AUTOMATING PSEUDO-BOOLEAN INFERENCE WITHIN A DPLL FRAMEWORK

by

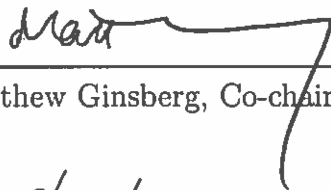
HEIDI DIXON

A DISSERTATION

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

December 2004

"Automating Pseudo-Boolean Inference within a DPLL Framework," a dissertation prepared by Heidi Dixon in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science. This dissertation has been approved and accepted by:



Dr. Matthew Ginsberg, Co-chair of the Examining Committee

Dr. Christopher Wilson, Co-chair of the Examining Committee

12-01-04

Date

Committee in charge:

Dr. Matthew Ginsberg, Co-chair
Dr. Christopher Wilson, Co-chair
Dr. David Etherington
Dr. Michal Young
Dr. Charles Wright

Accepted by:

Dean of the Graduate School

Copyright 2004 Heidi Dixon

An Abstract of the Dissertation of

Heidi Dixon for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken December 2004

Title: AUTOMATING PSEUDO-BOOLEAN INFERENCE WITHIN A DPLL FRAMEWORK

Approved:

Dr. Matthew Ginsberg, Co-chair

Dr. Christopher Wilson, Co-chair

State of the art satisfiability solvers provide important tools for problem solving in a number of real world problem domains [44, 10, 18, 60, 42]. These methods are all based on the classic DPLL algorithm. Unfortunately, these methods perform poorly on many important families of problems including the pigeonhole problem. Pigeonhole problems state that $n + 1$ pigeons cannot be placed in n holes and are believed to be common subproblems in many problem domains such as planning and scheduling. The most competitive satisfiability solvers show exponential scaling on these simple structured problems. These problems should be easy but traditional satisfiability methods make them unnecessarily hard.

Traditional satisfiability methods fail to solve these types of problems because the proof system they automate is very weak. The proof system used by traditional meth-

ods is resolution and all resolution proofs of the pigeonhole problem are exponential in length [35]. Consequently, traditional methods scale exponentially on pigeonhole problems. The only way to improve performance on these problems is to improve the strength of the underlying representation and inference system. This approach was thought to be impractical because the overhead of managing and automating a stronger and more complex inference system would outweigh any benefits derived from the stronger inference.

We implement a DPLL style satisfiability solver that uses pseudo-Boolean representation and automates an inference system properly stronger than resolution. We show that this approach is practical and that in fact, there is no significant advantage to resolution based satisfiability methods over their pseudo-Boolean counterpart. Building a pseudo-Boolean solver entails adapting all sub-procedures of a DPLL style method to use pseudo-Boolean representation. We provide an implementation of the learning procedure for pseudo-Boolean constraints and show experimentally that choices made in how learning is implemented determine the strength of the underlying inference system.

We give experimental results showing that the pseudo-Boolean solver can always closely match the performance of traditional methods yet the reverse is not true. The pseudo-Boolean solver allows exponential speedups over traditional methods on pigeonhole problems. We also give experimental results showing that traditional methods are unnecessarily slow on random planning problems due to embedded pigeonhole problems.

CURRICULUM VITA

NAME OF AUTHOR: Heidi Dixon

PLACE OF BIRTH: San Francisco, CA, U.S.A.

DATE OF BIRTH: December 14, 1970

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Oberlin College

DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science, 2004, University of Oregon

Bachelor of Arts in Geology, 1993, Oberlin College

AREAS OF SPECIAL INTEREST:

Artificial Intelligence
Boolean Satisfiability
Automated Deduction

PROFESSIONAL EXPERIENCE:

Research Assistant, Computational Intelligence Research Laboratory, 1998 - 2004
Software Engineer, Terralink, Portland Maine 1997 - 1998
Mathematics Teacher, The Chewonki Foundation, Maine Coast Semester Program, 1996 - 1997

AWARDS AND HONORS:

Sigma Xi Associate Membership, 1994
George B. Wharton Prize in Geology, Oberlin College, 1993

PUBLICATIONS:

Heidi E. Dixon, Matthew L. Ginsberg, Eugene M. Luks, and Andrew J. Parkes. Generalizing Boolean Satisfiability II: Theory. *Journal of Artificial Intelligence Research*, (accepted).

