Algorithmic Applications of Propositional Proof Complexity

Ashish Sabharwal

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2005

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Ashish Sabharwal

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of the Supervisory Committee:

_____
Paul W. Beame

_____
Henry Kautz

Reading Committee:

_____
Paul W. Beame

_____
Henry Kautz

_____
Venkatesan Guruswami

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, 1-800-521-0600, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature_____

Date_____

University of Washington

Abstract

Algorithmic Applications of Propositional Proof Complexity

Ashish Sabharwal

Co-Chairs of the Supervisory Committee:
Professor Paul W. Beame
Computer Science and Engineering
Professor Henry Kautz
Computer Science and Engineering

This thesis explores algorithmic applications of proof complexity theory to the areas of exact and approximation algorithms for graph problems as well as propositional reasoning systems studied commonly by the artificial intelligence and formal verification communities. On the theoretical side, our focus is on the propositional proof system called resolution. On the practical side, we concentrate on propositional satisfiability algorithms (SAT solvers) which form the core of numerous real-world automated reasoning systems.

There are three major contributions in this work. (A) We study the behavior of resolution on appropriate encodings of three graphs problems, namely, independent set, vertex cover, and clique. We prove lower bounds on the sizes of resolution proofs for these problems and derive from this unconditional hardness of approximation results for resolution-based algorithms. (B) We explore two key techniques used in SAT solvers called clause learning and restarts, providing the first formal framework for their analysis. Formulating them as proof systems, we put them in perspective with respect to resolution and its refinements. (C) We present new techniques for designing structure-aware SAT solvers based on high-level problem descriptions. We present empirical studies which demonstrate that one can achieve enormous speed-up in practice by incorporating variable orders as well as symmetry information obtained directly from the underlying problem domain.

TABLE OF CONTENTS

LIST OF ALGORITHMS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express sincere appreciation for my primary advisor, Paul Beame, who introduced me to proof complexity theory, nurtured my ideas as I struggled to produce my first research results, allowed me the flexibility to work on what I liked, and taught me to appreciate the invaluable virtues of clarity, perfection, and depth in research and technical writing. He will always be an inspiration to me.

Many thanks also to my co-advisor, Henry Kautz, for introducing me to the world of automated reasoning, for helping me achieve a smooth transition from theory to practice, for his constant encouragement, and for providing extremely useful advice and help on job search as I came towards the end of my years as a graduate student.

I would like to thank Venkat Guruswami who graciously agreed to be on my reading committee at the last moment and whose suggestions and criticism of the writeup made it more precise and more accessible to the reader. Thanks also to Richard Ladner, Dan Weld, and Elizabeth Thompson for being on my supervisory committee.

Everyone in the Department of Computer Science made my stay in Seattle very comfortable and enjoyable as I worked on this thesis. The National Science Foundation indirectly provided the necessary financial stability.

Last but not the least, the completion of this thesis would not have been possible without the hidden support of my family and friends. Thank you all for being there with me in this unforgettable journey.

Chapter 1

INTRODUCTION

Propositional proof complexity is the study of the structure of proofs of mathematical statements expressed in a propositional or Boolean form. This thesis explores the algorithmic applications of this field, focusing on exact and approximation algorithms for combinatorial graph problems as well as on propositional reasoning systems used frequently in artificial intelligence and formal verification.

Science relies heavily on modeling systems and providing proofs of properties of interest. This motivates the study of the nature of proofs themselves and the use of computers to automatically generate them when possible. What do mathematical proofs look like? How are problems of interest represented in formats suitable for proving properties about them? What are the computational challenges involved in finding such proofs? Do short proofs even always exist when one's reasoning abilities are limited? How can our understanding of proof structures be used to improve combinatorial search algorithms? This work is a step towards answering these very natural and influential questions.

We concentrate on propositional statements. These are logical statements over a set of variables each of which can be either TRUE or FALSE. Suppose a propositional statement S is a tautology, i.e., it is TRUE for all possible combinations of values of the underlying variables. S is alternatively referred to as being valid. How can one provide a proof of the validity of S? Computationally, what does it mean to have such a proof?

Cook and Reckhow [39] introduced the formal notion of a proof system in order to study mathematical proofs from a computational perspective. They defined a propositional proof system to be an efficient algorithm A that takes as input a propositional statement S and a purported proof π of its validity in a certain pre-specified format. The crucial property of A is that for all invalid statements S, it rejects the pair (S, π) for all π, and for all valid statements S, it accepts the pair (S, π) for some proof π. This notion of proof systems can be alternatively formulated in terms of unsatisfiable formulas — those that are FALSE for all assignments to the variables.

They further observed that if there is no propositional proof system that admits short (polynomial in size) proofs of validity of all tautologies, i.e., if there exist computationally hard tautologies for every propositional proof system, then the complexity classes NP and co-NP are different, and hence P $\neq$ NP. This observation makes finding

tautological formulas (equivalently, unsatisfiable formulas) that are computationally difficult for various proof systems one of the central tasks of proof complexity research, with far reaching consequences to complexity theory and Computer Science in general. These hard formulas naturally yield a hierarchy of proof systems based on the sizes of proofs they admit. Tremendous amount of research has gone into understanding this hierarchical structure. A slightly outdated but interesting survey by Beame and Pitassi [20] nicely summarizes many of the results obtained along these lines in the last two decades.

As the most theoretical part of this work, we explore the proof complexity of a large class of structured formulas based on certain NP-complete combinatorial search problems on graphs. We prove that these formulas are hard for a very commonly studied proof system known as resolution. Algorithmically, this implies lower bounds on the running time of a class of algorithms for solving these problems exactly as well as approximately. These results complement the known approximation hardness results for these problems which build on the relatively sophisticated and recent machinery of probabilistically checkable proofs (PCPs) [9, 8].

On the more applied side which relates to automated reasoning systems, our focus is on propositional satisfiability algorithms, or *SAT solvers* as they are commonly known. Given a propositional formula F as input, the task of a SAT solver is to either find a variable assignment that satisfies F or declare F to be unsatisfiable. SAT solvers have evolved tremendously in the last decade, becoming general-purpose professional tools in areas as diverse as hardware verification [24, 112], automatic test pattern generation [74, 104], planning [70], scheduling [59], and group theory [114]. Annual SAT competitions have led to dozens of clever implementations [e.g. 13, 84, 113, 88, 57, 64], exploration of many new techniques [e.g. 80, 58, 84, 88], and creation of extensive benchmarks [63].

Researchers involved in the development and implementation of such solvers tackle the same underlying problem as those who work on propositional proof complexity — the propositional satisfiability problem. They have typically focused on specific techniques that are efficient in real world domains and have arguably interacted with the proof complexity community in a somewhat superficial way. One must grant that relatively straightforward correlations such as the equivalence between the basic backtracking SAT procedure called DPLL and a simple proof system called tree-like resolution as well as the relationship between more advanced SAT solvers using inequalities with a proof system called cutting planes have been regarded as common knowledge. However, a more in-depth connection between the ideas developed by the two communities is lacking. They have traditionally used very different approaches, rarely letting the concepts developed by one influence the techniques or focus of the other in any significant way.

The aim of the latter half of this work is to advance both of these fields – satisfiability algorithms and proof complexity – through independent work as well as

cross-fertilization of ideas. It explores the inherent strength of various propositional reasoning systems and uses theoretical insights to comprehend and improve the most widely implemented class of complete SAT solvers. It tries to achieve a balance between providing rigorous formal analysis and building systems to evaluate concepts. The results are a blend of theoretical complexity bounds, some of which explain observed behavior, and systems such as SymChaff built to demonstrate performance gains on real world problems.

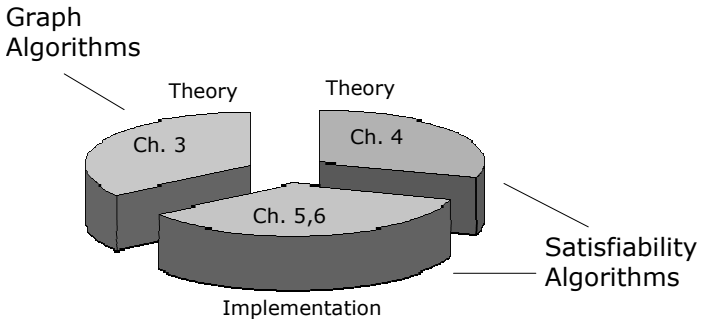

Figure 1.1: The three applications of proof complexity explored in this work

This thesis contains a broad spectrum of work from answering purely theoretical questions on one end to building systems that change the way we solve problems of human interest on the other (see Figure 1.1). A constant flow of ideas between the two extremes has far reaching benefits, as does research that addresses the middle ground. Realizing inherent strengths and limitations directs one's focus on concepts critical to good implementations and is the foundation of better systems. The work we will present in Chapters 5 and 6 is a testimony to this. On the other hand, systems whose development is motivated by real world applications provide a new avenue for utilizing the analysis techniques developed theoretically. The work in Chapter 4 stands in support of this.

The quest for short proofs continues amongst researchers for several reasons. One is pure mathematical curiosity in search for simplicity and elegance. Another is the far reaching effect on complexity theory that existence of short proofs might have. Yet another is computational efficiency demanded by the numerous real world applications that have come to depend on SAT solvers. The purpose of this thesis is to bring these motivations together in order to attain a better theoretical as well as practical understanding of the algorithmic applications of propositional proof complexity.

1.1 Theoretical Contributions

From a proof complexity perspective, the focus of this work is on the proof system called resolution. It is a very simple system with only one rule which applies to dis-

junctions of propositional variables and their negations: (a OR b) and ((NOT a) OR c) together imply (b OR c). Repeated application of this rule suffices to derive an empty disjunction if and only if the initial formula is unsatisfiable; such a derivation serves as a proof of unsatisfiability of the formula.

1.1.1 The Resolution Complexity of Structured Problems

In Chapter 3 we combine combinatorial and probabilistic techniques to show that propositional formulations of the membership in co-NP of almost all instances of some interesting co-NP complete graph problems, namely complements of Independent Set, Clique, and Vertex Cover, unconditionally require exponential size resolution proofs.

Resolution, although powerful enough to encompass most of the complete SAT solvers known today, does not admit short proofs of even simple counting-based formulas, most notably those encoding the pigeonhole principle: there is no one-one mapping from n pigeons to $(n-1)$ holes. Progress in the last decade has shown that almost all randomly chosen formulas are also difficult for resolution. However these random formulas are completely unstructured, unlike most real world problems. Are there large classes of hard but structured formulas that may be more representative of the instances encountered in practice? We give an affirmative answer to this.

More formally, we consider the problem of providing a resolution proof of the statement that a given graph with n vertices and average degree Δ does not contain an independent set of size k. For graphs chosen randomly from the distribution $\mathbb{G}(n,p)$, where $\Delta = np$, we show that such proofs asymptotically almost surely require size roughly exponential in n/Δ^6 for $k \leq n/3$. This, in particular, implies a $2^{\Omega(n)}$ lower bound for constant degree graphs. We deduce similar complexity results for the related vertex cover problem on random graphs.

This work was done jointly with Paul Beame and Russell Impagliazzo. It has been published in the proceedings of the 16^{th} Annual Conference on Computational Complexity (CCC), 2001 [16] and is currently under review for the journal Computational Complexity.

1.1.2 Hardness of Approximation

The complexity results described above translate into exponential lower bounds on the running time of a class of search algorithms for finding a maximum independent set or a minimum vertex cover. In fact, all resolution proofs studied in the above work turn out to be of exponential size even when one attempts to prove the non-existence of a much larger independent set or clique than the largest one, or a much smaller vertex cover than the smallest one. This in turn implies that a natural class of approximate optimization algorithms for these problems performs poorly on almost all problem instances.

In particular, we show that there is no resolution-based algorithm for approximating the maximum independent set size within a factor of $\Delta/(6 \log \Delta)$ or the minimum vertex cover size within a factor of 3/2. This latter result contrasts well with the commonly used factor of 2 approximation algorithm for the vertex cover problem.

We also give relatively simple algorithmic upper bounds for these problems and show them to be tight for the class of exhaustive backtracking techniques.

This work was done jointly with Paul Beame and Russell Impagliazzo and is currently under review for the journal Computational Complexity.

1.2 Proof Systems Underlying SAT Solvers

Our next contribution is in providing a formal understanding of the numerous satisfiability algorithms developed in the last decade. It is common knowledge that most of today's complete SAT solvers implement a subset of the resolution proof system. However, where exactly do they fit in the proof system hierarchy? How do they compare to refinements of resolution such as regular resolution? Why do certain techniques result in huge performance gains in practice while others have limited benefits? This work provides the first answers to some of these questions.

1.2.1 Clause Learning, Restarts, and Resolution

Chapter 4 develops an intuitive but formal understanding of the behavior of SAT solvers in practice. Using a new framework for rigorous analysis of techniques frequently used in solver implementations, we show that the use of a critical technique called clause learning makes a solver more powerful than many refinements of resolution, and with yet another technique – arbitrary restarts – and a slight modification, makes it as strong as resolution in its full generality.

Conflict-driven clause learning works by caching and reusing reasons of failure on subproblems. Random restarts help a solver avoid unnecessarily exploring a potentially huge but uninteresting search space as a result of a bad decision. These are two of the most important ideas that have lifted the scope of modern SAT solvers from experimental toy problems to large instances taken from real world challenges. Despite overwhelming empirical evidence, not much was known of the ultimate strengths and weaknesses of the two. We provide a formal explanation of the observed exponential speedups seen when using these techniques.

This presents the first precise characterization of clause learning as a proof system and begins the task of understanding its power by relating it to resolution. In particular, we show that with a new learning scheme called FirstNewCut, clause learning can provide exponentially shorter proofs than any proper refinement of general resolution satisfying a natural self-reduction property. These include regular and ordered resolution, which are already known to be much stronger than the ordinary DPLL procedure which captures most of the SAT solvers that do not incorporate clause

learning. We also show that a slight variant of clause learning with unlimited restarts is as powerful as general resolution itself.

This work was done jointly with Paul Beame and Henry Kautz. It has been published in the proceedings of the 18^{th} International Joint Conference on Artificial Intelligence (IJCAI), 2003 [18] as well as in the Journal of Artificial Intelligence Research (JAIR), 2004 [19].

1.3 Building Faster SAT Solvers

The input to almost all SAT solvers of today is a formula in conjunctive normal form (CNF), i.e., a conjunction of disjunctions of variables or their negations. This shallow representation results in several algorithmic and implementation-related advantages and can, in most cases, be made fairly compact with the use of additional auxiliary variables.

On the other hand, one can easily argue that most real world problem instances given as input to a SAT solver in the CNF form originate from a more structured, high level problem description known to the problem designer. For example, the underlying high level structured object could be a circuit, a planning graph, or a finite state model for which one wants to prove a desired property.

In Chapters 5 and 6 we show how we can gain substantially by providing the solver extra information that relates variables to the high level object they originated from. This information can, for instance, be in the form of an ordering of variables based on the original structure (e.g. a depth-first traversal of a planning graph) or their semantics (e.g. variable x represents loading truck k with block q). Of course, a well-translated formula itself contains all this information, but in a hidden way. We argue that making, as is commonly done, a solver rediscover the structure that was clear to start with is unnecessary, if not totally wasteful.

1.3.1 Variable Ordering Using Domain Knowledge

Motivated by the theoretical work in Chapter 4, we propose in Chapter 5 a novel way of exploiting the underlying problem structure to guide clause learning algorithms toward faster solutions. The key idea is to generate a branching sequence for a CNF formula encoding a problem from the high level description of the problem such as a graph or a planning language specification. We show that this leads to exponential empirical speed-ups on the class of grid and randomized pebbling problems, as well as substantial improvements on certain ordering formulas.

Conflict-directed clause learning is known to dramatically increase the effectiveness of branch-and-bound SAT solvers. However, to realize its full power, a clause learner needs to choose a favorable order of variables to branch on. While several proposed formula-based static and dynamic ordering schemes help, they face the daunting task of recovering relevant structural information from the flat CNF formula representation

consisting only of disjunctive clauses. We show that domain knowledge can allow one to obtain this critical ordering information with much more ease. This is a case in point for using domain knowledge to boost performance of a SAT solver rather than using the solver as a pure blackbox tool in the traditional manner.

This work was done jointly with Paul Beame and Henry Kautz. It has been published in the proceedings of the 6^{th} International Conference on Theory and Applications of Satisfiability Testing (SAT), 2003 [99] as well as in the Journal of Artificial Intelligence Research (JAIR), 2004 [19].

1.3.2 Utilizing Structural Symmetry in Domains

In Chapter 6 we present a novel low-overhead framework for encoding and utilizing structural symmetry in SAT solvers. We use the notion of complete multi-class symmetry and demonstrate the efficacy of our technique through a new solver called SymChaff which achieves exponential speedup by using simple tags in the specification of problems from both theory and practice.

A natural feature of many application domains in which SAT solvers are used is the presence of symmetry or equivalence with respect to the underlying objects, such as that amongst all trucks at a certain location in logistics planning and all wires connecting two switch boxes in an FPGA circuit. Many of these problems turn out to have a concise description in what is called many-sorted first order logic. This description can be easily specified by the problem designer and almost as easily inferred automatically. SymChaff, an extension of the popular SAT solver zChaff, uses information obtained from the "sorts" in the first order logic constraints to create symmetry sets that are used to partition variables into classes and to maintain and utilize symmetry information dynamically.

The challenge in such work is to do it in a way that pushes the underlying proof system up in the hierarchy without incurring the significant cost that typically comes from large search spaces associated with complex proof systems. While most of the current SAT solvers implement subsets of resolution, SymChaff brings it up closer to symmetric resolution, a proof system known to be exponentially stronger than resolution [111, 76]. More critically, it achieves this in a time- and space-efficient manner.

This work has been published in the proceedings of the 20^{th} National Conference on Artificial Intelligence (AAAI), 2005 [98].

Chapter 2

PRELIMINARIES

Throughout this thesis, we work with propositional or Boolean variables, that is, variables that take value in the set {TRUE, FALSE}. A propositional formula F representing a Boolean function is formed by combining these variables using various Boolean operators. We use the two binary operators conjunction (AND, $\wedge$) and disjunction (OR, $\vee$), and the unary operator negation (NOT, $\neg$). These three operators are sufficient to express all Boolean functions and, at the same time, provide enough expressiveness to encode many interesting functions in a fairly compact way.

Definition 2.1. A propositional formula F is *satisfiable* if there exists an assignment ρ to its variables such that F evaluates to TRUE under ρ. If no such ρ exists, F is *unsatisfiable*. F is a *tautology* if $\neg F$ is unsatisfiable.

We often use the abbreviations SAT and UNSAT for satisfiable and unsatisfiable, respectively. A variable assignment ρ under which F evaluates to TRUE is referred to as a *satisfying assignment* for F.

Definition 2.2. A propositional formula F is in *conjunctive normal form* (CNF) if it is a conjunction of *clauses*, where each clause is a disjunction of *literals* and each literal is either a variable or its negation. The *size* of F is the number of clauses in F.

It is natural to think of F as a set of clauses and each clause as a set of literals. We use the symbol Λ to denote the *empty clause* which is always unsatisfiable. A clause with only one literal is referred to as a *unit* clause. A clause that is a subset of another is called its *subclause*. Let ρ be a partial assignment to the variables of F.

Definition 2.3. The *restricted formula* F^ρ is obtained from F by replacing variables in ρ with their assigned values. F is said to be *simplified* if all clauses with at least one TRUE literal are deleted and all occurrences of FALSE literals are removed from clauses. $F|_\rho$ denotes the result of simplifying the restricted formula F^ρ.

The construction of Tseitin [109] can be used to efficiently convert any given propositional formula to one in CNF form by adding new variables corresponding to its subformulas. For instance, given an arbitrary propositional formula G, one would first locally re-write each of its operators in terms of $\wedge, \vee$, and $\neg$ to obtain, say, $G = (((a \wedge b) \vee (\neg a \wedge \neg b)) \wedge \neg c) \vee d$. To convert this to CNF, one would add four auxiliary variables w, x, y, and z, construct clauses that encode the four relations

$w \leftrightarrow (a \wedge b)$, $x \leftrightarrow (\neg a \wedge \neg b)$, $y \leftrightarrow (w \vee x)$, and $z \leftrightarrow (y \wedge \neg c)$, and add to that the clause $(z \vee d)$. Given this efficient conversion mechanism, we restrict ourselves to CNF formulas.

2.1 The Propositional Satisfiability Problem

The combinatorial problem that lies at the heart of this work is the satisfiability problem that asks whether a given propositional formula has a satisfying assignment. More precisely,

Definition 2.4. The *propositional satisfiability problem* is the following: Given a CNF formula F as input, determine whether F is satisfiable or not. If it is satisfiable, output a satisfying assignment for it.

The decision version of this problem, where one is only asked to report SAT or UNSAT, is also referred to as CNF-SAT in the literature. In their well-known work, Cook [38] and Levin [79] proved the problem to be NP-complete, setting the foundation for a vast amount of research in complexity theory.

In this thesis, we will look at this problem from various perspectives. When a formula F is unsatisfiable, we will be interested in analyzing the size of the shortest *proof* of this fact. The formal machinery using which such proofs are presented and verified is discussed in the following section. From the practical perspective, we will also be interested in designing algorithms to find such proofs efficiently. When F is satisfiable, the task will be to design algorithms that efficiently find a satisfying assignment for it. Although some applications may require one to output several satisfying assignments, we will focus on finding one.

An algorithm that solves the propositional satisfiability problem is called a *satisfiability algorithm*. Practical implementations of such algorithms typically involve smart data structures and carefully chosen parameters in addition to an efficient top-level algorithm. These implementations are referred to as *SAT solvers*. Note the use of SAT here as referring to the propositional satisfiability problem in contrast to being an abbreviation of satisfiable. Henceforth, we leave it up to the context to make the meaning of "SAT" unambiguous.

2.2 Proof Systems

The notion of a propositional proof system was first defined in the seminal work of Cook and Reckhow [39]. It is an efficient (in the size of the proof) procedure to check the correctness of proofs presented in a certain format. More formally,

Definition 2.5. A *propositional proof system* is a polynomial time computable predicate S such that a propositional formula F is unsatisfiable iff there exists a *proof* (or *refutation*) π for which $S(F, \pi)$ holds.

We refer to such systems simply as *proof systems* and omit the word propositional. Note that proof systems can alternatively be defined for tautologies because of the fact that F is an unsatisfiable formula iff $\neg F$ is a tautology. In this manuscript, however, we use the phrase proof system in the context of unsatisfiable formulas only.

The strength of a proof system is characterized by the sizes of proofs it admits for various unsatisfiable formulas: a stronger proof system can verify the correctness of shorter proofs presented in a more complex format. This motivates the following definition.

Definition 2.6. The *complexity* of a formula F under a proof system S, denoted $\mathcal{C}_S(F)$, is the length of the shortest refutation of F in S.

Let $\{F_n\}$ be a family of formulas over an increasing number of variables n. The asymptotic complexity of $\{F_n\}$ in S with respect to n is given by the function $f(n) = \mathcal{C}_S(F_n)$ and is denoted $\mathcal{C}_S(F_n)$, with abuse of notation. We will be interested in characterizing families of formulas as having polynomial or exponential asymptotic complexity under specific proof systems.

2.2.1 Resolution

Resolution (RES) is a widely studied simple proof system that can be used to prove unsatisfiability of CNF formulas. It forms the basis of many popular systems for practical theorem proving. Lower bounds on resolution proof sizes thus have a bearing on the running time of these systems.

The *resolution rule* states that given clauses $C_1 = (A \vee x)$ and $C_2 = (B \vee \neg x)$, one can derive the clause $C = (A \vee B)$ by *resolving on* x. C_1 and C_2 are called the *parent* clauses and C is called their *resolvent*. The resolution rule has the property that a derived clause is satisfied by any assignment that satisfies both the parent clauses.

Definition 2.7. A *resolution derivation* of C from a CNF formula F is a sequence $\pi = (C_1, C_2, \ldots, C_s = C)$ with the following property: each clause C_i in π is either a clause of F (an *initial* clause) or is derived by applying the resolution rule to C_j and C_k, $1 \leq j, k < i$ (a *derived* clause). The *size* of π is s, the number of clauses occurring in it.

We assume that the clauses in π are non-redundant, i.e., each $C_j \neq C$ in π is used to derive at least one other clause $C_i, i > j$. Any derivation of the empty clause Λ from F, also called a *refutation* or *proof* of F, shows that F is unsatisfiable.

Definition 2.8. Let F be a CNF formula and π a resolution proof of its unsatisfiability.

(a) The *size* of π, $size(\pi)$, is the number of clauses appearing in π.

(b) The *resolution complexity* of F, $\mathsf{RES}(F)$, is the minimum of $size(\pi)$ over all resolution proofs π of F; if no such proofs exist, $\mathsf{RES}(F) = \infty$.

(c) The *width* of a clause is the number of literals occurring in it. The width $w(F)$ of F and the width $w(\pi)$ of π are the maximum of the widths of all clauses in F and π, respectively.

(d) The *refutation width* of F, $w(F \vdash \Lambda)$, is the minimum of $w(\pi)$ over all proofs π of F.

As we shall see in Section 2.2.3, to prove a lower bound on $\mathsf{RES}(F)$, it is sufficient to prove a lower bound on the refutation width of F. It also typically turns out to be easier to analyze the width rather than the size of the smallest refutation. This makes the concept of width quite useful in proof complexity.

It is often insightful to think of the *structure* of a resolution refutation (or derivation) π in terms of a directed acyclic graph G_π defined as follows. G_π has a vertex for each clause in π, labeled with that clause. If the clause C_k labeling a vertex v in G_π is derived by resolving clauses C_i and C_j upon a variable x, then (a) v has a secondary label x, and (b) v has two outgoing edges directed to the vertices labeled C_i and C_j. All vertices labeled with initial clauses of π do not have a secondary label and have outdegree zero.

2.2.2 Refinements of Resolution

Despite its simplicity, unrestricted resolution as defined above (also called *general resolution*) is hard to implement efficiently due to the difficulty of finding good choices of clauses to resolve; natural choices typically yield huge storage requirements. Various restrictions on the structure of resolution proofs lead to less powerful but easier to implement refinements that have been studied extensively in proof complexity.

Definition 2.9. Let $\pi = (C_1, C_2, \ldots, C_s = C)$ be a resolution derivation, G_π be the graph associated with it, α be an assignment to the variables in π, α_F be the all FALSE assignment, and α_T be the all TRUE assignment.

(a) π is *tree-like* if each vertex in G_π corresponding to a derived clause has indegree 1.

(b) π is *regular* if no secondary vertex label appears twice in any directed path in G_π.

(c) π is *ordered* or *Davis-Putnam* if the sequence of secondary vertex labels along every directed path in G_π respects a fixed total ordering of the variables.

(d) π is *linear* if each C_i in π is either an initial clause or is derived by resolving C_{i-1} with $C_j, j < i-1$.

(e) π is an *α-derivation* if at least one parent clause involved in each resolution step in it is falsified by α.

(f) π is *positive* if it is an α-derivation for $\alpha = \alpha_F$.

(g) π is *negative* if it is an α-derivation for $\alpha = \alpha_T$.

(h) π is *semantic* if it is an α-derivation for some α.

While all these refinements are sound and complete as proof systems, they differ vastly in efficiency. For instance, in a series of results, Bonet et al. [27], Bonet and Galesi [28], and Buresh-Oppenheim and Pitassi [31] have shown that regular, ordered, linear, positive, negative, and semantic resolution are all exponentially stronger than tree-like resolution. On the other hand, Bonet et al. [27] and Alekhnovich et al. [3] have proved that tree-like, regular, and ordered resolution are exponentially weaker than RES.

2.2.3 The Size-Width Relationship

Most known resolution complexity lower bounds, including our results in subsequent chapters, can be proved using a general result of Ben-Sasson and Wigderson [23] that is derived from earlier arguments by Haken [60] and Clegg, Edmonds, and Impagliazzo [36]. It provides a relationship between the size of resolution proofs and their width (recall Definition 2.8), namely, any short proof of unsatisfiability of a CNF formula can be converted to one of small width. Therefore, a lower bound on the width of a resolution proof implies a lower bound on its size.

For a reason that will become clear in Section 2.3.1, we will use $\mathsf{DPLL}(F)$ to denote the tree-like resolution complexity of a formula F.

Proposition 2.1 ([23]). *For any CNF formula F, $\mathsf{DPLL}(F) \geq 2^{w(F\vdash\Lambda)-w(F)}$.*

Proposition 2.2 ([23]). *For any CNF formula F over n variables and $c = 1/(9\ln 2)$, $\mathsf{RES}(F) \geq 2^{c(w(F\vdash\Lambda)-w(F))^2/n}$.*

For completeness, we sketch the proof of this result for the case of tree-like resolution. Suppose we have a refutation π of F (over n variables) with $size(\pi) \leq 2^b$. The idea is to use induction on n and b to construct a refutation π' of F such that $width(\pi') \leq b$. Let the last variable resolved upon in π be x. Assume without loss of generality that x is derived in π by a tree-like derivation of size at most 2^{b-1}. This gives a refutation of $F|_{\neg x}$ of the same size by simply removing x from all clauses. By

induction on b, this can be converted into a refutation of $F|_{\neg x}$ of width at most $b-1$, which immediately gives a derivation π'' of x from F of width at most b by adding x back to the initial clauses from which it was removed and propagating the change.

On the other hand, π contains a derivation of $\neg x$ of size at most 2^b which can be converted to a refutation of $F|_x$ of the same size. By induction on n, this refutation, and hence the original derivation of $\neg x$, can be converted to one of width at most b. Now resolve, wherever possible, each of the initial clauses of this small width derivation of $\neg x$ with the result x of the derivation π'' and propagate the resulting simplification. This gives a refutation π' of F of width at most b.

2.3 The DPLL *Procedure and Clause Learning*

The Davis-Putnam-Logemann-Loveland or DPLL procedure is both a proof system as well as a collection of algorithms for finding proofs of unsatisfiable formulas. It can equally well be thought of as a collection of complete algorithms for finding a satisfying assignment for a given formula; its failure to find such an assignment constitutes a proof of unsatisfiability of the formula. While the former view is more suited to proof complexity theory, the latter is the norm when designing satisfiability algorithms. Davis and Putnam [44] came up with the basic idea behind this procedure. However, it was only a couple of years later that Davis, Logemann, and Loveland [43] presented it in the efficient top-down form in which it is widely used today.

Algorithm 1, `DPLL-recursive(`F, ρ`)`, sketches the basic DPLL procedure on CNF formulas. The idea is to repeatedly select an unassigned literal ℓ in the input formula F and recursively search for a satisfying assignment for $F|_\ell$ and $F_{\neg\ell}$. The step where such an ℓ is chosen is commonly referred to as the *branching* step. Setting ℓ to TRUE or FALSE when making a recursive call is called a *decision*. The end of each recursive call, which takes F back to fewer assigned variables, is called the *backtracking* step.

A partial assignment ρ is maintained during the search and output if the formula turns out to be satisfiable. If $F|_\rho$ contains the empty clause, the corresponding clause of F from which it came is said to be *violated* by ρ. To increase efficiency, unit clauses are immediately set to TRUE as outlined in Algorithm 1. *Pure literals* (those whose negation does not appear) are also set to TRUE as a preprocessing step and, in some implementations, in the simplification process after every branch.

At any point during the execution of the algorithm, a variable that has been assigned a value at a branching step is called a *decision variable* while one that has been assigned a value by unit propagation is called an *implied variable*. The *decision level* of an assigned variable is the recursive depth (starting at 0) of the call to `DPLL-recursive` that assigns it a value.

Variants of this algorithm form the most widely used family of complete algorithms for formula satisfiability. They are frequently implemented in an iterative rather than recursive manner, resulting in significantly reduced memory usage. The key difference

```
Input  : A CNF formula F and an initially empty partial assignment ρ
Output : UNSAT, or an assignment satisfying F
begin
    (F, ρ) ← UnitPropagate(F, ρ)
    if F contains the empty clause then return UNSAT
    if F has no clauses left then
        Output ρ
        return SAT
    ℓ ← a literal not assigned by ρ                    // the branching step
    if DPLL-recursive(F|_ℓ, ρ ∪ {ℓ}) = SAT then return SAT
    return DPLL-recursive(F|_¬ℓ, ρ ∪ {¬ℓ})
end

UnitPropagate(F)
begin
    while F contains no empty clause but has a unit clause x do
        F ← F|_x
        ρ ← ρ ∪ {x}
    return (F, ρ)
end
```

Algorithm 2.1: `DPLL-recursive(`F, ρ`)`

in the iterative version is the extra step of *unassigning* variables when one backtracks. The naive way of unassigning variables in a CNF formula is computationally expensive, requiring one to examine every clause in which the unassigned variable appears. However, the *watched literals* scheme of Moskewicz et al. [88] provides an excellent way around this and merits a brief digression.

The Watched Literals Scheme

The key idea behind the watched literals scheme, as the name suggests, is to maintain and "watch" two special literals for each active (i.e., not yet satisfied) clause that are not FALSE under the current partial assignment. Recall that empty clauses halt the DPLL process and unit clauses are immediately satisfied. Hence, one can always find such watched literals in all active clauses. Further, as long as a clause has two such literals, it cannot be involved in unit propagation. These literals are maintained as follows. When a literal ℓ is set to FALSE, we must find another watched literal for the clause that had ℓ as a watched literal. We must also let $\neg\ell$ be a watched literal for previously active clauses that are now satisfied because of this assignment to ℓ. By doing this, positive literals are given priority over unassigned literals for being the watched literals.

With this setup, one can test a clause for satisfiability by simply checking whether at least one of its two watched literals is TRUE. Moreover, the relatively small amount of extra book-keeping involved in maintaining watched literals is well paid off when one unassigns a literal ℓ by backtracking – in fact, one needs to do absolutely nothing! The invariant about watched literals is maintained as such, saving a substantial amount of computation that would have been done otherwise.

2.3.1 Relation to Tree-like Resolution

When a formula F is unsatisfiable, the transcript of the execution of DPLL on F forms a proof of its unsatisfiability. This proof is referred to as a DPLL *refutation* of F. The *size* of a DPLL refutation is the number of branching steps in it. As the following Proposition shows, the structure of DPLL refutations is intimately related to the structure of tree-like resolution refutations.

Proposition 2.3. *A CNF formula F has a* DPLL *refutation of size at most s iff it has a tree-like resolution refutation of size at most s.*

Proof. The idea is to associate with every DPLL refutation τ a tree T^τ and show how T^τ can be viewed as or simplified to the graph G_π associated with a tree-like resolution refutation π of F. Given a DPLL refutation τ, the tree T^τ is constructed recursively by invoking the construction for `DPLL-recursive(`F, ϕ`)`. We describe below the construction in general for `DPLL-recursive(`H, ρ`)` for any sub-formula H of F and partial assignment ρ to the variable of F.

Start by creating the root node v for the tree corresponding to the procedure call `DPLL-recursive(`H, ρ`)`. If the procedure terminates because there is an empty clause after unit propagation, label v with an initial clause of F that has become empty and stop. If not, let ℓ be the literal chosen in the branching step of the call. Recursively create the two subtrees T_ℓ and $T_{\neg\ell}$ associated with the recursive calls to `DPLL-recursive(`$H|_\ell, \{\ell\}$`)` and `DPLL-recursive(`$H|_{\neg\ell}, \{\neg\ell\}$`)`, respectively. If either of T_ℓ or $T_{\neg\ell}$ is labeled by a clause that does not contain $\neg\ell$ or ℓ, respectively, then discard v and the other subtree, associate this one with the procedure call `DPLL-recursive(`H, ρ`)`, and stop. Otherwise, add edges from v to the roots of T_ℓ and $T_{\neg\ell}$. Let x be the variable corresponding to ℓ. Label v with the clause obtained by resolving on x the clauses labeling T_ℓ and $T_{\neg\ell}$. Finally, assign x as the secondary label for v.

It can be verified that the label of the root node of the final tree T^τ corresponding to the call to `DPLL-recursive(`F, ϕ`)` is the empty clause Λ and that T^τ is precisely the graph G_π associated with a legal tree-like resolution refutation π of F.

On the other hand, if one starts with a graph $G|_\pi$ associated with a tree-like resolution refutation π of F, the graph can be viewed unchanged as the tree T^τ associated with a DPLL refutation τ of F. This finishes the proof. □

Corollary 2.1. *For a CNF formula F, the size of the smallest* DPLL *refutation of F is equal to the size of the smallest tree-like resolution refutation of F.*

This explains why we used $\mathsf{DPLL}(F)$ to denote the tree-like resolution complexity of F in Section 2.2.3.

2.3.2 Clause Learning

The technique of clause learning was first introduced in the context of the DPLL-based SAT solvers by Marques-Silva and Sakallah [84]. It can be thought of as an extension of the DPLL procedure that caches causes of assignment failures in the form of learned clauses. It proceeds by following the normal branching process of DPLL until there is a "conflict," i.e., a variable is implied to be TRUE as well as FALSE by unit propagation. We give here a brief sketch of how conflicts are handled, deferring more precise details to Section 4.2.

If a conflict occurs when no variable is currently branched upon, the formula is declared UNSAT. Otherwise, the algorithm looks at the graphical structure of variable assignment implications (caused by unit propagation). From this, it infers a possible "cause" of the conflict, i.e., a relatively small subset of the currently assigned variables that, by unit propagation, results in the conflict. This cause is learned in the form of a "conflict clause." The idea is to avoid any future conflicts that may result from a careless assignment to the subset of variables already known to potentially cause a conflict. The algorithm now backtracks and continues as in ordinary DPLL, treating the learned clause just like the initial ones. A clause is said to be *known* at a stage if it is either an initial clause or has previously been learned.

Algorithm 2.2 gives the basic structure of the clause learning algorithm by Moskewicz et al. [88] used in the popular SAT solver zChaff. This algorithm forms the basis of our implementations and experiments in subsequent chapters. We present it here as the top-level iterative process that lies at the heart of zChaff.

The procedure `DecideNextBranch` chooses the next variable to branch on. In zChaff, this is done using the Variable State Independent Decaying Sum (VSIDS) heuristic which assigns a slowly decaying weight to each literal that is boosted whenever the literal is involved in a conflict. Note that there is no explicit variable flip in the entire algorithm. The conflict clause learning strategy used by zChaff automatically (by unit propagation) flips the assignment of the current variable before backtracking. The procedure `Deduce` applies unit propagation, keeping track of any clauses that may become empty, causing what is known as a conflict. If all clauses have been satisfied, it declares the formula to be SAT. The procedure `AnalyzeConflict` looks at the structure of implications and computes from it a conflict clause to learn. It also computes and returns the decision level that one needs to backtrack.

In general, the learning process is expected to save us from redoing the same computation when we later have an assignment that causes conflict due in part to

```
Input  : A CNF formula
Output : UNSAT, or SAT along with a satisfying assignment
begin
    while TRUE do
        DecideNextBranch
        while TRUE do
            status ← Deduce
            if status = CONFLICT then
                blevel ← AnalyzeConflict
                if blevel = 0 then return UNSAT
                Backtrack(blevel)
            else if status = SAT then
                Output current assignment stack
                return SAT
            else break
end
```

Algorithm 2.2: `DPLL-ClauseLearning`

the same reason. Variations of such conflict-driven learning include different ways of choosing the clause to learn (different *learning schemes*) and possibly allowing multiple clauses to be learned from a single conflict. In the last decade, many algorithms based on this idea have been proposed and demonstrated to be empirically successful on large problems that could not be handled using other methodologies. These include Relsat by Bayardo Jr. and Schrag [13], Grasp by Marques-Silva and Sakallah [84], SATO by Zhang [113], and, as mentioned before, zChaff. We leave a more detailed discussion of the concepts involved in clause learning as well as its formulation as a proof system CL to Section 4.2.

Remark 2.1. Throughout this thesis, we will use the term DPLL to denote the basic branching and backtracking procedure given in Algorithm 1, and possibly the iterative version of it. It will not include learning conflict clauses when backtracking, but will allow intelligent branching heuristics as well as common extensions such as fast backtracking and restarts discussed in Section 4.2. Note that this is in contrast with the occasional use of the term DPLL to encompass practically all branching and backtracking approaches to SAT, including those involving learning.

Chapter 3

THE RESOLUTION COMPLEXITY OF GRAPH PROBLEMS

We are now ready to describe the technical contributions of this thesis in detail. We begin in this chapter with our main proof complexity results. These are for the resolution proof system and apply to the CNF formulations of three graph problems, namely, (the existence of) independent sets, vertex covers, and cliques.

