothe, Holger Spakowski, and Jörg Vogel. Exact complexity of the winner problem for Young elections. *Theory Comput. Syst.*, 36(4):375–386, 2003.

[99] Guilhem Semerjian. On the freezing of variables in random constraint satisfaction problems. *Journal of Statistical Physics*, 130(2):251–293, 2008.

[100] Natarajan Shankar and Marc Vaucher. The mechanical verification of a DPLL-based satisfiability solver. *Electronic Notes in Theoretical Computer Science*, 269:3–17, 2011.

[101] Carsten Sinz. Towards an optimal CNF encoding of Boolean cardinality constraints. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005.

[102] A. Soifer. *Ramsey Theory: Yesterday, Today, and Tomorrow*, volume 285. Springer, 2011.

[103] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 1–9, New York, NY, USA, 1973. ACM.

[104] Stefan Szeider. Backdoor sets for DLL subsolvers. *Journal of Automated Reasoning*, 35(1-3):73–88, 2005.

[105] M. Thurley. sharpSAT – counting models with advanced component caching and implicit BCP. In Armin Biere and CarlaP. Gomes, editors, *9th International Conference on Theory and Applications of Satisfiability Testing*, volume 4121 of *Lecture Notes in Computer Science*, pages 424–429. Springer, 2006.

[106] Takahisa Toda and Takehide Soh. Implementing efficient all solutions SAT solvers. *J. Exp. Algorithmics*, 21:1.12:1–1.12:44, November 2016.

[107] G. S. Tseitin. *On the Complexity of Derivation in Propositional Calculus*, pages 466–483. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[108] Leslie Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[109] Leslie G. Valiant. Relative complexity of checking and evaluating. *Information Processing Letters*, 5(1):20–23, 1976.

[110] Ingo Wegener. *Complexity Theory*. Springer-Verlag, Berlin, 2005. Exploring the limits of efficient algorithms, Translated from the German by Randall Pruim.

[111] Nathan Wetzler, Marijn J. H. Heule, and Jr. Hunt, Warren A. *DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs*. Springer, 2014.

[112] Arthur T. White. *Graphs, Groups and Surfaces*. Amsterdam, North-Holland Pub. Co., 1973.

[113] Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In *IJCAI*, volume 3, pages 1173–1178, 2003.

[114] David H. Wolpert and William Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1:67–82, 1997.

[115] Xiaodong Xu, Meilian Liang, and Stanisław Radziszowski. On the nonexistence of some generalized Folkman numbers. *Graphs and Combinatorics*, 34(5):1101–1110, 2018.

[116] Avi Yadgar, Orna Grumberg, and Assaf Schuster. Hybrid BDD and All-SAT method for model checking. In *Languages: From Formal to Natural*, pages 228–244. Springer, 2009.

[117] Shuyuan Zhang, Abdulrahman Mahmoud, Sharad Malik, and Sanjai Narain. Verification and synthesis of firewalls using SAT and QBF. In *20th IEEE International Conference on Network Protocols (ICNP)*, pages 1–6. IEEE, 2012.

Appendices

Appendix A

A Comparison of Interactive Theorem Provers

The purpose of this appendix is to provide an admittedly subjective comparison among some popular interactive theorem provers from the perspective of using these specifically in proving theorems in discrete mathematics and combinatorics. Since there are several much better (written and informed) publications covering the formalities of each interactive theorem prover, we stress that this appendix reflects the personal view of the author. We believe this appendix will be useful for researchers having a first contact with theorem provers as it aims to provide a map of alternatives together with examples of the strengths and limitations of each.

To setup the stage, we start by listing the interactive theorem provers considered in this appendix. They are listed here because of our familiarity with them as they were used, some in more extent than others, in work related to this thesis.

These interactive theorem provers are, in alphabetical order:

- **Coq**: An interactive theorem prover implementing the calculus of inductive constructions by Coquand [23,92]. It is widely popular, in particular in the software verification and programming languages communities. It is mostly written in OCaml with some bits in C. The basic installation includes a standard library with common datatypes like lists and sets (though sets are called Ensembles, see Section A.3). The standard

library is augmented by a number of community-based contributions available in the Coq package index[1] and elsewhere.