Heidi E. Dixon, Matthew L. Ginsberg, David K. Hofer, Eugene M. Luks, and Andrew J. Parkes. Implementing a generalized version of resolution. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI–2004)*, pages 55-60, 2004.

Heidi E. Dixon, Matthew L. Ginsberg, and Andrew J. Parkes. Generalizing Boolean Satisfiability I: Background and Survey of Existing work. *Journal of Artificial Intelligence Research*, 21:193-243, 2004.

Heidi E. Dixon, Matthew L. Ginsberg, and Andrew J. Parkes. Likely Near-term Advances in SAT Solvers. Workshop on Microprocessor Test and Verification (MTV '02). Held in Austin, Texas, USA. June 2002.

Heidi E. Dixon and Matthew L. Ginsberg. Inference methods for a pseudo-Boolean satisfiability solver. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI–2002)*, pages 635-640, 2002.

Heidi E. Dixon and Matthew L. Ginsberg. Combining Satisfiability Techniques from AI and OR. *The Knowledge Engineering Review*, 15(1):31-45, 2000.

ACKNOWLEDGMENTS

Often we walk through life unaware of how substantially we affect the lives of those around us. Moments of acknowledgment are far too rare. Many people have impacted my life over the past few years by offering support and encouragement, and for this, I feel very fortunate.

First, I thank my adviser Matt Ginsberg. As a mentor, Matt believed in me when I struggled to believe in myself. He held me to high standards and expected that I would achieve them. I will always be thankful for his generosity and his friendship. Matt and David Etherington both supported my choice to start a family during the past year. They are both dedicated fathers who understand the importance of family. The accommodations they made for me have allowed me to balance family and career.

I would like to thank Chris Wilson, David Etherington, Michal Young, and Charlie Wright being on my committee and giving my thesis a thorough review. I thank Andrew Parkes for the many conversations we shared. Andrew always encouraged me to be meticulous about the details. My mother-in-law Ellen looked after my two month old son Izzy Robin while I finished writing. No one could have cared more lovingly for my son, and I couldn't have managed without her help. Bud Keith gave me valuable feedback on the C++ code written for this dissertation. The code and my general programming practices were both improved. Jan Saunders and Star Holmberg helped me navigate the labyrinth of graduate school and department procedures, policies, and requirements. Tristan Smith has been a faithful friend and I suspect that the dissertation formatting files he gave me saved me from a tremendous headache. I would like to thank everyone at CIRL and OTS past and present. It has been a fun and challenging place to work.

Finally, I thank my husband Stew. It is not possible to quantify or express how much Stew has helped me in my life and my work. He gives his support everyday and it sustains me.

To family [66]

TABLE OF CONTENTS

Chapter Page

1. INTRODUCTION 1

1.1 Overview of Thesis 5

2. SYSTEMATIC SAT SOLVERS 7

2.1 In the Beginning: DPLL 9
2.1.1 Davis-Putnam-Logemann-Loveland 9
2.1.2 Unit Propagation 10
2.1.3 Early Branch Heuristics 10
2.2 Tackling Structured Problems with Learning 11
2.2.1 Resolution: The Inference of Learning 12
2.2.2 Bounded Learning 17
2.2.3 Non-Standard Backtracking 20
2.2.4 Branching Heuristics That Work with Learning 23
2.3 The Benefits of Fast Propagation 25
2.3.1 Unit Propagation: The Main Loop 25
2.3.2 Data Structures 26
2.3.3 Notes on the Interplay of Propagation and Learning . . . 32
2.4 Summary 33

3. RECONSIDERING REPRESENTATION 34

3.1 Systematic Solvers: A Special Kind of Proof System 35
3.1.1 DPLL Descendants: Resolution-Based Methods 36
3.2 Proof Complexity and Systematic Solvers 37
3.2.1 Proof Complexity 38
3.2.2 Exponential Lower Bounds for Resolution 40