An independent set in an undirected graph is a set of vertices no two of which share an edge. The problem of determining whether or not a given graph contains an independent set of a certain size is NP-complete as shown by Karp [69][1]. Consequently, the complementary problem of determining non-existence of independent sets of that size in the graph is co-NP-complete. This chapter studies the problem of providing a resolution proof of the non-existence of independent sets.

Any result that holds for nearly all graphs can be alternatively formalized as a result that holds with very high probability when a graph is chosen at random from a "fair" distribution. We use this approach and study the resolution complexity of the independent set problem in random graphs chosen from a standard distribution. Independent sets and many other combinatorial structures in random graphs have very interesting mathematical properties as discussed at length in the texts by Bollobás [26] and Janson, Łuczak, and Ruciński [66]. In particular, the size of the largest independent set can be described with high certainty and accuracy in terms of simple graph parameters.

This work proves that given almost any graph G and a number k, exponential-size resolution proofs are required to show that G does not contain an independent set of size k. In fact, when G has no independent set of size k, exponential-size resolution proofs are required to show that independent sets of even a much larger size $k' \gg k$ do not exist in G. This yields running time lower bounds for certain classes of algorithms for approximating the size of the largest independent sets in random graphs.

Closely related to the independent set problem are the problems of proving the non-existence of cliques or vertex covers of a given size. Our results for the independent set problem also lead to bounds for these problems. As the approximations for the vertex cover problem act differently from those for independent sets, we state the results in terms of vertex covers as well as independent sets. (Clique approximations are essentially identical to independent set approximations.)

---

[1] Karp actually proved the related problem of clique to be NP-complete.

Many algorithms for finding a maximum-size independent set have been proposed. Influenced by algorithms of Tarjan [105] and Tarjan and Trojanowski [106], Chvátal [34] devised a specialized proof system for the independent set problem. In this system he showed that with probability approaching 1, proofs of non-existence of large independent sets in random graphs with a linear number of edges must be exponential in size. Chvátal's system captures many backtracking algorithms for finding a maximum independent set, including those of Tarjan [105], Tarjan and Trojanowski [106], Jian [67], and Shindo and Tomita [100]. In general, the transcript of any *f-driven algorithm* [34] for independent sets running on a given graph can be translated into a proof in Chvátal's system.

Our results use the well-known resolution proof system for propositional logic rather than Chvátal's specialized proof system. Given a graph G and an integer k, we consider encoding the existence of an independent set of size k in G as a CNF formula and examine the proof complexity of such formulas in resolution. Resolution on one of the encodings we present captures the behavior of Chvátal's proofs on the corresponding graphs. For all our encodings, we show that given a randomly chosen graph G of moderate edge density, almost surely, the size of any resolution proof of the statement that G does not have an independent set of a certain size must be exponential in the number of vertices in G. This implies an exponential lower bound on the running time of many algorithms for searching for, or even approximating, the size of a maximum independent set or minimum vertex cover in G.

Although resolution is a relatively simple and well-studied proof system, one may find the concept of resolution proofs of graph theoretic problems somewhat unnatural. The tediousness of propositional encodings and arguments related to them contributes even more to this. Chvátal's proof system, on the other hand, is completely graph theoretic in nature and relates well to many known algorithms for the independent set problem. By proving that resolution can efficiently simulate Chvátal's proof system, we provide another justification for studying the complexity of resolution proofs of graph problems.

In the proof complexity realm, exponential bounds for specialized structured formulas and for unstructured random k-CNF formulas have previously been shown by several researchers including Haken [60], Urquhart [110], Razborov [95], Chvátal and Szemerédi [35], Beame et al. [17], and Ben-Sasson and Wigderson [23]. However, much less is known for large classes of structured formulas. Our results significantly extend the families of structured random formulas for which exponential resolution lower bounds are known beyond the graph coloring example recently shown by Beame et al. [14]. (Note that our results neither imply nor follow from those in [14]. Although the non-existence of an independent set of size n/K implies in a graph of n vertices implies that the graph is not K-colorable, the argument requires an application of the pigeonhole principle which is not efficiently provable in resolution [60].)

For obtaining our lower bounds, instead of looking at the general problem of dis-

proving the existence of *any* large independent set in a graph, we focus on a restricted class of independent sets that we call *block-respecting independent sets*. We show that even ruling out this smaller class of independent sets requires exponential-size resolution proofs. These restricted independent sets are simply the ones obtained by dividing the n vertices of the given graph into k blocks of equal size (assuming k divides n) and choosing one vertex from each block. Since it is easier to rule out a smaller class of independent sets, the lower bounds we obtain for the restricted version are stronger in the sense that they imply lower bounds for the general problem. While block-respecting independent sets are a helpful tool in analyzing general resolution proofs, we are able to give better lower bounds for DPLL proofs by applying a counting argument directly to the general problem.

We show that our results extend the known lower bounds for Chvátal's system [34] to resolution and also extend them to graphs with many more than a linear number of edges, yielding bounds for approximation algorithms as well as for exact computation. More precisely, we show that no resolution-based technique can achieve polynomial-time approximations of independent set size within a factor of $\Delta/(6 \log \Delta)$. For the vertex cover problem, we show an analogous result for approximation factors better than $3/2$.

Recently, by computing a property related to the Lovász number of a random graph, more precisely its vector chromatic number, Coja-Oghlan [37] gave an expected polynomial time $O(\sqrt{\Delta}/\log \Delta)$-approximation algorithm for the size of the maximum independent set in random graphs of density Δ. Thus our results show that this new approach is provably stronger than that obtainable using resolution-based algorithms.

The proof our of main lower bound is based on the size-width relationship of resolution proofs discussed in Section 2.2.3. It uses the property that any proof of non-existence of an independent set of a certain size in a random graph is very likely to refer to a relatively large fraction of the vertices of the input graph, and that any clause capturing the properties of this large fraction of vertices must have large width.

More precisely, the proof can be broadly divided into two parts, both of which use the fact that random graphs are almost surely locally sparse. We first show that the minimum number s of input clauses that are needed for any refutation of the problem is large for most graphs. We then use combinatorial properties of independent sets in random graphs to say that any clause minimally implied by a relatively large subset of these s clauses has to be large. Here minimally implied means that implied by the size-s set of clauses under consideration but not by any proper subset of it. These two arguments together allow us to deduce that the width of any such refutation has to be large. The size-width relationship translates this into a lower bound on the refutation size.

We begin with basic properties of independent sets in Section 3.1. In Section 3.2 we describe three natural encodings of the independent set problem as CNF formulas and compare the proof sizes of the different encodings. In Sections 3.3 and 3.4 we com-

pare these to proofs in Chvátal's proof system for independent sets and to the proof complexity of related graph theory problems, namely, vertex cover and clique. After giving some simple proof complexity upper bounds based on exhaustive backtracking algorithms in Section 3.5, we prove the main resolution lower bounds in Sections 3.6 to 3.8. Note that Sections 3.2 to 3.5 contain somewhat tedious details that the reader may want to skip during the first read. Finally, in Section 3.10 we prove a somewhat stronger lower bound that applies to exhaustive backtracking algorithms (as well as the DPLL procedure) and qualitatively matches our upper bounds for the same.

Remark 3.1. Although we described DPLL algorithms in Section 2.3 as working on propositional CNF formulas, they capture a much more general class of algorithms that are based on branching and backtracking. For instance, basic algorithms for finding a maximum independent set, such as that of Tarjan [105], branch on each vertex v by either including v in the current independent set and deleting it and all its neighbors from further consideration, or excluding v from the current independent set and recursively finding a maximum independent set in the remaining graph. This can be formulated as branching and backtracking on appropriate variables of a CNF formulation of the problem. In fact, more complicated algorithms, such as that of Tarjan and Trojanowski [106], branch in a similar manner not only on single vertices but on small subsets of vertices, reusing subproblems already solved. Such algorithms also fall under the category of resolution-based (not necessarily tree-like) algorithms and our lower bounds apply to them as well because of the following reasoning. The computation history of these algorithms can be translated into a proof in Chvátal's system by replacing each original branch in the computation with a small tree of single-vertex branches. We then resort to our result that resolution can efficiently simulate Chvátal's proof system.

3.1 Independent Sets in Random Graphs

For any undirected graph $G = (V, E)$, let $n = |V|$ and $m = |E|$. A *k-independent set* in G is a set of k vertices no two of which share an edge. We will describe several natural ways of encoding in clausal form the statement that G has a k-independent set. Their refutations will be proofs that G does *not* contain any k-independent set. We will be interested in size bounds for such proofs.

Combinatorial properties of random graphs have been studied extensively (see, for instance, [26, 66]). We use the standard model $\mathbb{G}(n, p)$ for graphs with n vertices where each of the $\binom{n}{2}$ edges is chosen independently at random with probability $p \in [0, 1]$. $G \sim \mathbb{G}(n, p)$ denotes a graph G chosen at random from this distribution. We will state most of our results in terms of parameters n and Δ, where $\Delta \stackrel{\text{def}}{=} np$ is (roughly) the average degree of G.

We will need both worst case and almost certain bounds on the size of the largest independent set in graphs of density Δ.

Proposition 3.1 (Turan's Theorem). *Every graph G with n vertices and average degree Δ has an independent set of size $\lfloor \frac{n}{\Delta+1} \rfloor$. In general, for any integer k satisfying $\Delta < \frac{n}{k-1} - 1$, G has an independent set of size k.*

For $\epsilon > 0$, let $k_{\pm\epsilon}$ be defined as follows[2]:

$$k_{\pm\epsilon} = \lfloor \frac{2n}{\Delta}(\log \Delta - \log\log \Delta + 1 - \log 2 \pm \epsilon) \rfloor$$

Proposition 3.2 ([66], Theorem 7.4). *For every $\epsilon > 0$ there is a constant C_ϵ such that the following holds. Let $\Delta = np$, $C_\epsilon \leq \Delta \leq n/\log^2 n$, and $G \sim \mathbb{G}(n,p)$. With probability $1 - o(1)$ in n, the largest independent set in G is of size between $k_{-\epsilon}$ and $k_{+\epsilon}$.*

This shows that while random graphs are very likely to have an independent set of size $k_{-\epsilon}$, they are very unlikely to have one of size $k_{+\epsilon}+1$. The *number* of independent sets of a certain size also shows a similar threshold behavior. While there are almost surely no independent sets of size $(2n/\Delta)\log\Delta$, the following lemma, which follows by a straightforward extension of the analysis in [66, Lemma 7.3], shows that there are exponentially many of size $(n/\Delta)\log\Delta$. We use this bound later to put a limit on the best one can do with exhaustive backtracking algorithms that systematically consider all potential independent sets of a certain size.

Lemma 3.1. *There is a constant $C > 0$ such that the following holds. Let $\Delta = np$, $\Delta \leq n/\log^2 n$, and $G \sim \mathbb{G}(n,p)$. With probability $1 - o(1)$ in n, G contains at least $2^{C(n/\Delta)\log^2\Delta}$ independent sets of size $\lfloor (n/\Delta)\log\Delta \rfloor$.*

Proof. Let X_k be a random variable whose value is the number of independent sets of size k in $G = (V,E)$. The expected value of X_k is given by:

$$\begin{aligned}
\mathbb{E}[X_k] &= \sum_{S \subseteq V, |S|=k} \Pr[S \text{ is an independent set in } G] \\
&= \binom{n}{k}(1-p)^{\binom{k}{2}} \\
&\geq \left(\frac{n}{k}\right)^k e^{-cpk^2} \quad \text{for } c > 1/2,\ p = o(1) \text{ in } n, \text{ and large enough } n \\
&= \left(\frac{n}{k}\, e^{-c\Delta k/n}\right)^k
\end{aligned}$$

[2]Throughout this thesis, logarithms denoted by log will have the natural base e and those denoted by $\log_2$ will have base 2.

Let $c = 0.55$ and $C = 0.05/\log 2$ so that $\Delta^{1-c}/\log\Delta \geq 2^{C\log\Delta}$. Setting $k = \lfloor (n/\Delta)\log\Delta \rfloor$ and observing that $\left((n/k)e^{-c\Delta k/n}\right)^k$ decreases with k,

$$\begin{aligned}\mathbb{E}\left[X_{\lfloor (n/\Delta)\log\Delta\rfloor}\right] &\geq \left(\frac{\Delta}{\log\Delta}e^{-c\log\Delta}\right)^{(n/\Delta)\log\Delta} \\ &\geq 2^{C(n/\Delta)\log^2\Delta}.\end{aligned}$$

We now use the standard second moment method to prove that X_k for $k = \lfloor (n/\Delta)\log\Delta \rfloor$ asymptotically almost surely lies very close to its expected value. We begin by computing the expected value of X_k^2 and deduce from it that the variance of X_k is small.

$$\begin{aligned}\mathbb{E}\left[X_k^2\right] &= \sum_{S\subseteq V,\ |S|=k} \Pr\left[S \text{ is independent}\right] \sum_{i=0}^{k} \sum_{T\subseteq V,\ |T|=k,\ |S\cap T|=i} \Pr\left[T \text{ is independent}\right] \\ &= \binom{n}{k}(1-p)^{\binom{k}{2}} \sum_{i=0}^{k} \binom{k}{i}\binom{n-k}{k-i}(1-p)^{\binom{k}{2}-\binom{i}{2}}\end{aligned}$$

$$\begin{aligned}\text{Therefore}\quad \frac{var\left[X_k\right]}{\mathbb{E}\left(\left[X_k\right]\right)^2} &= \frac{\mathbb{E}\left[X_k^2\right]}{\left(\mathbb{E}\left[X_k\right]\right)^2} - 1 \\ &= \frac{\binom{n}{k}(1-p)^{\binom{k}{2}} \sum_{i=0}^{k} \binom{k}{i}\binom{n-k}{k-i}(1-p)^{\binom{k}{2}-\binom{i}{2}}}{\left[\binom{n}{k}(1-p)^{\binom{k}{2}}\right]^2} - 1\end{aligned}$$

This is the same expression as equation (7.8) of [66, page 181]. Following the calculation of Lemma 7.3 of [66], we obtain that $var[X_k]/(\mathbb{E}[X_k^2])^2 \to 0$ for $k = \lfloor (n/\Delta)\log\Delta \rfloor$ as $n \to \infty$ when $\Delta \geq \sqrt{n}\log^2 n$. When $\Delta \leq \sqrt{n}\log^2 n$, an argument along the lines of Theorem 7.4 of [66] provides the same result. Applying the second moment method, this leads to the desired bound. □

3.2 Encoding Independent Sets as Formulas

In order to use a propositional proof system to prove that a graph does not have an independent set of a particular size, we first need to formulate the problem as a propositional formula. This is complicated by the difficulty of counting set sizes using CNF formulas.

One natural way to encode the independent set problem is to have indicator variables that say which vertices are in the independent set and auxiliary variables that count the number of vertices in the independent set. This encoding is discussed in Section 3.2.1. The clauses in this encoding, although capturing the simple concept of

counting, are somewhat involved. Moreover, the existence of two different types of variables makes this encoding difficult to reason about directly.

A second encoding, derived from the counting-based encoding, is described in Section 3.2.2. It is based on a mapping from the vertices of the graph to k additional nodes as an alternative to straightforward counting, and uses variables of only one type. This is essentially the same encoding as the one used by Bonet, Pitassi, and Raz [29] for the clique problem, except that in our case we need to add an extra set of clauses, called ordering clauses, to make the lower bounds non-trivial. (Otherwise, lower bounds trivially follow from known lower bounds for the pigeonhole principle [60] which have nothing to do with the independent set problem; in [29] this problem did not arise because the proof system considered was cutting planes where, as shown by Cook et al. [40], the pigeonhole principle has short proofs.)

Section 3.2.3 finally describes a much simpler encoding which is the one we analyze directly for our lower bounds. This encoding considers only a restricted class of independent sets that we call *block-respecting independent sets*, for which the problem of counting the set size is trivial. Hence, the encoding uses only one type of variable that indicates whether or not a given vertex is in the independent set. Refutation of this third encoding rules out the existence of the smaller class of block-respecting independent sets only. Intuitively, this should be easier to do than ruling out all possible independent sets. In fact, we show that the resolution and DPLL refutations of this encoding are bounded above in size by those of the mapping encoding and are at worst a small amount larger than those of the counting encoding. As a result, we can translate our lower bounds for this third encoding to each of the other encodings. Further, we give upper bounds for the two general encodings which also apply to the simpler block-respecting independent set encoding.

For the rest of this chapter, identify the vertex set of the input graph with $\{1, 2, \ldots, n\}$. Each encoding will be defined over variables from one or more of the following three categories:

- $x_v, 1 \leq v \leq n$, which is TRUE iff vertex v is chosen by the truth assignment to be in the independent set,
- $y_{v,i}, 0 \leq i \leq v \leq n$, $0 \leq i \leq k$, which is TRUE iff precisely i of the first v vertices are chosen in the independent set, and
- $z_{v,i}, 1 \leq v \leq n, 1 \leq i \leq k$, which is TRUE iff vertex v is chosen as the i^{th} node of the independent set.

A desirable property of all independent set encodings is their *monotonicity*, i.e., for $k' > k$, proving the non-existence of an independent set of size k' in that encoding must not be any harder than doing so for size k, up to a polynomial factor. This property indeed holds for each of the three encodings we consider below.

3.2.1 Encoding Based on Counting

The *counting encoding*, $\alpha_{count}(G,k)$, of the independent set problem is defined over variables x_v and $y_{v,i}$. As mentioned previously, this encoding is somewhat tedious in nature. It has the following three kinds of clauses:

(a) Edge Clauses: For each edge (u,v), $\alpha_{count}(G,k)$ has one clause saying that at most one of u and v is selected; $\forall (u,v) \in E, u < v : (\neg x_u \vee \neg x_v) \in \alpha_{count}(G,k)$

(b) Size-k Clause: There is a clause saying that the independent set chosen is of size k; $y_{n,k} \in \alpha_{count}(G,k)$

(c) Counting Clauses: There are clauses saying that variables $y_{v,i}$ correctly count the number of vertices chosen. For simplicity, we first write this condition not as a set of clauses but as more general propositional formulas. For the base case, $\alpha_{count}(G,k)$ contains $y_{0,0}$ and the clausal form of $(y_{v,0} \leftrightarrow (y_{v-1,0} \wedge \neg x_v))$ for $v \in \{1, \ldots n\}$. Further, $\forall i, v, 1 \leq i \leq v \leq n$, $1 \leq i \leq k$, $\alpha_{count}(G,k)$ contains the clausal form of $(y_{v,i} \leftrightarrow ((y_{v-1,i} \wedge \neg x_v) \vee (y_{v-1,i-1} \wedge x_v)))$, unless $i = v$, in which case $\alpha_{count}(G,k)$ contains the clausal form of the simplified formula $(y_{v,v} \leftrightarrow (y_{v-1,v-1} \wedge x_v))$.

Translated into clauses, these conditions take the following form. Formulas defining $y_{v,0}$ for $v \geq 1$ translate into $\{(\neg y_{v,0} \vee y_{v-1,0}), (\neg y_{v,0} \vee \neg x_v), (y_{v,0} \vee \neg y_{v-1,0} \vee x_v)\}$. Further, formulas defining $y_{v,i}$ for $v > i \geq 1$ translate into $\{(y_{v,i} \vee \neg y_{v-1,i} \vee x_v), (y_{v,i} \vee \neg y_{v-1,i-1} \vee \neg x_v), (\neg y_{v,i} \vee y_{v-1,i} \vee y_{v-1,i-1}), (\neg y_{v,i} \vee y_{v-1,i} \vee x_v), (\neg y_{v,i} \vee y_{v-1,i-1} \vee \neg x_v)\}$, whereas in the case $i = v$ they translate into $\{(\neg y_{v,v} \vee y_{v-1,v-1}), (\neg y_{v,v} \vee x_v), (\neg x_v \vee \neg y_{v-1,v-1} \vee y_{v,v})\}$.

Lemma 3.2. *For any graph G over n vertices and $k' > k$,*

$$\begin{aligned} \mathsf{RES}(\alpha_{count}(G,k')) &< n\ \mathsf{RES}(\alpha_{count}(G,k)) + 2n^2 \quad \textit{and} \\ \mathsf{DPLL}(\alpha_{count}(G,k')) &< n\ \mathsf{DPLL}(\alpha_{count}(G,k)) + 2n^2. \end{aligned}$$

Proof. If G contains an independent set of size k, then there are no resolution refutations of $\alpha_{count}(G,k)$. By our convention, $Res(\alpha_{count}(G,k)) = DPLL(\alpha_{count}(G,k)) = \infty$, and the result holds. Otherwise consider a refutation π of $\alpha_{count}(G,k)$. Using π, we construct a refutation π' of $\alpha_{count}(G,k')$ such that $size(\pi') \leq (n-k+1)\ size(\pi) + 2(k'-k)(n-k)$, which is less than $n\ size(\pi) + 2n^2$. Further, if π is a tree-like refutation, then so is π'.

$\alpha_{count}(G,k')$ contains all clauses of $\alpha_{count}(G,k)$ except the size-k clause, $y_{n,k}$. Therefore, starting with $\alpha_{count}(G,k')$ as initial clauses and using π modified not to use the clause $y_{n,k}$, we derive a subclause of $\neg y_{n,k}$. This clause, however, cannot be a strict subclause of $\neg y_{n,k}$ because $\alpha_{count}(G,k) \setminus \{y_{n,k}\}$ is satisfiable. Hence, we must

obtain $\neg y_{n,k}$. Call this derivation D_n. By construction, $size(D_n) \leq size(\pi)$. Making a copy of D_n, we restrict it by setting $x_n \leftarrow \text{FALSE}, y_{n,k} \leftarrow y_{n-1.k}$ to obtain a derivation D_{n-1} of $\neg y_{n-1,k}$. Continuing this process, construct derivations D_p of $\neg y_{p,k}$ for $p \in \{n-1, n-2, \ldots, k\}$ by further setting $x_{p+1} \leftarrow \text{FALSE}, y_{p+1,k} \leftarrow y_{p,k}$. Again, by construction, $size(D_p) \leq size(\pi)$. Combining derivations $D_n, D_{n-1}, \ldots, D_k$ into π' gives a derivation of size at most $(n-k+1)size(\pi)$ of clauses $\neg y_{p,k}, k \leq p \leq n$, which is tree-like if π is.

Continuing to construct π', resolve the above derived clause $\neg y_{k,k}$ with the counting clause $(\neg y_{k+1,k+1} \vee y_{k,k})$ of $\alpha_{count}(G, k')$ to obtain $\neg y_{k+1,k+1}$. Now for v going from $k+2$ to n, resolve the already derived clauses $\neg y_{v-1,k+1}$ and $\neg y_{v-1,k}$ with the counting clause $(\neg y_{v,k+1} \vee y_{v-1,k+1} \vee y_{v-1,k})$ of $\alpha_{count}(G, k')$ to obtain $\neg y_{v,k+1}$. This gives a tree-like derivation of size less than $2(n-k)$ of clauses $\neg y_{p,k+1}, k+1 \leq p \leq n$, starting from clauses $\neg y_{q,k}, k \leq q \leq n$. Repeating this process $(k'-k)$ times gives a tree-like derivation of size less than $2(k'-k)(n-k)$ of clauses $\neg y_{p,k'}, k' \leq p \leq n$, starting from clauses $\neg y_{q,k}, k \leq q \leq n$, derived previously. In particular, $\neg y_{n,k'}$ is now a derived clause. Resolving it with the size-k' clause $y_{n,k'}$ of $\alpha_{count}(G, k')$ completes refutation π'. □

3.2.2 Encoding Based on Mapping

This encoding, denoted $\alpha_{map}(G, k)$, uses a mapping from n vertices of G to k nodes of the independent set as an indirect way of counting the number of vertices chosen by a truth assignment to be in the independent set. It can be viewed as a set of constraints restricting the mapping (see Figure 3.1). The idea is to map the nodes of the independent set to the sequence $(1, 2, \ldots, k)$ in the increasing order of their index as vertices in the graph. This encoding is defined over variables $z_{v,i}$ and has the following five kinds of clauses:

(a) Edge Clauses: For each edge (u, v), there are clauses saying that at most one of u and v is chosen in the independent set; $\forall (u, v) \in E, i, j, 1 \leq i < j \leq k$: $(\neg z_{u,i} \vee \neg z_{v,j}) \in \alpha_{map}(G, k)$

(b) Surjective Clauses: For each node i, there is a clause saying that some vertex is chosen as the i^{th} node of the independent set; $\forall i, 1 \leq i \leq k$: $(z_{1,i} \vee z_{2,i} \vee \ldots \vee z_{n,i}) \in \alpha_{map}(G, k)$

(c) Function Clauses: For each vertex v, there are clauses saying that v is not mapped to two nodes, i.e. it is not counted twice in the independent set; $\forall v, i, j, 1 \leq v \leq n, 1 \leq i < j \leq k$: $(\neg z_{v,i} \vee \neg z_{v,j}) \in \alpha_{map}(G, k)$

(d) 1-1 Clauses: For each node i, there are clauses saying no two vertices map to the i^{th} node of the independent set; $\forall i, u, v, 1 \leq i \leq k, 1 \leq u < v \leq n$: $(\neg z_{u,i} \vee \neg z_{v,i}) \in \alpha_{map}(G, k)$

(e) Ordering Clauses: For every pair of consecutive nodes, there are clauses saying that vertices are not mapped to these in the reverse order. This, by transitivity, implies that there is a unique mapping to k nodes once we have chosen k vertices to be in the independent set. $\forall u, v, i, 1 \leq u < v \leq n, 1 \leq i < k$: $(\neg z_{u,i+1} \vee \neg z_{v,i}) \in \alpha_{map}(G, k)$.

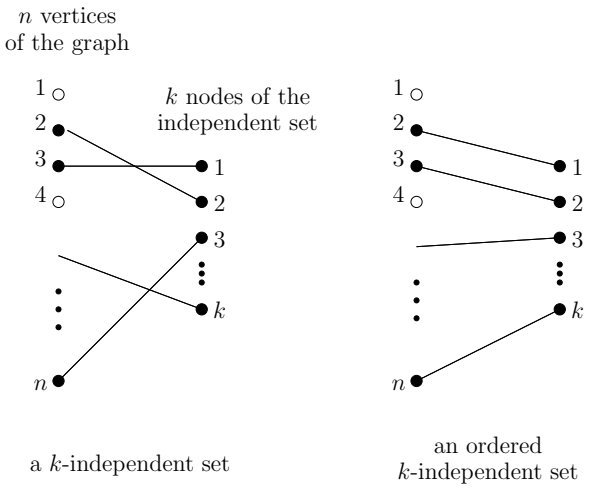

Figure 3.1: Viewing independent sets as a mapping from n vertices to k nodes

Lemma 3.3. *For any graph G and $k' \geq k$,*

$$\begin{aligned} \mathsf{RES}(\alpha_{map}(G, k')) &\leq \mathsf{RES}(\alpha_{map}(G, k)) \quad \textit{and} \\ \mathsf{DPLL}(\alpha_{map}(G, k')) &\leq \mathsf{DPLL}(\alpha_{map}(G, k)). \end{aligned}$$

Proof. If G contains an independent set of size k, then there are no resolution refutations of $\alpha_{map}(G, k)$. By our convention, $Res(\alpha_{map}(G, k)) = DPLL(\alpha_{map}(G, k)) = \infty$, and the result holds. Otherwise consider a refutation π of $\alpha_{map}(G, k)$. Observe that all clauses of $\alpha_{map}(G, k)$ are also clauses of $\alpha_{map}(G, k')$. Hence π is also a refutation of $\alpha_{map}(G, k')$, proving the desired bounds. □

3.2.3 Encoding Using Block-respecting Independent Sets

Fix $b = n/k$ for the rest of the chapter and assume for simplicity that k divides n (denoted $k \,|\, n$). Arbitrarily partition the vertices of G into k subsets, called *blocks*, of size b each. A *block-respecting independent set* of size k in G under this partitioning is an independent set in G with precisely one vertex in each of the k blocks. Clearly, if a graph does not contain *any* k-independent set, then it certainly does not contain

any block-respecting independent set of size k either. Note that the restriction $k \mid n$ is only to make the presentation simple. We can extend our arguments to all $k < n$ by letting each block have either b or $b+1$ vertices for $b = \lfloor n/k \rfloor$. The calculations are nearly identical to what we present here.

We now define a CNF formula $\alpha_{block}(G,k)$ over variables x_v that says that G contains a block-respecting independent set of size k. Assume without loss of generality that the first b vertices of G form the first block, the second b vertices form the second block, and so on. Henceforth, in all references to G, we will implicitly assume this fixed order of vertices and partition into k blocks. Since this order and partition are chosen arbitrarily, the bounds we derive hold for any partitioning of G into blocks.

The encoding $\alpha_{block}(G,k)$ contains the following three kinds of clauses:

(a) Edge Clauses: For each edge (u,v), there is one clause saying that not both u and v are selected; $\forall (u,v) \in E, u < v : \ (\neg x_u \vee \neg x_v) \in \alpha_{block}(G,k)$

(b) Block Clauses: For each block, there is one clause saying that at least one of the vertices in it is selected; $\forall\ i, 0 \leq i < k : \ (x_{bi+1} \vee x_{bi+2} \vee \ldots \vee x_{bi+b}) \in \alpha_{block}(G,k)$

(c) 1-1 Clauses: For each block, there are clauses saying that at most one of the vertices in it is selected; $\forall\, i,p,q, 0 \leq i < k, 1 \leq p < q \leq b : \ (\neg x_{bi+p} \vee \neg x_{bi+q}) \in \alpha_{block}(G,k)$

$\alpha_{block}(G,k)$ is satisfiable iff G has a block-respecting independent set of size k under the fixed order and partition of vertices implicitly assumed. Note that there is no exact analog of Lemmas 3.2 and 3.3 for the block encoding. In fact, if one fixes the order of vertices and division into blocks is based on this order, then the non-existence of a block-respecting independent set of size k doesn't even logically imply the non-existence of one of size k' for all $k' > k$. This monotonicity, however, holds when $k \mid k'$.

Lemma 3.4. *For any graph G, $k' \geq k$, $k \mid k'$, and $k' \mid n$,*

$$\begin{aligned} \mathsf{RES}(\alpha_{block}(G,k')) &\leq \mathsf{RES}(\alpha_{block}(G,k)) \quad \textit{and} \\ \mathsf{DPLL}(\alpha_{block}(G,k')) &\leq \mathsf{DPLL}(\alpha_{block}(G,k)). \end{aligned}$$

The result holds even when the 1-1 clauses are omitted from both encodings.

Proof. If G contains a block-respecting independent set of size k, then there is no resolution refutation of $\alpha_{block}(G,k)$. By our convention, $Res(\alpha_{block}(G,k)) = DPLL(\alpha_{block}(G,k)) = \infty$, and the result holds. Otherwise consider a refutation π of $\alpha_{block}(G,k)$. The two encodings, $\alpha_{block}(G,k)$ and $\alpha_{block}(G,k')$, are defined over the same set of variables and have identical edge clauses. We will apply a transformation

σ to the variables so that the block and 1-1 clauses of $\alpha_{block}(G,k)$ become a subset of the block and 1-1 clauses of $\alpha_{block}(G,k')$, respectively.

σ works as follows. Each block of vertices in $\alpha_{block}(G,k)$ consists exactly of k'/k blocks of vertices in $\alpha_{block}(G,k')$ because $k \mid k'$. σ sets all but the first n/k' vertices of each block of $\alpha_{block}(G,k)$ to FALSE. This shrinks all block clauses of $\alpha_{block}(G,k)$ to block clauses of $\alpha_{block}(G,k')$. Further, it trivially satisfies all 1-1 clauses of $\alpha_{block}(G,k)$ that are not 1-1 clauses of $\alpha_{block}(G,k')$. Hence $\pi|_\sigma$ is a refutation of $\alpha_{block}(G,k')$ which in fact uses only a subset of the original block and 1-1 clauses of the formula. □

3.2.4 Relationships Among Encodings

For reasonable bounds on the block size, resolution refutations of the block encoding are essentially as efficient as those of the other two encodings. We state the precise relationship in the following lemmas.

Lemma 3.5. *For any graph G over n vertices, $k \mid n$, and $b = n/k$,*

$$\begin{aligned} \mathsf{RES}(\alpha_{block}(G,k)) &\leq b^2\ \mathsf{RES}(\alpha_{count}(G,k)) \quad \textit{and} \\ \mathsf{DPLL}(\alpha_{block}(G,k)) &\leq (2\ \mathsf{DPLL}(\alpha_{count}(G,k)))^{\log_2 2b}. \end{aligned}$$

Proof. Fix a resolution proof π of $\alpha_{count}(G,k)$. We describe a transformation ρ on the underlying variables such that for each initial clause $C \in \alpha_{count}(G,k)$, $C|_\rho$ is either TRUE or an initial clause of $\alpha_{block}(G,k)$. This lets us generate a resolution proof of $\alpha_{block}(G,k)$ from $\pi|_\rho$ of size not much larger than $size(\pi)$. ρ is defined as follows: for each $i \in \{0,1,\ldots,k\}$, set $y_{bi,i} = \text{TRUE}$ and $y_{bi,j} = \text{FALSE}$ for $j \neq i$; set all $y_{v,i} = \text{FALSE}$ if vertex v does not belong to either block $i+1$ or block i; finally, for $1 \leq j \leq b$, replace all occurrences of $y_{bi+j,i+1}$ and $\neg y_{bi+j,i}$ with $(x_{bi+1} \vee x_{bi+2} \vee \ldots \vee x_{bi+j})$, and all occurrences of $\neg y_{bi+j,i+1}$ and $y_{bi+j,i}$ with $(x_{bi+j+1} \vee x_{bi+j+2} \vee \ldots \vee x_{bi+b})$. Note that setting $y_{bi,i} = \text{TRUE}$ for each i logically implies the rest of the transformations stated above.

We first prove that ρ transforms initial clauses of $\alpha_{count}(G,k)$ as claimed. The edge clauses are the same in both encodings. The size-k clause $y_{n,k}$ and the counting clause $y_{0,0}$ of $\alpha_{count}(G,k)$ transform to TRUE. The following can also be easily verified by plugging in the substitutions for the y variables. The counting clauses that define $y_{v,0}$ for $v \geq 1$ are either satisfied or translate into the first block clause $(x_1 \vee \ldots \vee x_b)$. Further, the counting clauses that define $y_{v,i}$ for $v \geq 1, i \geq 1$ are either satisfied or transform into the i^{th} or the $(i+1)^{st}$ block clause, i.e., into $(x_{b(i-1)+1} \vee \ldots \vee x_{b(i-1)+b})$ or $(x_{bi+1} \vee \ldots \vee x_{bi+b})$. Hence, all initial clauses of $\alpha_{count}(G,k)$ are either satisfied or transform into initial clauses of $\alpha_{block}(G,k)$.

We now describe how to generate a valid resolution proof of $\alpha_{block}(G,k)$ from this transformation. Note that the substitutions for $y_{bi+j,i+1}$ and $y_{bi+j,i}$ replace these variables by a disjunction of at most b positive literals. Any resolution step performed

on these y's in the original proof must now be converted into a set of equivalent resolution steps, which will lengthen the transformed refutation. More specifically, a step resolving clauses $(y \vee A)$ and $(\neg y \vee B)$ on the literal y (where y is either $y_{bi+j,i+1}$ or $y_{bi+j,i}$) will now be replaced by a set of resolution steps deriving $(A' \vee B')$ from clauses $(x_{u_1} \vee \ldots \vee x_{u_p} \vee A')$ and $(x_{v_1} \vee \ldots \vee x_{v_q} \vee B')$ and any initial clauses of $\alpha_{block}(G,k)$, where all x's mentioned belong to the same block of G, $\{u_1, \ldots, u_p\}$ is disjoint from $\{v_1, \ldots, v_q\}$, $p+q=b$, and A' and B' correspond to the translated versions of A and B, respectively.

The obvious way of doing this is to resolve the clause $(x_{u_1} \vee \ldots \vee x_{u_p} \vee A')$ with all 1-1 clauses $(\neg x_{u_i} \vee \neg x_{v_1})$ obtaining $(\neg x_{v_1} \vee A')$. Repeating this for all x_{v_j}'s gives us clauses $(\neg x_{v_j} \vee A')$. Note that this reuses $(x_{u_1} \vee \ldots \vee x_{u_p} \vee A')$ q times and is therefore not tree-like. Resolving all $(\neg x_{v_j} \vee A')$ in turn with $(x_{v_1} \vee \ldots \vee x_{v_q} \vee B')$ gives us $(A' \vee B')$. This takes $pq + q < b^2$ steps. Hence the blow-up in size for general resolution is at most a factor of b^2. Note that this procedure is symmetric in A' and B'; we could also have chosen the clause $(\neg y \vee B)$ to start with, in which case we would need $qp + p < b^2$ steps.

The tree-like case is somewhat trickier because we need to replicate clauses that are reused by the above procedure. We handle this using an idea similar to the one used by Clegg et al. [36] for deriving the size-width relationship for tree-like resolution proofs. Let $newSize(s)$ denote the maximum over the sizes of all transformed tree-like proofs obtained from original tree-like proofs of size s by applying the above procedure and creating enough duplicates to take care of reuse. We prove by induction that $newSize(s) \leq (2s)^{\log_2 2b}$. For the base case, $newSize(1) = 1 \leq 2b = 2^{\log_2 2b}$. For the inductive step, consider the subtree of the original proof that derives $(A \vee B)$ by resolving $(y \vee A)$ and $(\neg y \vee B)$ on the literal y as above. Let this subtree be of size $s \geq 2$ and assume without loss of generality that the subtree deriving $(y \vee A)$ is of size $s_A \leq s/2$. By induction, the transformed version of this subtree deriving $(x_{u_1} \vee \ldots \vee x_{u_p} \vee A')$ is of size at most $newSize(s_A)$ and that of the other subtree deriving $(x_{v_1} \vee \ldots \vee x_{v_q} \vee B')$ is of size at most $newSize(s - s_A - 1)$. Choose $(x_{u_1} \vee \ldots x_{u_p} \vee A')$ as the clause to start the new derivation of $(A' \vee B')$ as described in the previous paragraph. The size of this refutation is at most $b \cdot newSize(s_A) + newSize(s - s_A - 1) + b^2$. Since this can be done for any original proof of size s, $newSize(s) \leq b \cdot newSize(s_A) + newSize(s - s_A - 1) + b^2$ for $s \geq 2$ and $s_A \leq s/2$. It can be easily verified that $newSize(s) = 2bs\ b^{\log_2 s} = (2s)^{\log_2 2b}$ is a solution to this. This proves the bound for the DPLL case. □

Lemma 3.6. *For any graph G over n vertices and $k \mid n$,*

$$\begin{aligned} \mathsf{RES}(\alpha_{block}(G,k)) &\leq \mathsf{RES}(\alpha_{map}(G,k)) \quad \textit{and} \\ \mathsf{DPLL}(\alpha_{block}(G,k)) &\leq \mathsf{DPLL}(\alpha_{map}(G,k)). \end{aligned}$$

Proof. In the general encoding $\alpha_{map}(G,k)$, a vertex v can potentially be chosen as the i^{th} node of the k-independent set for *any* $i \in \{1, 2, \ldots, k\}$. In the restricted encoding,

however, vertex v belonging to block j can be thought of as either being selected as the j^{th} node of the independent set or not being selected at all. Hence, if we start with a resolution (or DPLL) refutation of $\alpha_{map}(G,k)$ and set $z_{v,i} = \text{FALSE}$ for $i \neq j$, we get a simplified refutation where the only variables are of the form $z_{v,j}$, where vertex v belongs to block j. Renaming these $z_{v,j}$'s as x_v's, we get a refutation in the variables of $\alpha_{block}(G,k)$ that is no larger in size than the original refutation of $\alpha_{map}(G,k)$.

All we now need to do is verify that for every initial clause of $\alpha_{map}(G,k)$, this transformation either converts it into an initial clause of $\alpha_{block}(G,k)$ or satisfies it. The transformed refutation will then be a refutation of $\alpha_{block}(G,k)$ itself. This reasoning is straightforward:

(a) Edge clauses $(\neg z_{u,i} \vee \neg z_{v,j})$ of $\alpha_{map}(G,k)$ that represented edge $(u,v) \in E$ with u in block i and v in block j transform into the corresponding edge clause $(\neg x_u \vee \neg x_v)$ of $\alpha_{block}(G,k)$. If vertex u (or v) is not in block i (or j, resp.), then the transformation sets $z_{u,i}$ (or $z_{v,j}$, resp.) to FALSE and the clause is trivially satisfied.