- **Isabelle**: An interactive theorem prover with an underlying meta-logic able to implement a number of other logics. Arguably the most popular logic among the ones implemented in Isabelle is Isabelle/HOL which, as the name indicates, implements Higher-Order Logic. It is mostly implemented in Standard ML, with certain pieces written in Java and Scala. The standard installation includes a number of theories from several fields in mathematics and computer science, plus an integrated development environment named Isabelle/jEdit. Additionaly, the Archive of Formal Proofs[2] collects many other peer-reviewed formalizations and is under active development.

- **PVS**: The Prototype Verification System [89] is a verification system with a language to write specifications and an interactive theorem prover. It is based on typed set theory. It is mostly written in LISP and the integrated development environment included in the standard installation is implemented as an Emacs mode. The basic set of theories included in PVS, named the prelude, includes a fairly useful portions of set theory and arithmetic, plus many data structures that are common in computer science. The Langley Formal Methods group at NASA collects additional theories in a separate repository [3] that is often integrated with the standard PVS installation. NASA's PVSLib includes a number of specifications ranging from pure mathematics to aeronautical applications.

The main issue to consider when picking between these interactive theorem provers is the underlying logic. The calculus of inductive constructions is fundamentally based on constructive logic which, unlike classic logic, does not include the law of excluded middle nor double-negation elimination. The reason these "features" are missing from constructive logic is because the focus of constructive logic is to link mathematics and proofs to computer programs and execution. Thus, Coq and other systems based on the calculus of inductive

[1] https://coq.inria.fr/opam/www/
[2] https://www.isa-afp.org/
[3] https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html

constructions lend themselves to code extraction which is a feature that allows for verified code to be extracted directly from the proof.

Another significant difference among the theorem provers listed above is that in PVS the proofs are not found alongside the theorems and lemmas. In Coq and Isabelle, one specifies a theorem or lemma and proceed to input the proof, which can be checked interactively. In PVS, after stating a theorem, one needs to input a proof in a separate, interactive Emacs buffer. Each step in this process is turned into LISP statements that are saved in a separate file. This is an issue when one wishes to perform a slight modification of a proof, since one basically needs to replay the interaction.

A.1 Dependent Types

Theorem provers have inherited many concepts from functional languages, so it is often a natural transition from writing programs to writing specifications in computer-aided theorem proving. Perhaps the most important concept native to both programming and theorem proving is that of types (see, for instance, the development of typed lambda calculus). Nevertheless, one concept that may not be familiar to programmers with basic knowledge of common programming languages is that of dependent types: types that depend on (runtime) values. Dependent types, although common, are not supported in every interactive theorem prover. Furthermore, in those that support dependent types, the level of support (from the point of view of usability) varies. We will argue in this section that dependent types are fundamental for theorem proving in discrete mathematics and theoretical computer science.

For a concrete example, consider arithmetic in $\mathbb{Z}_p$ for a prime number. It is reasonable to think of the following definition of addition in $\mathbb{Z}_p$ as a function in C:

```
unsigned int add(unsigned int p, unsigned int a, unsigned int b);
```

The signature above encodes some basic properties of the numbers involved: marking the parameters `p`, `a` and `b` as `unsigned` specifies that $p, a, b \geq 0$. This is, however, not enough since we would also like to specify that $a, b < p$. Furthermore, the number returned from this function is also in $\mathbb{Z}_p$, thus should

be specified similarly. Finally, p is not any unsigned integer, but a prime number to be precise. Compare the above function definition with the following definition in PVS:

```
add(p: (prime?), a: mod(p), b: mod(p)): mod(p) = res(p)(a + b)
```

which makes use of the `prime?` predicate; the `mod` data type that, for an integer p, restricts integers to the range $0 \leq i < p$; and the function `res` that calculates the residue of a division by p.