4. A PSEUDO-BOOLEAN SAT SOLVER 42

4.1 Pseudo-Boolean Representation 43
4.1.1 P-simulating Resolution 44
4.1.2 Translating Pseudo-Boolean Constraints into CNF 45
4.1.3 Short Proofs of the Pigeonhole Principle 48
4.2 Unit Propagation 53
4.2.1 Count-Based Methods 53
4.2.2 Watched Literals 57
4.2.3 Summary 60
4.3 Learning 61
4.3.1 Resolution Analog 62
4.3.2 Capturing Conflicts 62
4.3.3 Bounded Learning 68
4.3.4 Summary 71
4.4 Non-Standard Backtracking and Unique Implication Points 71
4.5 Branching Heuristics 75
4.6 Strengthening 76

5. SOLVING THE PIGEONHOLE PROBLEM 82

5.1 CNF Encodings 83
5.1.1 Do No Harm 83
5.1.2 Experimental Results 84
5.2 Pigeonhole Problems 88
5.2.1 Experimental Results 89
5.3 Pigeonhole Problems Embedded in Planning Problems 95
5.3.1 A Simple Logistics Problem 95
5.3.2 The Logistics Pigeonhole Problem 97
5.3.3 CNF Encoding 98
5.3.4 Pseudo-Boolean Encoding 99
5.3.5 Experimental Results 100
5.3.6 Discussion 106
5.4 Random Planning Problems 107
5.4.1 Experimental Results 108

6. RELATED WORK 115

6.1 Integer Programming Techniques 115
6.1.1 Branch-and-bound 116
6.1.2 Branch-and-cut 119
6.1.3 Solving Satisfiability Problems 123

6.1.4 Summary 125
6.2 Lifted Solvers 127
6.2.1 Pseudo-Boolean 127
6.2.2 Parity Constraints 128
6.2.3 Zero Suppressed Binary Decision Diagrams 131
6.2.4 Finitely Quantified Clauses 133

7. CONCLUSION 134

7.1 Contributions 134
7.2 Future Work 136

INDEX 138

BIBLIOGRAPHY 140

LIST OF FIGURES

Figure | Page

2.1 Learning in a DPLL search tree 16

2.2 Percent of CPU time spent in unit propagation for ZCHAFF 26

2.3 Percent of CPU time spent in unit propagation for RELSAT 27

3.1 A DPLL search tree (a) and corresponding resolution proof of the empty clause Λ (b) 37

3.2 Performance on PHP_n^{n+1} for some well known solvers 41

5.1 Comparison of execution time for PBCHAFF and ZCHAFF on CNF encodings. Each point corresponds to a CNF problem instance. The x coordinate corresponds to the execution time in seconds for ZCHAFF and the y coordinate corresponds to the execution time in seconds for PBCHAFF. The line $f(x) = 4.37 + 1.61x$ is the best linear fit to the data, and the curve $f(x) = 1.89x^{0.98}$ is the best log transformed linear fit. 85

5.2 Comparison of node counts for PBCHAFF and ZCHAFF on CNF encodings. Each point corresponds to a CNF problem instance. The x coordinate corresponds to the number of nodes expanded for ZCHAFF and the y coordinate corresponds to the number of nodes expanded by PBCHAFF. The line $f(x) = x$ is plotted as a reference. The curve $f(x) = 0.99x$ is the best log transformed linear fit. 87

5.3 Size of search trees in terms of node counts for PBCHAFF on the pigeonhole problem. The curve shown, $f(n) = 0.83n^{2.04}$, is the best polynomial fit for the data. 91

5.4 Preprocessing and solution time for PBCHAFF on the pigeonhole problem. The best polynomial fit for the preprocessing data is the curve $f(n) = cn^{6.14}$, and the curve $f(n) = dn^{4.84}$ is the best polynomial fit for the solution data. 92