(b) Surjective clauses of $\alpha_{map}(G,k)$ clearly transform to the corresponding block clauses of $\alpha_{block}(G,k)$ – for the i^{th} such clause, variables corresponding to vertices that do not belong to block i are set to FALSE and simply vanish, and we are left with the i^{th} block clause of $\alpha_{block}(G,k)$.

(c) It is easy to see that all function clauses and ordering clauses are trivially satisfied by the transformation.

(d) 1-1 clauses $(\neg z_{u,i} \vee \neg z_{v,i})$ of $\alpha_{map}(G,k)$ that involved vertices u and v both from block i transform into the corresponding 1-1 clause $(\neg x_u \vee \neg x_v)$ of $\alpha_{block}(G,k)$. If vertex u (or v) is not in block i, then the transformation sets $z_{u,i}$ (or $z_{v,i}$, resp.) to FALSE and the clause is trivially satisfied.

Thus, this transformed proof is a refutation of $\alpha_{block}(G,k)$ and the desired bounds follow. □

3.3 Simulating Chvátal's Proof System

In this section, we show that resolution on $\alpha_{block}(G,k)$ can efficiently simulate Chvátal's proofs [34] of non-existence of k-independent sets in G. This indirectly provides bounds on the running time of various algorithms for finding a maximum independent set in a given graph. We begin with a brief description of Chvátal's proof system. Let (S,t) for $t \geq 1$ be the *statement* that the subgraph of G induced by a vertex subset S does not have an independent set of size t. $(\phi, 1)$ is given as an axiom and the goal is to derive, using a series of applications of one of two rules, the statement (V,k), where V is the vertex set of G and k is given as input. The two inference rules are

Branching Rule: for any vertex $v \in S$, from statements $(S \setminus N(v), t-1)$ and $(S \setminus \{v\}, t)$ one can infer (S, t), where $N(v)$ is the set containing v and all its neighbors in G;

Monotone Rule: from statement (S, t) one can infer any (S', t') that (S, t) *dominates*, i.e., $S \supseteq S'$ and $t \leq t'$.

For a graph G with vertex set $V(G)$, let $Chv(G, k)$ denote the size of the smallest proof in Chvátal's system of the statement $(V(G), k)$. Following our convention, $Chv(G, k) = \infty$ if no such proof exists. As an immediate application of the monotone rule, we have:

Proposition 3.3. *For $k' > k$, $Chv(G, k') \leq Chv(G, k) + 1$.*

Proposition 3.4. *Let G and G' be graphs with $V(G) = V(G')$ and $E(G) \subseteq E(G')$. For any k, $Chv(G', k) \leq 2 \cdot Chv(G, k)$ and the number of applications of the branching rule in the two shortest proofs is the same.*

Proof. Let π be a proof of $(V(G), k)$ in G. We convert π into a proof π' of $(V(G'), k)$ in G' by translating proof statements in the order in which they appear in π. The axiom statement translates directly without any change. For the derived statements, any application of a monotone inference can be applied equally for both graphs. For an application of the branching rule in π, some (S, t) is derived from $(S \setminus N(v), t-1)$ and $(S \setminus \{v\}, t)$. To derive (S, t) for G', the only difference is the replacement of $(S \setminus N(v), t-1)$ by $(S \setminus N'(v), t-1)$, where $N'(v)$ is the set containing v and all its neighbors in G'. If these two statements are different then since $N'(v) \supseteq N(v)$, the latter follows from the former by a single application of the monotone rule. In total, at most $size(\pi)$ additional inferences are added, implying $size(\pi') \leq 2size(\pi)$. □

The following lemma shows that by traversing the proof graph beginning with the axioms one can locally replace each inference in Chvátal's system by a small number of resolution inferences.

Lemma 3.7. *For any graph G over n vertices and $k \mid n$,*

$$\mathsf{RES}(\alpha_{block}(G, k)) \;\leq\; 4n\ Chv(G, k).$$

Proof. Let V denote the vertex set of G. Arbitrarily partition V into k blocks of equal size. Let G_{block} be the graph obtained by adding to G all edges (u, v) such that vertices u and v belong to the same block of G. In other words, G_{block} is G modified to contain a clique on each block so that every independent set of size k in G_{block} is block-respecting with respect to G. By Proposition 3.4, the shortest proof in Chvátal's system, say π_{Chv}, of (V, k) in G_{block} is at most twice in size as the shortest proof of

(V,k) in G. We will use π_{Chv} to guide the construction of a resolution refutation π_{RES} of $\alpha_{block}(G,k)$ such that $size(\pi_{RES}) \leq 2n\ size(\pi_{Chv})$, proving the desired bound.

Observe that without loss of generality, for any statement (S,t) in π_{Chv}, t is at least the number of blocks of G containing vertices in S. This is so because it is true for the final statement (V,k), and if it is true for (S,t), then it is also true for both $(S \setminus \{v\}, t)$ and $(S \setminus N(v), t-1)$ from which (S,t) is derived. Call (S,t) a *trivial* statement if t is strictly bigger than the number of blocks of G containing vertices in S. The initial statement $(\phi,1)$ of the proof is trivial, whereas the final statement (V,k) is not. Furthermore, all statements derived by applying the monotone rule are trivial.

π_{RES} will have a clause associated with each non-trivial statement (S,t) occurring in π_{Chv}. This clause will be a subclause of the clause $C_S \stackrel{\text{def}}{=} (\bigvee_{u \in N_S} x_u)$, where N_S is the set of all vertices in $V \setminus S$ that are in blocks of G containing at least one vertex of S. π_{RES} will be constructed inductively, using the non-trivial statements of π_{Chv}. Note that the clause associated in this manner with (V,k) will be the empty clause, making π_{RES} a refutation.

Suppose (S,t) is non-trivial and is derived in π_{Chv} by applying the branching rule to vertex $v \in S$. Write the target clause C_S as $(C_S^b \vee C_S^r)$, where C_S^b is the disjunction of all variables corresponding to vertices of N_S that are in the same block as v, and C_S^r is the disjunction of all variables corresponding to vertices of N_S that are in the remaining blocks. Before deriving the desired subclause of C_S, derive two clauses Cl_1 and Cl_2 as follows depending on the properties of the inference that produced (S,t):

Case 1: Both $(S \setminus \{v\}, t)$ and $(S \setminus N(v), t-1)$ are trivial. It is easy to see that since (S,t) is non-trivial, if $(S \setminus \{v\}, t)$ is trivial then v is the only vertex of S in its block. Let Cl_1 be the initial block clause for the block containing v, which is precisely $(x_v \vee C_S^b)$. The fact that $(S \setminus N(v), t-1)$ is also trivial implies that the neighbors of v include not only every vertex of S appearing in the block containing v but also all vertices in $S \cap B$, where B is some other block that does not contain v. Resolving the block clause for block B with all edge clauses $(\neg x_v \vee \neg x_u)$ for $u \in S \cap B$ gives a subclause Cl_2 of $(\neg x_v \vee C_S^r)$.

Case 2: $(S \setminus \{v\}, t)$ is trivial but $(S \setminus N(v), t-1)$ is non-trivial. Set Cl_1 exactly as in case 1. Given that $(S \setminus N(v), t-1)$ is non-trivial, by the inductive assumption the prefix of π_{RES} constructed so far contains a subclause of $C_{S \setminus N(v)}$. Since the given proof applies to G_{block}, $N(v) \cup v$ contains every vertex in the block containing v as well as all neighbors of v in G that are not in v's block. Therefore, the subclause of $C_{S \setminus N(v)}$ we have by induction is a subclause of $(C_S^r \vee x_{u_1} \vee \ldots \vee x_{u_p})$, where each u_i is a neighbor of v in S in blocks other than v's block. Derive a new clause Cl_2 by resolving this clause with all edge clauses $(\neg x_v \vee \neg x_{u_i})$. Observe that Cl_2 is a subclause of $(\neg x_v \vee C_S^r)$.

Case 3: $(S \setminus \{v\}, t)$ is non-trivial but $(S \setminus N(v), t-1)$ is trivial. Set Cl_2 as in case 1. Since $(S \setminus \{v\}, t)$ is non-trivial, by the inductive assumption the prefix of π_{RES}

constructed so far contains a subclause Cl_2 of $C_{S\setminus\{v\}}$, i.e., a subclause of $(x_v \vee C_S)$.

Case 4: Both $(S \setminus \{v\}, t)$ and $(S \setminus N(v), t-1)$ are non-trivial. In this case, derive Cl_1 as in case 3 and Cl_2 as in case 2.

It is easy to verify that Cl_1 is a subclause of $(x_v \vee C_S)$ and Cl_2 is a subclause of $(\neg x_v \vee C_S^r)$. If either Cl_1 or Cl_2 does not mention x_v at all, then we already have the desired subclause of C_S. Otherwise resolve Cl_1 with Cl_2 to get a subclause of C_S. This completes the construction. Given any non-trivial statement in π_{Chv}, it takes at most $2n$ steps to derive the subclause associated with it in the resolution proof, given that we have already derived the corresponding subclauses for the two branches of that statement. Hence, $size(\pi_{RES}) \leq 2n\ size(\pi_{Chv})$. □

It follows that lower bounds on the complexity of α_{block} apply to Chvátal's system and hence also to many algorithms for finding a maximum independent set in a given graph that are captured by his proof system, such as those of Tarjan [105], Tarjan and Trojanowski [106], Jian [67], and Shindo and Tomita [100].

3.4 Relation to Vertex Cover and Coloring

This section discusses how the independent set problem relates to vertex covers and colorings of random graphs in terms of resolution complexity.

3.4.1 Vertex Cover

As for independent sets, for any undirected graph $G = (V, E)$, let $n = |V|$, $m = |E|$, and $\Delta = m/n$. A *t-vertex cover* in G is a set of t vertices that contains at least one endpoint of every edge in G. I is an independent set in G if and only if $V \setminus I$ is a vertex cover of G. Hence, the problem of determining whether or not G has a t-vertex cover is the same as that of determining whether or not it has a k-independent set for $k = n - t$. We use this correspondence to translate our bounds on the resolution complexity of independent sets to those on the resolution complexity of vertex covers.

Consider encoding in clausal form the statement that G has a t-vertex cover. The only defining difference between an independent set and a vertex cover is that the former requires at most one of the endpoints of every edge to be included, where as the latter requires at least one. Natural methods to count remain the same, that is, explicit counting variables, implicit mapping variables, or blocks. Similar to the independent set encoding variables, let $x'_v, 1 \leq v \leq n$, be a set of variables such that $x'_v = \text{TRUE}$ iff vertex v is chosen to be in the vertex cover. Let $y'_{v,i}, 1 \leq v \leq n, 1 \leq i \leq t$, denote the fact that exactly i of the first v vertices are chosen in the vertex cover. Let $z'_{v,i}, 1 \leq v \leq n, 1 \leq i \leq t$, represent that vertex v is mapped to the i^{th} node of the vertex cover.

The *counting encoding* of vertex cover, $VC_{count}(G, t)$, is defined analogous to $\alpha_{count}(G, k)$ except for the change that for an edge $(u, v) \in E$, the edge clause

for vertex cover is $(x'_u \vee x'_v)$ and not $(\neg x'_u \vee \neg x'_v)$. The rest of the encoding is obtained by setting $k \leftarrow t, x_v \leftarrow x'_v, y_{v,i} \leftarrow y'_{v,i}$. The *mapping encoding* of vertex cover, $VC_{mapping}(G,t)$ is similarly defined analogous to $\alpha_{mapping}(G,k)$ by setting $k \leftarrow t, z_{v,i} \leftarrow z'_{v,i}$, except for the change in edge clauses for edges $(u,v) \in E$ from $(\neg z_{u,i} \vee \neg z_{v,i})$ to $(z'_{u,i} \vee z'_{v,i})$. For $b = n/(n-t)$, the *block encoding* of vertex cover over $(n-t)$ blocks of size b each, $VC_{block}(G,t)$, is also defined analogous to $\alpha_{block}(G,k)$ by setting $k \leftarrow (n-t), x_v \leftarrow \neg x'_v$. It says that each edge is covered, and exactly $b-1$ vertices from each block are selected in the vertex cover, for a total of $(n-t)(b-1) = t$ vertices. Note that the 1-1 clauses of α_{block} translate into "all-but-one" clauses of VC_{block}.

It is not surprising that the resolution complexity of various encodings of the vertex cover problem is intimately related to that of the corresponding encodings of the independent set problem. We formalize this in the following lemmas.

Lemma 3.8. *For any graph G over n vertices,*

$$\mathsf{RES}(VC_{count}(G,t)) \;\leq\; \mathsf{RES}(\alpha_{count}(G,n-t)) + 6nt^2.$$

Proof. If G has an independent set of size $n-t$, then there is no resolution refutation of $\alpha_{count}(G,n-t)$. Consequently, $Res(\alpha_{count}(G,n-t)) = DPLL(\alpha_{count}(G,n-t)) = \infty$, trivially satisfying the claimed inequalities. Otherwise, consider a refutation π of $\alpha_{count}(G,n-t)$. We use π to construct a refutation π' of $VC_{count}(G,t)$ that is not too big.

Recall that the variables of π are $x_u, 1 \leq u \leq n$, and $y_{v,i}, 0 \leq i \leq v \leq n, 0 \leq i \leq n-t$. The variables of π' will be $x'_u, 1 \leq u \leq n$, and $y'_{v,i}, 0 \leq i \leq v \leq n, 0 \leq i \leq t$. Notice that the number of independent set counting variables $y_{v,i}$ is not the same as the number of vertex cover counting variables $y'_{v,i}$. We handle this by adding dummy counting variables, transforming π, and removing extra variables. To obtain π', apply transforms σ_1, σ_2 and σ_3 defined below to π.

σ_1 simply creates new counting variables $y_{v,i}, 0 \leq v \leq n, (n-t+1) \leq i \leq v$, and adds counting clauses corresponding to these variables as unused initial clauses of π. σ_2 sets $x_u \leftarrow \neg x'_u, y_{v,i} \leftarrow y'_{v,v-i}$. Intuitively, σ_2 says that i of the first v vertices being in the independent set is equivalent to exactly $v-i$ of the first v vertices being in the vertex cover. σ_3 sets $y'_{v,i} \leftarrow$ FALSE for $0 \leq v \leq n, (t+1) \leq i \leq v$. Since σ_1, σ_2 and σ_3 only add new clauses, rename literals or set variables, their application transforms π into another, potentially simpler, refutation on a different set of variables and initial clauses. Call the resulting refutation π''.

The initial edge clauses $(\neg x_u \vee \neg x_v)$ of π transform into edge clauses $(x'_u \vee x'_v)$ of $VC_{count}(G,t)$. The initial size-$(n-t)$ clause of π transforms into the initial size-t clause of $VC_{count}(G,t)$. Finally, the initial counting clauses of π, including those corresponding to the variables added by σ_1, transform into counting clauses of $VC_{count}(G,t)$ and n extra clauses. To see this, note that σ_2 transforms counting formulas $y_{0,0}$

into $y'_{0,0}$, $(y_{v,0} \leftrightarrow (y_{v-1,0} \wedge \neg x_v))$ into $(y'_{v,v} \leftrightarrow (y'_{v-1,v-1} \wedge x'_v))$, for $i \geq 1 : (y_{v,i} \leftrightarrow ((y_{v-1,i} \wedge \neg x_v) \vee (y_{v-1,i-1} \wedge x_v)))$ into $(y'_{v,v-i} \leftrightarrow ((y'_{v-1,v-i-1} \wedge x'_v) \vee (y'_{v-1,v-i} \wedge \neg x'_v)))$, and $(y_{v,v} \leftrightarrow (y_{v-1,v-1} \wedge x_v))$ into $(y'_{v,0} \leftrightarrow (y_{v-1,0} \wedge \neg x'_v))$. Applying σ_3 to set $y'_{v,i} \leftarrow$ FALSE for $(t+1) \leq i \leq v$ removes all but the initial counting clauses of $VC_{count}(G,t)$ and the counting formulas corresponding to the variables $y'_{v,t+1}, t+1 \leq v \leq n$, that simplify to $(\neg y'_{v-1,t} \vee \neg x'_v)$. Call this extra set of $n-t$ clauses $Bdry(G,t)$, or *boundary* clauses for (G,t).

At this stage, we have a refutation π'' of size at most $size(\pi)$ starting from clauses $VC_{count}(G,t) \cup Bdry(G,t)$. The boundary clauses together say that no more than t vertices are chosen in the vertex cover. This, however, is implied by the rest of the initial clauses. Using this fact, we first give a derivation π_{Bdry} of every boundary clause starting from the clauses of $VC_{count}(G,t)$. Appending π'' to π_{Bdry} gives a refutation π' of $VC_{count}(G,t)$.

Let $S_i = \bigvee_{i'=0}^{\min\{i,t\}} y'_{n-i,t-i'}$ for $0 \leq i \leq n-t$. Let $R_{v,i,j} = (\neg y'_{v,i} \vee \neg y'_{v,j})$ for $0 \leq i < j \leq v \leq n$ and $j \leq t$. We first give a derivation of these S and R clauses, and then say how to derive the boundary clauses from these. $S_0 = y'_{n,t}$ is an initial clause, and $S_i, i \geq 1$, is obtained by sequentially resolving S_{i-1} with the counting clauses $(\neg y'_{n-i+1,t-i'} \vee y'_{n-i,t-i'} \vee y'_{n-i,t-i'-1})$ for $0 \leq i' < \min\{i,t\}$. Similarly, when $i = 0$, $R_{v,0,v}$ is derived by resolving counting clauses $(\neg y'_{v,0} \vee \neg x'_v)$ and $(\neg y'_{v,v} \vee x'_v)$ on x'_v, clauses $R_{v,0,j}$ for $0 < j < v$ are derived by sequentially resolving $R_{v-1,0,j}$ with the counting clauses $(\neg y'_{v,j} \vee y'_{v-1,j} \vee x'_v)$ and $(\neg y'_{v,0} \vee \neg x'_v)$. Note that $R_{v,0,v}$ and $R_{v,0,j}$ are defined and derived only when $j \leq t$. When $i > 0$, $R_{v,i,v}$ is derived by sequentially resolving $R_{v-1,i-1,v-1}$ with the counting clauses $(\neg y'_{v,i} \vee y'_{v-1,i-1} \vee \neg x'_v)$ and $(\neg y'_{v,v} \vee y'_{v-1,v-1} \vee \neg x'_v)$, and resolving the result on x'_v with the counting clause $(\neg y'_{v,v} \vee x'_v)$. Finally, $R_{v,i,j}$ for $j < v$ is derived by resolving $R_{v-1,i,j}$ with the counting clauses $(\neg y'_{v,i} \vee y'_{v-1,i} \vee x'_v)$ and $(\neg y'_{v,j} \vee y'_{v-1,j} \vee x'_v)$, resolving $R_{v-1.i-1.j-1}$ with the counting clauses $(\neg y'_{v,i} \vee y'_{v-1,i-1} \vee \neg x'_v)$ and $(\neg y'_{v,j} \vee y'_{v-1,j-1} \vee \neg x'_v)$, and resolving the result of the two on x'_v.

To derive the boundary clause $(\neg y'_{v-1,t} \vee \neg x'_v)$ for any v, resolve each pair of clauses $(\neg y'_{v,t-i'} \vee y'_{v-1,t-i'-1} \vee \neg x'_v)$ and $R_{v-1,t-i'-1,t}$ for $0 \leq i' \leq \min\{n-v,t\}$, and resolve all resulting clauses with S_{n-v}. Note that when $\min\{n-v,t\} = t$, there is no $R_{v-1,t-i'-1,t}$, but the corresponding counting clause itself is of the desired form, $(\neg y'_{v,0} \vee \neg x'_v)$. This finishes the derivation π_{Bdry} of all clauses in $Bdry(G,t)$. As stated before, appending π'' to π_{Bdry} gives a refutation π' of $VC_{count}(G,t)$.

For general resolution, $size(\pi') = size(\pi'') + size(\pi_{Bdry}) \leq size(\pi) + size(\pi_{Bdry})$. Each S_i in π_{Bdry}, starting with $i = 0$, is derived in $\min\{i,t\}$ resolution steps from previous clauses, and each $R_{v,i,j}$, starting with $i = 0, v = j = 1$, requires at most 5 resolution steps from previous clauses. Hence, $size(\pi_{Bdry}) \leq nt + 5nt^2 \leq 6nt^2$ for large enough n, implying that $size(\pi') \leq size(\pi) + 6nt^2$. Note that this approach doesn't quite work for tree-like resolution proofs because π_{Bdry} itself becomes exponential in size due to the heavy reuse of clauses involved in the derivation of the $R_{v,i,j}$'s. $\square$

Given that the encodings $\alpha_{count}(G, n-t)$ and $VC_{count}(G, t)$ are duals of each other, the argument made for the Lemma above can also be made the other way, immediately giving us the following reverse result:

Lemma 3.9. *For any graph G over n vertices,*

$$\mathsf{RES}(\alpha_{count}(G,k)) \;\leq\; \mathsf{RES}(VC_{count}(G,n-k)) + 6nk^2.$$

Lemma 3.10. *For any graph G over n vertices and $(n-t) \mid n$,*

$$\begin{aligned} \mathsf{RES}(VC_{block}(G,t)) &= \mathsf{RES}(\alpha_{block}(G,n-t)) \quad \textit{and} \\ \mathsf{DPLL}(VC_{block}(G,t)) &= \mathsf{DPLL}(\alpha_{block}(G,n-t)). \end{aligned}$$

This result also holds without the 1-1 clauses of α_{block} and the corresponding all-but-one clauses of VC_{block}.

Proof. If G has an independent set of size $n-t$, then it also has a vertex cover of size t. In this case, there are no resolution refutations of $VC_{block}(G,t)$ or $\alpha_{block}(G,n-t)$, making the resolution complexity of both infinite and trivially satisfying the claim.

Otherwise, consider a refutation π of $\alpha_{block}(G,n-t)$. We use π to construct a refutation π' of $VC_{block}(G,t)$, which is of the same size and is tree-like if π is. π' is obtained from π by simply applying the transformation $x_v \leftarrow \neg x'_v, 1 \leq v \leq n$. Since this is only a 1-1 mapping between literals, π' is a legal refutation of size exactly $size(\pi)$. All that remains to argue is that the initial clauses of π' are the clauses of $VC_{block}(G,t)$. This, however, follows immediately from the definition of $VC_{block}(G,t)$.

Given the duality of the encodings $VC_{block}(G,t)$ and $\alpha_{block}(G,n-t)$, we can repeat the argument above to translate any refutation of the former into one of the latter. Combining this with the above, the resolution complexity of the two formulas is exactly the same. □

3.4.2 Coloring

A *K-coloring* of a graph $G = (V,E)$ is a function $col : V \to \{1,2,\ldots,K\}$ such that for every edge $(u,v) \in E$, $col(u) \neq col(v)$. For a random graph G chosen from a distribution similar to $\mathbb{G}(n,p)$, the resolution complexity of the formula $\chi(G,K)$ saying that G is K-colorable has been addressed by Beame et al. [14].

Suppose G is K-colorable. Fix a K-coloring col of G and partition the vertices into color classes $V_i, 1 \leq i \leq K$, where $V_i = \{v \in V : col(v) = i\}$. Each color class, by definition, must be an independent set, with the largest of size at least n/K. Thus, non-existence of a $k \stackrel{\text{def}}{=} n/K$ size independent set in G implies the non-existence of a K-coloring of G.

Let $\alpha(G,k)$ be an encoding of the k-independent set problem on graph G. The correspondence above can be used to translate properly encoded resolution proofs of

$\alpha(G,k)$ into those of $\chi(G,K)$. A lower bound on $\mathsf{RES}(\chi(G,K))$, such as the one in [14], would then imply a lower bound on $\mathsf{RES}(\alpha(G,k))$. However, such a translation between proofs must involve a resolution counting argument showing that K sets of vertices, each of size less than n/K, cannot cover all n vertices. This argument itself is at least as hard as PHP^n_{n-K}, the (weak) pigeonhole principle on n pigeons and $n-K$ holes, for which exponential lower bound has been shown by Raz [94]. This makes any translation of a proof of $\alpha(G,k)$ into one of $\chi(G,K)$ necessarily large, ruling out any interesting lower bound for independent sets as a consequence of [14].

On the other hand, non-existence of a K-coloring does not imply the non-existence of a k-independent set. In fact, there are very simple graphs with no K-coloring but with an independent set as large as $n-K$ (e.g. a clique of size $K+1$ along with $n-K-1$ nodes of degree zero). Consequently, our lower bounds for independent sets do not give any interesting lower bounds for K-coloring.

3.5 Upper Bounds

Based on a very simple exhaustive backtracking strategy, we give upper bounds on the DPLL (and hence resolution) complexity of the independent set and vertex cover encodings we have considered.

Lemma 3.11. *There is a constant C_0 such that if G is a graph over n vertices with no independent set of size k, then*

$$\mathsf{DPLL}(\alpha_{map}(G,k)) \;\leq\; 2^{C_0 k \log(ne/k)}.$$

This bound also holds when $\alpha_{map}(G,k)$ does not include 1-1 clauses.

Proof. A straightforward way to disprove the existence of a k-independent set is to go through all $\binom{n}{k}$ subsets of vertices of size k and use as evidence an edge from each subset. We use this strategy to construct a refutation of $\alpha_{map}(G,k)$.

To begin with, apply transitivity to derive all ordering clauses of the form $(\neg z_{u,j} \vee \neg z_{v,i})$ for $u < v$ and $i < j$. If $j = i+1$, this is simply one of the original ordering clauses. For $j = i+2$, derive the new clause $(\neg z_{u,i+2} \vee \neg z_{v,i})$ as follows. Consider any $w \in \{1,2,\ldots,n\}$. If $u < w$, we have the ordering clause $(\neg z_{w,i+1} \vee \neg z_{u,i+2})$, and if $u \geq w$, then $v > w$ and we have the ordering clause $(\neg z_{v,i} \vee \neg z_{w,i+1})$. Resolving these n ordering clauses (one for each w) with the surjective clause $(z_{1,i+1} \vee \ldots \vee z_{n,i+1})$ gives the new ordering clause $(\neg z_{u,i+2} \vee \neg z_{v,i})$ associated with u and v. This clearly requires only n steps and can be done for all $u < v$ and $j = i+2$. Continue to apply this argument for $j = i+3, i+4, \ldots, k$ and derive all new ordering clauses in n steps each.

We now construct a tree-like refutation starting with the initial clauses and the new ordering clauses we derived above. We claim that for any $i \in \{1,2,\ldots,k\}$ and for any $1 \leq v_i < v_{i+1} < \ldots < v_k \leq n$, a subclause of $(\neg z_{v_i,i} \vee \neg z_{v_{i+1},i+1} \vee \ldots \vee \neg z_{v_k,k})$

can be derived. We first argue why this claim is sufficient to obtain a refutation. For $i = k$, the claim says that a subclause of $\neg z_{v_k,k}$ can be derived for all $1 \leq v_k \leq n$. If any one of these n subclauses is a strict subclause of $\neg z_{v_k,k}$, it has to be the empty clause, resulting in a refutation. Otherwise, we have $\neg z_{v_k,k}$ for every v_k. Resolving all these with the surjective clause $(z_{1,k} \vee \ldots \vee z_{n,k})$ results in the empty clause.

We now prove the claim by induction on i. For the base case, fix $i = 1$. For any given k vertices $v_1 < v_2 < \ldots < v_k$, choose an edge (v_p, v_q) that witnesses the fact that these k vertices do not form an independent set. The corresponding edge clause $(\neg z_{v_p,p} \vee \neg z_{v_q,q})$ works as the required subclause.

For the inductive step, fix $v_{i+1} < v_{i+2} < \ldots < v_k$. We will derive a subclause of $(\neg z_{v_{i+1},i+1} \vee \neg z_{v_{i+2},i+2} \vee \ldots \neg z_{v_k,k})$. By induction, derive a subclause of $(\neg z_{v_i,i} \vee \neg z_{v_{i+1},i+1} \vee \ldots \vee z_{v_k,k})$ for any choice of $v_i < v_{i+1}$. If for some such v_i, $\neg z_{v_i,i}$ does not appear in the corresponding subclause, then the same subclause works here for the inductive step and we are done. Otherwise, for every $v_i < v_{i+1}$, we have a subclause of $(\neg z_{v_i,i} \vee \neg z_{v_{i+1},i+1} \vee \ldots \vee \neg z_{v_k,k})$ that contains $\neg z_{v_i,i}$. Resolving all these subclauses with the surjective clause $(z_{1,i} \vee z_{2,i} \vee \ldots \vee z_{n,i})$ results in the clause $(z_{v_{i+1},i} \vee \ldots \vee z_{v_k,i} \vee \neg z_{u_1,j_1} \vee \ldots \vee \neg z_{u_p,j_p})$, where each z_{u_c,j_c} lies in $\{z_{v_{i+1},i+1}, \ldots, z_{v_k,k}\}$. Observe that for each positive literal $z_{v_q,i}, i+1 \leq q \leq k$, in this clause, $(\neg z_{v_q,i} \vee \neg z_{v_{i+1},i+1})$ is either a 1-1 clause or an ordering clause. Resolving with all these clauses finally gives $(\neg z_{v_{i+1},i+1} \vee \neg z_{u_1,j_1} \vee \ldots \vee \neg z_{u_p,j_p})$, which is the kind of subclause we wanted to derive. This proves the claim.

Associate each subclause obtained using the iterative procedure above with the tuple $(v_i, v_{i+1}, \ldots, v_k)$ for which it was derived, giving a total of $\sum_{i=1}^{k} \binom{n}{i} \leq (ne/k)^k$ subclauses. Each of these subclauses is used at most once in the proof. Further, the derivation of each such subclause uses at most n new ordering clauses, each of which can be derived in at most n^2 steps. Thus, with enough copies to make the refutation tree-like, the size of the proof is $O(n^3(ne/k)^k)$, which is at most $2^{C_0 k \log(ne/k)}$ for a large enough constant C_0. □

Lemma 3.12. *There is a constant C_0' such that if G is graph over n vertices with no independent set of size k, then*

$$\mathsf{DPLL}(\alpha_{count}(G,k)) \quad \leq \quad 2^{C_0' k \log(ne/k)}.$$

Proof. As in the proof of Lemma 3.11, we construct a refutation by looking at each size k subset of vertices and using as evidence an edge from that subset.

For every i, v such that $0 \leq i \leq v < n$, first derive a new counting clause $(\neg y_{v+1,i+1} \vee y_{v,i} \vee y_{v-1,i} \vee \ldots \vee y_{i,i})$ by resolving original counting clauses $(\neg y_{u+1,i+1} \vee y_{u,i+1} \vee y_{u,i})$ for $u = v, v-1, \ldots, i+1$ together, and resolving the result with the counting clause $(\neg y_{i+1,i+1} \vee y_{i,i})$. Next, for any edge (i,j), $i > j$, resolve the edge clause $(\neg x_i \vee \neg x_j)$ with the counting clauses $(\neg y_{i,i} \vee x_i)$ and $(\neg y_{j,j} \vee x_j)$ to get the clause $(\neg y_{i,i} \vee \neg y_{j,j})$. Call this new clause $E_{i,j}$. We now construct a tree-like refutation using the initial clauses, these new counting clauses, and the new $E_{i,j}$ clauses.

We claim that for any $i \in \{1, 2, \ldots, k\}$ and for any $1 \leq v_i < v_{i+1} < \ldots < v_k \leq n$ with $v_j \geq j$ for $i \leq j \leq k$, we can derive a subclause of $(\neg y_{v_i,i} \vee y_{v_i-1,i} \vee \neg y_{v_{i+1},i+1} \vee y_{v_{i+1}-1,i+1} \vee \ldots \vee \neg y_{v_k,k} \vee y_{v_k-1,k})$ such that if $y_{v_{j-1},j}$ occurs in the subclause for some j, then so does $\neg y_{v_j,j}$. Note that for $v_j = j$, the variable $y_{v_j-1,j}$ does not even exist and will certainly not appear in the subclause. Given this claim, we can derive for $i = k$ a subclause B_j of $(\neg y_{j,k} \vee y_{j-1,k})$ for each $j \in \{k+1, \ldots, n\}$ and a subclause B_k of $\neg y_{k,k}$. If any of these B_j's is the empty clause, the refutation is complete. Otherwise every B_j contains $\neg y_{j,k}$. Let j' be the largest index such that $B_{j'}$ does not contain $y_{j'-1,k}$. Since B_k has to be the clause $\neg y_{k,k}$, such a j' must exist. Resolving all B_j's for $j \in \{j', \ldots, k\}$ with each other gives the clause $y_{n,k}$. Resolving this with the size-k clause $y_{n,k}$ gives the empty clause.

We now prove that the claim holds by induction on i. For the base case $i = 1$, fix $1 \leq v_1 < v_2 < \ldots < v_k \leq n$. Choose an edge (v_p, v_q) that witnesses the fact that these v_i's do not form an independent set. Resolve the corresponding edge clause $(\neg x_{v_p} \vee \neg x_{v_q})$ with the counting clauses $(\neg y_{v_p,p} \vee y_{v_p-1,p} \vee x_p)$ and $(\neg y_{v_q,q} \vee y_{v_q-1,q} \vee x_q)$ to get $(\neg y_{v_p,p} \vee y_{v_p-1,p} \vee \neg y_{v_q,q} \vee y_{v_q-1,q})$, which is a subclause of the desired form.

For the inductive step, fix $v_{i+1} < v_{i+2} < \ldots < v_k$. By induction, derive a subclause C_j of $(\neg y_{j,i} \vee y_{j-1,i} \vee \neg y_{v_{i+1},i+1} \vee y_{v_{i+1}-1,i+1} \vee \ldots \vee \neg y_{v_k,k} \vee y_{v_k-1,k})$ for any j in $\{i, i+1, \ldots, v_{i+1}-1\}$. If for some such j, neither $\neg y_{j,i}$ nor $y_{j-1,i}$ appears in C_j, then this subclause also works here for the inductive step and we are done. Otherwise for every j, C_j definitely contains $\neg y_{j,i}$, possibly $y_{j-1,i}$, and other positive or negative occurrences of variables of the form $y_{v',i'}$ where $i' > i$. Now use these C_j's to derive clauses C'_j's such that C'_j contains $\neg y_{j,i}$ but not $y_{j-1,i}$. The other variables appearing in C'_j will all be of the form $y_{v',i'}$ for $i' > i$.

If $\{v_{i+1}, \ldots, v_k\}$ is not an independent set, then there is an edge (v_p, v_q) witnessing this. In this case, simply use $E_{p,q}$ as the desired subclause and the inductive step is over. Otherwise there must be an edge (i, v_q) from vertex i touching this set. Let C'_i be the clause E_{i,v_q}. For j going from $i+1$ to k, do the following iteratively. If $y_{j-1,i}$ does not appear in C_j, then set $C'_j = C_j$. Otherwise set C'_j to be the clause obtained by resolving C_j with C'_{j-1}. If C'_{j-1} does not contain $\neg y_{j,i}$, then it can be used as the desired subclause for this inductive step and the iteration is stopped here, otherwise it continues onto the next value of j. If desired subclause is not derived somewhere along this iterative process, then we end up with all C'_j's containing $\neg y_{j,i}$ but not $y_{j-1,i}$. Resolving all these with the new counting clause $(\neg y_{v_{i+1},i+1} \vee y_{v_{i+1}-1,i} \vee y_{v_{i+1}-2,i} \vee \ldots \vee y_{i,i})$ finally gives a subclause of the desired form. This proves the claim.

Associate each subclause obtained using the iterative procedure above with the tuple $(v_i, v_{i+1}, \ldots, v_k)$ for which it was derived, giving a total of $\sum_{i=1}^{k} \binom{n}{i} \leq (ne/k)^k$ subclauses. Each of these subclauses is used at most once in the proof. Further, the derivation of each such subclause uses one new counting clause and one new clause $E_{i,j}$, each of which can be derived in at most n steps. Thus, with enough copies to

make the refutation tree-like, the size of the proof is $O(n(ne/k)^k)$, which is at most $2^{C_0' k \log(ne/k)}$ for a large enough constant C_0'. □

Theorem 3.1 (Independent Set Upper Bounds). *There are constants c_0, c_0' such that the following holds. Let $\Delta = np$, $\Delta \leq n/\log^2 n$, and $G \sim \mathbb{G}(n,p)$. Let k be such that G has no independent set of size k. With probability $1 - o(1)$ in n,*

$$\begin{aligned} \mathsf{DPLL}(\alpha_{map}(G,k)) &\leq 2^{c_0 (n/\Delta)\log^2 \Delta}, \\ \mathsf{DPLL}(\alpha_{count}(G,k)) &\leq 2^{c_0' (n/\Delta)\log^2 \Delta}, \quad \textit{and} \\ \mathsf{DPLL}(\alpha_{block}(G,k)) &\leq 2^{c_0 (n/\Delta)\log^2 \Delta}. \end{aligned}$$

The bounds also holds when 1-1 clauses are removed from $\alpha_{map}(G,k)$ or $\alpha_{block}(G,k)$. The block encoding bound holds when $k \mid n$.

Proof. By Proposition 3.1, $n/(\Delta+1) < k \leq n$. Hence $k\log(ne/k) \leq n\log(e(\Delta+1))$. We will use this fact when Δ is a relatively small constant.

Fix any $\epsilon > 0$ and let C_ϵ be the corresponding constant from Proposition 3.2. When $\Delta < C_\epsilon$, the desired upper bounds in this theorem are of the form $2^{O(n)}$. Moreover, the upper bounds provided by Lemmas 3.11 and 3.12 for the mapping and counting encodings, respectively, are exponential in $k\log(ne/k) \leq n\log(e(\Delta+1))$, and thus also of the form $2^{O(n)}$ when $\Delta < C_\epsilon$. Hence, for large enough constants c_0 and c_0', the claimed bounds hold with probability 1 for the mapping and counting encodings when $\Delta < C_\epsilon$. Lemma 3.6 extends this to the block encoding as well.

Assume for the rest of this proof that $C_\epsilon \leq \Delta \leq n/\log^2 n$. Let $k_{min} \leq k$ be the smallest integer such that G does not have an independent set of size k_{min}. By Proposition 3.2, with probability $1 - o(1)$ in n, $k_{min} \leq k_{+\epsilon} + 1$.

For the mapping-based encoding,

$$\begin{aligned} \mathsf{DPLL}(\alpha_{map}(G,k)) &\leq \mathsf{DPLL}(\alpha_{map}(G,k_{min})) && \text{by Lemma 3.3} \\ &\leq 2^{C_0 k_{min}\log(n/k_{min})} && \text{by Lemma 3.11} \\ &\leq 2^{C_0(k_{+\epsilon}+1)\log(n/(k_{+\epsilon}+1))} && \text{almost surely} \\ &\leq 2^{(c_0 n/\Delta)\log^2\Delta} && \text{for large enough } c_0. \end{aligned}$$

The bound for $\alpha_{block}(G,k)$ follows immediately from this bound for $\alpha_{map}(G,k)$ and Lemma 3.6. Further, Lemma 3.11 implies that these bounds hold even when the corresponding 1-1 clauses are removed from the mapping and block encodings. For

the counting-based encoding,

$$\begin{aligned}
\mathsf{DPLL}(\alpha_{count}(G,k) &\leq n\ \mathsf{DPLL}(\alpha_{count}(G,k_{min}) + 2n^2 && \text{by Lemma 3.2}\\
&\leq n2^{C_0' k_{min}\log(n/k_{min})} + 2n^2 && \text{by Lemma 3.12}\\
&\leq n2^{C_0' k_{min}\log(n/k_{min})} + 2n^2 &&\\
&\leq n2^{C_0'(k_{+\epsilon}+1)\log(n/(k_{+\epsilon}+1))} + 2n^2 && \text{almost surely}\\
&\leq 2^{(c_0' n/\Delta)\log^2\Delta} && \text{for a large enough constant } c_0'.
\end{aligned}$$

This finishes the proof. □

Corollary 3.1 (Vertex Cover Upper Bounds). *There are constants c_0, c_0'' such that the following holds. Let $\Delta = np$, $\Delta \leq n/\log^2 n$, and $G \sim \mathbb{G}(n,p)$. Let t be such that G has no vertex cover of size t. With probability $1-o(1)$ in n,*

$$\begin{aligned}
\mathsf{RES}(VC_{count}(G,t)) &\leq 2^{c_0''(n/\Delta)\log^2\Delta}, \quad \textit{and}\\
\mathsf{DPLL}(VC_{block}(G,t)) &\leq 2^{c_0(n/\Delta)\log^2\Delta}.
\end{aligned}$$

The bounds also holds when all-but-one clauses are removed from $VC_{block}(G,t)$. The block encoding bound holds when $(n-t)\,|\,n$.