For readers unfamiliar with type theory, it may not be obvious that one of the main reasons these features are not standard in mainstream programming languages is that type-checking programs with arbitrary dependent types is undecidable. Thus using specifications with dependent types will necessarily involve additional checks, some of which will need user input. In the case of PVS, these are usually handled through *type-checking constraints (TCCs)* which ask the user to prove some property about the use of specifications with dependent types. For instance, the alternative definition of the `add` function defined as:

```
add(p: (prime?), a: mod(p), b: mod(p)): mod(p) = a + b
```

would require us to effectively prove that for any $0 \leq a, b < p$, $a + b < p$ (in order to comply with the type of the function which is `mod(p)`), which is obviously not true.

Coq advertises syntactically similar support for dependent types. Nevertheless, it differs from PVS in a fundamental fashion: that the proofs required to guarantee the proper use of the specifications are part of the notation itself. We start with this valid example:

```
Definition add (p: { x: Z | prime x }) (a b: Z): Z := a + b.
```

which guarantees that, in order to use the `add` function so defined, p must be a prime number. It does not, however, specify any relationship between a and p, b and p or the result $a + b$ and p. This example of the notation for *subset types* (types defined using the set notation) is thus valid but useless. In order for it to be of any use, we need to connect the parameters and the result type with p, as in the PVS example. The following alternative, however, is invalid:

```
Definition add (p: { x: Z | prime x })
(a b: { x: Z | 0 <= x /\ x < p }): Z := a + b.
```

with error

```
The term "p" has type "{x : Z | prime x}" while it is expected to have
type "Z".
```

This happens because the notation `{x : Z | prime x}` means not only an integer that has that property, but also a proof of that property. The parameter `p` is thus a tuple, one of whose members is an integer with the desired property. One then needs other features of the language to extract the integer of interest:

```
Definition add (p: { x: Z | prime x })
(a b: { x: Z | 0 <= x /\ x < (match p with | exist _ p' _ => p' end) })
:= (0 : Z).
```

We have intentionally fixed the result to be 0 for the sake of simplicity in order to focus on the `match` statement that extract the integer `y` out of the constructor of a subset type expression. To restore the functionality of the `add` function, one needs to add a and b, but the parameters `a` and `b` are themselves subset types so one needs additional `match` statements to extract each integer. In order to really match the preciseness of the PVS specification presented before, one also needs to modify the return type of the function. We believe that providing a fully developed example here would be of little use. Yet, it should be clear from this analysis that subtyping à la PVS is not natural in Coq. Arguably the natural way to deal with these kinds of specifications in Coq is to require the proofs as arguments to the function (which are ultimately ignored):

```
Definition add (p a b: Z) : prime p -> 0 <= a < p -> 0 <= b < p -> Z :=
 fun _ => (fun _ => (fun _ => a + b)).
```

Finally, Isabelle/HOL does not support dependent typing, so we will leave it out of this conversation.

A.2 Inheritance

In this section we will consider inheritance as found in object-oriented programming languages like Java. We will argue that comprehensive support for inheritance would be necessary to develop natural formalizations in discrete mathematics. One of the most natural examples of inheritance in mathematics is algebraic structures: rings are groups with multiplication, fields are rings with division and subtraction, etc. These were, in fact, the original motivation for Isabelle/HOL's *type classes* [91]. It is then natural to expect that theorems about groups will apply transparently to fields, and those for fields will apply transparently to rings.

The way Isabelle/HOL deals with this kind of inheritance is through the concept of *locales* which are groups of definitions and theorems which are parametrized by some fixed values and some axioms over these fixed values. A simple use of locales, based on a similar example from Isabelle's locale documentation, is:

```
locale poset =
  fixes S :: "'a set"
  and R :: "('a × 'a) set"
assumes R_reflexive : "∀x∈S.(x, x) ∈ R"
and R_antisymmetric:
  "∀ x ∈ S.∀ y ∈ S. (x, y) ∈ R ⟶ (y, x) ∈ R ⟶ x = y"
and R_transitive:
  "∀ x ∈ S. ∀ y ∈ S. ∀ z ∈ S.
    (x, y) ∈ R ⟶ (y, z) ∈ R ⟶ (x, z)∈ R"
begin

(* Theorems about posets... *)

end

locale semilattice = poset +
  assumes inf_assumption :
"∀ x ∈ S. ∀ y ∈ S. ∃ z.
  ((z, x) ∈ R ∧ (z, y) ∈ R)
    ∧ (∀ w∈ S. (w, x)∈ R ⟶ (w, y)∈ R ⟶ (w, z)∈ R)"
begin

(* Theorems about semilattices... *)
```

```
end
```