5.5 Comparison of solvers on the pigeonhole problem. Resolution based methods ZCHAFF, RELSAT, and BERKMIN show exponential scaling. OPBDP, and the PBCHAFF version that learns only clausal constraints both show exponential scaling. PBCHAFF shows polynomial scaling. The best polynomial fit for the PBCHAFF solution data is the curve $f(n) = dn^{4.84}$. . . 93

5.6 Comparison of solvers on the pigeonhole problem. 94

5.7 Comparison of solvers on the logistics pigeonhole problem. 101

5.8 Comparison of zCHAFF and PBCHAFF on the logistics pigeonhole problem. Encoding 1 is the original encoding described in Section 5.3.1. Encoding 2 is the same encoding with the addition of ground instances of resolvent (5.26). 105

5.9 Comparison of PBCHAFF and zCHAFF on random planning problems. Each point corresponds to a CNF problem instance. The x coordinate corresponds to the execution time in seconds for zCHAFF and the y coordinate corresponds to execution time in seconds for PBCHAFF. The points above the line $f(x) = x$ represent instances where zCHAFF outperformed PBCHAFF. Points below $f(x) = x$ represent instances where PBCHAFF outperformed zCHAFF. The curve $f(x) = 0.49x^{0.77}$ is the best log transformed linear fit. 109

5.10 Comparison of solvers on satisfiable instances of random planning problems. The plot shows execution time in seconds for zCHAFF on the x axis versus PBCHAFF on the y axis. The best log transformed linear fit is the curve $f(x) = 2.53x^{0.91}$. The line $f(x) = x$ is plotted as a reference. . . . 110

5.11 Comparison of solvers on satisfiable instances of random planning problems. The plot shows the number of nodes expanded by zCHAFF on the x axis versus nodes expanded by PBCHAFF on the y axis. The curve $f(x) = 3.34x^{0.89}$ is the best log transformed linear fit. The line $f(x) = x$ is plotted as a reference. 111

5.12 Comparison of solvers on unsatisfiable instances of random planning problems. The plot shows execution time in seconds for zCHAFF on the x axis versus execution time for PBCHAFF on the y axis. The line $f(x) = x$ is plotted as a reference. 112

5.13 Comparison of solvers on unsatisfiable instances of random planning problems. The plot shows the number of nodes expanded for zCHAFF on the x axis versus PBCHAFF on the y axis. The line $f(x) = x$ is plotted as a reference. 113

6.1 The non-integer solution $(\frac{8}{7}, \frac{18}{7})$ is eliminated by cutting-plane $x_2 \leq 2$ and $x_1 + x_2 \leq 3$. 120

6.2 ZBDD for set of clauses (6.2). Solid edges represent 1-edges, and dashed edges represent 0-edges. 132

LIST OF TABLES

Table Page

5.1 Results for PBCHAFF on pigeonhole problems. The table lists node counts, execution time in seconds for strengthening as a preprocessing method, and execution time in seconds for solution. 90
5.2 Size of the logistics pigeonhole problem as a function of the number of objects. The columns give the number of variables for each instance, the number of clauses in the CNF encoding of the instance, and the number of constraints in the pseudo-Boolean encoding of the instance. 101
5.3 Range of possible values for domain sizes for the randomly generated planning problems. 107

CHAPTER 1

Introduction

This thesis explores new ways of solving the propositional satisfiability problem (SAT). An instance of SAT consists of a set of Boolean variables and a set of disjunctive clauses over the variables. The challenge is to find an assignment of values to variables that satisfies all the clauses, a task known to be NP-complete [17]. SAT is an easy encoding language for many problem domains because it is a simple logical language. For this reason, methods that solve the SAT problem have the potential to be very useful as general purpose solvers.

Because SAT problems are NP-complete, they currently require solution methods that use search. Search methods can explore the search space in a systematic way or move about the space in a nonsystematic way. The DPLL [22] algorithm is the classic systematic approach to solving SAT problems. Its basic structure underlies the most successful SAT methods to date. DPLL-style solvers have undergone a series of advances in recent years that allow the solution of dramatically larger problems than previously. Current solvers now provide important tools for problem solving in many real-world problem domains such as planning [48], microprocessor verification [10, 20, 70], and software design and analysis [46].