Proof. Apply Theorem 3.1 with k set to $n-t$ and use Lemmas 3.8 and 3.10 to translate the result of the Theorem to encodings of vertex cover. Note that $\mathsf{RES}(\alpha_{count}(G, n-t)) \leq \mathsf{DPLL}(\alpha_{count}(G, n-t))$. □

3.6 Key Concepts for Lower Bounds

This section defines key concepts that will be used in the lower bound argument given in the next section. Fix a graph G and a partition of its n vertices into k subsets of size b each. For any edge (u,v) in G, call it an *inter-block edge* if u and v belong to different blocks of G, and an *intra-block edge* otherwise.

Definition 3.1. A truth assignment to variables of $\alpha_{block}(G,k)$ is *critical* if it sets exactly one variable in each block to TRUE.

Critical truth assignments satisfy all block, 1-1 and intra-block edge clauses but may leave some inter-block edge clauses unsatisfied.

Definition 3.2. The block multi-graph of G, denoted $B(G)$, is the multi-graph obtained from G by identifying all vertices that belong to the same block and removing any self-loops that are thus generated.

$B(G)$ contains exactly k nodes and possibly multiple edges between pairs of nodes. The degree of a node in $B(G)$ is the number of inter-block edges touching the corresponding block of G. Given the natural correspondence between G and $B(G)$, we will write *nodes of* $B(G)$ and *blocks of* G interchangeably. For a subgraph H of G, $B(H)$ is obtained analogously by identifying all vertices of H that are in the same block of G and removing self-loops.

Definition 3.3. Let S be a set of blocks of G. H is *block induced by* S if it is the subgraph of G induced by all vertices present in the blocks S. H is a *block induced subgraph* of G if there exists a subset S of blocks such that H is block induced by S.

If H is block induced by S, then $B(H)$ is induced by S in $B(G)$. The reverse, however, may not be true. If H is a block induced subgraph, then there is a *unique* minimal block set S such that H is block induced by S. This S contains exactly those blocks that have non-zero degree in $B(H)$. With each block induced subgraph, associate such a *minimal* S and say that the subgraph is induced by $|S|$ blocks. Note that every block in any such minimal S must have non-zero degree.

Definition 3.4. The *block width* of a clause C with respect to G, denoted $w^G_{block}(C)$, is the number of different blocks of G the variables appearing in C come from.

Clearly, $w(C) \geq w^G_{block}(C)$. For a block induced subgraph H of G, let $E(H)$ denote the conjunction of the edge clauses of $\alpha_{block}(G,k)$ that correspond to the edges of H. Let H be induced by the block set S.

Definition 3.5. H *critically implies* a clause C, denoted $H \xrightarrow{c} C$, if $E(H) \rightarrow C$ evaluates to true for all critical truth assignments to the variables of $\alpha_{block}(G,k)$.

Definition 3.6. H *minimally implies* C, denoted $H \xrightarrow{m} C$, if $H \xrightarrow{c} C$ and for every subgraph H' of G induced by a proper subset of S, $H' \not\xrightarrow{c} C$.

Note that "minimally implies" should really be called "minimally critically implies," but we use the former phrase for brevity. Note further that if $H \xrightarrow{m} C$, then every block of H has non-zero degree.

Definition 3.7. The *complexity* of a clause C, denoted $\mu_G(C)$, is the minimum over the sizes of subsets S of blocks of G such that the subgraph of G induced by S critically implies C.

Proposition 3.5. *Let G be a graph and Λ denote the empty clause.*

(a) For $C \in \alpha_{block}(G,k)$, $\mu_G(C) \leq 2$.

(b) $\mu_G(\Lambda)$ is the number of blocks in the smallest block induced subgraph of G that has no block-respecting independent set.

(c) Subadditive property*: If clause C is a resolvent of clauses C_1 and C_2, then* $\mu_G(C) \leq \mu_G(C_1) + \mu_G(C_2)$.

Proof. Each initial clause is either an edge clause, a block clause or a 1-1 clause. Any critical truth assignment, by definition, satisfies all block, 1-1 and intra-block edge clauses. Further, an edge clause corresponding to an inter-block edge (u, v) is implied by the subgraph induced by the two blocks to which u and v belong. Hence, complexity of an initial clause is at most 2, proving part (a).

Part (b) follows from the definition of μ_G. Part (c) follows from the simple observation that if G_1 critically implies C_1, G_2 critically implies C_2, and both G_1 and G_2 are block induced subgraphs, then $G_{1\cup 2}$, defined as the block graph induced by the union of the blocks G_1 and G_2 are induced by, critically implies both C_1 and C_2, and hence critically implies C. □

3.7 Proof Sizes and Graph Expansion

This section contains the main ingredients of our lower bound results and is technically the most interesting and challenging part at the core of this chapter. We use combinatorial properties of block graphs and independent sets to obtain a lower bound on the size of resolution refutations for a given graph in terms of its expansion properties. Next, we argue that random graphs almost surely have good expansion properties. Section 3.8 combines these two to obtain an almost certain lower bound for random graphs.

The overall argument in a little more details is as follows. We define the notion of "boundary" for block induced subgraphs as a measure of the number of blocks in it that have an isolated vertex and thus contribute trivially to any block-respecting independent set. Lemmas 3.13 and 3.14 relate this graph-theoretic concept to resolution refutations. The main lower bound follows in three steps from here. First, Lemma 3.16 argues that one must almost surely consider a large fraction of the blocks of a graph to prove the non-existence of a block-respecting independent set in it. Second, Lemma 3.17 shows that almost all subgraphs induced by large fractions of blocks must have large boundary. Finally, Lemma 3.18 combines these two to obtain an almost certain lower bound on the width of any refutation.

We begin by defining the notion of boundary.

Definition 3.8. The *boundary* of a block induced subgraph H, denoted $\beta(H)$, is the set of blocks of H that have at least one isolated vertex.

3.7.1 Relating Proof Size to Graph Expansion

We first derive a relationship between the width of clauses and the boundary size of block-induced subgraphs that minimally imply them.

Lemma 3.13. *Let C be a clause in the variables of $\alpha_{block}(G, k)$ and H be a block induced subgraph of G. If $H \xrightarrow{m} C$, then $w^G_{block}(C) \geq |\beta(H)|$.*

Proof. We use a *toggling property* of block-respecting independent sets (Figure 3.2) to show that each boundary block of H contributes at least one literal to C.

Let H be induced by the set of blocks S. Fix a boundary block $B \in S$. Let H_B be the subgraph induced by $S \setminus \{B\}$. By minimality of H, $H_B \overset{c}{\not\to} C$. Therefore, there exists a critical truth assignment γ such that $\gamma(E(H_B)) = \text{TRUE}$ but $\gamma(C) = \text{FALSE}$. Since $\gamma(C) = \text{FALSE}$ and $H \xrightarrow{c} C$, it follows that $\gamma(E(H)) = \text{FALSE}$. Further, since $\gamma(E(H_B)) = \text{TRUE}$, $\gamma(E(H) \setminus E(H_B))$ must be FALSE, implying that γ violates the edge clause corresponding to an inter-block edge $(v, w), v \in B, w \notin B$. In particular, $\gamma(v) = \text{TRUE}$.

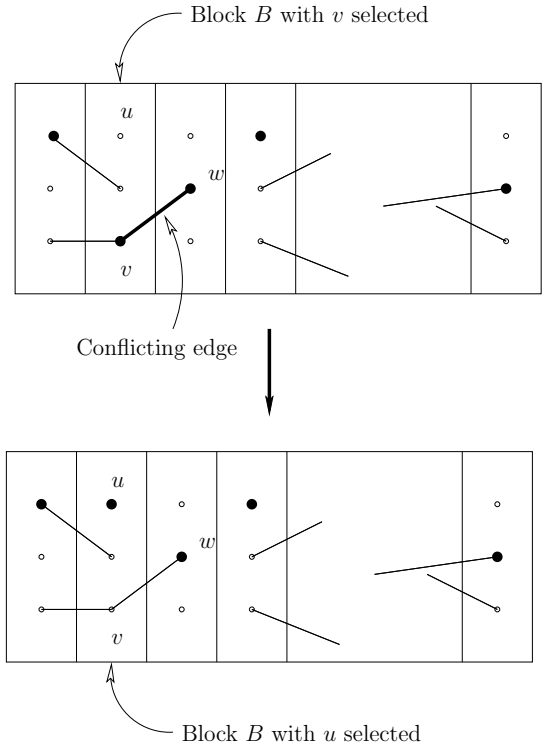

Figure 3.2: Toggling property of block-respecting independent sets; selected vertices are shown in bold

Fix an isolated vertex $u \in B$. Create a new critical truth assignment $\bar{\gamma}$ as follows: $\bar{\gamma}(v) = \text{FALSE}$, $\bar{\gamma}(u) = \text{TRUE}$, and $\bar{\gamma}(x) = \gamma(x)$ for every other vertex x in H. By construction, $\bar{\gamma}(E(H_B)) = \gamma(E(H_B)) = \text{TRUE}$. Further, since u does not have any inter-block edges and γ is critical, even $\bar{\gamma}(E(H))$ is TRUE. It follows from $H \xrightarrow{c} C$

that $\bar{\gamma}(C) = \text{TRUE}$. Recall that $\gamma(C) = \text{FALSE}$. This is what we earlier referred to as the toggling property. Since γ and $\bar{\gamma}$ differ only in their assignment to variables in block B, clause C must contain at least one literal from B. □

The subgraph of G induced by the empty set of blocks clearly has a block-respecting independent set while the subgraph induced by all blocks does not. This motivates the following definition. Let $s+1$ denote the minimum number of blocks such that some subgraph of G induced by $s+1$ blocks does not have a block respecting independent set.

Definition 3.9. The *sub-critical expansion*, $e(G)$, of G is the maximum over all $t, 2 \leq t \leq s$, of the minimum boundary size of any subgraph H of G induced by t' blocks, where $t/2 < t' \leq t$.

Lemma 3.14. *Any resolution refutation of $\alpha_{block}(G,k)$ must contain a clause of width at least $e(G)$.*

Proof. Let t be chosen as in the definition of $e(G)$ and π be a resolution refutation of $\alpha_{block}(G,k)$. By Proposition 3.5 (b), $\mu_G(\Lambda) = s+1$. Further, Proposition 3.5 (a) says that any initial clause has complexity at most 2. Therefore for $2 < t \leq s$ there exists a clause C in π such that $\mu_G(C) > t \geq 2$ and no ancestor of C has complexity greater than t.

Since $\mu_G(C) > 2$, C cannot be an initial clause. It must then be a resolvent of two parent clauses C_1 and C_2. By Proposition 3.5 (c) and the fact that no ancestor of C has complexity greater than t, one of these clauses, say C_1, must have $\mu_G(C_1)$ between $(t+1)/2$ and t. If H is a block induced subgraph that witnesses the value of $\mu_G(C_1)$, then by Lemma 3.13, $w^G_{width}(C_1) \geq |\beta(H)|$. Hence, $w(C_1) \geq |\beta(H)|$. By definition of $e(G)$, $|\beta(H)| \geq e(G)$. Thus $w(C_1) \geq e(G)$ as required. □

Corollary 3.2. *Let $c = 1/(9 \log 2)$ and $k \mid n$. For any graph G with its n vertices partitioned into k blocks of size $b = n/k$ each,*

$$\begin{aligned} \mathsf{RES}(\alpha_{block}(G,k)) &\geq 2^{c(e(G)-b)^2/n} \quad \textit{and} \\ \mathsf{DPLL}(\alpha_{block}(G,k)) &\geq 2^{e(G)-b}. \end{aligned}$$

Proof. This follows immediately from Lemma 3.14 and Propositions 2.2 and 2.1 by observing that the initial width of $\alpha_{block}(G,k)$ is b. □

3.7.2 Lower Bounding Sub-critical Expansion

Throughout this section, the probabilities are with respect to the random choice of a graph G from the distribution $\mathbb{G}(n,p)$ for some fixed parameters n and p. Let $B(G)$ be a block graph corresponding to G with block size b. For the rest of this chapter, we will fix b to be 3, which corresponds to the largest independent set size ($k = n/3$) for which the results in this section hold. Although the results can be generalized to any $b \geq 3$, our best bounds are obtained for the simpler case of $b = 3$ that we present.

Definition 3.10. $B(G)$ is (r, q)-*dense* if some subgraph of G induced by r blocks (i.e., some subgraph of $B(G)$ with r nodes) contains at least q edges.

The following lemma shows that for almost all random graphs G, the corresponding block graph $B(G)$ is locally sparse.

Lemma 3.15. *Let $G \sim \mathbb{G}(n, p)$ and $B(G)$ be a corresponding block graph with block size 3. For $r, q \geq 1$,*

$$\Pr[B(G) \text{ is } (r, q)\text{-dense}] < \left(\frac{ne}{3r}\right)^r \left(\frac{9er^2p}{2q}\right)^q.$$

Proof. Let H be a subgraph of G induced by r blocks. H contains $3r$ vertices. For $G \sim \mathbb{G}(n, p)$, the number of edges contained in H has the binomial distribution with parameters $\binom{3r}{2}$ and p. Therefore,

$$\Pr\left[H \text{ has at least } q \text{ edges}\right] \leq \binom{\binom{3r}{2}}{q} p^q < \binom{\frac{9r^2}{2}}{q} p^q \leq \left(\frac{9er^2p}{2q}\right)^q.$$

Summing this over all $\binom{n/3}{r} \leq (ne/3r)^r$ subgraphs H induced by r blocks gives the desired bound. □

We use this local sparseness property of the block graphs of almost all random graphs to prove that the smallest pair-induced subgraph one needs to consider for proving that G does not have a paired vertex cover is almost surely large.

Lemma 3.16. *There is a constant C such that the following holds. Let $\Delta = np$ and $s < Cn/\Delta^3$. The probability that $G \sim \mathbb{G}(n, p)$ contains a subgraph induced by at most s blocks that has no block-respecting independent set is $o(1)$ in s.*

Proof. The probability that G contains a subgraph induced by at most s blocks that has no block-respecting independent set is the same as the probability that there is some *minimal* subgraph H of G induced by $r \leq s$ blocks that has no block-respecting independent set. By minimality, H has no isolated vertices and hence no boundary blocks. Consequently, each of the r blocks that induce H must have at least 3 inter-block edges. Hence, the subgraph of $B(G)$ with the r nodes corresponding to the r blocks that induce H must have at least $3r/2$ edges.

Thus, the probability that G contains such a block induced subgraph H is at most

$$\sum_{r=1}^{s} \Pr[B(G) \text{ is } (r, 3r/2)\text{-dense}].$$

By Lemma 3.15, we have $\Pr[B(G) \text{ is } (r, 3r/2)\text{-dense}] < D(r)$ where

$$\begin{aligned} D(r) &= \left(\frac{ne}{3r}\right)^r (3erp)^{3r/2} \\ &= \left(\frac{ne}{3}(3ep)^{3/2} r^{1/2}\right)^r \\ &= \left(Q(n,p)\ r^{1/2}\right)^r \end{aligned}$$

for $Q(n,p) = (ne/3)(3ep)^{3/2}$. Now

$$\begin{aligned} \frac{D(r+1)}{D(r)} &= \frac{\left(Q(n,p)\ (r+1)^{1/2}\right)^{r+1}}{\left(Q(n,p)\ r^{1/2}\right)^r} \\ &= Q(n,p)\ (r+1)^{1/2}\ \left(\frac{r+1}{r}\right)^{r/2} \\ &\leq Q(n,p)\ (r+1)^{1/2}\ e^{1/2} \\ &\leq \frac{ne}{3}\left(\frac{3e\Delta}{n}\right)^{3/2} e^{1/2}(r+1)^{1/2} \\ &= \left(\frac{3e^6\Delta^3(r+1)}{n}\right)^{1/2} \end{aligned}$$

This quantity, and hence $D(r+1)/D(r)$, is at most $1/2$ for $1 \leq r < Cn/\Delta^3$, where $C \stackrel{\text{def}}{=} 1/(12e^6)$ is a constant. Let $s+1 = Cn/\Delta^3$. It follows that the probability that G contains such a block induced subgraph H is bounded above by a geometric series in r with common ratio $1/2$. It is therefore at most twice the largest term of the series which is less than $D(1)$. Now

$$D(1) \ = \ Q(n,p) \ = \ \frac{ne}{3}\left(\frac{3e\Delta}{n}\right)^{3/2} \ = \ \left(\frac{3e^5\Delta^3}{n}\right)^{1/2} \ = \ \left(\frac{3e^5C}{s+1}\right)^{1/2}.$$

Therefore, $D(1)$ is $o(1)$ in s as claimed. □

We again use the local sparseness property to prove that any subgraph induced by not too many blocks has large boundary for almost all random graphs G. The intuition is that for sparse subgraphs, most blocks have degree less than 3 and thus belong to the boundary.

Lemma 3.17. *There is a constant c such that the following holds. Let $\Delta = np$, $0 < \epsilon \leq 1/6$, $b' = 3(1-\epsilon)$, $t \leq cn/\Delta^{\frac{b'}{b'-2}}$, and $G \sim \mathbb{G}(n,p)$. The probability that there exists $r \in (t/2, t]$ such that G has a subgraph H induced by r blocks with $\beta(H) \leq \epsilon r$ is $o(1)$ in t.*

Proof. Fix b', ϵ, and t satisfying the conditions of the Lemma. Let H be a subgraph of G induced by r blocks. By definition, all r blocks inducing H must have non-zero degree in $B(H)$. Moreover, if H has at most ϵr boundary blocks, the other $(1-\epsilon)r$ blocks of non-zero degree inducing it must have degree at least 3. Hence, the r nodes of $B(G)$ that induce H form a subgraph with at least $(1-\epsilon)r3/2 = b'r/2$ edges. Therefore, H has at most ϵr boundary blocks only if $B(G)$ is $(r, b'r/2)$-dense. Thus, by Lemma 3.15, the probability that such an H exists is at most

$$\begin{aligned}\Pr[B(G) \text{ is } (r,b'r)\text{-dense}] &< \left(\frac{ne}{3r}\right)^r \left(\frac{9erp}{b'}\right)^{b'r/2} \\ &= \left(\frac{e}{3}\left(\frac{3e\Delta}{1-\epsilon}\right)^{b'/2}\left(\frac{r}{n}\right)^{(b'-2)/2}\right)^r\end{aligned}$$

For $r > t/2$, it suffices to obtain an upper bound on this probability that is exponentially small in r. Rearranging the terms in the expression above, $\Pr[B(G) \text{ is } (r,b'r)\text{-dense}] \leq 2^{-r}$ when

$$\begin{aligned}\frac{r}{n} &\leq \left(\frac{3}{2e}\right)^{2/(b'-2)}\left(\frac{1-\epsilon}{3e\Delta}\right)^{b'/(b'-2)} \\ &= \left(\frac{1-\epsilon}{2e^2}\right)^{2/(b'-2)}\frac{1-\epsilon}{3e\Delta^{b'/(b'-2)}}.\end{aligned}$$

Note that $\epsilon \leq 1/6$ and $b' = 3(1-\epsilon) \geq 5/2$. Hence $(1-\epsilon)/(2e^2)^{2/(b'-2)}$ is at least $(5/(12e^2))^4$ and it suffices to have

$$\frac{r}{n} \leq \left(\frac{5}{12e^2}\right)^4 \frac{5}{18e\Delta^{b'/(b'-2)}} = \frac{c}{\Delta^{b'/(b'-2)}}$$

for a constant $c \stackrel{\text{def}}{=} 5^5/(12^4 18e^9)$. Therefore, the probability that $B(G)$ is $(r, b'r)$-dense is at most 2^{-r} for $r \leq cn/\Delta^{b'/(b'-2)}$. It follows that the probability that there exists such an H with $r \in (t/2, t]$ is at most $\sum_{r=\lceil (t+1)/2 \rceil}^{t} 2^{-r}$. This sum is $o(1)$ in t as required. □

Lemmas 3.16 and 3.17 combine to give the following lower bound on sub-critical expansion:

Lemma 3.18. *For each $\epsilon \in (0, 1/6]$ there is a constant c_ϵ such that the following holds. Let $\Delta = np$, $b' = 3(1-\epsilon)$, $W = n/\Delta^{b'/(b'-2)}$, and $G \sim \mathbb{G}(n,p)$. The probability that $e(G) < c_\epsilon W$ is $o(1)$ in W.*

Proof. Let C be the constant from Lemma 3.16 and c be the one from Lemma 3.17. Let $s+1$ be the minimum number of blocks such that some subgraph of G induced

by $s+1$ blocks does not have a block induced independent set. By Lemma 3.16, $s \leq Cn/\Delta^3$ with probability $o(1)$ in n. Now let $t = \min(C,c)W$. Conditioned on $s > Cn/\Delta^3$ and because $b' < 3$, we have that $t \leq s$ as in the definition of $e(G)$. By Lemma 3.17, the probability that some subgraph of G induced by r blocks with $t/2 < r \leq t \leq s$ has less than $\epsilon r > \epsilon t/2 = c_\epsilon W$ boundary blocks is $o(1)$ in n, where $c_\epsilon = (\epsilon/2)\min(C,c)$. It follows from a union bound on the two bad events (s is small or some subgraph has small boundary) that $e(G) < c_\epsilon W$ with probability $o(1)$ in n. □

3.8 Lower Bounds for Resolution and Associated Algorithms

We now use the ideas developed in Sections 3.6 and 3.7, and bring the pieces of the argument together in a general technical result from which our resolution complexity lower bounds follow.

Lemma 3.19. *For each $\delta > 0$ there are constants $C_\delta, C'_\delta > 0$ such that the following holds. Let $\Delta = np$ and $G \sim \mathbb{G}(n,p)$. With probability $1 - o(1)$ in n,*

$$\begin{aligned} \mathsf{RES}(\alpha_{block}(G, n/3)) &\geq 2^{C_\delta n/\Delta^{6+2\delta}} \quad \textit{and} \\ \mathsf{DPLL}(\alpha_{block}(G, n/3)) &\geq 2^{C'_\delta n/\Delta^{3+\delta}}. \end{aligned}$$

Proof. Observe that the expressions $n/\Delta^{6+2\delta}$ and $n/\Delta^{3+\delta}$ in the desired bounds decrease as δ increases. Hence, it suffices to prove the bounds for $\delta \in (0,2]$, and for $\delta > 2$, simply let $C_\delta = C_2$ and $C'_\delta = C'_2$.

Let $\epsilon = \delta/(6+3\delta)$, $b' = 3(1-\epsilon)$, and $W = n/\Delta^{b'/(b'-2)}$. For $\delta \in (0,2]$, we have that $\epsilon \in (0, 1/6]$. From Lemma 3.18, there is a constant c_ϵ such that with probability $1 - o(1)$ in n, $e(G) \geq c_\epsilon W$. It follows from Corollary 3.2 that for $c = 1/(9\log 2)$ and with probability $1 - o(1)$ in n,

$$\begin{aligned} \mathsf{RES}(\alpha_{block}(G, n/3)) &\geq 2^{c(c_\epsilon W - 3)^2/n} \quad \text{and} \\ \mathsf{DPLL}(\alpha_{block}(G, n/3)) &\geq 2^{c_\epsilon W - 3}. \end{aligned}$$

Given the relationship between ϵ and δ, there are constants $C_\delta, C'_\delta > 0$ depending only on δ such that $c(c_\epsilon W - 3)^2 \geq C_\delta W^2$ and $c_\epsilon W - 3 \geq C'_\delta W$. Note also that $b'/(b'-2) = (3-3\epsilon)/(1-3\epsilon) = 3+\delta$. Hence,

$$\log_2(\mathsf{RES}(\alpha_{block}(G, n/3))) \geq C_\delta W^2/n = C_\delta n/\Delta^{\frac{2b'}{b'-2}} = C_\delta n/\Delta^{6+2\delta} \quad \text{and}$$

$$\log_2(\mathsf{DPLL}(\alpha_{block}(G, n/3))) \geq C'_\delta W = C'_\delta n/\Delta^{\frac{b'}{b'-2}} = C'_\delta n/\Delta^{3+\delta}.$$

This finishes the proof. □

Theorem 3.2 (Independent Set Lower Bounds). *For each $\delta > 0$ there are constants C_δ, C'_δ, C''_δ, C'''_δ, $C''''_\delta > 0$ such that the following holds. Let $\Delta = np$, $k \le n/3$, $k \mid n$, and $G \sim \mathbb{G}(n,p)$. With probability $1 - o(1)$ in n,*

$$\begin{aligned}
\mathsf{RES}(\alpha_{map}(G,k)) &\ge 2^{C_\delta n/\Delta^{6+2\delta}},\\
\mathsf{DPLL}(\alpha_{map}(G,k)) &\ge 2^{C'_\delta n/\Delta^{3+\delta}},\\
\mathsf{RES}(\alpha_{count}(G,k)) &\ge 2^{C''_\delta n/\Delta^{6+2\delta}},\\
\mathsf{DPLL}(\alpha_{count}(G,k)) &\ge 2^{C'''_\delta n/\Delta^{3+\delta}},\\
\mathsf{RES}(\alpha_{block}(G,k)) &\ge 2^{C_\delta n/\Delta^{6+2\delta}},\\
\mathsf{DPLL}(\alpha_{block}(G,k)) &\ge 2^{C'_\delta n/\Delta^{3+\delta}}, \quad \textit{and}\\
Chv(G,k) &\ge 2^{C''''_\delta n/\Delta^{6+2\delta}}.
\end{aligned}$$

The bounds for the block encoding require $k \mid (n/3)$.

Proof. All of the claimed bounds follow by applying monotonicity of the encoding at hand, using its relationship with the block encoding, and applying Lemma 3.19. Let C_δ and C'_δ be the constants from Lemma 3.19. For the mapping-based encoding,

$$\begin{aligned}
\mathsf{RES}(\alpha_{map}(G,k)) &\ge \mathsf{RES}(\alpha_{map}(G,n/3)) && \text{by Lemma 3.3}\\
&\ge \mathsf{RES}(\alpha_{block}(G,n/3)) && \text{by Lemma 3.6}\\
&\ge 2^{C_\delta n/\Delta^{6+2\delta}} && \text{by Lemma 3.19,}
\end{aligned}$$

$$\begin{aligned}
\mathsf{DPLL}(\alpha_{map}(G,k)) &\ge \mathsf{DPLL}(\alpha_{map}(G,n/3)) && \text{by Lemma 3.3}\\
&\ge \mathsf{DPLL}(\alpha_{block}(G,n/3)) && \text{by Lemma 3.6}\\
&\ge 2^{C'_\delta n/\Delta^{3+\delta}} && \text{by Lemma 3.19.}
\end{aligned}$$

For the counting-based encoding,

$$\begin{aligned}
\mathsf{RES}(\alpha_{count}(G,k) &\ge \frac{1}{n}\left(\mathsf{RES}(\alpha_{count}(G,n/3)) - 2n^2\right) && \text{by Lemma 3.2}\\
&\ge \frac{1}{n}\left(\frac{1}{2}\mathsf{RES}(\alpha_{block}(G,n/3)) - 2n^2\right) && \text{by Lemma 3.5}\\
&\ge 2^{C''_\delta n/\Delta^{6+2\delta}} && \text{by Lemma 3.19}
\end{aligned}$$

for a large enough constant C''_δ. Similarly,

$$\begin{aligned}
\mathsf{DPLL}(\alpha_{count}(G,k) &\ge \frac{1}{n}\left(\mathsf{DPLL}(\alpha_{count}(G,n/3)) - 2n^2\right) && \text{by Lemma 3.2}\\
&\ge \frac{1}{n}\left(\frac{1}{2}\mathsf{DPLL}(\alpha_{block}(G,n/3))^{1/\log_2 6} - 2n^2\right) && \text{by Lemma 3.5}\\
&\ge 2^{C'''_\delta n/\Delta^{3+\delta}} && \text{by Lemma 3.19}
\end{aligned}$$

for a large enough constant C'''_δ.

The bounds for the block encoding follow immediately from Lemmas 3.4 and 3.19. Finally, for the bound on the proof size in Chvátal's system,

$$\begin{aligned} Chv(G,k) &\geq Chv(G,n/3) - 1 && \text{by Proposition 3.3} \\ &\geq \frac{1}{4n}\mathsf{RES}(\alpha_{block}(G,n/3)) - 1 && \text{by Lemma 3.7} \\ &\geq 2^{C''''_\delta n/\Delta^{6+2\delta}} && \text{by Lemma 3.19} \end{aligned}$$

for a large enough constant c''''_δ. □

Corollary 3.3 (Vertex Cover Lower Bounds). *For each $\delta > 0$ there are constants $\widetilde{C}_\delta, C_\delta, C'_\delta > 0$ such that the following holds. Let $\Delta = np$, $t \geq 2n/3$, $(n-t) \mid n$, and $G \sim \mathbb{G}(n,p)$. With probability $1 - o(1)$ in n,*

$$\begin{aligned} \mathsf{RES}(VC_{count}(G,t)) &\geq 2^{\widetilde{C}_\delta n/\Delta^{6+2\delta}}, \\ \mathsf{RES}(VC_{block}(G,t)) &\geq 2^{C_\delta n/\Delta^{6+2\delta})}, \quad \textit{and} \\ \mathsf{DPLL}(VC_{block}(G,t)) &\geq 2^{C'_\delta n/\Delta^{3+\delta})}. \end{aligned}$$

The bounds for the block encoding require $(n-t) \mid (n/3)$.

Proof. Let C_δ, C'_δ, and C''_δ be the constants from Theorem 3.2 and let $\widetilde{C}_\delta$ be any constant less than C''_δ. For the counting encoding bound, apply Theorem 3.2 with k set to $n-t$ and use Lemma 3.9 to translate the results to the encoding of vertex cover. For the block encoding bounds, apply Theorem 3.2 in conjunction with Lemma 3.10. □

3.9 Hardness of Approximation

Instead of considering the decision problem of whether a given graph G has an independent set of a given size k, one may consider the related *optimization problem*: given G, find an independent set in it of the largest possible size. We call this optimization problem the *maximum independent set* problem. One may similarly define the optimization problem *minimum vertex cover* problem.

Since the decision versions of these problems are NP-complete, the optimization versions are NP-hard and do not have any known polynomial time solutions. From the perspective of algorithm design, it is then natural to ask whether there is an efficient algorithm that finds an independent set of size "close" to the largest possible or a vertex cover of size close to the smallest possible. That is, is there an efficient algorithm that finds an "approximate" solution to the optimization problem? In this section, we rule out the existence of any such efficient "resolution-based" algorithm for the independent set and vertex cover problems.

Remark 3.2. The results we prove in this section contrast well with the known approximation hardness results for the two problems which are both based on the PCP (probabilistically checkable proofs) characterization of NP [9, 8]. Håstad [62] showed that unless P = NP, there is no polynomial time $n^{1-\epsilon}$-approximation algorithm for the clique (and hence the independent set) problem for any $\epsilon > 0$. For graphs with maximum degree Δ_{max}, Trevisan [108] improved this to a factor of $\Delta_{max}/2^{O(\sqrt{\log \Delta_{max}})}$. More recently, Dinur and Safra [47] proved that unless P = NP, there is no polynomial time $10\sqrt{5} - 21 \approx 1.36$ factor approximation algorithm for the vertex cover problem. Our results, on the other hand, hold irrespective of the relationship between P and NP but apply only to the class of resolution-based algorithms defined shortly.

3.9.1 Maximum Independent Set Approximation

We begin by making several of the above notions precise. Let A be an algorithm for finding a maximum independent set in a given graph.

Definition 3.11. Let $\gamma \geq 1$. A is a *γ-approximation algorithm* for the maximum independent set problem if on input G with maximum independent set size $\hat{k}$, A produces an independent set of size at least $\hat{k}/\gamma$.

In other words, if A produces an independent set of size $\bar{k}$ on input G, it proves that G does not have one of size $\bar{k}\gamma + 1$. This reasoning allows us to use our lower bounds from the previous section to prove that even approximating a maximum independent set is exponentially hard for certain resolution-based algorithms.

Definition 3.12. A γ-approximation algorithm A for the maximum independent set problem is *resolution-based* if it has the following property: if A outputs an independent set of size $\bar{k}$ on input G, then its computation history along with a proof of correctness within a factor of γ yields a resolution proof of $\alpha_{map}(G, k)$, $\alpha_{count}(G, k)$, or $\alpha_{block}(G, k)$ for $k \leq \bar{k}\gamma + 1, k \mid n$. (For the block encoding, we further require $k \mid (n/3)$.)

The manner in which the computation history and the proof of correctness are translated into a resolution refutation of an appropriate encoding depends on specific details and varies with the context. We will see a concrete example of this for the vertex cover problem when discussing Proposition 3.7 later in this section.

Let $\mathcal{A}_{\gamma}^{RES-ind}$ denote the class of all resolution-based γ-approximation algorithms for the maximum independent set problem. We show that while there is a trivial algorithm in this class for $\gamma \geq \Delta + 1$, there isn't an efficient one for $\gamma \leq \Delta/(6 \log \Delta)$.

Proposition 3.6. *For $\gamma \geq \Delta + 1$, there is a polynomial-time algorithm in $\mathcal{A}_{\gamma}^{RES-ind}$.*

Proof. Let A be the polynomial-time algorithm that underlies the bound in Turan's theorem (Proposition 3.1), that is, on a graph G with n nodes and average degree Δ as input, A produces an independent set of size $\bar{k} \geq n/(\Delta + 1)$. Since the size of a

maximum independent set in G is at most n, A is a $(\Delta+1)$-approximation. We will argue that A is also resolution-based.

To be a resolution-based, the computation history of A on G along with a proof of correctness within a factor of $(\Delta+1)$ must yield a resolution proof of a suitable encoding $\alpha(G,k)$ for some $k \leq k^* = \bar{k}(\Delta+1)+1, k \mid n$. When G has no edges, $\Delta = 0$ and A produces an independent set of size $\bar{k} = n$. In this case, there is nothing to prove. When G has at least one edge (u,v), $k^* \geq n+1$ and we can choose $k = n$. In this case, A indeed yields a straightforward resolution proof of $\alpha(G,k)$ for both the mapping and the counting encodings by utilizing the edge clause(s) corresponding to (u,v). Therefore, A is resolution-based as a $(\Delta+1)$-approximation algorithm. □

While Proposition 3.2 guarantees that there is almost never an independent set of size larger than $(2n/\Delta)\log\Delta$, Theorem 3.2 shows that there is no efficient way to prove this fact using resolution. Indeed, there exist efficient resolution proofs only for the non-existence of independent sets of size larger than $n/3$. We use this reasoning to prove the following hardness of approximation result.

Theorem 3.3 (Independent Set Approximation). *There is a constant c such that the following holds. Let $\delta > 0$, $\Delta = np$, $\Delta \geq c$, $\gamma \leq \Delta/(6\log\Delta)$, and $G \sim \mathbb{G}(n,p)$. With probability $1-o(1)$ in n, every algorithm $A \in \mathcal{A}_\gamma^{RES-ind}$ takes time exponential in $n/\Delta^{6+2\delta}$.*

Proof. Recall the definitions of $k_{+\epsilon}$ and C_ϵ from Proposition 3.2. Fix $\epsilon > 0$ such that $k_{+\epsilon} < (2n/\Delta)\log\Delta$ and let $c \geq C_\epsilon$. The claimed bound holds trivially for $\Delta \geq n^{1/6}$. We will assume for the rest of the proof that $C_\epsilon \leq \Delta \leq n/\log^2 n$.

From Proposition 3.2, with probability $1-o(1)$ in n, a maximum independent set in G is of size $k_{max} \leq k_{+\epsilon} < (2n/\Delta)\log\Delta$. If A approximates this within a factor of γ, then, in particular, it proves that G does not have an independent set of size $k = k_{max}\gamma + 1 \leq n/3$. Convert the transcript of the computation of A on G along with an argument of its correctness within a factor of γ into a resolution proof π of an appropriate encoding $\alpha(G,k)$. From Theorem 3.2, $size(\pi)$ must be exponential in $n/\Delta^{6+2\delta}$. □

3.9.2 Minimum Vertex Cover Approximation

A similar reasoning can be applied to approximation algorithms for finding a minimum vertex cover.

Definition 3.13. Let $\gamma \geq 1$. A is a *γ-approximation algorithm* for the minimum vertex cover problem if on input G with minimum vertex cover size $\hat{t}$, A produces a vertex cover of size at most $\hat{t}\gamma$.

Definition 3.14. A γ-approximation algorithm A for the minimum vertex cover problem is *resolution-based* if it has the following property: if A outputs a vertex cover

of size $\bar{t}$ on input G, then its computation history along with a proof of correctness within a factor of γ yields a resolution proof of $VC_{count}(G,t)$ or $VC_{block}(G,t)$ for $t \geq \bar{t}/\gamma - 1, (n-t) \mid n$. (For the block encoding, we further require $(n-t) \mid (n/3)$.)

Let $\mathcal{A}_{\gamma}^{RES-VC}$ denote the class of all resolution-based γ-approximation algorithms for the minimum vertex cover problem.

As the following proposition shows, the usual greedy 2-approximation algorithm for vertex cover, for instance, is in $\mathcal{A}_{2}^{RES-VC}$. It works by choosing an arbitrary edge, say (u,v), including both u and v in the vertex cover, throwing away all edges incident on u and v, and repeating this process until all edges have been removed from the graph. This gives a 2-approximation because any optimal vertex cover will also have to choose at least one of u and v. For concreteness, we describe this algorithm below as Algorithm 3.1 and denote it by `VC-greedy`. We use $E(G)$ for the set of edges in G and $E(w)$ for the set of edges incident on a vertex w.

Input : An undirected graph G with minimum vertex cover size $t+1$
Output : A vertex cover for G of size at most $2(t+1)$
begin
  $cover \leftarrow \phi$
  while $E(G) \neq \phi$ **do**
    Choose an edge $(u,v) \in E(G)$ arbitrarily
    $cover \leftarrow cover \cup \{u,v\}$
    $E(G) \leftarrow E(G) \setminus (E(u) \cup E(v))$
  Output *cover*
end

Algorithm 3.1: `VC-greedy`, a greedy 2-approximation algorithm for the minimum vertex cover problem

Proposition 3.7. *Let $t = \bar{t}/2 - 1$ and $(n-t) \mid n$. If* `VC-greedy` *outputs a vertex cover of size $\bar{t}$ on input G, then* $\mathsf{RES}(VC_{count}(G,t)) \leq 8t^2$.

Proof. Consider a run of `VC-greedy` on G that produces a vertex cover of size $\bar{t} = 2(t+1)$. This yields a sequence of $\bar{t}/2 = t+1$ *vertex disjoint* edges of G that are processed sequentially till G has no edges left. Without loss of generality, assume that these $t+1$ edges are $(v_1,v_2),(v_3,v_4),\ldots,(v_{2t+1},v_{2t+2})$. Extend this ordering of the vertices of G to the remaining $n-2t-2$ vertices. Under this ordering, we will construct a refutation of $\alpha_{count}(G,t)$ of size at most $8t^2$. Note that $\alpha_{count}(G,t)$ includes $(x_{2p-1} \vee x_{2p}), 1 \leq p \leq t+1$, among its edge clauses.

In order to construct this derivation, it will be helpful to keep in mind that one can resolve any clause $(y_{q,i} \vee B), 1 \leq q < n, 1 \leq i \leq t$, with the initial clause $(y_{q+1,i} \vee \neg y_{q,i} \vee x_{q+1})$ to derive $(y_{q+1,i} \vee x_{q+1} \vee B)$. For convenience, we will refer to this as a Z_1 derivation. Similarly, for $i < t$, $(y_{q,i} \vee B)$ can be resolved with the initial

clause $(y_{q+1,i+1} \vee \neg y_{q,i} \vee \neg x_{q+1})$ to derive $(y_{q+1,i+1} \vee \neg x_{q+1} \vee B)$. We will refer to this as a Z_2 derivation.

Using the above derivations as building blocks, we show that for $0 \leq p < t, 0 \leq i < t-1$, and any clause $(y_{2p,i} \vee A)$, we can derive the clause $(y_{2p+2,i+1} \vee y_{2p+2,i+2} \vee A)$ in 8 resolution steps. First, apply a Z_1 derivation to $(y_{2p,i} \vee A)$ to obtain $(y_{2p+1,i} \vee x_{2p+1} \vee A)$. Apply a Z_2 derivation to this to get $(y_{2p+2,i+1} \vee x_{2p+1} \vee \neg x_{2p+2} \vee A)$. Resolve this with the edge clause $(x_{2p+1} \vee x_{2p+2})$ to finally obtain the clause $C_1 = (y_{2p+2,i+1} \vee x_{2p+1} \vee A)$. Starting again from $(y_{2p,i} \vee A)$, apply a Z_2 derivation to obtain $(y_{2p+1,i+1} \vee \neg x_{2p+1} \vee A)$. Apply Z_1 and Z_2 derivations separately to this clause and resolve the results together on the variable x_{2p+2} to obtain the clause $C_2 = (y_{2p+2,i+1} \vee y_{2p+2,i+2} \vee \neg x_{2p+1} \vee A)$. Resolving clauses C_1 and C_2 on the variable x_{2p+1} finishes the 8 step derivation of $(y_{2p+2,i+1} \vee y_{2p+2,i+2} \vee A)$. We will refer to this derivation as Z_3.