This example illustrates some of the notation for locales, and sets. (Notice we have chosen to implement the relation R as a set and not as a function.) In this example, partially ordered sets are defined in terms of a set S and a relation R. Semilattices are then defined as partially ordered sets with additional restrictions on R. We can prove in each context whatever theorem we wish about partially ordered sets and semilattices. What we cannot do with the above notation, despite its flexibility, is instantiate an object of type `semilattice`. That is, `semilattice` is a context that inherits theorems from a parent context `poset`, not a type from which objects can be instantiated. This type of inheritance is thus different from the one we are after. An example of the kind of inheritance we would like is to be able to modify the above definitions to talk about finite posets:

```
 locale finite_poset =
  fixes S :: "'a fset"
  and R :: "('a × 'a) fset"
assumes R_reflexive : "∀x∈S.(x, x) ∈ R"
begin

(* Theorems about finite posets... *)

end
```

Unfortunately the above definition is not valid, and fails with the following error:

```
Type unification failed: Clash of types "_ fset" and "_ set"

Type error in application: incompatible operand type

Operator:  Ball :: ??'a set ⇒ (??'a ⇒ bool) ⇒ bool
Operand:   S :: 'a fset
```

The reason is that the notation $\forall$`x`$\in$`S` is reserved for sets. Although finite sets (the type `fset`) in Isabelle are sets that are finite, the fact that they are not sets means functions and notation no longer apply to them.

A.3 The Issue of Unordered Sets

Unordered sets are fundamental to discrete mathematics and computer science, though in this section we will elaborate on how they are unnatural for computations. Because of this, and depending on the logic implemented by an interactive theorem prover, it may be difficult to write statements regarding unordered sets and their manipulations.

We start from discussing the treatment of unordered sets in typed set theory, where they are most natural. In PVS, which as we mentioned before implements typed set theory, sets of elements of type `T` are just functions `T->bool` from elements of type `T` to the (built-in) Boolean type. Such functions are also referred to as predicates over the type `T`. Thus, an element is in a set if and only if the function evaluates to true. As such, predicates and sets can be used interchangeably. For instance, these expressions are equivalent and define the set of even natural numbers:

```
{ x: nat | rem(2)(x) = 0 }
LAMBDA (x: nat): (rem(2)(x) = 0)
```

and the following statements are true:

```
(LAMBDA (x: nat): (rem(2)(x) = 0))(2)
NOT ({x: nat | rem(2)(x) = 0 })(3)
```

Furthermore, predicates and sets can be used to define types which allows for elegantly defining dependent types as discussed in Section A.1.

In the calculus of inductive constructions, which as we mentioned before is based on constructive logic, things are different. In particular, `Set` in Coq is a keyword not related to set theory per se. Unordered sets in Coq are called `Ensembles`, and are defined in terms of the `In` function which takes an Ensemble and an element and outputs a proposition. For instance, the set of even numbers would be:

```
Inductive Even_Numbers : Ensemble nat :=
 Even_Numbers_const: forall (x: nat), In nat Even_Numbers (2 * x).
```

What the above definition provides is a constructor that allows us to prove a natural number is in the set `Even_Numbers`.