DPLL-style solvers have been improved in a number of ways. Extremely fast propagation gives solvers the ability to traverse large search trees quickly, and the size of search trees is reduced by bounded learning, a technique that caches failed sets of assignments to prevent repeatedly re-solving the same subproblem. Non-

standard backjumping and restart methods allow solvers to move more freely about the search space, giving DPLL-style solvers some of the advantages of nonsystematic search methods. Additionally, branching heuristics continue to play an important role in solver performance.

Despite their significant success, systematic satisfiability methods perform poorly on many important families of problems including certain parity problems, the pigeonhole problem, and clique coloring problems. Consider the well known pigeonhole problem stating that $n+1$ pigeons cannot be placed in n holes if only one pigeon can occupy a hole. This simple problem is very hard for current solvers. The most competitive solvers such as ZCHAFF [76] and BERKMIN [35] show time scaling that is exponential in the number of pigeons. This discouraging result is made worse by the fact that the pigeonhole problem and related counting principles are common subproblems in many important problem domains such as planning and scheduling.

If we want to understand what is going on here, we must first understand the role of proof construction in a systematic solver. If an instance of a satisfiability problem is unsatisfiable, the solver must construct some form of a proof that no solution exists. The inference system used by DPLL-style solvers to construct proofs of unsatisfiability is a form of the resolution proof system. For this reason DPLL-style solvers are called resolution-based methods. Resolution is a simple proof system with a single inference rule that allows new clauses to be derived from a set of clauses. Unfortunately, the shortest resolution proof of unsatisfiability for the pigeonhole problem is exponential in the number of pigeons [38]. Clearly the time to construct a proof is at least proportional to the length of the proof. This guarantees exponential scaling on pigeonhole problems and unnecessarily poor performance on problems containing embedded pigeonhole problems. Refining current algorithms may yield improvements, but cannot provide polynomial-time scaling on these problems and many others, unless the underlying proof system used by the solver is changed.

However, the general belief has been that stronger inference systems are impractical to automate. The following quote is from a paper written by Bart Selman, Henry Kautz, and David McAllester detailing a set of challenge problems for the satisfiability community.

> ... attempts to mechanize these more powerful proof systems usually yield no computational savings, because it is *harder* to find the small proof tree in the new system than to simply crank out a large resolution proof. In essence, the overhead in dealing with the more powerful rules of inference consumes all the potential savings [64].

A similar sentiment is expressed by David Mitchell. He argues here that resolution-based methods are prevalent because of their simplicity.

> The prevalence of resolution is ... in part a result of it being a weak enough proof system that it is not too hard to find complete strategies which can be practically implemented. Much stronger proof systems, such as extended resolution (for which no non-trivial lower bounds are known), do not offer much promise for practical implementation [54].

A new direction in the area of SAT research is the construction of lifted solvers. A lifted solver takes the framework of a Boolean SAT solver, such as DPLL, and adapts it to use a stronger (more concise) representation. The original motivation for this approach was to improve performance through the use of concise representations [2, 34, 74]. In fact, many problems of interest are initially encoded concisely in a high-level language. When this high-level encoding is translated to a satisfiability problem, the resulting encoding can easily become too large to be managed by a SAT solver. By working directly with a higher level encoding, lifted solvers avoid the memory issues of overly large CNF encodings.

Lifted solvers were originally thought to be impractical because individual constraints are more complex and are likely to increase the cost of many subprocedures. Studies have since disproved this assumption, showing that stronger representations can improve solver performance, mainly by improving the speed of the unit propagation procedure [2, 34]. The cost of propagation per constraint may be higher, but a single high-level constraint may be a standin for an exponential number of Boolean clauses that would need to be dealt with individually by a conventional satisfiability solver.

A point that has been often overlooked by implementations of systematic lifted solvers is that stronger representations can also be used to improve the strength of

the solver's underlying proof system. A major goal of this thesis is to give guidelines for how to do this within the DPLL framework. We will show that improving proof strength requires more than just using a stronger representation. Stronger inference rules must somehow be incorporated.