A similar argument shows that for the boundary case $i = t-1$, one can derive from $(y_{2p,t-1} \vee A)$ in at most 8 steps the clause $(y_{2p+2,t} \vee A)$.

We are ready to describe the overall construction of the refutation. Starting from the initial clause $y_{0,0}$, apply Z_3 to derive $(y_{2,1} \vee y_{2,2})$. Now apply Z_3 successively to the two literals of this clause to obtain $(y_{4,2} \vee y_{4,3} \vee y_{4,4})$. Applying Z_3 repeatedly to the literals of the clause obtained in this manner results in the derivation of $(y_{2p,p} \vee y_{2p,p+1} \vee \ldots \vee y_{2p,r})$ in at most $8p^2$ steps, where $1 \leq p \leq t$ and $r = \min(2p, t)$. For $p = t$, this gives a derivation of $y_{2t,t}$ in a total of $8t^2$ steps.

Resolving $y_{2t,t}$ with the initial clauses $(\neg y_{2t,t} \vee \neg x_{2t+1})$ and $(\neg y_{2t,t} \vee \neg x_{2t+2})$, and resolving the two resulting clauses with the edge clause $(x_{2t+1} \vee x_{2t+2})$ derives the empty clause Λ and finishes the refutation. □

Theorem 3.4 (Vertex Cover Approximation). *There is a constant c such that the following holds. Let $\delta > 0$, $\Delta = np$, $\Delta \geq c$, $\gamma < 3/2$, and $G \sim \mathbb{G}(n,p)$. With probability $1 - o(1)$ in n, every algorithm $A \in \mathcal{A}_\gamma^{RES-VC}$ takes time exponential in $n/\Delta^{6+2\delta}$.*

Proof. This proof is very similar to that of Theorem 3.3. Recall the definitions of $k_{+\epsilon}$ and C_ϵ from Proposition 3.2. Fix $\epsilon > 0$ such that $k_{+\epsilon} < (2n/\Delta)\log\Delta$ and let $c \geq C_\epsilon$. The claimed bound holds trivially for $\Delta \geq n^{1/6}$. We will assume for the rest of the proof that $C_\epsilon \leq \Delta \leq n/\log^2 n$.

From Proposition 3.2 and the relation between independent sets and vertex covers, with probability $1-o(1)$ in n, a minimum vertex cover in G is of size $t_{min} \geq n - k_{+\epsilon} > n - (2n/\Delta)\log\Delta$. If A approximates this within a factor of γ, then, in particular, it proves that G does not have a vertex cover of size $t = t_{min}/\gamma - 1 \geq 2n/3$. Convert the transcript of A's computation on G along with an argument of its correctness within a factor of γ into a resolution proof π of an appropriate encoding $VC(G,t)$. From Corollary 3.3, $size(\pi)$ must be exponential in $n/\Delta^{6+2\delta}$. □

3.10 Stronger Lower Bounds for Exhaustive Backtracking Algorithms and DPLL

We conclude this chapter with a stronger lower bound for a natural class of backtracking algorithms for the independent set and vertex cover problems, namely the class of exhaustive backtracking search algorithms. The key difference between the algorithms captured by resolution that we have considered so far and the ones in this class is that the latter do *not* reuse computation performed for previous branches; instead, they systematically rule out all potential independent sets or vertex covers of the desired size by a possibly smart but nonetheless exhaustive search. As an illustration, we will give an example of a non-trivial exhaustive backtracking algorithm for the independent set problem shortly.

The argument for our lower bound is based on the density of independent sets and vertex covers in random graphs and is quite straightforward in the light of Lemma 3.1. We derive as a consequence a tighter lower bound for the DPLL complexity of the mapping and counting encodings of the two problems that allows the edge density in the underlying graph to be much higher than in Theorem 3.2 and Corollary 3.3.

Returning to the class of exhaustive backtracking algorithms, recall that the approach we used for our upper bounds (cf. Section 3.5) was to systematically rule out all potential independent sets of a certain size $k' = k_{min}$. This is the simplest algorithm in the class. Of course, instead of simply considering all $\binom{n}{k'}$ subsets of vertices of size k' as we did, one can imagine more complex techniques for exhaustive search. For instance, an idea similar to the one used by Beame et al. [14] for the graph coloring problem would be to consider all subsets of size $u < k'$ in the first stage. For a random graph, most of these subsets are very likely to already contain an edge and need not be processed further. For any remaining subset S, one can recursively refute the existence of an independent set of size $k' - u$ in the residual graph with $|n - k - N(S)|$ vertices, where $N(S)$ denotes all neighbors of S outside S. This is also an exhaustive backtracking algorithm.

Such algorithms may require a more complex analysis than we gave in our upper bound proofs and could potentially be more efficient. However, as the following result shows, any technique that systematically rules out all possible k'-independent sets by an exhaustive backtracking search cannot improve the relatively simple upper bounds in Theorem 3.1 and Corollary 3.1 by more than a constant factor in the exponent.

Let $\mathcal{A}^{ind}_{exhaustive}$ (or $\mathcal{A}^{VC}_{exhaustive}$) denote the class of backtracking algorithms for proving non-existence of independent sets (vertex covers, resp.) of a given size in a given graph, that work by recursively subdividing the problem based on whether or not a set of vertices is included in the independent set (vertex cover, resp.) and that do not reuse computation performed in previous branches. For example, our approach in Section 3.5 as well as the algorithm based on [14] sketched above, both belong to $\mathcal{A}^{ind}_{exhaustive}$.

Theorem 3.5 (Exhaustive Backtracking Algorithms). *There are constants C and c such that the following holds. Let $\Delta = np$, $c \leq \Delta \leq n/\log^2 n$, and $G \sim \mathbb{G}(n,p)$. With probability $1-o(1)$ in n, every algorithm $A \in \mathcal{A}^{ind}_{exhaustive}$ (or $\mathcal{A}^{VC}_{exhaustive}$) running on input (G,k) must branch at least $2^{C(n/\Delta)\log^2 \Delta}$ times when G does not have an independent set (vertex cover, resp.) of size k.*

Proof. Let C be the constant from Lemma 3.1. Recall the definitions of $k_{+\epsilon}$ and C_ϵ from Proposition 3.2. Fix $\epsilon > 0$ such that $k_{+\epsilon} + 1 > (2n/\Delta)\log\Delta$ and let $c \geq C_\epsilon$. With probability $1-o(1)$ in n, algorithm $A \in \mathcal{A}^{ind}_{exhaustive}$ succeeds in proving the non-existence of a k-independent set in G only when $k \geq k_{+\epsilon}+1$. However, Lemma 3.1 says that G almost surely contains at least $2^{C(n/\Delta)\log^2\Delta}$ independent sets of size $k^* = \lfloor (n/\Delta)\log\Delta \rfloor$, which is less than $(k_{+\epsilon}+1)/2$. Hence, while recursively subdividing the problem based on whether or not to include a vertex in the k-independent set, A must explore at least $2^{C(n/\Delta)\log^2\Delta}$ distinct k^*-independent sets before finding a contradictory edge for each and backtracking.

For the vertex cover case, note that the algorithms in $\mathcal{A}^{VC}_{exhaustive}$ are the duals of the algorithms in $\mathcal{A}^{ind}_{exhaustive}$; including a vertex in a vertex cover to create a smaller subproblem is equivalent to not including it in an independent set. Further, the number of vertex covers of size $n-k$ in G is exactly the same as the number of independent sets of size k in G. Hence, the above lower bound applies to the algorithms in $\mathcal{A}^{VC}_{exhaustive}$ as well. □

Theorem 3.6 (Stronger DPLL Lower Bounds). *There are constants C and c such that the following holds. Let $\Delta = np$, $c \leq \Delta \leq n/(2\log^2 n)$, and $G \sim \mathbb{G}(n,p)$. With probability $1-o(1)$ in n,*

$$\begin{aligned}
\mathsf{DPLL}(\alpha_{map}(G,k)) &\geq 2^{C(n/\Delta)\log^2\Delta}, \\
\mathsf{DPLL}(\alpha_{count}(G,k)) &\geq 2^{C(n/\Delta)\log^2\Delta}, \\
\mathsf{DPLL}(VC_{map}(G,t)) &\geq 2^{C(n/\Delta)\log^2\Delta}, \quad \textit{and} \\
\mathsf{DPLL}(VC_{count}(G,t)) &\geq 2^{C(n/\Delta)\log^2\Delta}.
\end{aligned}$$

Proof. The DPLL complexity of the encodings, by our convention, is ∞ if G does have an independent set of size k. If it does not, the tree T associated with any DPLL refutation of $\alpha_{map}(G,k)$ or $\alpha_{count}(G,k)$ can be viewed as the trace of an exhaustive backtracking algorithm $A \in \mathcal{A}^{ind}_{exhaustive}$ on input (G,k) as follows. An internal node in T with variable x_v as its secondary label corresponds to the decision of A to branch based on whether or not to include vertex v in the independent set it is creating. Nodes in T with counting variables as secondary labels represent the counting process of A.

Given this correspondence, Theorem 3.5 immediately implies the desired lower bounds for the independent set problem. The results for the vertex cover problem can be derived in an analogous manner. Note that refuting the block encoding may be easier than ruling out all independent sets (vertex covers, resp.) of size k. Hence, Theorem 3.5 does not translate into a bound for this encoding. □

3.11 Discussion

In this chapter, we used a combination of combinatorial and probabilistic arguments to obtain lower and upper bounds on the resolution complexity of several natural CNF encodings of the independent set, vertex cover, and clique problems. Our results hold almost surely when the underlying graph is chosen at random from the $\mathbb{G}(n,p)$ model. Consequently, they hold (deterministically) for nearly all graphs. A key step in the main lower bound arguments was to simplify the task by considering the induced block graph in place of the original graph. The expansion properties of the block graph then allowed us to relate refutation width with structural properties of the graph.

Our results imply exponential lower bounds on the running time of resolution-based backtracking algorithms for finding a maximum independent set (or, equivalently, a maximum clique or a minimum vertex cover) in a given graph. Such algorithms include some of the best known ones for these combinatorial problems [105, 106, 67, 100].

A noteworthy contribution of this work is the hardness of approximation result. We showed unconditionally that there is no polynomial time resolution-based approximation algorithm that guarantees a solution within a factor less than $\Delta/(6\log\Delta)$ for the maximum independent set problem or within a factor less than $3/2$ for the minimum vertex cover problem. This complements the hardness results conditioned on P $\neq$ NP that rule out efficient approximations within factors of $\Delta_{max}/2^{O(\sqrt{\log\Delta_{max}})}$ [108] and $10\sqrt{5}-21 \approx 1.36$ [47] for the two problems, respectively. (Here Δ_{max} denotes the maximum degree of the underlying graph rather than the average degree.)

On the flip side, some algorithms, such as those of Robson [97], Beigel [21], Chen et al. [33], and Tomita and Seki [107], employ techniques that do not seem to be captured by resolution. The techniques they use, such as unrestricted without loss of generality arguments [97], vertex folding [33], creation of new vertices [21], and pruning of search space using approximate coloring [107], represent global properties of graphs or global changes therein that appear hard to argue locally using a bounded number of resolution inferences. For instance, the algorithm of Robson [97] involves the reasoning that if an independent set contains only one element of $N(v)$, then without loss of generality, that element can be taken to be the vertex v itself. It is unclear how to model this behavior efficiently in resolution.

Restricted versions of these general properties, however, can indeed be simulated by resolution. This applies when one restricts, for instance, to vertices of small, bounded degree, as is done in many case-by-case algorithms cited at the beginning of this chapter [105, 106, 67, 100].

Finally, as we mentioned in the introduction, the spectral algorithm of Coja-Oghlan [37] achieves an $O(\sqrt{\Delta}/\log\Delta)$ approximation and, in the light of our lower bounds, cannot be simulated by resolution.

Chapter 4

CLAUSE LEARNING AS A PROOF SYSTEM

We now move on to satisfiability algorithms and present in this chapter a new (and first-ever) proof theoretic framework for formally analyzing the core of the numerous practical implementations of such algorithms being developed today.

As discussed in Chapter 1, in recent years the task of deciding whether or not a given CNF formula is satisfiable has gone from a problem of theoretical interest to a practical approach for solving real-world problems. SAT procedures are now a standard tool for tasks such as hardware and software verification, circuit diagnosis, experiment design, planning, scheduling, etc.

The most surprising aspect of such relatively recent practical progress is that the best complete satisfiability testing algorithms remain variants of the DPLL procedure for backtrack search in the space of partial truth assignments (cf. Section 2.3). The key idea behind its efficacy is the pruning of the search space based on falsified clauses. Since its introduction in the early 1960's, the main improvements to DPLL have been smart branch selection heuristics such as by Li and Anbulagan [80], extensions like randomized restarts by Gomes et al. [58] and clause learning (cf. Section 2.3.2), and well-crafted data structures such as watched literals for fast unit propagation by Moskewicz et al. [88]. One can argue that of these, clause learning has been the most significant in scaling DPLL to realistic problems. This chapter attempts to understand the potential of clause learning and leads on to the next chapter which suggests practical ways of harnessing its power.

Clause learning grew out of work in artificial intelligence on explanation-based learning (EBL), which sought to improve the performance of backtrack search algorithms by generating explanations for failure (backtrack) points, and then adding the explanations as new constraints on the original problem. The results of de Kleer and Williams [46], Stallman and Sussman [103], Genesereth [55], and Davis [45] proved this approach to be quite effective. For general constraint satisfaction problems the explanations are called "conflicts" or "no goods"; in the case of Boolean CNF satisfiability, the technique becomes clause learning – the reason for failure is learned in the form of a "conflict clause" which is added to the set of given clauses. Through a series of papers and accompanying solvers, Bayardo Jr. and Schrag [13], Marques-Silva and Sakallah [84], Zhang [113], Moskewicz et al. [88], and Zhang et al. [115] showed that clause learning can be efficiently implemented and used to solve hard problems that cannot be approached by any other technique.

Despite its importance there has been little work on formal properties of clause

learning, with the goal of understanding its fundamental strengths and limitations. A likely reason for such inattention is that clause learning is a rather complex rule of inference – in fact, as we describe below, a complex family of rules of inference. A contribution of this work is a precise mathematical specification of various concepts used in describing clause learning.

Another problem in characterizing clause learning is defining a formal notion of the strength or power of a reasoning method. We address this issue by defining a new proof system called CL that captures the complexity of a clause learning algorithm on various classes of formulas. From the basic proof complexity point of view, only families of unsatisfiable formulas are of interest because only proofs of unsatisfiability can be large; minimum proofs of satisfiability are linear in the number of variables of the formula. In practice, however, many interesting formulas are satisfiable. To justify our approach of using a proof system CL, we refer to the work of Achlioptas, Beame, and Molloy [1] who have shown how negative proof complexity results for unsatisfiable formulas can be used to derive time lower bounds for specific inference algorithms, especially DPLL, running on satisfiable formulas as well. The key observation in their work is that before hitting a satisfying assignment, an algorithm is very likely to explore a large unsatisfiable part of the search space that corresponds to the first bad variable assignment.

Proof complexity does not capture everything we intuitively mean by the power of a reasoning system because it says nothing about how difficult it is to *find* shortest proofs. However, it is a good notion with which to begin our analysis because the size of proofs provides a lower bound on the running time of any implementation of the system. In the systems we consider, a branching function, which determines which variable to split upon or which pair of clauses to resolve, guides the search. A negative proof complexity result for a system tells us that a family of formulas is intractable even with a perfect branching function; likewise, a positive result gives us hope of finding a good branching function.

Recall from Chapter 2 that general resolution or RES is exponentially stronger than the DPLL procedure, the latter being exactly as powerful as tree-like resolution. Although RES can yield shorter proofs, in practice DPLL is better because it provides a more efficient way to search for proofs. The weakness of the tree-like proofs that DPLL generates is that they do not reuse derived clauses. The conflict clauses found when DPLL is augmented by clause learning correspond to reuse of derived clauses in the associated resolution proofs and thus to more general forms of resolution proofs. As a theoretical upper bound, all DPLL based approaches, including those involving clause learning, are captured by RES. An intuition behind the results we present is that the addition of clause learning moves DPLL closer to RES while retaining its practical efficiency.

It has been previously observed by Lynce and Marques-Silva [82] that clause learning can be viewed as adding resolvents to a tree-like resolution proof. However, we

provide the first mathematical proof that clause learning, viewed as a propositional proof system CL, is exponentially stronger than tree-like resolution. This explains, formally, the performance gains observed empirically when clause learning is added to DPLL based solvers. Further, we describe a generic way of extending families of formulas to obtain ones that exponentially separate CL from many refinements of resolution (see Section 2.2.2) known to be intermediate in strength between RES and tree-like resolution. These include regular and ordered resolution, and any other proper refinement of RES that behaves naturally under restrictions of variables, *i.e.*, for any formula F and restriction ρ on its variables, the shortest proof of $F|_\rho$ in the system is not any larger than a proof of F itself.

The argument used in our result above involves a new clause learning scheme called FirstNewCut that we introduce specifically for this purpose. Our second technical result shows that combining a slight variant of CL, denoted CL--, with unlimited restarts results in a proof system as strong as RES itself. This intuitively explains the speed-ups obtained empirically when randomized restarts are added to DPLL based solvers, with or without clause learning.

Remark 4.1. MacKenzie [83] has recently used arguments similar to those of Beame et al. [15] to prove that a variant of clause learning can simulate all of regular resolution.

4.1 Natural Proper Refinements of a Proof System

We discussed various refinements of resolution in Section 2.2.2. The concept of refinement applies to proof systems in general. We formalize below what it means for a refinement of a proof system to be natural and proper. Recall that the complexity $\mathcal{C}_S(F)$ of a formula F under a proof system S is the length of the shortest refutation of F in S.

Definition 4.1. For proof systems S and T, and a function $f : \mathbb{N} \to [1, \infty)$,

- S is *natural* if for any formula F and restriction ρ on its variables, $\mathcal{C}_S(F|_\rho) \leq \mathcal{C}_S(F)$.
- S is a *refinement* of T if proofs in S are also (restricted) proofs in T.
- A refinement S of T is $f(n)$*-proper* if there exists a witnessing family $\{F_n\}$ of formulas such that $\mathcal{C}_S(F_n) \geq f(n) \cdot \mathcal{C}_T(F_n)$. The refinement is *exponentially-proper* if $f(n) = 2^{n^{\Omega(1)}}$ and *super-polynomially-proper* if $f(n) = n^{\omega(1)}$.

Proposition 4.1. *Tree-like, regular, linear, positive, negative, semantic, and ordered resolution are natural refinements of* RES.

The following proposition follows from the separation results of Bonet et al. [27] and Alekhnovich et al. [3].

Proposition 4.2 ([27, 3]). *Tree-like, regular, and ordered resolution are exponentially-proper natural refinements of* RES.

4.2 A Formal Framework for Studying Clause Learning

Although many SAT solvers based on clause learning have been proposed and demonstrated to be empirically successful, a theoretical discussion of the underlying concepts and structures needed for our analysis is lacking. This section focuses on this formal framework.

For concreteness, we will use Algorithm 2.2 on page 17 as the basic clause learning algorithm. We state it again below for ease of reference. Recall that `DecideNextBranch` implements the variable selection process, `Deduce` apply unit propagation, `AnalyzeConflict` does clause learning upon reaching a conflict, and `Backtrack` unassigns variables up to the appropriate decision level computed during conflict analysis.

```
Input  : A CNF formula
Output : UNSAT, or SAT along with a satisfying assignment
begin
    while TRUE do
        DecideNextBranch
        while TRUE do
            status ← Deduce
            if status = CONFLICT then
                blevel ← AnalyzeConflict
                if blevel = 0 then return UNSAT
                Backtrack(blevel)
            else if status = SAT then
                Output current assignment stack
                return SAT
            else break
end
```

Algorithm 4.1: `DPLL-ClauseLearning`

4.2.1 Decision Levels and Implications

Although we have already defined concepts such as decision level and implied variable in the context of the DPLL procedure, we did so with the simpler-to-understand re-

cursive version of the algorithm in mind. We re-define these concepts for the iterative version with clause learning given above.

Variables assigned values through the actual branching process are called *decision* variables and those assigned values as a result of unit propagation are called *implied* variables. *Decision* and *implied literals* are analogously defined. Upon backtracking, the last decision variable no longer remains a decision variable and might instead become an implied variable depending on the clauses learned so far. The *decision level of a decision variable* x is one more than the number of current decision variables at the time of branching on x. The *decision level of an implied variable* is the maximum of the decision levels of decision variables used to imply it. The *decision level* at any step of the underlying DPLL procedure is the maximum of the decision levels of all current decision variables. Thus, for instance, if the clause learning algorithm starts off by branching on x, the decision level of x is 1 and the algorithm at this stage is at decision level 1.

A clause learning algorithm stops and declares the given formula to be UNSAT whenever unit propagation leads to a conflict at decision level zero, i.e., when no variable is currently branched upon. This condition will be referred to as a *conflict at decision level zero*.

4.2.2 Branching Sequence

We use the notion of branching sequence to prove an exponential separation between DPLL and clause learning. It generalizes the idea of a static *variable order* by letting the order differ from branch to branch in the underlying DPLL procedure. In addition, it also specifies which branch (TRUE or FALSE) to explore first. This can clearly be useful for satisfiable formulas, and can also help on unsatisfiable ones by making the algorithm learn useful clauses earlier in the process.

Definition 4.2. A *branching sequence* for a CNF formula F is a sequence $\sigma = (l_1, l_2, \ldots, l_k)$ of literals of F, possibly with repetitions. A DPLL based algorithm $\mathcal{A}$ on F *branches according to* σ if it always selects the next variable v to branch on in the literal order given by σ, skips v if v is currently assigned a value, and otherwise branches further by setting the chosen literal to FALSE and deleting it from σ. When σ becomes empty, $\mathcal{A}$ reverts back to its default branching scheme.

Definition 4.3. A branching sequence σ is *complete* for a formula F under a DPLL based algorithm $\mathcal{A}$ if $\mathcal{A}$ branching according to σ terminates before or as soon as σ becomes empty. Otherwise it is *incomplete* or *approximate*.

Clearly, how well a branching sequence works for a formula depends on the specifics of the clause learning algorithm used, such as its learning scheme and backtracking process. One needs to keep these in mind when generating the sequence. It is also important to note that while the size of a variable order is always the same as the

number of variables in the formula, that of an effective branching sequence is typically much more. In fact, the size of a branching sequence complete for an unsatisfiable formula F is equal to the size of an unsatisfiability proof of F, and when F is satisfiable, it is proportional to the time needed to find a satisfying assignment.

4.2.3 Implication Graph and Conflicts

Unit propagation can be naturally associated with an *implication graph* that captures all possible ways of deriving all implied literals from decision literals.

Definition 4.4. The *implication graph* G at a given stage of DPLL is a directed acyclic graph with edges labeled with sets of clauses. It is constructed as follows:

Step 1: Create a node for each decision literal, labeled with that literal. These will be the indegree zero source nodes of G.

Step 2: While there exists a known clause $C = (l_1 \vee \ldots l_k \vee l)$ such that $\neg l_1, \ldots, \neg l_k$ label nodes in G,

i. Add a node labeled l if not already present in G.

ii. Add edges $(l_i, l), 1 \leq i \leq k$, if not already present.

iii. Add C to the label set of these edges. These edges are thought of as grouped together and associated with clause C.

Step 3: Add to G a special "conflict" node $\bar{\Lambda}$. For any variable x that occurs both positively and negatively in G, add directed edges from x and $\neg x$ to $\bar{\Lambda}$.

Since all node labels in G are distinct, we identify nodes with the literals labeling them. Any variable x occurring both positively and negatively in G is a *conflict variable*, and x as well as $\neg x$ are *conflict literals*. G contains a *conflict* if it has at least one conflict variable. DPLL at a given stage has a *conflict* if the implication graph at that stage contains a conflict. A conflict can equivalently be thought of as occurring when the residual formula contains the empty clause Λ.

By definition, an implication graph may not contain a conflict at all, or it may contain many conflict variables and several ways of deriving any single literal. To better understand and analyze a conflict when it occurs, we work with a subgraph of an implication graph, called the *conflict graph* (see Figure 4.1), that captures only one among possibly many ways of reaching a conflict from the decision variables using unit propagation.

Definition 4.5. A *conflict graph* H is any subgraph of an implication graph with the following properties:

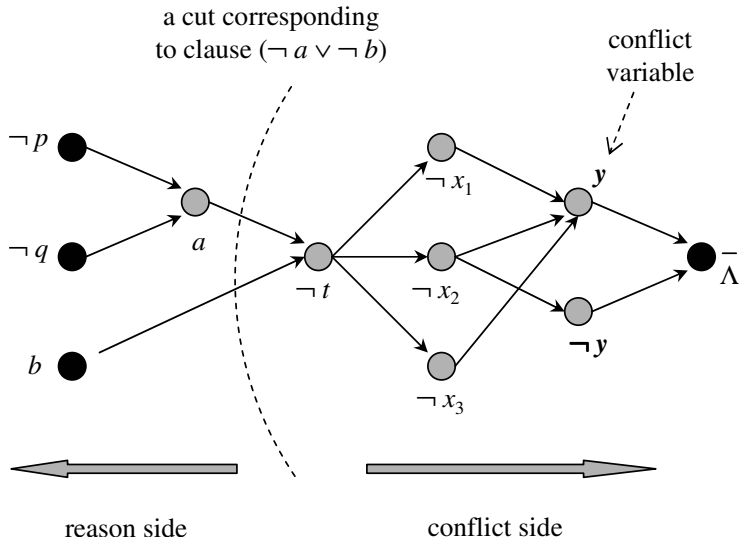

Figure 4.1: A conflict graph

(a) H contains $\bar{\Lambda}$ and exactly one conflict variable.

(b) All nodes in H have a path to $\bar{\Lambda}$.

(c) Every node l in H other than $\bar{\Lambda}$ either corresponds to a decision literal or has precisely the nodes $\neg l_1, \neg l_2, \ldots, \neg l_k$ as predecessors where $(l_1 \vee l_2 \vee \ldots \vee l_k \vee l)$ is a known clause.

While an implication graph may or may not contain conflicts, a conflict graph always contains exactly one. The choice of the conflict graph is part of the strategy of the solver. A typical strategy will maintain one subgraph of an implication graph that has properties (b) and (c) from Definition 4.5, but not property (a). This can be thought of as a *unique inference* subgraph of the implication graph. When a conflict is reached, this unique inference subgraph is extended to satisfy property (a) as well, resulting in a conflict graph, which is then used to analyze the conflict.

Conflict clauses

Recall that for a subset U of the vertices of a graph, the *edge-cut* (henceforth called a cut) corresponding to U is the set of all edges going from vertices in U to vertices not in U.

Consider the implication graph at a stage where there is a conflict and fix a conflict graph contained in that implication graph. Choose any cut in the conflict graph that has all decision variables on one side, called the *reason side*, and $\bar{\Lambda}$ as well as at least

one conflict literal on the other side, called the *conflict side*. All nodes on the reason side that have at least one edge going to the conflict side form a *cause* of the conflict. The negations of the corresponding literals forms the *conflict clause* associated with this cut.

4.2.4 Trivial Resolution and Learned Clauses

Definition 4.6. A resolution derivation $(C_1, C_2, \ldots, C_k)$ is *trivial* iff all variables resolved upon are distinct and each $C_i, i \geq 3$, is either an initial clause or is derived by resolving C_{i-1} with an initial clause.

A trivial derivation is tree-like, regular, linear, as well as ordered. As the following propositions show, trivial derivations correspond to conflicts in clause learning algorithms.

Proposition 4.3. *Let F be a CNF formula. If there is a trivial resolution derivation of a clause $C \notin F$ from F then setting all literals of C to* FALSE *leads to a conflict by unit propagation.*

Proof. Let $\pi = (C_1, C_2, \ldots, C_k = C)$ be a trivial resolution derivation of C from F. Let $C_k = (l_1 \vee l_2 \vee \ldots \vee l_q)$ and ρ be the partial assignment that sets all $l_i, 1 \leq i \leq q$, to FALSE. Assume without loss of generality that clauses in π are ordered so that all initial clauses precede any derived clause. We give a proof by induction on the number of derived clauses in π.

For the base case, π has only one derived clause, $C = C_k$. Assume without loss of generality that $C_k = (A \vee B)$ and C_k is derived by resolving two initial clauses $(A \vee x)$ and $(B \vee \neg x)$ on variable x. Since ρ falsifies C_k, it falsifies all literals of A, implying $x =$ TRUE by unit propagation. Similarly, ρ falsifies B, implying $x =$ FALSE and resulting in a conflict.

When π has at least two derived clauses, C_k, by triviality of π, must be derived by resolving $C_{k-1} \notin F$ with a clause in F. Assume without loss of generality that $C_{k-1} = (A \vee x)$ and the clause from F used in this resolution step is $(B \vee \neg x)$, where $C_k = (A \vee B)$. Since ρ falsifies $C = C_k$, it falsifies all literals of B, implying $x =$ FALSE by unit propagation. This in turn results in falsifying all literals of C_{k-1} because all literals of A are also set to FALSE by ρ. Now $(C_1, \ldots, C_{k-1})$ is a trivial resolution derivation of $C_{k-1} \notin F$ from F with one less derived clause than π, and all literals of C_{k-1} are falsified. By induction, this must lead to a conflict by unit propagation. □

Proposition 4.4. *Any conflict clause can be derived from initial and previously derived clauses using a trivial resolution derivation.*

Proof. Let σ be the cut in a fixed conflict graph associated with the given conflict clause. Let $V_{conflict}(\sigma)$ denote the set of variables on the conflict side of σ, but including the conflict variable only if it occurs both positively and negatively on the

conflict side. We will prove by induction on $|V_{conflict}(\sigma)|$ the stronger statement that the conflict clause associated with a cut σ has a trivial derivation from known (i.e. initial or previously derived) clauses resolving precisely on the variables in $V_{conflict}(\sigma)$.

For the base case, $V_{conflict}(\sigma) = \phi$ and the conflict side contains only $\bar{\Lambda}$ and a conflict literal, say x. Informally, this cut corresponds to the immediate cause of the conflict, namely, the single unit propagation step that led to the derivation of $\neg x$. More concretely, the clause associated with this cut consists of the node $\neg x$ which has an edge to $\bar{\Lambda}$, and nodes $\neg l_1, \neg l_2, \ldots, \neg l_k$, corresponding to a known clause $C_x = (l_1 \vee l_2 \vee \ldots \vee l_k \vee x)$, that each have an edge to x. The conflict clause for this cut is simply the known clause C_x itself, having a length zero trivial derivation.

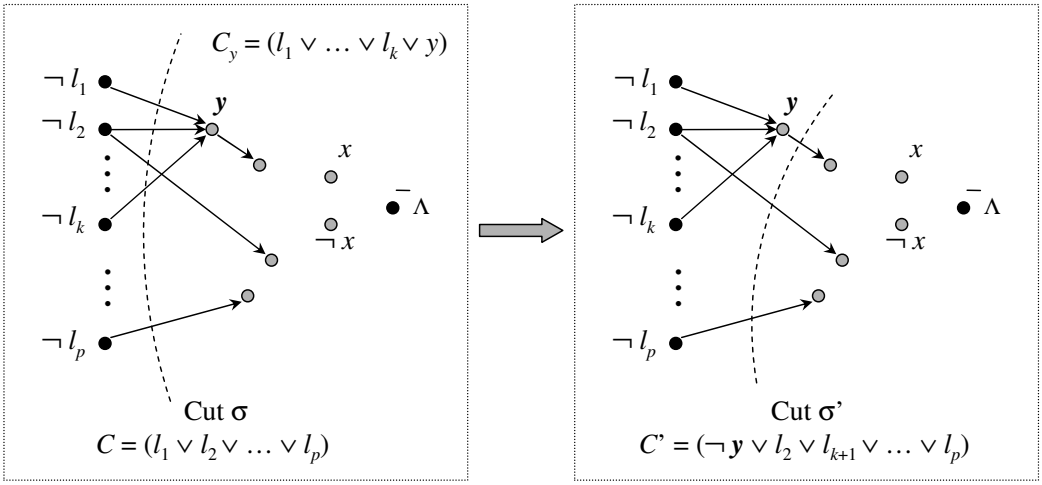

Figure 4.2: Deriving a conflict clause using trivial resolution. Resolving C' with C_y on variable y gives the conflict clause C.

When $V_{conflict}(\sigma) \neq \phi$, choose a node y on the conflict side all whose predecessors are on the reason side (see Figure. 4.2). Let the conflict clause be $C = (l_1 \vee l_2 \vee \ldots \vee l_p)$ and assume without loss of generality that the predecessors of y are $\neg l_1, \neg l_2, \ldots, \neg l_k$ for some $k \leq p$. By definition of unit propagation, $C_y = (l_1 \vee l_2 \vee \ldots \vee l_k \vee y)$ must be a known clause. Obtain a new cut σ' from σ by moving node y from the conflict side to the reason side. The new associated conflict clause must be of the form $C' = (\neg y \vee D)$, where D is a subclause of C. Now $V_{conflict}(\sigma') \subset V_{conflict}(\sigma)$. Consequently, by induction, C' must have a trivial resolution derivation from known clauses resolving precisely upon the variables in $V_{conflict}(\sigma')$. Recall that no variable occurs twice in a conflict graph except the conflict variable. Hence $V_{conflict}(\sigma')$ has precisely the variables of $V_{conflict}(\sigma)$ except y. Using this trivial derivation of C' and finally resolving C' with the known clause C_y on variable y gives us a trivial derivation of C from known clauses. This completes the inductive step. □

4.2.5 Learning Schemes

The essence of clause learning is captured by the *learning scheme* used to analyze and learn the "cause" of a failure. More concretely, different cuts in a conflict graph separating decision variables from a set of nodes containing $\bar{\Lambda}$ and a conflict literal correspond to different learning schemes (see Figure 4.3). One may also define learning schemes based on cuts not involving conflict literals at all such as a scheme suggested by Zhang et al. [115], but the effectiveness of such schemes is not clear. These will not be considered here.

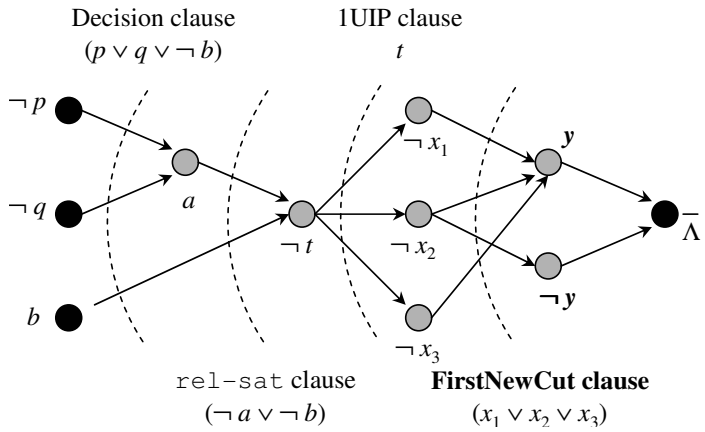

Figure 4.3: Various learning schemes

It is insightful to think of the *nondeterministic* scheme as the most general learning scheme. Here we select the cut nondeterministically, choosing, whenever possible, one whose associated clause is not already known. Since we can repeatedly branch on the same last variable, nondeterministic learning subsumes learning multiple clauses from a single conflict as long as the sets of nodes on the reason side of the corresponding cuts form a (set-wise) decreasing sequence. For simplicity, we will assume that only one clause is learned from any conflict.

In practice, however, we employ deterministic schemes. The *decision* scheme [115], for example, uses the cut whose reason side comprises all decision variables. rel-sat [13] uses the cut whose conflict side consists of all implied variables at the current decision level. This scheme allows the conflict clause to have exactly one variable from the current decision level, causing an automatic flip in its assignment upon backtracking.

This nice flipping property holds in general for all *unique implication points* (UIPs) [84]. A UIP of an implication graph is a node at the current decision level d such that any path from the decision variable at level d to the conflict variable as well as its negation must go through it. Intuitively, it is a *single* reason at level d that causes the conflict. Whereas rel-sat uses the decision variable as the obvious UIP, Grasp [84]

and zChaff [88] use *FirstUIP*, the one that is "closest" to the conflict variable. Grasp also learns multiple clauses when faced with a conflict. This makes it typically require fewer branching steps but possibly slower because of the time lost in learning and unit propagation.

The concept of UIP can be generalized to decision levels other than the current one. The *1UIP scheme* corresponds to learning the FirstUIP clause of the current decision level, the *2UIP scheme* to learning the FirstUIP clauses of both the current level and the one before, and so on. Zhang et al. [115] present a comparison of all these and other learning schemes and conclude that 1UIP is quite robust and outperforms all other schemes they consider on most of the benchmarks.

The FirstNewCut Scheme

We propose a new learning scheme called *FirstNewCut* whose ease of analysis helps us demonstrate the power of clause learning. We would like to point out that we use this scheme here only to prove our theoretical bounds using specific formulas. Its effectiveness on other formulas has not been studied yet. We would also like to point out that the experimental results that we present are for the 1UIP learning scheme, but can also be extended to certain other schemes, including FirstNewCut.

The key idea behind FirstNewCut is to make the conflict clause as relevant to the current conflict as possible by choosing a cut close to the conflict literals. This is what the FirstUIP scheme also tries to achieve in a slightly different manner. For the following definitions, fix a cut in a conflict graph and let S be the set of nodes on the reason side that have an edge to some node on the conflict side. S is the reason side *frontier* of the cut. Let C_S be the conflict clause associated with this cut.

Definition 4.7. *Minimization* of conflict clause C_S is the following process: while there exists a node $v \in S$ all of whose predecessors are also in S, move v to the conflict side, remove it from S, and repeat.

Definition 4.8. *FirstNewCut scheme*: Start with a cut whose conflict side consists of $\bar{\Lambda}$ and a conflict literal. If necessary, repeat the following until the associated conflict clause, after minimization, is not already known: choose a node on the conflict side, and move all its predecessors that lie on the reason side, other than those that correspond to decision variables, to the conflict side. Finally, learn the resulting new minimized conflict clause.

This scheme starts with the cut that is closest to the conflict literals and iteratively moves it back toward the decision variables until a new associated conflict clause is found. This backward search always halts because the cut with all decision variables on the reason side is certainly a new cut. Note that there are potentially several ways of choosing a literal to move the cut back, leading to different conflict clauses. The FirstNewCut scheme, by definition, always learns a clause not already known. This motivates the following:

Definition 4.9. A clause learning scheme is *non-redundant* if on a conflict, it always learns a clause not already known.

4.2.6 Clause Learning Proofs

The notion of clause learning proofs connects clause learning with resolution and provides the basis for our complexity bounds. If a given CNF formula F is unsatisfiable, clause learning terminates with a conflict at decision level zero. Since all clauses used in this final conflict themselves follow directly or indirectly from F, this failure of clause learning in finding a satisfying assignment constitutes a logical proof of unsatisfiability of F. We denote by CL the proof system consisting of all such proofs. Our bounds compare the sizes of proofs in CL with the sizes of (possibly restricted) resolution proofs. Recall that clause learning algorithms can use one of many learning schemes, resulting in different proofs.

Definition 4.10. A *clause learning (CL) proof* π of an unsatisfiable CNF formula F under learning scheme $\mathcal{S}$ and induced by branching sequence σ is the result of applying DPLL with unit propagation on F, branching according to σ, and using scheme $\mathcal{S}$ to learn conflict clauses such that at the end of this process, there is a conflict at decision level zero. The *size* of the proof, $size(\pi)$, is $|\sigma|$.

4.2.7 Fast Backtracking and Restarts

Most clause learning algorithms use *fast backtracking* or *conflict-directed backjumping* introduced by Stallman and Sussman [103], where one uses the conflict graph to undo not only the last branching decision but also all other recent decisions that did not contribute to the current conflict. In particular, the SAT solver zChaff that we will use for our experiments in Chapters 5 and 6 backtracks to decision level zero when it learns a unit clause. This property influences the structure of a branching sequence generation algorithm we will present in Section 5.2.1.