In Isabelle, defining a datatype of even numbers is possible using predicate subtyping, though it has the tricky condition that the set cannot be empty. Thus one needs to prove there is at least one even number:

```
typedef even_numbers = "{ n :: int | even n }" by (auto)
```

Lets now consider the issue of defining k-sets, that is, sets of a certain type which have k elements. These are commonly found in computer science, for instance, in the Exact-Cover-By-3-Sets (X3C) problem. k-sets of, say, natural numbers are easy to define and use in PVS as type dependent on a natural number k:

```
  kset(k: nat) : TYPE = { s: finite_set[nat] | card(s) = k }

  firstk (k: nat) : RECURSIVE[finite_set[nat]] =
  IF k > 0 THEN add(k , firstk(k - 1)) ELSE emptyset[nat] ENDIF
  MEASURE k

  some_fun (k: nat, s: kset(k)) : bool = TRUE

  example : bool = some_fun(3, firstk(3))
```

Using the function `firstk` as an argument for `some_fun` generates the following TCC:

```
 % Subtype TCC generated (at line 28, column 31) for  firstk(3)
    % expected type  kset(3)
  % unfinished
example_TCC1: OBLIGATION card[nat](firstk(3)) = 3;
```

which can be proven with the following additional lemmas:

```
  firstk_not_in : LEMMA
  FORALL (k: nat, m: nat):k <= m IMPLIES NOT member(m, firstk(k))

  firstk_card : LEMMA
  FORALL (k: nat): card(firstk(k)) = k
```

In Coq, it is not easy to define k-sets as an inductive type but it is possible to use dependent typing to require a parameter to be a k-set. Using a function

defined that way is more complicated than in PVS since one needs to prove upfront that the cardinality of the set passed as an argument is in fact the expected one, and this proof needs to be built into the argument.

```
Definition some_fun (k: nat)
           (s: { x: Ensemble nat | cardinal nat x k }) : Prop := True.

Fixpoint firstk (k: nat) : Ensemble nat :=
  match k with
  | 0 => Empty_set nat
  | S n => Add nat (firstk n) n
  end.

Lemma firstk_notIn: forall k k', k <= k' -> ~ In nat (firstk k) k'.
Proof.
  ...
Qed.

Lemma firstk_cardinal : forall k, cardinal nat (firstk k) k.
Proof.
  ...
Qed.

Eval compute in some_fun 3 (exist _ (firstk 3) (firstk_cardinal 3)).
```

A.4 Code Extraction

The previous sections seem to heavily favor the use of PVS (or typed set theory in general) for the type of problems combinatorics and discrete math are interested in. Nevertheless, the use of interactive theorem provers in the context of this thesis is ultimately for the purpose of verifying computations. One way to achieve this is by generating verified code that computes solutions to problems, as explained in Chapter 3. In this respect, interactive theorem provers like Coq and Isablle excel because they have great support for code extraction. In the case of Coq, it is particularly useful that the underlying theory (constructive logic) is designed, to a great extent, to support computations (hence the lack of the axiom of choice and the law of excluded middle).

We do not mean to make this section a reference for code extraction, and

we refer the reader to the documentation of each of the interactive theorem provers for an in-depth explanation of how the code extraction works. Instead, we focus on providing a realistic (though subjective) perspective about the issues related to extracting and running verified code from the practical point of view. The problem here is two-fold in the sense that proving the correctness of algorithms is very time-consuming and the obtained code is usually not comparable in performance to unverified code. This is, for instance, the case of the verified SAT solver written in Isabelle [12] versus the SAT solvers that participate in the yearly SAT competitions; or the verified verifiers for unsatisfiability in the DRAT [59] and LRAT [26] formats versus their unverified counterparts. Furthermore, it is usually the case that once these tools are developed, they need to be maintained and updated. In the case of verified software, this does not only involve modifying the code but also adapting the previous proofs to the new code, which is usually an expensive endeavor.

In summary, we believe that while verified code is certainly desirable, it is usually very hard to obtain. It is also unrewarding in scenarios where a software product or a publication needs to be completed within a certain timeline. In this respect, advances in tooling that automates the proofs and the maintenance of verified code will be the most helpful in order to make this practice more popular.

A.5 Conclusions

We have provided our view on several aspects of interactive theorem provers related to combinatorial computing. We have done this in the hope that people interested in this particular topic can take an informed decision when picking a certain interactive theorem prover for their research. The topics covered in this appendix are of course not exhaustive, and some of the limitations discussed in the various sections may eventually be overcome by the tools.