The primary inference step of DPLL style solvers is learning. In current resolution-based solvers, learning corresponds to a resolution step. Because learning is the primary inference step in DPLL style solvers, it is an ideal place to incorporate a powerful inference rule. However, the additional complexity of individual constraints means there are more choices in implementation. We will see that choices in the implementation of learning play a key role in determining the strength of the solver's underlying proof system.

We present an implementation of a pseudo-Boolean SAT solver that automates a proof system properly stronger than the proof systems of current resolution-based solvers. Pseudo-Boolean representation is commonly used by the operations research community and a constraint consists of a linear inequality over Boolean variables. The major question is whether the stronger proof system actually leads to improvements in solver performance. There are many ways things can go wrong. First, the solver may not be able to find short proofs for a particular problem instance, even though short proofs exist. The cutting-plane proof system associated with pseudo-Boolean representation allows short proofs of the pigeonhole problem; however, that does not mean that all cutting-plane proofs of the pigeonhole problem are short. The solver may produce a cutting-plane proof that is as long as a resolution proof. Second, the solver may produce a short proof (corresponding to a small search tree) but the cost of expanding a node may increase. The additional complexity of constraints may make subprocedures more expensive. The benefits of the reduced search tree may be outweighed by the higher cost of expanding a node, yielding no real improvement in performance. To examine these possibilities, we will compare the performance of our solver with the performance of resolution-based methods on many problem domains. We give in-depth comparisons of solvers on the pigeonhole problem and planning problems.

1.1 Overview of Thesis

Most systematic SAT solvers are based on the DPLL algorithm, but they also employ a variety of methods that improve performance over simple DPLL. These methods include clause learning combined with a restriction method for bounding the size of the learned clause set, indexing schemes for quickly identifying unit propagations, branching heuristics, and backjumping methods. The DPLL algorithm, together with these improvements, form what we call the DPLL framework. In Chapter 2 we review the elements of this framework, survey the relevant research, and try to identify the elements of this framework that make it successful.

Chapter 3 explores the role of systematic satisfiability solvers as proof systems and reviews some lower bounds for current satisfiability solvers. We give an introduction to some concepts from the field of proof complexity that will allow us to analyze and compare the strength of different proof systems. We discuss work showing exponential lower bounds on the performance of resolution-based solvers like RELSAT [5], zCHAFF [76], and BERKMIN [35]. These bounds follow from the fact that all resolution proofs of the pigeonhole principle are known to be exponential in length. We show that the theoretical lower bound is mirrored in experimental results showing exponential runtime scaling for these solvers on pigeonhole problems.

Chapter 4 describes our implementation of a pseudo-Boolean version of zCHAFF. The hope is that our implementation will respect the lessons from both Chapter 2 and Chapter 3, improving the strength of the underlying proof system while retaining the elements that make current satisfiability solvers successful. First we describe pseudo-Boolean representation and review some associated proof complexity results. We show that pseudo-Boolean representation can be exponentially more concise than CNF and, when combined with cutting-plane inference, it is possible to write polynomial-length proofs of the pigeonhole principle. We then make some observations about why strong representations lead to strong proof systems. These observations motivate our solver implementation. The remainder of Chapter 4 gives implementation details. We revisit the methods of Chapter 2, describing a pseudo-Boolean implementation for each element of the DPLL framework. A solver that uses pseudo-Boolean representation

can also apply additional methods that cannot be applied in traditional resolution-based methods. In Section 4.6, we show how a form of the coefficient reduction method from operations research can be applied within the DPLL framework.

In Chapter 5, we answer the question of whether automating strong proof systems leads to improvements in solver performance. We start by comparing performance of the pseudo-Boolean solver with its resolution-based counterpart on a large set of CNF benchmarks. Next we compare performance on pigeonhole problems, and finally we compare performance on planning problems.