More precisely, the level that a clause learning algorithm employing this technique backtracks to is one less than the maximum of the decision levels of all decision variables (i.e. the *sources* of the conflict) present in the underlying conflict graph. Note that the current conflict might use clauses learned earlier as a result of branching on the apparently redundant variables. This implies that fast backtracking in general cannot be replaced by a "good" branching sequence that does not produce redundant branches. For the same reason, fast backtracking cannot either be replaced by simply learning the decision scheme clause. However, the results we present here are independent of whether or not fast backtracking is used.

Restarts, introduced by Gomes et al. [58] and further developed by Baptista and Marques-Silva [12], allow clause learning algorithms to arbitrarily restart their branching process from decision level zero. All clauses learned so far are retained and now

treated as additional initial clauses. As we will show, unlimited restarts, performed at the correct step, can make clause learning very powerful. In practice, this requires extending the strategy employed by the solver to include when and how often to restart. Unless otherwise stated, however, clause learning proofs in the rest of this chapter will be assumed to allow no restarts.

4.3 Clause Learning and Proper Natural Refinements of RES

We prove that the proof system CL, even without restarts, is stronger than *all* proper natural refinements of RES. We do this by first introducing a way of extending any CNF formula based on a given RES proof of it. We then show that if a formula F $f(n)$-separates RES from a natural refinement S, its extension $f(n)$-separates CL from S. The existence of such an F is guaranteed for all $f(n)$-proper natural refinements by definition.

4.3.1 The Proof Trace Extension

Definition 4.11. Let F be a CNF formula and π be a RES refutation of it. Let the last step of π resolve v with $\neg v$. Let $S = \pi \setminus (F \cup \{\neg v, \Lambda\})$. The *proof trace extension* $PT(F, \pi)$ of F is a CNF formula over variables of F and new trace variables t_C for clauses $C \in S$. The clauses of $PT(F, \pi)$ are all initial clauses of F together with a trace clause $(\neg x \vee t_C)$ for each clause $C \in S$ and each literal $x \in C$.

We first show that if a formula has a short RES refutation, then the corresponding proof trace extension has a short CL proof. Intuitively, the new trace variables allow us to simulate every resolution step of the original proof individually, without worrying about extra branches left over after learning a derived clause.

Lemma 4.1. *Suppose a formula F has a* RES *refutation π. Let $F' = PT(F, \pi)$. Then $\mathcal{C}_{\mathsf{CL}}(F') < size(\pi)$ when* CL *uses the FirstNewCut scheme and no restarts.*

Proof. Suppose π contains a derived clause C_i whose strict subclause C_i' can be derived by resolving two previously occurring clauses. We can replace C_i with C_i', do trivial simplifications on further derivations that used C_i and obtain a simpler proof π' of F. Doing this repeatedly will remove all such redundant clauses and leave us with a simplified proof no larger in size. Hence we will assume without loss of generality that π has no such clause.

Viewing π as a sequence of clauses, its last two elements must be a literal, say v, and Λ. Let $S = \pi \setminus (F \cup \{v, \Lambda\})$. Let $(C_1, C_2, \ldots, C_k)$ be the subsequence of π that has precisely the clauses in S. Note that $C_i = \neg v$ for some $i, 1 \leq i \leq k$. We claim that the branching sequence $\sigma = (t_{C_1}, t_{C_2}, \ldots, t_{C_k})$ induces a CL proof of F of size k using the FirstNewCut scheme. To prove this, we show by induction that after i branching steps, the clause learning procedure branching according to σ has

learned clauses $C_1, C_2, \ldots, C_i$, has trace variables $t_{C_1}, t_{C_2}, \ldots, t_{C_i}$ set to TRUE, and is at decision level i.

The base case for induction, $i = 0$, is trivial. The clause learning procedure is at decision level zero and no clauses have been learned. Suppose the inductive claim holds after branching step $i-1$. Let $C_i = (x_1 \vee x_2 \vee \ldots \vee x_l)$. C_i must have been derived in π by resolving two clauses $(A \vee y)$ and $(B \vee \neg y)$ coming from $F \cup \{C_1, C_2, \ldots, C_{i-1}\}$, where $C_i = (A \vee B)$. The i^{th} branching step sets $t_{C_i} =$ FALSE. Unit propagation using trace clauses $(\neg x_j \vee t_{C_i})$, $1 \leq j \leq l$, sets each x_j to FALSE, thereby falsifying all literals of A and B. Further unit propagation using $(A \vee y)$ and $(B \vee \neg y)$ implies y as well as $\neg y$, leading to a conflict. The cut in the conflict graph containing y and $\neg y$ on the conflict side and everything else on the reason side yields C_i as the FirstNewCut clause, which is learned from this conflict. The process now backtracks and flips the branch on t_{C_i} by setting it to TRUE. At this stage, the clause learning procedure has learned clauses $C_1, C_2, \ldots, C_i$, has trace variables $t_{C_1}, t_{C_2}, \ldots, t_{C_i}$ set to TRUE, and is at decision level i. This completes the inductive step.

The inductive proof above shows that when the clause learning procedure has finished branching on all k literals in σ, it will have learned all clauses in S. Adding to this the initial clauses F that are already known, the procedure will have as known clauses $\neg v$ as well as the two unit or binary clauses used to derive v in π. These immediately generate Λ in the residual formula by unit propagation using variable v, leading to a conflict at decision level k. Since this conflict does not use any decision variable, fast backtracking retracts all k branches. The conflict, however, still exists at decision level zero, thereby concluding the clause learning procedure and finishing the CL proof. □

Lemma 4.2. *Let S be an $f(n)$-proper natural refinement of* RES *whose weakness is witnessed by a family $\{F_n\}$ of formulas. Let $\{\pi_n\}$ be the family of shortest* RES *proofs of $\{F_n\}$. Let $\{F'_n\} = \{PT(F_n, \pi_n)\}$. For* CL *using the FirstNewCut scheme and no restarts, $\mathcal{C}_S(F'_n) \geq f(n) \cdot \mathcal{C}_{\mathsf{CL}}(F'_n)$.*

Proof. Let ρ_n the restriction that sets every trace variable of F'_n to TRUE. We claim that $\mathcal{C}_S(F'_n) \geq \mathcal{C}_S(F'_n|_{\rho_n}) = \mathcal{C}_S(F_n) \geq f(n) \cdot \mathcal{C}_{\mathsf{RES}}(F_n) > f(n) \cdot \mathcal{C}_{\mathsf{CL}}(F'_n)$. The first inequality holds because S is a natural proof system. The following equality holds because ρ_n keeps the original clauses of F_n intact and trivially satisfies all trace clauses, thereby reducing the initial clauses of F'_n to precisely F_n. The next inequality holds because S is an $f(n)$-proper refinement of RES. The final inequality follows from Lemma 4.1. □

This gives our first main result and its corollaries using Proposition 4.2:

Theorem 4.1. *For any $f(n)$-proper natural refinement S of* RES *and for* CL *using the FirstNewCut scheme and no restarts, there exist formulas $\{F_n\}$ such that $\mathcal{C}_S(F_n) \geq f(n) \cdot \mathcal{C}_{\mathsf{CL}}(F_n)$.*

Corollary 4.1. CL *can provide exponentially shorter proofs than tree-like, regular, and ordered resolution.*

Corollary 4.2. *Either* CL *is not a natural proof system or it is equivalent in strength to* RES*.*

Proof. As clause learning yields resolution proofs of unsatisfiable formulas, CL is a refinement of RES. Assume without loss of generality that it is an $f(n)$-proper refinement for some function f; this is true for instance when $f(n) = 1$ for all n. If CL is a natural proof system, Theorem 4.1 implies that there exists a family $\{F_n\}$ of formulas such that $\mathcal{C}_{\mathsf{CL}}(F_n) \geq f(n) \cdot \mathcal{C}_{\mathsf{CL}}(F_n)$. Since $f : \mathbb{N} \to [1, \infty)$ by the definition of $f(n)$-proper, $f(n)$ must be 1 for all n, proving the result. □

4.4 Clause Learning and General Resolution

We begin this section by showing that CL proofs, irrespective of the learning scheme, branching strategy, or restarts used, can be efficiently simulated by RES. In the reverse direction, we show that CL, with a slight variation and with unlimited restarts, can efficiently simulate RES in its full generality. The variant relates to the variables one is allowed to branch upon.

Lemma 4.3. *For any formula F over n variables and* CL *using any learning scheme and any number of restarts, $\mathcal{C}_{\mathsf{RES}}(F) \leq n \cdot \mathcal{C}_{\mathsf{CL}}(F)$.*

Proof. Given a CL proof π of F, a RES proof can be constructed by sequentially deriving all clauses that π learns, which includes the empty clause Λ. From Proposition 4.4, all these derivations are trivial and hence require at most n steps each. Consequently, the size of the resulting RES proof is at most $n \cdot size(\pi)$. Note that since we derive clauses of π individually, restarts in π do not affect the construction. □

Definition 4.12. Let CL-- denote the variant of CL where one is allowed to branch on a literal whose value is already set explicitly or because of unit propagation.

Of course, such a relaxation is useless in ordinary DPLL; there is no benefit in branching on a variable that doesn't even appear in the residual formula. However, with clause learning, such a branch can lead to an immediate conflict and allow one to learn a key conflict clause that would otherwise have not been learned. We will use this property to show that RES can be efficiently simulated by CL-- with enough restarts.

We first state a generalization of Lemma 4.3. CL-- can, by definition, do all that usual CL can, and is potentially stronger. The simulation of CL by RES can in fact be extended to CL-- as well. The proof goes exactly as the proof of Lemma 4.3 and uses the easy fact that Proposition 4.4 doesn't change even when one is allowed to branch on variables that are already set. This gives us:

Proposition 4.5. *For any formula F over n variables and* CL-- *using any learning scheme and any number of restarts, $\mathcal{C}_{\mathsf{RES}}(F) \leq n \cdot \mathcal{C}_{\mathsf{CL}\text{--}}(F)$.*

Lemma 4.4. *For any formula F over n variables and* CL *using any non-redundant scheme and at most $\mathcal{C}_{\mathsf{RES}}(F)$ restarts, $\mathcal{C}_{\mathsf{CL}\text{--}}(F) \leq n \cdot \mathcal{C}_{\mathsf{RES}}(F)$.*

Proof. Let π be a RES proof of F of size s. Assume without loss of generality as in the proof of Lemma 4.1 that π does not contain a derived clause C_i whose strict subclause C_i' can be derived by resolving two clauses occurring previously in π. The proof of this Lemma is very similar to that of Lemma 4.1. However, since we do not have trace variables to allow us to simulate each resolution step individually and independently, we use explicit restarts.

Viewing π as a sequence of clauses, its last two elements must be a literal, say v, and Λ. Let $S = \pi \setminus (F \cup \{v, \Lambda\})$. Let $(C_1, C_2, \ldots, C_k)$ be the subsequence of π that has precisely the clauses in S. Note that $C_i = \neg v$ for some $i, 1 \leq i \leq k$. For convenience, define an *extended branching sequence* to be a branching sequence in which certain places, instead of being literals, can be marked as restart points. Let σ be the extended branching sequence consisting of all literals of C_1, followed by a restart point, followed by all literals of C_2, followed by a second restart point, and so on up to C_k. We claim that σ induces a CL-- proof of F using any non-redundant learning scheme. To prove this, we show by induction that after the i^{th} restart point in σ, the CL-- procedure has learned clauses $C_1, C_2, \ldots, C_i$ and is at decision level zero.

The base case for induction, $i = 0$, is trivial. No clauses have been learned and the clause learning procedure is at decision level zero. Suppose the inductive claim holds after the $(i-1)^{st}$ restart point in σ. Let $C_i = (x_1 \vee x_2 \vee \ldots \vee x_l)$. C_i must have been derived in π by resolving two clauses $(A \vee y)$ and $(B \vee \neg y)$ coming from $F \cup \{C_1, C_2, \ldots, C_{i-1}\}$, where $C_i = (A \vee B)$. Continuing to branch according to σ till before the i^{th} restart point makes the CL-- procedure set all if $x_1, x_2, \ldots, x_l$ to FALSE. Note that when all literals appearing in A and B are distinct, the last branch on x_l here is on a variable that is already set because of unit propagation. CL--, however, allows this. At this stage, unit propagation using $(A \vee y)$ and $(B \vee \neg y)$ implies y as well as $\neg y$, leading to a conflict. The conflict graph consists of $\neg x_j$'s, $1 \leq j \leq l$, as the decision literals, y and $\neg y$ as implied literals, and $\bar{\Lambda}$. The only new conflict clause that can learned from this very simple conflict graph is C_i. Thus, C_i is learned using any non-redundant learning scheme and the i^{th} restart executed, as dictated by σ. At this stage, the CL-- procedure has learned clauses $C_1, C_2, \ldots, C_i$, and is at decision level zero. This completes the inductive step.

The inductive proof above shows that when the CL-- procedure has finished with the k^{th} restart in σ, it will have learned all clauses in S. Adding to this the initial clauses F that are already known, the procedure will have as known clauses $\neg v$ as well as the two unit or binary clauses used to derive v in π. These immediately generate Λ

in the residual formula by unit propagation using variable v, leading to a conflict at decision level zero, thereby concluding the clause learning procedure and finishing the CL-- proof. The bounds on the size of this proof and the number of restarts needed immediately follow from the definition of σ. □

Combining Lemma 4.4 with Proposition 4.5, we get

Theorem 4.2. CL-- *with any non-redundant scheme and unlimited restarts is polynomially equivalent to* RES.

Remark 4.2. Baptista and Marques-Silva [12] showed that by choosing the restart points in a smart way, CL together with restarts can be converted into a *complete* algorithm for satisfiability testing, i.e., for all unsatisfiable formulas given as input, it will halt and provide a proof of unsatisfiability. Our theorem makes a much stronger claim about a slight variant of CL, namely, with enough restarts, this variant can always find proofs of unsatisfiability that are as short as those of RES.

4.5 Discussion

In this chapter, we developed a mathematical framework for studying the most widely used class of complete SAT solvers, namely the one based on DPLL and clause learning. We studied clause learning from a proof complexity perspective and obtained two significant results for the proof system CL summarized in Figure 4.4. The first of these is that CL can provide exponentially smaller proofs than any proper natural refinement of RES. We derived from this as a corollary that CL is either not natural or is as powerful as RES itself. This is an interesting and somewhat surprising statement. The second noteworthy result is that a variant of clause learning with unrestricted restarts has exactly the same strength as RES.

Our argument used the notion of a proof trace extension of a formula which allowed one to convert a formula that is easy for RES to an extended formula that is easy for CL, at the same time retaining the hardness with respect to any natural refinement of RES. We also defined and made use of a new learning scheme, FirstNewCut.

Understanding where clause learning stands in relation to well studied proof systems should lead to better insights on why it works well on certain domains and fails on others. For instance, we will see in Chapter 5 an example of a domain (pebbling problems) where our results say that learning is necessary and sufficient, given a good branching order, to obtain sub-exponential solutions using clause learning based methods.

On the other hand, the connection with resolution also implies that any problem that contains as a sub-problem a formula that is inherently hard even for RES, such as the pigeonhole principle to be described in detail in Chapter 6, must be hard for any variant of clause learning. For such domains, theoretical results suggest practical extensions such as symmetry breaking and counting techniques for obtaining efficient

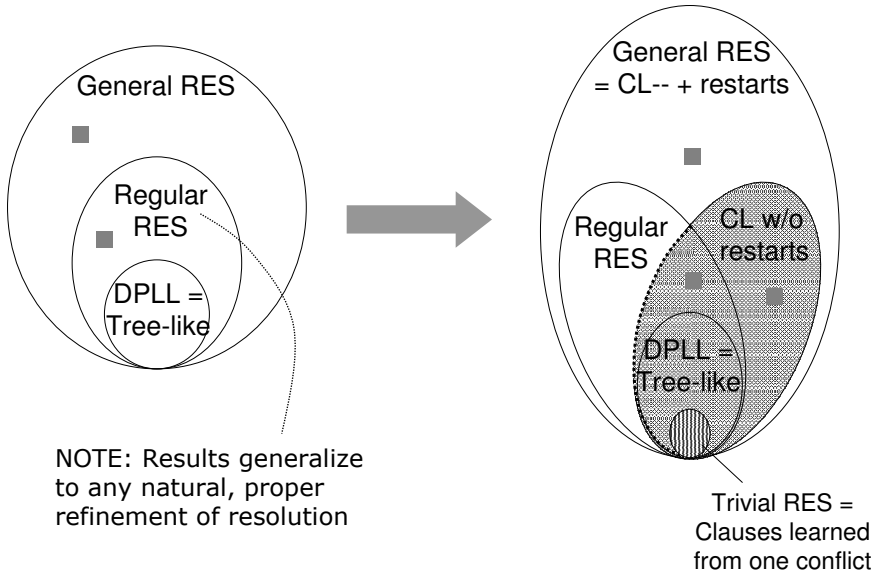

Figure 4.4: Results: Clause learning in relation to resolution

solutions. The first of these serves as a motivation for the work we will present in Chapter 6.

Chapter 5

USING PROBLEM STRUCTURE FOR EFFICIENT CLAUSE LEARNING

Given the results about the strengths and limitations of clause learning in Chapter 4, it is natural to ask how the understanding we gain through this kind of analysis may lead to practical improvement in SAT solvers. The theoretical bounds tell us the potential power of clause learning; they don't give us a way of *finding* short solutions when they exist. In order to leverage their strength, clause learning algorithms must follow the "right" variable order for their branching decisions for the underlying DPLL procedure. While a good variable order may result in a polynomial time solution, a bad one can make the process as slow as basic DPLL without learning. The present chapter addresses this problem of moving from analytical results to practical improvement. The approach we take is the use of the problem structure for guiding SAT solvers in their branch decisions.

Both random CNF formulas and those encoding various real-world problems are quite hard for current SAT solvers. However, while DPLL based algorithms with lookahead but no learning (such as satz by Li and Anbulagan [80]) and those that try only one carefully chosen assignment without any backtracks (such as SurveyProp by Mézard and Zecchina [87]) are our best tools for solving random formula instances, formulas arising from various real applications seem to require clause learning as a critical ingredient. The key thing that makes this second class of formulas different is their inherent structure, such as dependence graphs in scheduling problems, causes and effects in planning, and algebraic structure in group theory.

Most theoretical and practical problem instances of satisfiability problems originate, not surprisingly, from a higher level description, such as a Planning Domain Description Language (PDDL) specification for planning [51], timed automata or logic description for model checking, task dependency graph for scheduling, circuit description for VLSI, algebraic structure for group theory, and processor specification for hardware. Typically, this description contains more structure of the original problem than is visible in the flat CNF representation in DIMACS format [68] to which it is converted before being fed into a SAT solver. This structure can potentially be used to gain efficiency in the solution process.

Several ideas have been brought forward in the last decade for extracting structure after conversion into a CNF formula. These include the works of Giunchiglia et al. [56] and Ostrowski et al. [92] on exploiting variable dependency, Ostrowski et al. [92] on using constraint redundancy, Aloul et al. [6] and others on using symmetry, Brafman

[30] on exploiting binary clauses, and Amir and McIlraith [7] on using partitioning.

While all these approaches extract structure after conversion into a CNF formula, we argue that using the original higher level description itself to generate structural information is likely to be more effective. The latter approach, despite its intuitive appeal, remains largely unexplored, except for suggested use in bounded model checking by Shtrichman [101] and the separate consideration of cause variables and effect variables in planning by Kautz and Selman [71].

We further open this line of research by proposing an effective method for exploiting problem structure to guide the branching decision process of clause learning algorithms. Our approach uses the original high level problem description to generate not only a CNF encoding but also a *branching sequence* (recall Definition 4.2) that guides the SAT solver toward an efficient solution. This branching sequence serves as auxiliary structural information that was possibly lost in the process of encoding the problem as a CNF formula. It makes clause learning algorithms learn useful clauses instead of wasting time learning those that may not be reused in future at all.

We consider two families of formulas called the pebbling formulas and the GT_n formulas. The pebbling formulas, more commonly occurring in theoretical proof complexity literature such as in the works of Ben-Sasson et al. [22] and Beame et al. [15], can be thought of as representing precedence graphs in dependent task systems and scheduling scenarios. They can also be viewed as restricted planning problems. The GT_n formulas were introduced by Krishnamurthy [76] and have also been used frequently to obtain resolution lower bounds such as by Bonet and Galesi [28] and Alekhnovich et al. [3]. They represent a straightforward ordering principle on n elements. Although admitting a polynomial size solution, both pebbling and GT_n formulas are not so easy to solve in practice, as is indicated by our experimental results for unmodified zChaff.

We give an exact sequence generation algorithm for pebbling formulas, using the underlying pebbling graph as the high level description. We also give a much simpler but approximate branching sequence generation algorithm for GT_n formulas, utilizing their underlying ordering structure. Our sequence generators as presented work for the FirstUIP learning scheme (cf. Section 4.2.5), which is one of the best known. They can also be extended to other schemes, including FirstNewCut. Our empirical results are based on our extension of the SAT solver zChaff.

We show that the use of branching sequences produced by our generators leads to exponential empirical speedups for the class of grid and randomized pebbling formulas. We also report significant gains obtained for the class of GT_n formulas.

From a broader perspective, our results for pebbling and GT_n formulas serve as a proof of concept that analysis of problem structure can be used to achieve dramatic improvements even in the current best clause learning based SAT solvers.

5.1 Two Interesting Families of Formulas

We begin by describing in detail the two families of CNF formulas from the proof complexity literature mentioned above.

5.1.1 Pebbling Formulas

Pebbling formulas are unsatisfiable CNF formulas whose variations have been used repeatedly in proof complexity to obtain theoretical separation results between different proof systems such as by Ben-Sasson et al. [22] and Beame et al. [15]. The version we will use in this chapter is known to be easy for regular resolution but hard for tree-like resolution [22], and hence for DPLL without learning. We use these formulas to show how one can utilize problem structure to allow clause learning algorithms to handle much bigger problems than they otherwise can.

Pebbling formulas represent the constraints for sequencing a system of tasks that need to be completed, where each task can be accomplished in a number of alternative ways. The associated pebbling graph has a node for each task, labeled by a disjunction of variables representing the different ways of completing the task. Placing a pebble on a node in the graph represents accomplishing the corresponding task. Directed edges between nodes denote task precedence; a node is pebbled when all of its predecessors in the graph are pebbled. The pebbling process is initialized by placing pebbles on all indegree zero nodes. This corresponds to completing those tasks that do not depend on any other.

Formally, a *Pebbling formula* Pbl_G is an unsatisfiable CNF formula associated with a directed, acyclic *pebbling graph* G (see Figure 5.1). Nodes of G are labeled with disjunctions of variables, i.e. with clauses. A node labeled with clause C is thought of as *pebbled* under a (partial) variable assignment σ if $C|_\sigma = \text{TRUE}$. Pbl_G contains three kinds of clauses – precedence clauses, source clauses and target clauses. For instance, a node labeled $(x_1 \vee x_2)$ with three predecessors labeled $(p_1 \vee p_2 \vee p_3)$, q_1 and $(r_1 \vee r_2)$ generates six precedence clauses $(\neg p_i \vee \neg q_j \vee \neg r_k \vee x_1 \vee x_2)$, where $i \in \{1,2,3\}$, $j \in \{1\}$ and $k \in \{1,2\}$. The precedence clauses imply that if all predecessors of a node are pebbled, then the node itself must also be pebbled. For every indegree zero *source node* s of G, Pbl_G contains the clause labeling s as a source clause. Thus, Pbl_G implies that all source nodes are pebbled. For every outdegree zero *target node* of G labeled, say, $(t_1 \vee t_2)$, Pbl_G has target clauses $\neg t_1$ and $\neg t_2$. These imply that target nodes are not pebbled, and provide a contradiction.

Grid pebbling formulas are based on simple pyramid-shaped layered pebbling graphs with distinct variable labels, 2 predecessors per node, and disjunctions of size 2 (see Figure 5.1). *Randomized pebbling formulas* are more complicated and correspond to random pebbling graphs. We only consider pebbling graphs where no variable appears more than once in any node label. In general, random pebbling graphs allow multiple target nodes. However, the more the targets, the easier it is to produce a

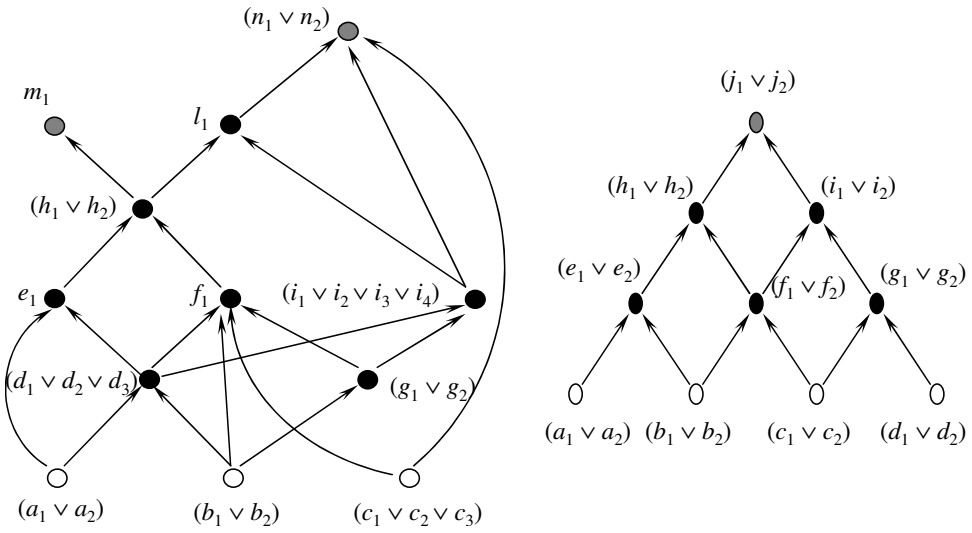

Figure 5.1: A general pebbling graph with distinct node labels, and a 4-layer grid pebbling graph

contradiction because we can focus only on the (relatively smaller) subgraph under the lowest target. Hence, for our experiments, we add a simple grid structure at the top of randomly generated pebbling formulas to make them have exactly one target.

All pebbling formulas with a single target are minimally unsatisfiable, i.e. any strict subset of their clauses admits a satisfying assignment. For each formula Pbl_G we use for our experiments, we also use a satisfiable version of it, called Pbl_G^{SAT}, obtained by randomly choosing a clause of Pbl_G and deleting it. When G is viewed as a task graph, Pbl_G^{SAT} corresponds to a task system with a single fault, and finding a satisfying assignment for it corresponds to locating the fault.

5.1.2 The GT_n Formulas

The GT_n formulas are unsatisfiable CNF formulas based on the ordering principle that any partial order on the set $\{1, 2, \ldots, n\}$ must have a maximal element. They were first considered by Krishnamurthy [76] and later used by Bonet and Galesi [28] to show the optimality of the size-width relationship of resolution proofs. Recently, Alekhnovich et al. [3] used a variation, called GT'_n, to show an exponential separation between RES and regular resolution.

The variables of GT_n are $x_{i,j}$ for $i, j \in [n], i \neq j$, which should be thought of as the binary predicate $i \succ j$. Clauses $(\neg x_{i,j} \vee \neg x_{j,i})$ ensure that $\succ$ is anti-symmetric and $(\neg x_{i,j} \vee \neg x_{j,k} \vee x_{i,k})$ ensure that $\succ$ is transitive. This makes $\succ$ a partial order on $[n]$. *Successor clauses* $(\vee_{k \neq j} x_{k,j})$ provide the contradiction by saying that every element j has a successor in $[n] \setminus \{j\}$, which is clearly false for the maximal elements

of $[n]$ under the ordering $\succ$.

These formulas, although capturing a simple mathematical principle, are empirically difficult for many SAT solvers including zChaff. We employ our techniques to improve the performance of zChaff on these formulas. We use for our experiments the unsatisfiable version GT_n described above as well as a satisfiable version GT_n^{SAT} obtained by deleting a randomly chosen successor clause. The reason we consider these ordering formulas in addition to seemingly harder pebbling formulas is that the latter admit short tree-like proofs in certain extensions of RES whereas the former seem to critically require reuse of derived or learned clauses for short refutations. We elaborate on this in Section 5.2.2.

5.2 From Analysis to Practice

The complexity bounds established in the previous chapter indicate that clause learning is potentially quite powerful, especially when compared to ordinary DPLL. However, natural choices such as which conflict graph to choose, which cut in it to consider, in what order to branch on variables, and when to restart, make the process highly nondeterministic. These choices must be made deterministically (or randomly) when implementing a clause learning algorithm. To harness its full potential on a given problem domain, one must, in particular, implement a learning scheme and a branch decision process suited to that domain.

5.2.1 Solving Pebbling Formulas

As a first step toward our grand goal of translating theoretical understanding into effective implementations, we show, using pebbling problems as a concrete example, how one can utilize high level problem descriptions to generate effective branching strategies for clause learning algorithms. Specifically, we use insights from our theoretical analysis to give an efficient algorithm to generate an effective branching sequence for unsatisfiable as well as satisfiable pebbling formulas (see Section 5.1.1). This algorithm takes as input the underlying pebbling graph (which is the high level description of the pebbling problem), and not the CNF pebbling formula itself. As we will see in Section 5.2.3, the generated branching sequence gives exponential empirical speedup over zChaff for both grid and randomized pebbling formulas.

zChaff, despite being one of the current best clause learners, by default does not perform very well on seemingly simple pebbling formulas, even on the uniform grid version. Although clause learning should ideally need only polynomial time to solve these problem instances (in fact, linear time in the size of the formula), choosing a good branching order is critical for this to happen. Since nodes are intuitively pebbled in a bottom up fashion, we must also learn the right clauses (i.e. clauses labeling the nodes) in a bottom up order. However, branching on variables labeling lower nodes before those labeling higher ones prevents any DPLL based learning algorithm from

backtracking the right distance and proceeding further in an effective manner. To make this clear, consider the general pebbling graph of Figure 5.1. Suppose we branch on and set d_1, d_2, d_3 and a_1 to FALSE. This will lead to a contradiction through unit propagation by implying a_2 is TRUE and b_1 and b_2 are both FALSE. We will learn $(d_1 \vee d_2 \vee d_3 \vee \neg a_2)$ as the associated 1UIP conflict clause and backtrack. There will still be a contradiction without any further branches, making us learn $(d_1 \vee d_2 \vee d_3)$ and backtrack. At this stage, we will have learned the correct clause but will be *stuck* with two branches on d_1 and d_2. Unless we had branched on e_1 before branching on the variables of node d, we will not be able to learn e_1 as the clause corresponding to the next higher pebbling node.

Automatic Sequence Generation: `PebSeq1UIP`

Algorithm 5.1, `PebSeq1UIP`, describes a way of generating a good branching sequence for pebbling formulas. It works on any pebbling graph G with distinct label variables as input and produces a branching sequence linear in the size of the associated pebbling formula. In particular, the sequence size is linear in the number of variables as well when the indegree as well as label size are bounded by a constant.

`PebSeq1UIP` starts off by handling the set U of all nodes labeled with unit clauses. Their outgoing edges are deleted and they are treated as pseudo sources. The procedure first generates a branching sequence for non-target nodes in U in increasing order of height. The key here is that when zChaff learns a unit clause, it fast-backtracks to decision level zero, effectively restarting at that point. We make use of this fact to learn these unit clauses in a bottom up fashion, unlike the rest of the process which proceeds top down in a depth-first way.

```
Input  : Pebbling graph G with no repeated labels
Output : Branching sequence for Pbl_G for the 1UIP learning scheme
begin
    foreach v in BottomUpTraversal(G) do
        v.height ← 1 + max_{u∈v.preds} {u.height}
        Sort(v.preds, increasing order by height)
    // first handle unit clause labeled nodes and generate their sequence
    U ← {v ∈ G.nodes : |v.labels| = 1}
    G.edges ← G.edges \ {(u, v) ∈ G.edges : u ∈ U}
    Add to G.sources any new nodes with now 0 preds
    Sort(U, increasing order by height)
    foreach u ∈ U \ G.targets do
        Output u.label
        PebSubseq1UIPWrapper(u)
    // now add branching sequence for targets by increasing height
    Sort(G.targets, increasing order by height)
    foreach t ∈ G.targets do PebSubseq1UIPWrapper(t)
end
PebSubseq1UIPWrapper(node v) begin
    if |v.preds| > 0 then PebSubseq1UIP(v, |v.preds|)
end
PebSubseq1UIP(node v, int i) begin
    u ← v.preds[i]
    if i = 1 then
        // this is the lowest predecessor
        if !u.visited and u ∉ G.sources then
            u.visited ← TRUE
            PebSubseq1UIPWrapper(u)
        return
    Output u.labels \ {u.lastLabel}
    if !u.visitedAsHigh and u ∉ G.sources then
        u.visitedAsHigh ← TRUE
        Output u.lastLabel
        if !u.visited then
            u.visited ← TRUE
            PebSubseq1UIPWrapper(u)
    PebSubseq1UIP(v, i − 1)
    for j ← (|u.labels| − 2) downto 1 do
        Output u.labels[1], ..., u.labels[j]
        PebSubseq1UIP(v, i − 1)
    PebSubseq1UIP(v, i − 1)
end
```

Algorithm 5.1: `PebSeq1UIP`, generating branching sequence for pebbling formulas

`PebSeq1UIP` now adds branching sequences for the targets. Note that for an unsatisfiability proof, we only need the sequence corresponding to the first (lowest) target. However, we process all targets so that this same sequence can also be used when the

formula is made satisfiable by deleting enough clauses. The subroutine `PebSubseq1UIP` runs on a node v, looking at its i^{th} predecessor u in increasing order by height. No labels are output if u is the lowest predecessor; the negations of these variables will be indirectly implied during clause learning. However, it is recursed upon if not previously visited. This recursive sequence results in learning something close to the clause labeling this lowest node, but not quite that exact clause. If u is a higher predecessor (it will be marked as $visitedAsHigh$), `PebSubseq1UIP` outputs all but one variables labeling u. If u is not a source and has not previously been visited as high, the last label is output as well, and u recursed upon if necessary. This recursive sequence results in learning the clause labeling u. Finally, `PebSubseq1UIP` generates a recursive pattern, calling the subroutine with the next lower predecessor of v. The precise structure of this pattern is dictated by the 1UIP learning scheme and fast backtracking used in zChaff. Its size is exponential in the degree of v with label size as the base.

The Grid Case. It is insightful to look at the simplified version of the sequence generation algorithm that works only for grid pebbling formulas. This is described below as Algorithm 5.2, `GridPebSeq1UIP`. Note that both predecessors of any node are at the same level for grid pebbling graphs and need not be sorted by height. There are no nodes labeled with unit clauses and there is exactly one target node t, simplifying the whole algorithm to a single call to `PebSubseq1UIP(t,2)` in the notation of Algorithm 5.1. The last *for* loop of this procedure and the recursive call that follows it are now redundant. We combine the original wrapper method and the calls to `PebSubseq1UIP` with parameters $(v, 2)$ and $(v, 1)$ into a single method `GridPebSubseq1UIP` with parameter v.

The resulting branching sequence can actually be generated by a simple depth first traversal (DFS) of the grid pebbling graph, printing no labels for the nodes on the rightmost path (including the target node), both labels for internal nodes, and one arbitrarily chosen label for source nodes. However, this resemblance to DFS is a somewhat misleading coincidence. The resulting sequence diverges substantially from DFS order as soon as label size or indegree of some nodes is changed. For the 10 node depth 4 grid pebbling graph shown in Figure 5.1, the branching sequence generated by the algorithm is $h_1, h_2, e_1, e_2, a_1, b_1, f_1, f_2, c_1$. Here, for instance, b_1 is generated after a_1 not because it labels the right (second) predecessor of node e but because it labels the left (first) predecessor of node f. Similarly, f_1 and f_2 appear after the subtree rooted at h as left predecessors of node i rather than as right predecessors of node h.

Example 5.1. To clarify the algorithm for the general case, we describe its execution on a small example. Let G be the pebbling graph in Figure 5.2. Denote by t the node labeled $(t_1 \vee t_2)$, and likewise for other nodes. Nodes c, d, f and g are at height 1, nodes a and e at height 2, node b at height 3, and node t at height 4. $U = \{a, b\}$. The edges (a, t) and (b, t) originating from these unit clause labeled nodes are removed, and t, with no predecessors anymore, is added to the list of sources. We output the

```
Input  : Grid pebbling graph G with target node t
Output : Branching sequence for Pbl_G for the 1UIP learning scheme
begin
  GridPebSubseq1UIP(t)
end

GridPebSubseq1UIP(node v) begin
  if v ∈ G.sources then return
  u ← v.preds.left
  Output u.firstLabel
  if !u.visitedAsLeft and u ∉ G.sources then
    u.visitedAsLeft ← TRUE
    Output u.secondLabel
    if !u.visited then
      u.visited ← TRUE
      GridPebSubseq1UIP(u)
  u ← v.preds.right
  if !u.visited and u ∉ G.sources then
    u.visited ← TRUE
    GridPebSubseq1UIP(u)
end
```

Algorithm 5.2: `GridPebSeq1UIP`, generating branching sequence for grid pebbling formulas

label of the non-target unit nodes in U in increasing order of height, and recurse on each of them in order, i.e. we output a_1, setting $B = (a_1)$, call `PebSubseq1UIPWrapper` on a, and then repeat this process for b. This is followed by a recursive call to `PebSubseq1UIPWrapper` on the target node t.

The call `PebSubseq1UIPWrapper` on a in turn invokes `PebSubseq1UIP` with parameters $(a, 2)$. This sorts the predecessors of a in increasing order of height to, say, d, c, with d being the lowest predecessor. v is set to a and u is set to the second predecessor c. We output all but the last label of u, i.e. of c, making the current branching sequence $B = (a_1, c_1)$. Since u is a source, nothing more needs to be done for it and we make a recursive call to `PebSubseq1UIP` with parameters $(a, 1)$. This sets u to d, which is the lowest predecessor and requires nothing to be done because it is also a source. This finishes the sequence generation for a, ending at $B = (a_1, c_1)$. After processing this part of the sequence, zChaff will have a as a learned clause.

We now output b_1, the label of the unit clause b. The call, `PebSubseq1UIPWrapper` on b, proceeds similarly, setting predecessor order as (d, f, e), with d as the lowest predecessor. Procedure `PebSubseq1UIP` is called first with parameters $(b, 3)$, setting u to e. This adds all but the last label of e to the branching sequence, making it $B = (a_1, c_1, b_1, e_1, e_2)$. Since this is the first time e is being visited as high,

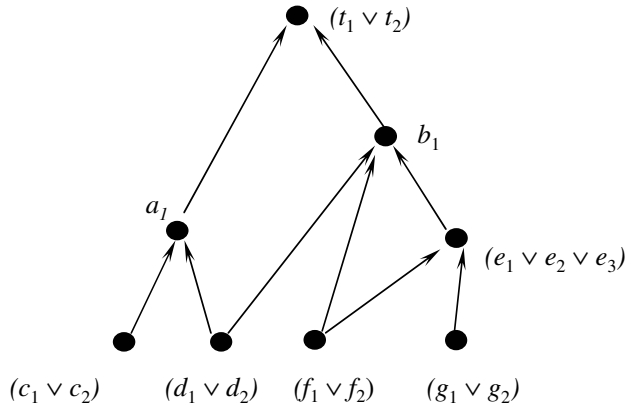

Figure 5.2: A simple pebbling graph to illustrate branch sequence generation

its last label is also added, making $B = (a_1, c_1, b_1, e_1, e_2, e_3)$, and it is recursed upon with `PebSubseq1UIPWrapper(e)`. This recursion extends the sequence to $B = (a_1, c_1, b_1, e_1, e_2, e_3, f_1)$. After processing this part of B, zChaff will have both a and $(e_1 \vee e_2 \vee e_3)$ as learned clauses. Getting to the second highest predecessor f of b, which happens to be a source, we simply add another f_1 to B. Finally, we get to the third highest predecessor d of b, which happens to be the lowest as well as a source, thus requiring nothing to be done. Coming out of the recursion, back to $u = f$, we generate the pattern given by the last `for` loop, which is empty because the label size of f is only 2. Coming out once more of the recursion to $u = e$, the `for` loop pattern generates e_1, f_1 and is followed by a call to `PebSubseq1UIP` with the next lower predecessor f as the second parameter, which generates f_1. This makes the current sequence $B = (a_1, c_1, b_1, e_1, e_2, e_3, f_1, f_1, e_1, f_1, f_1)$. After processing this, zChaff will also have b as a learned clause.

The final call to `PebSubseq1UIPWrapper` with parameter t doesn't do anything because both predecessors of t were removed in the beginning. Since both a and b have been learned, zChaff will have an immediate contradiction at decision level zero. This gives us the complete branching sequence $B = (a_1, c_1, b_1, e_1, e_2, e_3, f_1, f_1, e_1, f_1, f_1)$ for the pebbling formula Pbl_G.

Complexity of Sequence Generation

Let graph G have n nodes, indegree of non-source nodes between d_{min} and d_{max}, and label size between l_{min} and l_{max}. For simplicity of analysis, we will assume that $l_{min} = l_{max} = l$ and $d_{min} = d_{max} = d$ ($l = d = 2$ for a grid graph).

Let us first compute the size of the pebbling formula associated with G. The running time of `PebSeq1UIP` and the size of the branching sequence generated will be given in terms of this size. The number of clauses in the pebbling formula Pbl_G

is roughly nl^d. Taking clause sizes into account, the size of the formula, $|Pbl_G|$, is roughly $n(l+d)l^d$. Note that the size of the CNF formula itself grows exponentially with the indegree and gets worse as label size increases. The best case is when G is the grid graph, where $|Pbl_G| = \Theta(n)$. This explains the degradation in performance of zChaff, both original and modified, as we move from grid graphs to random graphs (see section 5.2.3). Since we construct Pbl_G^{SAT} by deleting exactly one randomly chosen clause from Pbl_G (see Section 5.1.1), the size $|Pbl_G^{SAT}|$ of the satisfiable version is also essentially the same.

Let us now compute the running time of `PebSeq1UIP`. Initial computation of heights and predecessor sorting takes time $\Theta(nd \log d)$. Assuming n_u unit clause labeled nodes and n_t target nodes, the remaining node sorting time is $\Theta(n_u \log n_u + n_t \log n_t)$. Since `PebSubseq1UIPWrapper` is called at most once for each node, the total running time of `PebSeq1UIP` is $\Theta(nd \log d + n_u \log n_u + n_t \log n_t + nT_{wrapper})$, where $T_{wrapper}$ denotes the running time of `PebSubseq1UIP-Wrapper` without taking into account recursive calls to itself. When n_u and n_t are much smaller than n, which we will assume as the typical case, this simplifies to $\Theta(nd \log d + nT_{wrapper})$. If $T(v,i)$ denotes the running time of `PebSubseq1UIP(v,i)`, again without including recursive calls to the wrapper method, then $T_{wrapper} = T(v,d)$. However, $T(v,d) = lT(v,d-1) + \Theta(l)$, which gives $T_{wrapper} = T(v,d) = \Theta(l^{d+1})$. Substituting this back, we get that the running time of `PebSeq1UIP` is $\Theta(nl^{d+1})$, which is about the same as $|Pbl_G|$.

Finally, we consider the size of the branching sequence generated. Note that for each node, most of its contribution to the sequence is from the recursive pattern generated near the end of `PebSubseq1UIP`. Let $Q(v,i)$ denote this contribution. $Q(v,i) = (l-2)(Q(v,i-1) + \Theta(l))$, which gives $Q(v,i) = \Theta(l^{d+2})$. Hence, the size of the sequence generated is $\Theta(nl^{d+2})$, which again is about the same as $|Pbl_G|$.

Theorem 5.1. *Given a pebbling graph G with label size at most l and indegree of non-source nodes at most d, Algorithm 5.1,* `PebSeq1UIP`*, produces a branching sequence σ of size at most S in time $\Theta(dS)$, where $S = |Pbl_G| \approx |Pbl_G^{SAT}|$. Moreover, the sequence σ is complete for Pbl_G as well as for Pbl_G^{SAT} under any clause learning algorithm using fast backtracking and 1UIP learning scheme (such as* zChaff*).*

Proof. The size and running time bounds follow from the previous discussion in this section. That this sequence is complete can be verified by a simple hand calculation simulating clause learning with fast backtracking and 1UIP learning scheme. □

5.2.2 Solving GT_n Formulas

We now consider automatic sequence generation for the ordering formulas introduced in Section 5.1.2. Since these formulas, like pebbling formulas, also originate in the proof complexity literature and in fact represent a problem that is structurally simpler to state and reason about than the pebbling problem, one may wonder what this section adds to the chapter. The answer lies in two key motivations. First, as we

will see, the automatically generated sequence for these formulas, unlike pebbling formulas, is extremely simple in nature and incomplete as a branching sequence. Nonetheless, it provides dramatic improvement in performance. Second, there is reason to believe that pebbling formulas may be easier than the GT_n formulas for resolution type proof systems. We formalize the intuition behind this in the next few paragraphs.

While pebbling formulas are not so easy to solve by popular SAT solvers, they may not inherently be too difficult for clause learning algorithms. In fact, even without any learning, they admit tree-like proofs under a somewhat stronger related proof system called RES(k) for large enough k as shown by Esteban et al. [50]:

Proposition 5.1 ([50]). *Pbl_G has a tree-like* RES(k) *refutation of size $O(|G|)$, where k is the maximum width of a clause labeling a node of G. In particular, when G is a grid graph with n nodes, Pbl_G has a tree-like* RES(2) *refutation of size $O(n)$.*

Here RES(k) denotes the extension of RES defined by Krajíček [75] that allows resolving, instead of clauses, disjunctions of conjunctions of up to k literals. Recall that clauses are disjunctions of literals, i.e., RES(1) is simply RES. Atserias and Bonet [10] discuss how a tree-like RES(k) proof of a formula F can be converted into a not-too-large tree-like RES proof of a related formula $F(k)$ over a few extra variables. More precisely, their result and Proposition 5.1 together imply that the addition of natural extension variables corresponding to k-conjunctions of variables of Pbl_G leads to a tree-like RES proof of size $O(|G| \cdot k)$ of a related pebbling formulas $Pbl_G(k)$.

For GT_n formulas, however, no such short tree-like proofs are known in RES(k) for any k. Reusing derived clauses (equivalently, learning clauses with DPLL) seems to be the key to finding short proofs of GT_n. This makes them a good candidate for testing clause learning based SAT solvers. Our experiments indicate that GT_n formulas, despite their simplicity, are quite hard for zChaff with its default parameter settings. Using a good branching sequence based on the ordering structure underlying these formulas leads to significant performance gains.

Automatic Sequence Generation: `GTnSeq1UIP`

Since there is exactly one, well defined, unsatisfiable GT formula for a fixed parameter n, it is not surprising that the approximate branching sequence given in Figure 5.3 that we use for it is straightforward. However, the fact that the same branching sequence works well for the satisfiable version of the GT_n formulas, obtained by deleting a randomly chosen successor clause, is worth noting.

Recall that `PebSeq1UIP` was a fairly complex algorithm that generated a perfect branching sequence for randomized pebbling graphs. In contrast, Algorithm 5.3, `GTnSeq1UIP`, for generating the branching sequence in Figure 5.3 is nearly trivial. As remarked earlier, it produces an incomplete sequence (see Definition 4.3) that nonetheless boosts performance in practice.

$$
\begin{array}{cccccc}
- & x_{2,1} & x_{3,1} & x_{4,1} & \dots & x_{n-1,1} \\
x_{1,2} & - & x_{3,2} & x_{4,2} & \dots & x_{n-1,2} \\
x_{1,3} & x_{2,3} & - & x_{4,3} & \dots & x_{n-1,3} \\
x_{1,4} & x_{2,4} & x_{3,4} & - & \dots & x_{n-1,4} \\
\vdots & & & & & \\
x_{1,n} & x_{2,n} & x_{3,n} & x_{4,n} & \dots & - \\
x_{1,n} & x_{2,n} & x_{3,n} & x_{4,n} & \dots & x_{n-1,n}
\end{array}
$$

Figure 5.3: Approximate branching sequence for GT_n formulas. The sequence goes top-down, and left to right within each row. '$-$' corresponds to a non-existent variable $x_{i,i}$.

Input : A natural number n
Output : Branching sequence for GT_n for the 1UIP learning scheme
begin
  for $i = 1$ *to* n **do**
    for $j = 1$ *to* $(n-1)$ **do**
      if $i \neq j$ **then** Output $x_{j,i}$
end

Algorithm 5.3: `GTnSeq1UIP`, generating branching sequence for GT_n formulas

5.2.3 Experimental Results

We conducted experiments on a Linux machine with a 1600 MHz AMD Athelon processor, 256 KB cache and 1024 MB RAM. Time limit was set to 6 hours and memory limit to 512 MB; the program was set to abort as soon as either of these was exceeded. We took the base code of zChaff [88], version 2001.6.15, and modified it to incorporate a branching sequence given as part of the input, along with a CNF formula. When an incomplete branching sequence is specified that gets exhausted before a satisfying assignment is found or the formula is proved to be unsatisfiable, the code reverts to the default variable selection scheme VSIDS of zChaff (cf. Section 2.3.2).

For consistency, we analyzed the performance with random restarts turned off. For all other parameters, we used the default values of zChaff. For all formulas, results are reported for DPLL (zChaff with clause learning disabled), for CL (unmodified zChaff), and for CL with a specified branching sequence (modified zChaff).

Tables 5.1 and 5.2 show the performance on grid pebbling and randomized pebbling formulas, respectively, using the branching sequence generated by Algorithm 5.1, `PebSeq1UIP`. Table 5.3 shows the performance on the GT_n formulas using the branching sequence generated by Algorithm 5.3, `GTnSeq1UIP`.

Table 5.1: zChaff on *grid pebbling* formulas. ‡ denotes out of memory.

Solver	*Grid formula* *Layers*	*Variables*	*Runtime in seconds* *Unsatisfiable*	*Satisfiable*
DPLL	5	30	0.24	0.12
	6	42	110	0.02
	7	56	> 6 hrs	0.07
	8	72	> 6 hrs	> 6 hrs
CL (unmodified zChaff)	20	420	0.12	0.05
	40	1,640	59	36
	65	4,290	‡	47
	70	4,970	‡	‡
CL + **branching sequence**	100	10,100	0.59	0.62
	500	250,500	254	288
	1,000	1,001,000	4,251	5,335
	1,500	2,551,500	21,097	‡

Table 5.2: zChaff on *randomized pebbling* formulas with distinct labels, indegree ≤ 5, and disjunction label size ≤ 6. ‡ denotes out of memory.

Solver	*Randomized pebbling formula* *Nodes*	*Variables*	*Clauses*	*Runtime in seconds* *Unsatisfiable*	*Satisfiable*
DPLL	9	33	300	0.00	0.00
	10	29	228	0.58	0.00
	10	48	604	> 6 hrs	> 6 hrs
CL (unmodified zChaff)	50	154	3,266	0.91	0.03
	87	296	9,850	‡	65
	109	354	11,106	584	0.78
	110	354	18,467	‡	‡
CL + **branching sequence**	110	354	18,467	0.28	0.29
	4,427	14,374	530,224	48	49
	7,792	25,105	944,846	181	> 6 hrs
	13,324	43,254	1,730,952	669	249

For both grid and randomized pebbling formulas, the size of problems that can be solved increases substantially as we move down the respective tables. Note that randomized pebbling graphs typically have a more complex structure than grid pebbling graphs. In addition, higher indegree and larger disjunction labels make both the CNF formula size as well as the required branching sequence larger. This explains

Table 5.3: zChaff on GT_n formulas. ‡ denotes out of memory.

Solver	GT_n *formula* n	*Variables*	*Clauses*	*Runtime in seconds* *Unsatisfiable*	*Satisfiable*
DPLL	8	62	372	1.05	0.34
	9	79	549	48.2	0.82
	10	98	775	3395	248
	11	119	1,056	$>$ 6 hrs	743
CL (unmodified zChaff)	10	98	775	0.20	0.00
	13	167	1,807	93.7	7.14
	15	223	2,850	1492	0.01
	18	322	5,067	‡	‡
CL + **branching sequence**	18	322	5,067	0.52	0.13
	27	727	17,928	701	0.17
	35	1,223	39,900	3.6	0.15
	45	2,023	86,175	‡	0.81

the difference between the performance of zChaff, both original and modified, on grid and randomized pebbling instances. For all instances considered, the time taken to generate the branching sequence from the input graph was significantly less than that for generating the pebbling formula itself.

For the GT_n formulas, since the branching used was incomplete, the solver had to revert back to zChaff's VSIDS heuristic to choose variables to branch on after using the given branching sequence as a guide for the first few decisions. Nevertheless, the sizes of problems that could be handled increased significantly. The satisfiable versions proved to be relatively easier, with or without a specified branching sequence.

5.3 Discussion

This chapter has developed the idea of using a high level description of a satisfiability problem for generating auxiliary information that can guide a SAT algorithm trying to solve it. Our experimental results show a clear exponential improvement in performance when such information is used to solve both grid and randomized pebbling problems, as well as the GT_n ordering problems.

Although somewhat artificial, these problems are interesting in their own right and provide hard instances for some of the best existing SAT solvers like zChaff. Pebbling graphs are structurally similar to the layered graphs induced naturally by problems involving unwinding of state space over time, such as CNF formulations of planning by Kautz and Selman [70] and bounded model checking by Biere et al. [25]. This bolsters our belief that high level structure can be recovered and exploited to

make clause learning more efficient.

In practice, a solver must employ good branching heuristics as well as implement a powerful proof system. Our result that pebbling formulas have short CL proofs depends critically upon the solver choosing a branching sequence that solves the formula in a "bottom-up" fashion, so that the learned clauses have maximal reuse. Nevertheless, we were able to automatically generate such sequences for grid and randomized pebbling formulas. For the GT_n formulas, we used a different approach and instead provided a very simple but imperfect automatically generated branching sequence that boosted performance significantly in practice.

Our approach of exploiting high level problem description to generate auxiliary information for SAT solvers, of course, requires the knowledge of this high level description to begin with. The standard CNF benchmarks such as those in the online collection at SATLIB [63], unfortunately, do not come with such a description and thus do not allow an extended evaluation of our technique on several interesting formulas routinely used by researchers. We regard this not as a drawback of our approach but instead as an easily avoidable limitation of the currently prevalent notion of SAT solvers as blackboxes taking only a pure CNF formula as input. There is no good reason for the high level problem description to be unavailable to generate auxiliary structural information since CNF formulas for practically all interesting problems, from theory and practice, are created from a more abstract specification. We continue to build upon this philosophy in the next chapter.

Chapter 6

SYMMETRY IN SATISFIABILITY SOLVERS

As discussed earlier, we have seen tremendous improvement in the capabilities of general purpose SAT solvers in the past decade. The state-of-the-art techniques make them quite effective in solving challenging problems from various domains. Despite the success, one aspect of many theoretical as well as real-world problems that we argue has not been fully exploited is the presence of *symmetry* or *equivalence* amongst the underlying objects.

The concept of symmetry in the context of SAT solvers is best explained through some examples of the many application areas where it naturally occurs. For instance, in FPGA (field programmable gate array) routing used in electronics design, all wires or channels connecting two switch boxes are equivalent; in circuit modeling, all inputs to a multiple fanin AND or OR gate are equivalent; in planning, all boxes that need to be moved from city A to city B are equivalent; in multi-processor scheduling, all available processors are equivalent; in cache coherency protocols in distributed computing, all available caches are typically equivalent. When such problems are translated into CNF formulas to be fed to a SAT solver, the underlying equivalence or symmetry translates into a symmetry between the variables of the formula.

There has been work on using this symmetry in *domain-specific* algorithms and techniques. However, our experimental results suggest that current general purpose complete SAT solvers are unable to fully capitalize on symmetry. This chapter focuses on developing a new general purpose technique towards this end and on empirically evaluating its effectiveness in comparison with other known approaches.

Example 6.1. For concreteness, we give one simple but detailed example of symmetry in SAT solvers. At the risk of appearing narrow in scope, we choose the *pigeonhole principle* PHP_m^n: given n pigeons and m holes, there is no one-one mapping of the pigeons to the holes when $n > m$. Translated into a CNF formula over variables $x_{i,j}$ denoting that pigeon i is mapped to hole j, this has two kinds of clauses. We use the notation $[p]$ to denote the set $\{1, 2, \ldots, p\}$.

(a) Pigeon clauses: for $i \in [n]$, clause $(x_{i,1} \vee x_{i,2} \vee \ldots \vee x_{i,m})$ says that pigeon i is mapped to some hole, and

(b) Hole clauses: for $i \neq k \in [n], j \in [m]$, hole clauses $(\neg x_{i,j} \vee \neg x_{k,j})$ say that no two pigeons are mapped to one hole.

This formula, despite being extremely simple to state, is a cornerstone of proof complexity research. Haken [60] used PHP^n_{n-1} to show the first ever exponential lower bound for resolution. Since then several researchers have improved upon and generalized his result to $m \ll n$, to other counting-based formulas, and to stronger proof systems. Needless to say, the results from the 2005 SAT competition [78] testify that the pigeonhole formulas provide a class of hard instances for most of the complete SAT solvers which are based on the clause learning proof system, and hence on resolution (cf. Chapter 4).

Returning to the context of symmetry, PHP^n_m contains two natural sets of equivalent or symmetric objects, the n pigeons and the m holes. Accordingly, *all* variables $x_{i,j}$ in this formula are symmetric to each other. As we will see, it helps to distinguish between the "pigeon-symmetry" between $x_{i,j}$ and $x_{k,j}$, and the "hole-symmetry" between $x_{i,j}$ and $x_{i,\ell}$.

Remark 6.1. While we use PHP^n_m as a motivation for the work presented in this chapter, we would like to remind the reader that the techniques we develop are much more general and capable of handling symmetry in more complex forms that we will describe in due course. There are known techniques to handle the pigeonhole problem in SAT solvers, such as the use of cardinality constraints by Chai and Kuehlmann [32] and Dixon et al. [49]. However, such approaches either do not generalize or do not perform as well in the presence of more complex forms of symmetry.

Previous Work

A technique due to Crawford et al. [41] that has worked quite well in handling symmetry is to add *symmetry breaking predicates* to the input specification to weed out all but the lexically-first solutions. The idea is to identify the group of permutations of variables that keep the CNF formula unchanged. For each such permutation π, clauses are added so that for every satisfying assignment σ for the original problem, whose permutation $\pi(\sigma)$ is also a satisfying assignment, only the lexically-first of σ and $\pi(\sigma)$ satisfies the added clauses. Tools such as Shatter by Aloul et al. [4] improve upon this technique and use graph isomorphism detectors like Saucy by Darga et al. [42] to generate symmetry breaking predicates. This latter problem of computing graph isomorphism is not known to have any polynomial time solution, and is conjectured to be strictly between the complexity classes P and NP [cf. 73]. Hence, one must resort to heuristic or approximate solutions. Further, the number of symmetry breaking predicates one needs to add in order to break all symmetries may be prohibitively large. This is typically handled by discarding "large" symmetries. This may, however, result in a much slower SAT solution as indicated by some of our experiments.

Solvers such as PBS by Aloul et al. [5], pbChaff by Dixon et al. [49], and Galena by Chai and Kuehlmann [32] utilize non-CNF formulations known as pseudo-Boolean

inequalities. They are based on the cutting planes proof system which, as mentioned in Section 3.2, is strictly stronger than resolution on which DPLL type CNF solvers are based. Since this more powerful proof system is difficult to implement in its full generality, pseudo-Boolean solvers often implement only a subset of it, typically learning only CNF clauses or restricted pseudo-Boolean constraints upon a conflict. Pseudo-Boolean solvers may lead to purely syntactic representational efficiency in cases where a single constraint such as $y_1 + y_2 + \ldots + y_k \leq 1$ is equivalent to $\binom{k}{2}$ binary clauses. More importantly, they are relevant to symmetry because they sometimes allow implicit encoding. For instance, the single constraint $x_1 + x_2 + \ldots + x_n \leq m$ over n variables captures the essence of the pigeonhole formula PHP_m^n over nm variables which is provably exponentially hard to solve using resolution-based methods without symmetry considerations. This implicit representation, however, is not suitable in certain applications such as clique coloring and planning that we discuss.

One could conceivably keep the CNF input unchanged but modify the solver to detect and handle symmetries during the search phase as they occur. Although this approach is quite natural, we are unaware of its implementation in a general purpose SAT solver besides sEqSatz by Li et al. [81] whose technique appears to be somewhat specific and whose results are not too impressive compared to zChaff itself. Related work has been done in the specific areas of automatic test pattern generation by Marques-Silva and Sakallah [85] and SAT-based model checking by Shtrichman [102]. In both cases, the solver utilizes global information obtained at a stage to make subsequent stages faster.

In other domain-specific work, Fox and Long [52] presented a framework for planning problems that is very similar to ours in essence. However, their work has two disadvantages. The obvious one is that they provide a planner, not a general purpose reasoning engine. The second is that their approach does not guarantee plans of optimal length when multiple (non-conflicting) actions are allowed to be performed at each time step.

Dixon et al. [48] give a generic method of representing and dynamically maintaining symmetry using group theoretic techniques that guarantee polynomial size proofs of many difficult formulas. The strength of their work lies in a strong group theoretic foundation and comprehensiveness in handling all possible symmetries. The computations involving group operations that underlie their current implementation are, however, often quite expensive.

Our Contribution

We propose a new technique for representing and dynamically maintaining symmetry information for DPLL-based satisfiability solvers. We present an evaluation of our ideas through our tool SymChaff and demonstrate empirical exponential speedup in a variety of problem domains from theory and practice. While our framework as presented applies to both CNF and pseudo-Boolean formulations, the current implementation

of SymChaff uses pure CNF representation.

A key difference between our approach and that based on symmetry breaking predicates is that we use a high level description of a problem rather than its CNF representation to obtain symmetry information. (We give concrete examples of this later in this chapter.) This leads to several advantages. The high level description of a problem is typically very concise and reveals its structure much better than a relatively large set of clauses encoding the same problem. It is simple, in many cases almost trivial, for the problem designer to specify global symmetries at this level using straightforward "tagging." If one prefers to compute these symmetries automatically, off-the-shelf graph isomorphism tools can be used. Using these tools on the concise high level description will, of course, be much faster than using the same tools on a substantially larger CNF encoding.

While it is natural to choose a variable and branch two ways by setting it to TRUE and FALSE, this is not necessarily the best option when k variables, $x_1, x_2, \ldots, x_k$, are known to be arbitrarily interchangeable. The same applies to more complex symmetries where multiple classes of variables *simultaneously* depend on an index set $I = \{1, 2, \ldots, k\}$ and can be arbitrarily interchanged in parallel within their respective classes. We formalize this as a k-complete multi-class symmetry and handle it using a $(k+1)$-*way branch* based on I that maintains completeness of the search and shrinks the search space by as much as $O(k!)$. The index sets are implicitly determined from the many-sorted first order logic representation of the problem at hand. We extend the standard notions of conflict and clause learning to the multiway branch setting, introducing *symmetric learning*. Our solver SymChaff integrates seamlessly with most of the standard features of modern SAT solvers, extending them in the context of symmetry wherever necessary. These include fast unit propagation, good restart strategy, effective constraint database management, etc.

6.1 Preliminaries

The technique we present in this work can be applied to all DPLL based systematic SAT solvers designed for CNF as well as pseudo-Boolean formulas.

Definition 6.1. A *pseudo-Boolean formula* is a conjunction of pseudo-Boolean constraints, where each pseudo-Boolean constraint is a weighted inequality over propositional variables with typically integer coefficients.

This generalizes the notion of a clause; $(a \vee b \vee c)$ is equivalent to the pseudo-Boolean inequality $a + b + c \geq 1$.

Recall that a CNF clause is called "unit" if all but one of its literals are set to FALSE; the remaining literal must be set to TRUE to satisfy the clause. Similarly, a pseudo-Boolean constraint is called "unit" if variables have been set in such a way that all its unset literals must be set to TRUE to satisfy the constraint. Unit propagation

is a technique common to SAT and pseudo-Boolean solvers that recursively simplifies the formula by appropriately setting unset variables in unit constraints.

A DPLL-based systematic SAT or pseudo-Boolean solver implements the basic branch-and-backtrack procedure described in Section 2.3. Various features and optimizations, such as conflict clause learning, random restarts, watched literals, conflict-directed backjumping, etc., are added to this simple DPLL process in order to increase efficiency.

6.1.1 Constraint Satisfaction Problems and Symmetry

A constraint satisfaction problem (CSP) is a collection of constraints over a set $V = \{x_1, x_2, \ldots, x_n\}$ of variables. Although the following notions are generic, our focus in this work will be on CNF and pseudo-Boolean constraints over propositional variables.

Symmetry may exist in various forms in a CSP. We define it in terms of permutations of variables that preserve certain properties. Let σ be a permutation of $[n]$. Extend σ by defining $\sigma(x_i) = x_{\sigma(i)}$ for $x_i \in V$ and $\sigma(V') = \{\sigma(x) \mid x \in V'\}$ for $V' \subseteq V$. For a constraint C over V, let $\sigma(C)$ be the constraint resulting from C by applying σ to each variable of C. For a CSP Γ, define $\sigma(\Gamma)$ to be the new CSP consisting of the constraints $\{\sigma(C) \mid C \in \Gamma\}$.

Definition 6.2. A permutation σ of the variables of a CSP Γ is a *global symmetry* of Γ if $\sigma(\Gamma) = \Gamma$.

Definition 6.3. Let V be a the set of variables of a CSP Γ. $V' \subseteq V, |V'| = k$, is a *k-complete (global) symmetry* of Γ if *every* permutation σ of V satisfying $\sigma(V') = V'$ and $\sigma(x) = x$ for $x \notin V'$ is a global symmetry of Γ.

In other words, the k variables in V' can be arbitrarily interchanged without changing the original problem. Such symmetries exist in simple problems such as the pigeonhole principle where all pigeons (and holes) are symmetric. This can be detected and exploited using various known techniques such as cardinality constraints by Chai and Kuehlmann [32] and Dixon et al. [49].

6.1.2 Many-Sorted First Order Logic

In first order logic, one can express universally and existentially quantified logical statements about variables and constants that range over a certain domain with some inherent structure. For instance, the domain could be $\{1, 2, \ldots, n\}$ with the successor relationship of the first n natural numbers as its structure, and a (false) universally quantified logical statement over it could be that every element in the domain has a successor.

In *many-sorted* logic, the domain of variables and constants may be divided up into various types or "sorts" of elements that are quantified over independently. In other

words, many-sorted first order logic extends first order logic with type information. The reader is referred to standard texts such as by Gallier [54] for further details. We remark here that many-sorted first order logic is known to be exactly as expressive as first order logic itself. In this sense, sorts or types add convenience but not power to the logic.

As an example, consider again the pigeonhole principle where the domain consists of a set P of pigeons and a set H of holes. The problem can be stated as the succinct 2-sorted first order formula $[\forall(p \in P)\ \exists(h \in H)\ .\ X(p,h)] \wedge [\forall(h \in H,\ p_1 \in P,\ p_2 \in P)\ .\ (p_1 \neq p_2 \rightarrow (\neg X(p_1,h) \vee \neg X(p_2,h)))]$, where $X(p,h)$ is the predicate "pigeon p maps to hole h." We can alternatively write this 2-sorted first order logic formula even more concisely as $[\forall^P i\ \exists^H j\ .\ x_{i,j}] \wedge [\forall^H j\ \forall^P i,k\ .\ (i \neq k \rightarrow (\neg x_{i,j} \vee \neg x_{k,j}))]$

Recall on the other hand from Example 6.1 that the CNF formulation of same problem requires $|P|+|H|\binom{|P|}{2}$ clauses. As we will see shortly, the sort-based quantified representation of problems lies at the heart of our approach by providing us the base "symmetry sets" to start with.

6.2 Symmetry Framework and SymChaff

We describe in this section our new symmetry framework in a generic way, briefly referring to specific implementation aspects of SymChaff as appropriate.

The motivation and description of our techniques can be best understood with a few concrete examples in mind. We use three relatively simple logistics planning problems depicted in Figure 6.1. In all three of these problems, there are k trucks $T_1, T_2, \ldots, T_k$ initially at a location L_{TB} (truckbase). There are several locations as well as a number of packages. Each package is initially at a certain location and needs to be transported to a certain destination location. Actions that can be taken at any step include driving a truck from one location to another, and loading or unloading multiple boxes (in parallel) onto or from a truck. The task is to find a minimum length plan such that all boxes arrive at their destined locations and all trucks return to L_{TB}. Actions that do not conflict in their pre- or post-conditions can be taken in parallel.

Let $s(i) = (i \bmod n) + 1$ denote the cyclic successor of i in $[n]$.

Example 6.2 (`PlanningA`). Let $k = \lceil 3n/4 \rceil$. For $1 \leq i \leq n$, there is a location L_i that has two packages $P_{i,1}$ and $P_{i,2}$. The goal is to deliver package $P_{i,1}$ to location $L_{s(i)}$ and package $P_{i,2}$ to location $L_{s(s(i))}$.

The shortest plan for this problem is of length 7 for any n. The idea behind the plan is to use 3 trucks to handle 4 locations. E.g., truck T_1 transports $P_{1,1}$, $P_{1,2}$, and $P_{2,1}$, truck T_2 transports $P_{3,1}$, $P_{3,2}$, and $P_{4,1}$, and truck T_3 transports $P_{2,2}$ and $P_{4,2}$. The 7 steps for T_1 involve (i) driving to L_1, (ii) loading the two boxes there, (iii) driving to L_2, (iv) unloading $P_{1,1}$ and loading $P_{2,1}$, (v) driving to L_3, (vi) unloading the two boxes it is carrying, and (vii) driving back to L_{TB}.

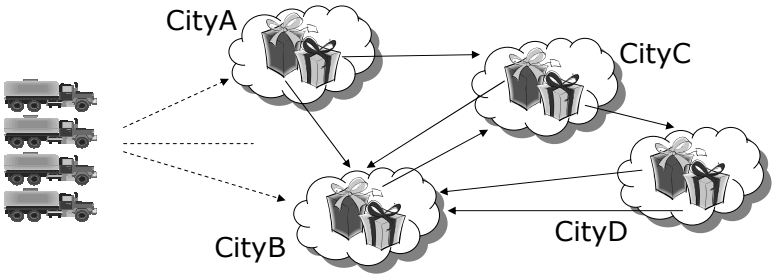

Figure 6.1: The setup for logistic planning examples

Example 6.3 (`PlanningB`). Let $k = \lceil n/2 \rceil$. For $1 \leq i \leq n$, there are 5 packages at location L_i that are all destined for location $L_{s(i)}$. This problem has more symmetries than `PlanningA` because all packages initially at the same location are symmetric.

The shortest plan for this problem is of length 7 and assigns one truck to two consecutive locations. E.g., the 7 steps for truck T_1 include (i) driving to L_1, (ii) loading all boxes there, (iii) driving to L_2, (iv) unloading the boxes it is carrying and loading all boxes originally present at L_2, (v) driving to L_2, (vi) unloading all boxes it is carrying, and (vii) driving back to L_{TB}.

Example 6.4 (`PlanningC`). Let $k = n$. For $1 \leq i \leq n$, there are locations $L_i^{\text{src}}, L_i^{\text{dest}}$ and packages $P_{i,1}, P_{i,2}$. Both these packages are initially at location L_i^{src} and must be delivered to location L_i^{dest}. Here not only the two packages at each source location are symmetric but all n tuples $(L_i^{\text{src}}, L_i^{\text{dest}}, P_{i,1}, P_{i,2})$ are symmetric as well.

It is easily seen that the shortest plan for this problem is of length 5 and assigns one truck to each source-destination pair. E.g., the 5 steps for T_1 involve (i) driving to L_1^{src}, (ii) loading the two boxes there, (iii) driving to L_1^{dest}, (iv) unloading the two boxes it is carrying, and (v) driving back to L_{TB}.

For a given plan length, such a planning problem can be converted into a CNF formula using tools such as Blackbox by Kautz and Selman [72] and then solved using standard SAT solvers. The variables in this formula are of the form *load-$P_{i,1}$-onto-T_j-at-L_k-time-t*, etc. We omit the details [see 70].

6.2.1 k-complete m-class Symmetries

Consider a CSP Γ over a set $V = \{x_1, x_2, \ldots, x_n\}$ of variables as before. We generalize the idea of complete symmetry for Γ to complete multi-class symmetry. Let $V_1, V_2, \ldots, V_m$ be disjoint subsets of V of cardinality k each. Let $V_0 = V \setminus \left(\bigcup_{i \in [m]} V_i\right)$. Order the variables in each $V_i, i \in [m]$, arbitrarily and let $y_i^j, j \in [k]$, denote the j^{th} variable of V_i.

Let σ be a permutation of the set $[k]$. Define $\bar{\sigma}$ to be the permutation of V induced by σ and $V_i, 0 \leq i \leq m$, as follows: $\bar{\sigma}(x) = x$ for $x \in V_0$ and $\bar{\sigma}(x) = y_i^{\sigma(j)}$ for $x = y_i^j \in V_i, i \in [m]$. In other words, $\bar{\sigma}$ maps variables in V_0 to themselves and applies σ in parallel to the indices of the variables in each class $V_i, i \in [m]$, simultaneously.

Definition 6.4. If $\bar{\sigma}$ is a global symmetry of Γ for *every* permutation σ of $[k]$ then the set $\{V_1, V_2, \ldots, V_m\}$ is a *k-complete m-class (global) symmetry* of Γ. The sets $V_i, i \in [m]$, are referred to as the *variable classes*. Variables in V_i are said to be *indexed by* the *symindex set* $[k]$.

Note that a k-complete 1-class symmetry is simply a k-complete symmetry. Complete multi-class symmetries correspond to the case where variables from multiple classes can be simultaneously and coherently changed in parallel without affecting the problem. This happens naturally in many problem domains.

Example 6.5. Consider the logistics planning problem `PlanningA` (Example 6.2) for $n = 4$ converted into a unsatisfiable CNF formula corresponding to plan length 6. The problem has $k = 3$ trucks and is 3-complete m-class symmetric for appropriate m. The variable classes V_i of size 3 are indexed by the symindex set $\{1, 2, 3\}$ and correspond to sets of 3 variables that differ only in which truck they use. For example, variables *unload-$P_{2,1}$-from-T_1-at-L_2-time*-5, *unload-$P_{2,1}$-from-T_2-at-L_2-time*-5, and *unload-$P_{2,1}$-from-T_3-at-L_2-time*-5 comprise one variable class which is denoted by *unload-$P_{2,1}$-from-$\mathbf{T_j}$-at-L_2-time*-5. The many-sorted representation of the problem has one universally quantified sort for the trucks. The problem `PlanningA` remains unchanged, e.g., when T_1 and T_2 are swapped in all variable classes simultaneously.

In more complex scenarios, a variable class may be indexed by multiple symindex sets and be part of more than one complete multi-class symmetry. This will happen, for instance, in the `PlanningB` problem (Example 6.3) where variables *load-$\mathbf{P_{2,a}}$-onto-$\mathbf{T_j}$-at-L_4-time*-4 are indexed by two symindex sets, $a \in [5]$ and $j \in [3]$, each acting independent of the other. This problem has a universally quantified 2-sorted first order representation.

Alternatively, multiple object classes, even in the high level description, may be indexed by the same symindex set. This happens, for example, in the `PlanningC` problem (Example 6.4), where $L_i^{\text{src}}, L_i^{\text{dest}}, P_{i,1}$, and $P_{i,2}$ are all indexed by i. This results in symmetries involving an even higher number of variable classes indexed by the same symindex set than in the case of `PlanningA` type problems.

6.2.2 Symmetry Representation

SymChaff takes as input a CNF file in the standard DIMACS format [68] as well as a `.sym` symmetry file S that encodes the complete multi-class symmetries of the input formula. Lines in S that begin with `c` are treated as comments. S contains a header

line `p sym nsi ncl nsv` declaring that it is a symmetry file with `nsi` symindex sets, `ncl` variable classes, and `nsv` symmetric variables.

Symmetry is represented in the input file S and maintained inside SymChaff in three phases. First, *symindex sets* are represented as consecutive, disjoint intervals of positive integers. In the `PlanningB` example for $n = 4$, the three trucks would be indexed by the set $[1 .. 3]$ and the 5 packages at location $L_i, 1 \leq i \leq 4$, by symindex sets $[3 + 5(i - 1) + 1 .. 3 + 5i]$, respectively. Here $[p .. q]$ denotes the set $\{p, p + 1, \ldots, q\}$. Second, one *variable class* is defined for each variable class V_i and associated with each symindex set that indexes variables in it. Finally, a *symindex map* is created that associates with each symmetric variable the variable class it belongs to and the indices in the symindex sets it is indexed by. For instance, variable *load-*$P_{2,4}$*-onto-*T_3*-at-*L_4*-time-*4 in problem `PlanningB` will be associated with the variable class *load-*$\mathbf{P_{2,a}}$*-onto-*$\mathbf{T_j}$*-at-*L_4*-time-*4 and with indices $j = 3$ and $a = 3 + 5(2 - 1) + 4 = 12$. The symmetry input file S is a straightforward encoding of symindex sets, variable classes, and symindex map.

Example 6.6. As another example and as an illustration of the exact syntax of S, we give the actual symmetry input file for the pigeonhole problem PHP_3^4 in Figure 6.2. There are two symindex sets, one for the 4 pigeons and the other for the 3 holes. These correspond to the consecutive, disjoint intervals $[1 .. 4]$ and $[5 .. 7]$, respectively, and are associated with the right end-points of the intervals, 4 and 7. All 12 variables of the problem are symmetric to each other and thus belong to the only variable class for the problem (commented as "vartype" in the Figure). This variable class is indexed by the two symindex sets associated with the right end-points 4 and 7. Finally, the symindex map says, for example, that variable 5, which happens to correspond to the variable $x_{2,2}$ in PHP_3^4, belongs to the first (and only) variable class and is indexed by the index 2 from the first symindex set and the index 6 from the second symindex set associated with its variable class.

Note that while the variable classes and the symindex map remain static, the symindex sets change dynamically as SymChaff proceeds assigning values to variables. In fact, when sufficiently many variables have been assigned truth values, all complete multi-class symmetries will be destroyed. For efficient access and manipulation, SymChaff stores variable classes in a vector data structure from the Standard Template Library (STL) of C++, the symindex map as a hash table, and symindex sets together as a multiset containing only the right end-points of the consecutive, disjoint intervals corresponding to the symindex sets. A symindex set split is achieved by adding the corresponding new right end-point to the multiset, and symindex sets are combined when backtracking by deleting the end-point.

```
c Symmetry file for php-004-003.cnf
c 4 pigeons, 3 holes, 12 symmetric variables
c 2 symindex sets, 1 vartype
c
p sym 12 2 1
c
c symindex sets
1 4 0
2 7 0
0
c vartypes
1 4 7 0
0
c
c symindex mappings
1 1 1 5 0
2 1 1 6 0
3 1 1 7 0
4 1 2 5 0
5 1 2 6 0
6 1 2 7 0
7 1 3 5 0
8 1 3 6 0
9 1 3 7 0
10 1 4 5 0
11 1 4 6 0
12 1 4 7 0
0
```

Figure 6.2: A sample symmetry file, `php-004-003.sym`

6.2.3 Multiway Index-based Branching

A distinctive feature of SymChaff is multiway symindex-based branching. Suppose at a certain stage the variable selection heuristic suggests that we branch by setting variable x to FALSE. SymChaff checks to see whether x has any complete multi-class symmetry left in the current stage. (Note that symmetry in our framework reduces as variables are assigned truth values.) x, of course, may not be symmetric at all to start with. If x doesn't have any symmetry, SymChaff proceeds with the usual DPLL style 2-way branch by setting x now to FALSE and later to TRUE. If it does have symmetry, SymChaff arbitrarily chooses a symindex set $I, |I| = k \geq 2$, that indexes x and creates a $(k+1)$-way branch. Let $x_1, x_2, \ldots, x_k$ be the variables indexed by

I in the variable class V' to which x belongs ($x = x_j$ for some j). For $0 \leq i \leq k$, the i^{th} branch sets $x_1, \ldots, x_i$ to FALSE and $x_{i+1}, \ldots, x_k$ to TRUE. The idea behind this multiway branching is that it only matters *how many* of the x_i are set to FALSE and not which exact ones. This reduces the search for a satisfying assignment from up to 2^k different partial assignments of $x_1, \ldots, x_k$ to only $k+1$ different ones. This clearly maintains completeness of the search and is the key to the good performance of SymChaff.