In Chapter 6, we discuss related work. We survey some standard operations research methods for solving integer programming problems. Pseudo-Boolean representation is a subclass of a more general operations research representation that allows linear inequalities over positive integer variables. We review some applications of integer programming methods to solve satisfiability problems. We also review other work on lifted SAT solvers. Chapter 7 gives concluding remarks and discusses ideas for future work.

CHAPTER 2

Systematic SAT Solvers

In this chapter we introduce the family of algorithms for solving satisfiability problems that are based on the Davis-Putnam-Logemann-Loveland procedure [22]. Descendants of the DPLL algorithm are currently the best methods for solving general satisfiability problems and provide competitive solutions in a number of important problem domains such as microprocessor testing and verification [10, 20, 70] and planning [48].

Definition 2.0.1. *An instance of* satisfiability *is a set of Boolean variables* $U = \{v_1, v_2, \dots, v_n\}$ *and a set of clauses* $C = \{c_1, c_2, \dots, c_m\}$. *A* clause *is a disjunction of literals, where a* literal *is a Boolean variable* v_i, *or its negation* $\overline{v}_i$. *A clause is* satisfied *if and only if any one of its literals evaluates to true. A* solution *to a SAT problem is an assignment of values to variables that satisfies every clause; if no such assignment exists, the instance is called* unsatisfiable.

The SAT problem is known to be NP-complete [17].

The performance of the DPLL procedure has been improved in many ways. We restrict our discussion in this chapter to improvements that do not require a representational change. The benefits of strengthening the underlying problem representation will be discussed in subsequent chapters. In Section 2.2 we introduce techniques that improve performance by taking advantage of structure. Structured problems might be best defined as problems in which subproblems recur. Earlier DPLL implementa-

tions were notoriously bad at solving structured problems because each encountered subproblem was solved from scratch, even if the subproblem had been previously encountered many times. The key technique here is learning, a technique that caches solutions to subproblems to avoid the need to repeatedly re-solve the subproblem. Learning alone proved to be impractical because the set of learned items could become unmanageably large. The development of learning combined with a suitable method to restrict the size of the set of cached solutions signaled an important turning point for satisfiability methods. Solvers then could solve interesting structured problems that were more representative of real-world problems [6]. Consequently, the focus of satisfiability research moved away from the randomly generated satisfiability problems typically considered in earlier papers.

The DPLL procedure proceeds by selecting a subset of variables and fixing their values, a process known as *branching*. Then the unit propagation procedure identifies variables whose values are forced by the branching choices. Unit propagation continues to be the major computational task of DPLL-style algorithms, taking up over eighty percent of computation time.

In Section 2.3, we look at techniques that improve performance by reducing the cost of expanding a node. These techniques focus on reducing the amount of time spent in the unit propagation procedure. Efforts here have focused mainly on building better data structures.

Branching heuristics continue to play an important role in reducing the size of the search space for satisfiability problems. Our presentation of branching heuristics is divided to reflect the fact that heuristics developed in different contexts. Early heuristics were designed to work with simple DPLL implementations, and most experiments were run on randomly generated SAT problems. More recent heuristics are designed to work in concert with learning, and focus on structured problem domains such as planning and microprocessor verification.

2.1 In the Beginning: DPLL

2.1.1 Davis-Putnam-Logemann-Loveland

As we stated earlier, the techniques discussed in this chapter are all based on the Davis-Putnam-Logemann-Loveland method [22]. The algorithm takes a valid partial assignment and attempts to extend it to a valid full assignment by incrementally assigning values to variables. A partial assignment is a set of variable value pairs of the form $v_i = V$ such that v_i is a variable and $V \in \{\texttt{true}, \texttt{false}\}$. Literals provide a shorthand notation for variable assignments with a representing $a = \texttt{true}$ and $\bar{a}$ representing $a = \texttt{false}$.