When one branches and sets variables, the symindex sets must be updated to reflect this change. When proceeding along the i^{th} branch in the above setting, two kinds of *symindex splits* happen. First, if x is also indexed by an index j in a symindex set $J = [a \,..\, b] \neq I$, we must split J into up to three symindex sets given by the intervals $[a \,..\, j-1]$, $[j \,..\, j]$, and $[j+1 \,..\, b]$ because j's symmetry has been destroyed by this assignment. To reduce the number of splits, SymChaff replaces x with another variable in its variable class for which $j = a$ and thus the split divides J into two new symindex sets only, $[a \,..\, a]$ and $[a+1 \,..\, b]$. This first kind of split is done once for the multiway branch for x and is independent of the value of i. The second kind of split divides $I = [c \,..\, d]$ into up to two symindex sets given by $[c \,..\, i]$ and $[i+1 \,..\, d]$. This, of course, captures the fact that both the first i and the last $k-i$ indices of I remain symmetric in the i^{th} branch of the multiway branching step.

Symindex sets that are split while branching must be restored when a backtrack happens. When a backtrack moves the search from the i^{th} branch of a multiway branching step to the $i+1^{st}$ branch, SymChaff deletes the symindex set split of the second type created for the i^{th} branch and creates a new one for the $i+1^{st}$ branch. When all $k+1$ branches are finished, SymChaff also deletes the split of the first type created for this multiway branch and backtracks.

6.2.4 Symmetric Learning

We extend the notion of conflict-directed clause learning to our symmetry framework. When all branches of a $(k+1)$-way symmetric branch b have been explored, SymChaff learns a *symconflict clause* C such that when all literals of C are set to FALSE, unit propagation falsifies *every* branch of b. This process clearly maintains soundness of the search. The symconflict clause is learned even for 2-way branches and is computed as follows.

Suppose a k-way branch b starts at decision level d. If the i^{th} branch of b leads to a conflict without any further branches, two things happen. First, SymChaff learns the FirstUIP clause following the conflict analysis strategy of zChaff (see Section 4.2.5). Second, it stores in a set S_b associated with b the decision literals at levels higher than d that are involved in the conflict. On the other hand, if the i^{th} branch of b develops further into another branch b', SymChaff stores in S_b those literals of the symconflict clause recursively learned for b' that have decision level higher than d. When all branches at b have been explored, the symconflict clause learned for b is

$\bigvee_{\ell \in S_b} \neg \ell$.

6.2.5 Static Ordering of Symmetry Classes and Indices

It is well known that the variable order chosen for branching in any DPLL-based solver has tremendous impact on efficiency. Along similar lines, the order in which variable classes and symindex sets are chosen for multiway branching can have significant impact on the speed of SymChaff.

While we leave dynamic strategies for selecting variable classes and symindex sets as ongoing and future work, SymChaff does support static ordering through a very simple and optional `.ord` order file given as input. This file specifies an ordering of variable classes as an initial guide to the VSIDS variable selection heuristic of zChaff (cf. Section 2.3.2), treating asymmetric variables in a class of their own. Further, for each variable class indexed by multiple symindex sets, it allows one to specify an order of priority on symindex sets. The exact file structure is omitted due to lack of space.

6.2.6 Integration of Standard Features

The efficiency of state-of-the-art SAT and pseudo-Boolean solvers relies heavily on various features that have been developed, analyzed, and tested over the last decade. SymChaff integrates well with most of these features, either using them without any change or extending them in the context of multiway branching and symmetric learning. The only significant and relatively new feature that neither SymChaff nor the version of zChaff on which it is based currently support is assignment stack shrinking based on conflict clauses which was introduced by Nadel [91] in the solver Jerusat.

For completeness, we make a digression to give a flavor of how assignment stack shrinking works. When a conflict occurs because a clause C' is violated and the resulting conflict clause C to be learned exceeds a certain threshold length, the solver backtracks to almost the highest decision level of the literals in C. It then starts assigning to FALSE the unassigned literals of the violated clause C' until a new conflict is encountered, which is expected to result in a smaller and more pertinent conflict clause to be learned.

Returning to SymChaff, it supports fast unit propagation using watched literals, good restart strategies, effective constraint database management, and smart branching heuristics in a very natural way (cf. Sections 2.3.2 and 4.2). In particular, it uses zChaff's watched literals scheme for unit propagation, deterministic and randomized restart strategies, and clause deletion mechanisms without any modification, and thus gains by their use as any other SAT solver would. While performing multiway branching for classes of variables that are known to be symmetric, SymChaff starts every new multiway branch based on the variable that would have been chosen by

VSIDS branch selection heuristic of zChaff, thereby retaining many advantages that effective branch selection heuristics like VSIDS have to offer.

Conflict clause learning is extended to symmetric learning as described earlier. Conflict-directed backjumping in the traditional context allows a solver to backtrack directly to a decision level d if variables at levels d or higher are the only ones involved in the conflicts in both branches at a point other than the branch variable itself. SymChaff extends this to multiway branching by computing this level d for all branches at a multiway branch point by looking at the symconflict clause for that branch, discarding all intermediate branches and their respective partial symconflict clauses, backtracking to level d, and updating the symindex sets.

While conflict-directed backjumping is always beneficial, fast backjumping may not be so. This latter technique, relevant mostly to the firstUIP learning scheme of zChaff, allows a solver to jump directly to a higher decision level d when even one branch leads to a conflict involving variables at levels d or higher only (in addition to the variable at the current branch). This discards intermediate decisions which may actually be relevant and in the worst case will be made again unchanged after fast backjumping. SymChaff provides this feature as an option which turns out to be helpful in certain domains and detrimental in others. To maintain consistency of symconflict clauses learned later, the level d' to backjump to is computed as the maximum of the level d as above and the maximum decision level $\bar{d}$ of any variable in the partial symconflict clause associated with the current multiway branch.

6.3 Benchmark Problems and Experimental Results

SymChaff is implemented on top of zChaff version 2003.11.04. The input to SymChaff is a `.cnf` formula file in the standard DIMACS format, a `.sym` symmetry file, and an optional `.ord` static symmetry order file. It uses the default parameters of zChaff. The program was compiled using g++ 3.3.3 for RedHat Linux 3.3.3-7. Experiments were conducted on a cluster of 36 machines running Linux 2.6.11 with four 2.8 GHz Intel Xeon processors on each machine, each with 1 GB memory and 512 KB cache.

Tables 6.1 and 6.2 report results for several parameterizations of two problems from proof complexity theory, three planning problems, and a routing problem from design automation. These problems are discussed below. Satisfiable instances of some of these problems were easy for all solvers considered and are thus omitted from the table. Except for the planning problems for which automatic "tags" were used (described later), the `.sym` symmetry files were automatically generated by a straightforward modification to the scripts used to create the `.cnf` files from the problem descriptions. For all instances, the time required to generate the `.sym` file was negligible compared to the `.cnf` file and is therefore not reported. The `.sym` files were in addition extremely small compared to the corresponding `.cnf` files.

The solvers used were SymChaff, zChaff version 2003.11.04, and March-eq-100 by

Huele et al. [64]. Symmetry breaking predicates were generated using Shatter version 0.3 that uses the graph isomorphism tool Saucy. Note that zChaff won the best solver award for industrial benchmarks in the SAT 2004 competition [77] while March-eq-100 won the award for handmade benchmarks.

SymChaff outperformed the other two solvers without symmetry breaking predicates in all but excessively easy instances. Generating symmetry breaking predicates from the input CNF formula was typically quite slow compared to a complete solution by SymChaff. The effect of adding symmetry breaking predicates before feeding the problem to zChaff was mixed, helping to various extents in some instances and hurting in others. In either case, it was never any better than using SymChaff without symmetry breaking predicates.

6.3.1 Problems from Proof Complexity

Pigeonhole Principle: `php-`n`-`m is the classic pigeonhole problem described in Example 6.1 for n pigeons and m holes. The corresponding formulas are satisfiable iff $n \leq m$. They are known to be exponentially hard for resolution [60, 94] but easy when the symmetry rule is added [76]. Symmetry breaking predicates can therefore be used for fast CNF SAT solutions. The price to pay is symmetry detection in the CNF formula, i.e., generation of symmetry breaking predicates using graph isomorphism tools. We found this process to be significantly costly in terms of the overall runtime.

pbChaff and Galena, on the other hand, use an explicit pseudo-Boolean encoding and rely on learning good pseudo-Boolean conflict constraints. They do overcome the drawbacks of the symmetry breaking predicates technique but are nonetheless slower than SymChaff.

SymChaff uses two symindex sets corresponding to the pigeons and the holes, and one variable class containing all the variables. It solves this problem in time $\Theta(m^2)$. Note that although it must read the entire input file containing $\Theta(nm^2)$ clauses, it does not need to process all of these clauses given the symmetry information. Although reading the input file is quite fast in practice, we do not include the time spent on it when claiming the $\Theta(m^2)$ bound.

This contrasts well with one of the fastest current techniques for this problem (other than the implicit pseudo-Boolean encoding) by Motter and Markov [89] which is based on ZBDDs and requires a fairly involved analysis to prove that it runs in time $\Theta(m^4)$ [90].

Clique Coloring Principle: The formula `clqcolor-`n`-`m`-`k encodes the clique coloring problem whose solution is a set of edges that form an undirected graph G over n nodes such that two conditions hold: G contains a clique of size m and G can be colored using k colors so that no two adjacent nodes get the same color. The formula

is satisfiable iff $m \leq n$ and $m \leq k$.

At first glance, this problem might appear to be a simple generalization of the pigeonhole problem. However, it evades fast solutions using SAT as well as pseudo-Boolean techniques even when the clique part is encoded implicitly using pseudo-Boolean methods. Indeed, Pudlák [93] has shown it to be exponentially hard for the cutting planes proof system on which pseudo-Boolean solvers are based.

Our experiments indicate that not only finding symmetries from the corresponding CNF formulas is time consuming, zChaff is extremely slow even after taking symmetry breaking predicates into account. SymChaff, on the other hand, uses three symindex sets corresponding to nodes, membership in clique, and colors, and three variable classes corresponding to edge variables, clique variables, and coloring variables. It solves the problem in time $\Theta(k^2)$, again ignoring the time spent on reading the input file.

We note that this problem can also be solved in polynomial time using the group theoretic technique of Dixon et al. [48]. However, the group operations that underlie their implementation are polynomials of degree as high as 6 or 7, making the approach significantly slower in practice.

6.3.2 Problems from Applications

All planning problems were encoded using the high level STRIPS formulation of Planning Domain Description Language (PDDL) introduced by Fikes and Nilsson [51]. These were then converted into CNF formulas using the tool Blackbox version 4.1 by Kautz and Selman [72]. A PDDL description of a planning problem is a straightforward Lisp-style specification that declares the objects involved, their initial state, and their goal state. In addition to this instance-specific description, it also uses a domain-specific file that describes the available actions in terms of their preconditions and effects.

We modified Blackbox to generate symmetry information as well by using a very simple "tagged" PDDL description where an original PDDL declaration such as

(:OBJECTS T_1 T_2 T_3
L_1^{src} L_2^{src} L_1^{dest} L_2^{dest}
$P_{1,1}$ $P_{2,1}$ $P_{1,2}$ $P_{2,2}$)

in the PlanningC example is replaced with

(:OBJECTS T_1 T_2 T_3 - SYMTRUCKS
L_1^{src} L_2^{src} - SYMLOCS
L_1^{dest} L_2^{dest} - SYMLOCS
$P_{1,1}$ $P_{2,1}$ - SYMLOCS
$P_{1,2}$ $P_{2,2}$ - SYMLOCS)

The rest of the PDDL description remains unchanged and a `.sym` file is automatically generated using these tags.

Example 6.7. For concreteness, we give the actual PDDL specification for our `PlanningA` example with $n = 3$ locations and $k = \lceil 3n/4 \rceil = 3$ trucks in Figure 6.3. The "tag" in bold is the only change to the usual specification of the problem needed to process symmetry information automatically.

```
(define (problem PlanningA-03)
 (:domain logistics-strips-sym)
 (:objects
  truck1
  truck2
  truck3 - SYMTRUCKS

  package1
  package2
  package3
  package4
  package5
  package6
  truckbase
  location1
  location2
  location3
  city1
 )
 (:init
  (TRUCK truck1)
  (TRUCK truck2)
  (TRUCK truck3)
  (OBJ package1)
  (OBJ package2)
  (OBJ package3)
  (OBJ package4)
  (OBJ package5)
  (OBJ package6)
continued...
```

```
...continued
  (LOCATION truckbase)
  (LOCATION location1)
  (LOCATION location2)
  (LOCATION location3)
  (CITY city1)

  (at package1 location1)
  (at package2 location1)
  (at package3 location2)
  (at package4 location2)
  (at package5 location3)
  (at package6 location3)
  (at truck1 truckbase)
  (at truck2 truckbase)
  (at truck3 truckbase)
  (in-city truckbase city1)
  (in-city location1 city1)
  (in-city location2 city1)
  (in-city location3 city1)
 )
 (:goal (and
  (at package1 location2)
  (at package2 location3)
  (at package3 location3)
  (at package4 location1)
  (at package5 location1)
  (at package6 location2)
 ))
)
```

Figure 6.3: A sample PDDL file for `PlanningA` with $n = 3$

We are now ready to present the four application-oriented problems for which we have experimental results. Three of these are planning problems.

Gripper Planning: The problem `gripper-`n`-`t is our simplest planning example. It consists of $2n$ balls in a room that need to be moved to another room in t steps using a robot that has two grippers that it can use to pick up balls. The corresponding formulas are satisfiable iff $t \geq 4n - 1$.

SymChaff uses two symindex sets corresponding to the balls and the grippers. The number of variable classes is relatively large and corresponds to each action that can be performed without taking into account the specific ball or gripper used. While SymChaff solves this problem easily in both unsatisfiable and satisfiable cases, the other two solvers perform poorly. Further, detecting symmetries from CNF using Shatter is not too difficult but does not speed up the solution process by any significant amount.

Logistics Planning `log-rotate`: The problem `log-rotate-`n`-`t is the logistics planning example `PlanningA` with n as the number of locations and t as the maximum plan length. As described earlier, it involves moving boxes in a cyclic rotation fashion between the locations. The formula is satisfiable iff $t \geq 7$.

SymChaff uses one symindex set corresponding to the trucks, and several variable classes. Here again symmetry breaking predicates, although not too hard to compute, provide less than a factor of two improvement. March-eq and zChaff were much slower than SymChaff on large instances, both unsatisfiable and satisfiable.

Logistics Planning `log-pairs`: The problem `log-pairs-`n`-`t is the logistics planning example `PlanningC` with n as the number of location pairs and t as the maximum plan length. As described earlier, it involves moving boxes between n disjoint location pairs. The corresponding formula is satisfiable iff $t \geq 5$.

SymChaff uses $n + 1$ symindex sets corresponding to the trucks and the location pairs, and several variable classes. This problem provides an interesting scenario where zChaff normally compares well with SymChaff but performs worse by a factor of two when symmetry breaking predicates are added. We also note that computing symmetry breaking predicates for this problem is quite expensive by itself.

Channel Routing: The problem `chnl-`t`-`n is from design automation and has been considered in previous works on symmetry and pseudo-Boolean solvers [4, 6]. It consists of two blocks of circuits with t tracks connecting them. Each track can hold one wire (or "net" as it is sometimes called). The task is to route n wires from one block to the other using these tracks. The underlying problem is a disguised pigeonhole principle. The formula is solvable iff $t \geq n$.

Table 6.1: Experimental results on UNSAT formulas. ‡ indicates > 6 hours.

Problem + parameters		SymChaff	zChaff	March-eq	Shatter	zChaff after Shatter
php	009-008	0.01	0.22	1.55	0.07	0.10
	013-012	0.01	1017	‡	0.09	0.01
	051-050	0.24	‡	‡	13.71	0.50
	091-090	0.84	‡	‡	245	3.47
	101-100	1.20	‡	‡	466	6.48
clqcolor	05-03-04	0.02	0.01	0.21	0.09	0.01
	12-07-08	0.03	‡	‡	5.09	4929
	20-15-16	0.26	‡	‡	748	‡
	30-18-21	0.60	‡	‡	20801	‡
	50-40-45	8.76	‡	‡	‡	‡
gripper	02t6	0.02	0.03	0.07	0.20	0.04
	04t14	0.84	2820	‡	3.23	983
	06t22	3.37	‡	‡	23.12	‡
	10t38	47	‡	‡	193	‡
log-rotate	06t6	0.74	1.47	21.55	8.21	0.93
	08t6	2.03	4.29	375	31.4	4.21
	09t6	8.64	15.67	3835	74	28.9
	11t6	51	12827	‡	324	17968
log-pair	05t5	0.46	0.38	3.65	25.19	0.65
	07t5	1.83	1.87	80	243	3.05
	09t5	6.29	6.23	582	1373	14.57
	11t5	15.65	18.05	1807	6070	34.4
chnl	010-011	0.04	8.61	‡	0.20	0.02
	011-020	0.06	135	‡	0.28	0.03
	020-030	0.05	‡	‡	4.60	0.10
	050-100	1.75	‡	‡	810	1.81

SymChaff uses two symindex sets corresponding to the end-points of the tracks in the two blocks, and $2n$ variable classes corresponding to the two end-points for each net. While March-eq was unable to solve any instance of this problem considered, zChaff performed as well as SymChaff after symmetry breaking predicates were added. The generation of symmetry breaking predicates was, however, orders of magnitude slower.

6.4 Discussion

SymChaff sheds new light into ways that high level symmetry, which is typically obvious to the problem designer, can be used to solve problems more efficiently. It handles

Table 6.2: Experimental results on SAT formulas. ‡ indicates > 6 hours.

Problem + parameters		SymChaff	zChaff	March-eq	Shatter	zChaff after Shatter
gripper	02t7	0.02	0.03	0.34	0.17	0.03
	04t15	2.03	1061	‡	0.23	1411
	06t23	7.27	‡	‡	19.03	‡
	10t39	92	‡	‡	193	‡
log-rotate	06t7	2.87	2.09	11	16.92	3.03
	07t7	7.64	6.85	27	55	47
	08t7	9.13	182	14805	62	358
	09t7	139	1284	814	186	1356

frequently occurring complete multi-class symmetries and is empirically exponentially faster on several problems from theory and practice, both unsatisfiable and satisfiable. The time and memory overhead it needs for maintaining data structures related to symmetry is fairly low and on problems with very few or no symmetries, it works as well as zChaff.

Our framework for symmetry is, of course, not tied to SymChaff. It can extend any state of the art DPLL-based CNF or pseudo-Boolean solver. Two key places where we differ from earlier approaches are in using high level problem description to obtain symmetry information (instead of trying to recover it from the CNF formula) and in maintaining this information dynamically without using complicated group theoretic machinery. This allows us to overcome many drawbacks of previously proposed solutions.

We show, in particular, that straightforward tagging in the specification of planning problems is enough to automatically generate relevant symmetry information which in turn makes the search for an optimal plan much faster. SymChaff incorporates several new ideas that allow this to happen. These include simple but effective symmetry representation, multiway branching based on variable classes and symmetry sets, and symmetric learning as an extension of clause learning to multiway branches.

One limitation of our approach is that it does not support symmetries that are initially absent but arise *after* some literals are set. Our symmetry sets only get refined from their initial value as decisions are made. Consider, for instance, a planning problem where two packages P_1 and P_2 are initially at locations L_1 and L_2, respectively, (and hence asymmetric) but are both destined for location L^{dest}. If at some point they both reach a common location, they should ideally be treated as equivalent with respect to the remaining portion of the plan. The `airlock` domain introduced by Fox and Long [53] is a creative example where such dynamically created symmetries are the norm rather than the exception. While they do describe a planner that is able to

exploit these symmetries, it is unclear how to incorporate such reasoning in a general purpose SAT solver besides resorting to on-the-fly computations involving the group of symmetries which, as observed in the work of Dixon et al. [48], can sometimes be quite expensive.

Chapter 7

CONCLUSION

We conclude with some general as well as concrete directions for extending the work presented in this thesis.

Our results in Chapter 3 imply exponential lower bounds on the running time of a class of backtracking algorithms for finding a maximum independent set (or, equivalently, a maximum clique or a minimum vertex cover) in a given graph, or approximating it. Analysis of the complexity of the independent set and other related problems under stronger proof systems, such as Cutting Planes [65, 29], bounded-depth Frege systems [2], or an extension of resolution that allows "without loss of generality" reasoning as mentioned in Section 3.11, will broaden our understanding in this respect.

The DPLL upper bounds that we give are based on a rather simple enumeration, with natural search space pruning, of all potential independent sets. As Theorem 3.5 points out, this is the best one can do using any exhaustive backtracking algorithm. Considering more complex techniques may let us close the gap of a factor of nearly $O(\Delta^5)$ in the exponent that currently exists between our lower and upper bounds for general resolution. It appears that we have not taken advantage of the full power of resolution, specifically the reuse of derived clauses.

The work presented in Chapter 4 inspires but leaves open several interesting questions of proof complexity. We showed that there are formulas on which CL is much more efficient than any proper natural refinement of RES. In general, can every short refutation in any such refinement be converted into a short CL proof? Or are these refinements and CL incomparable? We have shown that with arbitrary restarts, a slight variant of CL is as powerful as RES. However, judging when to restart and deciding what branching sequence to use after restarting adds more nondeterminism to the process, making it harder for practical implementations. Can CL with limited restarts also simulate RES efficiently?

We also introduced in that chapter FirstNewCut as a new learning scheme and used it to derive our theoretical results. A characterization of the real-world domains on which it performs better than other schemes is still open. In the process of deriving theoretical results, we gave a formal description of concepts such as implication and conflict graphs, and how they relate to learned clauses and trivial resolution derivations. This framework, we hope, will be useful in answering the complexity questions left open by this work.

In Chapter 5, the form in which we extract and use problem structure is a branch-

ing sequence. Although capable of capturing more information than a static variable order and avoiding the overhead of dynamic branching schemes, the exactness and detail branching sequences seem to require for pebbling formulas might pose problems when we move to harder domains where a polynomial size sequence is unlikely to exist. We may still be able to obtain substantial (but not exponential) improvements as long as an incomplete or approximate branching sequence made correct decisions most of the time, especially near the top of the underlying DPLL tree. The performance gains reported for GT_n formulas indicate that even a very simple and partial branching sequence can make a big difference in practice. Along these lines, variable orders in general have been studied in other scenarios, such as for algorithms based on BDDs [see e.g., 11, 61]. Reda et al. [96] have shown how to use BDD variable orders for DPLL algorithms without learning [96]. The ideas here can potentially provide new ways of capturing structural information.

From Chapter 6, the symmetry representation and maintenance techniques developed for SymChaff may be exploited in several other ways. The variable selection heuristic of the DPLL process is the most noticeable example. This framework can perhaps be applied even to local search-based satisfiability tools such as Walksat by McAllester et al. [86] to make better choices and reduce the search space. As for the framework itself, it can be easily extended to handle *k-ring* multi-class symmetries, where the k underlying indices can be rotated cyclically without changing the problem (e.g. as in the `PlanningB` problem, Example 6.3). However, the best-case gain of a factor of k may not offset the overhead involved.

SymChaff is the first cut at implementing our generic framework and can be extended in several directions. Learning strategies for symconflict clauses other than the "decision variable scheme" that it currently uses may lead to better performance, and so may dynamic strategies for selecting the order in which various branches of a multiway branch are traversed, as well as a dynamic equivalent of the static `.ord` file that SymChaff supports. Extending it to handle pseudo-Boolean constraints is a relatively straightforward but promising direction. Creating a PDDL preprocessor for planning problems that uses graph isomorphism tools to tag symmetries in the PDDL description would fully automate the planning-through-satisfiability process in the context of symmetry.

On the theoretical side, how does the technique of SymChaff compare in strength to proof systems such as RES with symmetry? It is unclear whether it is as powerful as the latter or can even efficiently simulate all of RES without symmetry. Answering this in the presence of symmetry may also help resolve an open question from Chapter 4 of whether clause learning (without symmetry) can efficiently simulate all of RES.

BIBLIOGRAPHY

[1] D. Achlioptas, P. Beame, and M. Molloy. A sharp threshold in proof complexity. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, pages 337–346, Crete, Greece, July 2001.

[2] M. Ajtai. The complexity of the pigeonhole principle. *Combinatorica*, 14(4): 417–433, 1994.

[3] M. Alekhnovich, J. Johannsen, T. Pitassi, and A. Urquhart. An exponential separation between regular and general resolution. In *Proceedings of the Thirty-Fourth Annual ACM Symposium on Theory of Computing*, pages 448–456, Montréal, Canada, May 2002.

[4] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Shatter: Efficient symmetry-breaking for boolean satisfiability. In *Proceedings of the 40th Design Automation Conference*, pages 836–839, Anahein, CA, June 2003.

[5] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. PBS: A backtrack-search pseudo-boolean solver and optimizer. In *Proceedings of the 5th International Conference on Theory and Applications of Satisfiability Testing*, pages 346–353, Cincinnati, OH, May 2002.

[6] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *Proceedings of the 39th Design Automation Conference*, pages 731–736, New Orleans, LA, June 2002.

[7] E. Amir and S. A. McIlraith. Partition-based logical reasoning. In *Proceedings of the 7th International Conference on Principles of Knowledge Representation and Reasoning*, pages 389–400, Breckenridge, CO, Apr. 2000.

[8] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings 33rd Annual Symposium on Foundations of Computer Science*, Pittsburgh, PA, Oct. 1992. IEEE.

[9] S. Arora and S. Safra. Probabilistic checking of proofs. In *Proceedings 33rd Annual Symposium on Foundations of Computer Science*, Pittsburgh, PA, Oct. 1992. IEEE.

[10] A. Atserias and M. L. Bonet. On the automatizability of resolution and related propositional proof systems. In *CSL '02: 16th Workshop on Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 569–583, Edinburgh, Scotland, Sept. 2002. Springer.

[11] A. Aziz, S. Tasiran, and R. K. Brayton. BDD variable orderings for interacting finite state machines. In *Proceedings of the 31th Design Automation Conference*, pages 283–288, San Diego, CA, June 1994.

[12] L. Baptista and J. P. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. In *6th Principles and Practice of Constraint Programming*, pages 489–494, Singapore, Sept. 2000.

[13] R. J. Bayardo Jr. and R. C. Schrag. Using CST look-back techniques to solve real-world SAT instances. In *Proceedings, AAAI-97: 14th National Conference on Artificial Intelligence*, pages 203–208, Providence, RI, July 1997.

[14] P. Beame, J. Culberson, D. Mitchell, and C. Moore. The resolution complexity of random graph k-colorability. Technical Report TR04-012, Electronic Colloquium in Computation Complexity, 2004. To appear in *Discrete Applied Mathematics*.

[15] P. Beame, R. Impagliazzo, T. Pitassi, and N. Segerlind. Memoization and DPLL: Formula caching proof systems. In *Proceedings 18th Annual IEEE Conference on Computational Complexity*, pages 225–236, Aarhus, Denmark, July 2003.

[16] P. Beame, R. Impagliazzo, and A. Sabharwal. Resolution complexity of independent sets in random graphs. In *Proceedings Sixteenth Annual IEEE Conference on Computational Complexity*, pages 52–68, Chicago, IL, June 2001.

[17] P. Beame, R. Karp, T. Pitassi, and M. Saks. On the Complexity of Unsatisfiability Proofs for Random k-CNF Formulas. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 561–571, Dallas, TX, May 1998.

[18] P. Beame, H. Kautz, and A. Sabharwal. Understanding the power of clause learning. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 1194–1201, Acapulco, Mexico, Aug. 2003.

[19] P. Beame, H. Kautz, and A. Sabharwal. Understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, Dec. 2004.

[20] P. Beame and T. Pitassi. Propositional Proof Complexity: Past, Present, Future. In *Current Trends in Theoretical Computer Science*, pages 42–70. World Scientific, 2001.

[21] R. Beigel. Finding maximum independent sets in sparse and general graphs. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 856–857, Baltimore, MD, Jan. 1999.

[22] E. Ben-Sasson, R. Impagliazzo, and A. Wigderson. Near-optimal separation of treelike and general resolution. Technical Report TR00-005, Electronic Colloquium in Computation Complexity, 2000. To appear in *Combinatorica*.

[23] E. Ben-Sasson and A. Wigderson. Short proofs are narrow - resolution made simple. *Journal of the ACM*, 48(2):149–169, 2001.

[24] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th Design Automation Conference*, pages 317–320, New Orleans, LA, June 1999.

[25] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Amsterdam, the Netherlands, Mar. 1999.

[26] B. Bollobás. *Random Graphs*. Academic Press, London, 1985.

[27] M. L. Bonet, J. L. Esteban, N. Galesi, and J. Johansen. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM Journal on Computing*, 30(5):1462–1484, 2000.

[28] M. L. Bonet and N. Galesi. Optimality of size-width tradeoffs for resolution. *Computational Complexity*, 10(4):261–276, 2001.

[29] M. L. Bonet, T. Pitassi, and R. Raz. Lower bounds for cutting planes proofs with small coefficients. *Journal of Symbolic Logic*, 62(3):708–728, Sept. 1997.

[30] R. I. Brafman. A simplifier for propositional formulas with many binary clauses. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 515–522, Seattle, WA, Aug. 2001.

[31] J. Buresh-Oppenheim and T. Pitassi. The complexity of resolution refinements. In *18th Annual IEEE Symposium on Logic in Computer Science*, pages 138–147, Ottawa, Canada, June 2003.

[32] D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. In *Proceedings of the 40th Design Automation Conference*, pages 830–835, Anahein, CA, June 2003.

[33] J. Chen, I. Kanj, and W. Jia. Vertex cover: Further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001.

[34] V. Chvátal. Determining the stability number of a graph. *SIAM Journal on Computing*, 6(4):643–662, 1977.

[35] V. Chvátal and E. Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, 1988.

[36] M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Gröbner basis algorithm to find proofs of unsatisfiability. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 174–183, Philadelphia, PA, May 1996.

[37] A. Coja-Oghlan. The Lovász number of random graphs. In *Proceedings of the 7th International Workshop on Randomization and Approximation Techniques in Computer Science*, volume 2764 of *Lecture Notes in Computer Science*, pages 228–239, Princeton, NY, Aug. 2003. Springer-Verlag.

[38] S. A. Cook. The complexity of theorem proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, OH, May 1971.

[39] S. A. Cook and R. A. Reckhow. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic*, 44(1):36–50, 1977.

[40] W. Cook, C. R. Coullard, and G. Turan. On the complexity of cutting plane proofs. *Discrete Applied Mathematics*, 18:25–38, 1987.

[41] J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159, Cambridge, MA, Nov. 1996.

[42] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for CNF. In *Proceedings of the 41st Design Automation Conference*, pages 518–522, San Diego, CA, June 2004.

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

[44] M. Davis and H. Putnam. A computing procedure for quantification theory. *Communications of the ACM*, 7:201–215, 1960.

[45] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24(1-3):347–410, 1984.

[46] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[47] I. Dinur and S. Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, 162(1):439–486, 2005.

[48] H. E. Dixon, M. L. Ginsberg, E. M. Luks, and A. J. Parkes. Generalizing boolean satisfiability II: Theory. *Journal of Artificial Intelligence Research*, 22: 481–534, 2004.

[49] H. E. Dixon, M. L. Ginsberg, and A. J. Parkes. Generalizing boolean satisfiability I: Background and survey of existing work. *Journal of Artificial Intelligence Research*, 21:193–243, 2004.

[50] J. L. Esteban, N. Galesi, and J. Messner. On the complexity of resolution with bounded conjunctions. In *Automata, Languages, and Programming: 29th International Colloquium*, volume 2380 of *Lecture Notes in Computer Science*, pages 220–231, Malaga, Spain, July 2002. Springer-Verlag.

[51] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):198–208, 1971.

[52] M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 956–961, July 1999.

[53] M. Fox and D. Long. Extending the exploitation of symmetries in planning. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning Systems*, pages 83–91, Apr. 2002.

[54] J. H. Gallier. *Logic for Computer Science*. Harper & Row, 1986.

[55] M. R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24(1-3):411–436, 1984.

[56] E. Giunchiglia, M. Maratea, and A. Tacchella. Dependent and independent variables in propositional satisfiability. In *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA)*, volume 2424 of *Lecture Notes in Computer Science*, pages 296–307, Cosenza, Italy, Sept. 2002. Springer-Verlag.

[57] E. Goldberg and Y. Novikov. BerkMin: A fast and robust sat-solver. In *Design, Automation and Test in Europe Conference and Exposition (DATE)*, pages 142–149, Paris, France, Mar. 2002.

[58] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings, AAAI-98: 15th National Conference on Artificial Intelligence*, pages 431–437, Madison, WI, July 1998.

[59] C. P. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems*, pages 208–213, Pittsburgh, PA, June 1998.

[60] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39: 297–305, 1985.

[61] J. E. Harlow and F. Brglez. Design of experiments in BDD variable ordering: Lessons learned. In *Proceedings of the International Conference on Computer Aided Design*, pages 646–652, San Jose, CA, Nov. 1998.

[62] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182: 105–142, 1999.

[63] H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In I. P. Gent, H. van Maaren, and T. Walsh, editors, *SAT2000*, pages 283–292. IOS Press, 2000. URL `http://www.satlib.org`.

[64] M. Huele, J. van Zwieten, M. Dufour, and H. van Maaren. March-eq: Implementing additional reasoning into an efficient lookahead SAT solver. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, volume 3542 of *Lecture Notes in Computer Science*, pages 345–359, Vancouver, BC, Canada, May 2004. Springer.

[65] R. Impagliazzo, T. Pitassi, and A. Urquhart. Upper and lower bounds for tree-like cutting planes proofs. In *9th Annual IEEE Symposium on Logic in Computer Science*, pages 220–228, Los Alamitos, CA, 1994.

[66] S. Janson, T. Łuczak, and A. Ruciński. *Random Graphs*. John Wiley & Sons, 2000.

[67] T. Jian. Algorithms for solving maximum independent set problem. *IEEE Transactions on Computers*, 35(9):847–851, 1986.

[68] D. S. Johnson and M. A. Trick, editors. *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS, 1996.

[69] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–104. Plenum Press, New York, 1972.

[70] H. A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 359–363, Vienna, Austria, Aug. 1992.

[71] H. A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings, AAAI-96: 13th National Conference on Artificial Intelligence*, pages 1194–1201, Portland, OR, Aug. 1996.

[72] H. A. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Working notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98*, Pittsburgh, PA, 1998.

[73] J. Köbler, U. Schöning, and J. Torán. *The Graph Isomorphism Problem: its Structural Complexity*. Birkhauser Verlag, 1993. ISBN 0-8176-3680-3.

[74] H. Konuk and T. Larrabee. Explorations of sequential ATPG using boolean satisfiability. In *11th VLSI Test Symposium*, pages 85–90, 1993.

[75] J. Krajíček. On the weak pigeonhole principle. *Fundamenta Mathematicae*, 170 (1-3):123–140, 2001.

[76] B. Krishnamurthy. Short proofs for tricky formulas. *Acta Informatica*, 22: 253–274, 1985.

[77] D. Le Berre and L. Simon (Organizers). SAT 2004 competition, May 2004. URL http://www.satcompetition.org/2004/.

[78] D. Le Berre and L. Simon (Organizers). SAT 2005 competition, June 2005. URL http://www.satcompetition.org/2005/.

[79] L. Levin. Universal sequential search problems. *Problems of Information Transmission*, 9(3):265–266, 1973. Originally in Russian.

[80] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 366–371, Nagoya, Japan, Aug. 1997.

[81] C. M. Li, B. Jurkowiak, and P. W. Purdom. Integrating symmetry breaking into a DLL procedure. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 149–155, Cincinnati, OH, May 2002.

[82] I. Lynce and J. P. Marques-Silva. An overview of backtrack search satisfiability algorithms. *Annals of Mathematics and Artificial Intelligence*, 37(3):307–326, 2003.

[83] P. D. MacKenzie, July 2005. Private communication.

[84] J. P. Marques-Silva and K. A. Sakallah. GRASP – a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer Aided Design*, pages 220–227, San Jose, CA, Nov. 1996.

[85] J. P. Marques-Silva and K. A. Sakallah. Robust search algorithms for test pattern generation. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, pages 152–161, Seattle, WA, June 1997.

[86] D. A. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *AAAI/IAAI*, pages 321–326, Providence, RI, July 1997.

[87] M. Mézard and R. Zecchina. Random k-satisfiability problem: From an analytic solution to an efficient algorithm. *Physical Review E*, 66:056126, Nov. 2002.

[88] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[89] D. B. Motter and I. Markov. A compressed breadth-first search for satisfiability. In *ALENEX*, volume 2409 of *Lecture Notes in Computer Science*, pages 29–42, San Francisco, CA, Jan. 2002. Springer.

[90] D. B. Motter, J. A. Roy, and I. Markov. Resolution cannot polynomially simulate compressed-BFS. *Ann. of Math. and A.I.*, 44(1-2):121–156, 2005.

[91] A. Nadel. The Jerusat SAT solver. Master's thesis, Hebrew University of Jerusalem, 2002.

[92] R. Ostrowski, E. Grégoire, B. Mazure, and L. Sais. Recovering and exploiting structural knowledge from cnf formulas. In *8th Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 185–199, Ithaca, NY, Sept. 2002. Springer-Verlag.

[93] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Journal of Symbolic Logic*, 62(3):981–998, Sept. 1997.

[94] R. Raz. Resolution lower bounds for the weak pigeonhole principle. *Journal of the ACM*, 51(2):115–138, 2004.

[95] A. A. Razborov. Resolution lower bounds for perfect matching principles. In *Proceedings Seventeenth Annual IEEE Conference on Computational Complexity*, pages 17–26, Montreal, PQ, Canada, May 2002.

[96] S. Reda, R. Drechsler, and A. Orailoglu. On the relation between SAT and BDDs for equivalence checking. In *Proceedings of the International Symposium on Quality Electronic Design*, pages 394–399, San Jose, CA, Mar. 2002.

[97] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425–440, 1986.

[98] A. Sabharwal. SymChaff: A structure-aware satisfiability solver. In *Proceedings, AAAI-05: 20th National Conference on Artificial Intelligence*, pages 467–474, Pittsburgh, PA, July 2005.

[99] A. Sabharwal, P. Beame, and H. Kautz. Using problem structure for efficient clause learning. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 242–256, Portofino, Italy, May 2003. Springer-Verlag.

[100] M. Shindo and E. Tomita. A simple algorithm for finding a maximum clique and its worst-case time complexity. *Systems and Computers in Japan*, 21(3): 1–13, 1990.

[101] O. Shtrichman. Tuning SAT checkers for bounded model checking. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 480–494, Chicago, IL, July 2000.

[102] O. Shtrichman. Accelerating bounded model checking of safety properties. *Formal Methods in System Design*, 1:5–24, 2004.

[103] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9:135–196, 1977.

[104] P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinatorial test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 15(9):1167–1176, 1996.

[105] R. E. Tarjan. Finding a maximum clique. Technical Report 72-123, Computer Science Department, Cornell University, Ithaca, NY, 1972.

[106] R. E. Tarjan and A. E. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[107] E. Tomita and T. Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In *4th International Conference on Discrete Mathematics and Theoretical Computer Science*, volume 2731 of *Lecture Notes in Computer Science*, pages 278–289, Dijon, France, July 2003. Springer-Verlag.

[108] L. Trevisan. Non-approximability results for optimization problems on bounded degree instances. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, pages 453–461, Crete, Greece, July 2001.

[109] G. S. Tseitin. On the complexity of derivation in the propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic, Part II*. 1968.

[110] A. Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[111] A. Urquhart. The symmetry rule in propositional logic. *Discrete Applied Mathematics*, 96-97:177–193, 1999.

[112] M. N. Velev and R. E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003.

[113] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275, Townsville, Australia, July 1997.

[114] H. Zhang and J. Hsiang. Solving open quasigroup problems by propositional reasoning. In *Proceedings of the International Computer Symp.*, Hsinchu, Taiwan, 1994.

[115] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design*, pages 279–285, San Jose, CA, Nov. 2001.

VITA

Ashish Sabharwal was born in India in 1977. After finishing high school in the city of Jaipur in 1994, he joined the Indian Institute of Technology, Kanpur, and was awarded the Bachelor of Technology degree in Computer Science and Engineering in 1998. Thereafter, he moved to the United States of America and started graduate education at the University of Washington, Seattle, where he has been since then. Working in the area of proof complexity theory under the guidance of Professor Paul Beame, he got his Master of Science degree in Computer Science and Engineering in the year 2001. Subsequently, he diversified his research interests to include automated reasoning systems as well. He has been working under the joint supervision of Professors Paul Beame and Henry Kautz towards a Doctor of Philosophy degree in Computer Science and Engineering of which this dissertation is a part. After the completion of this degree in 2005, he will move to Cornell University in the capacity of a postdoctoral researcher.