Definition 2.1.1. *A* partial assignment *is an ordered sequence of literals.*

$$\{l_1, l_2, \ldots, l_n\}$$

A partial assignment P for a set of clauses C is invalid *if there is a clause c in C such that every literal in c is unsatisfied by P. Otherwise, a partial assignment is considered* valid. *A partial assignment P'* extends *a partial assignment P if and only if $P' = \{P, l_i, l_{i+1}, \ldots\}$.*

DPLL creates a binary search tree where each node corresponds to a partial assignment. The tree is explored using depth-first search with backtracking. Backtracking occurs if the algorithm reaches a state where there is no valid assignment for a particular variable. The algorithm terminates when a solution is found or when the entire space has been explored.

Procedure 2.1.2 (Davis-Putnam-Logemann-Loveland). *Given a* SAT *problem C and a partial assignment P of values to variables, to compute* DPLL(C, P)*:*

1 $P \leftarrow$ UNIT-PROPAGATE(C, P)
2 **if** P contains a contradiction
3   **then return** FAILURE
4 **if** P assigns a value to every variable
5   **then return** SUCCESS
6 $l \leftarrow$ a variable not assigned a value by P
7 **if** DPLL$(C, \{P, l\})$ = SUCCESS
8   **then return** SUCCESS
9   **else return** DPLL$(C, \{P, \bar{l}\})$

2.1.2 Unit Propagation

The unit propagation procedure identifies clauses that have no satisfied literals and exactly one unvalued literal. A clause with these properties is called a *unit clause*. In each such clause, the unvalued literal must be valued `true` to satisfy the clause. This process is repeated until a contradiction is encountered, a solution is found, or no more clauses meet the necessary conditions.

Procedure 2.1.3 (Unit propagation). *To compute* UNIT-PROPAGATE(C, P)*:*

1 **while** no contradiction is found **and** there is a $c \in C$ that under P
  has no satisfied literals and exactly one unassigned literal
2   **do** $l \leftarrow$ the literal in c unassigned by P
3         $P \leftarrow \{P, l\}$
4 **return** P

2.1.3 Early Branch Heuristics

Before the development of successful learning techniques, branching heuristics were the primary way of reducing the size of the search space. The main approach

was to try to encourage a cascade of unit propagations. The result of such a cascade is a smaller and more tractable subproblem. There are two popular classes of branching rules that are based on this idea. The MOMS rule branches on the variable that has maximum occurrences in clauses of minimum size [21, 28, 45, 47, 58]. This gives an approximation of the number of unit propagations that a particular variable assignment would cause. It has the advantage of being inexpensive to calculate. Another heuristic is the unit propagation rule [21, 30]. This rule calculates the full amount of propagation caused by a branching choice. Given a branching candidate v_i, the variable is independently fixed to `true` and `false` and the unit propagation procedure is run on each subproblem. The number of unit propagations caused by an assignment becomes a weight used to evaluate branching choices. Unlike the MOMS heuristic, this rule is exactly correct at determining the number of unit propagations an assignment will cause. It is also considerably more expensive to compute. A logical approach would be to combine these two techniques [21, 50]. The cheaper MOMS heuristic is used to determine which branching candidates should be investigated with the more expensive unit propagation heuristic. This was shown to outperform either technique used alone [50].

The branching heuristics presented above were all developed to work with a simple DPLL algorithm. The studies cited compare performance of strategies on randomly generated k-SAT problems. These particular algorithms are among the best methods for randomly generated SAT problems; however, for reasons we discuss in the next section, they perform poorly on large structured problems. The heuristics discussed here may still be viable within the context of learning algorithms, but this has not yet been demonstrated. In the next section, we introduce a difficulty encountered by backtracking algorithms called *thrashing* that cannot be addressed by branching heuristics alone.

2.2 Tackling Structured Problems with Learning

The simple DPLL methods described in Section 2.1 suffer from a condition called *thrashing* that occurs when a method re-solves the same subproblem over and over

again. Thrashing is inevitable for simple DPLL methods because they have no way of identifying whether a subproblem has been seen before. As a result, each encountered subproblem must be solved from scratch. For this reason, simple DPLL methods perform poorly on structured problems where subproblems may be encountered over and over again.

Imagine solving a SAT problem with variables $x_1, x_2, \ldots, x_{100}$, having successfully valued the variables $x_1, \ldots, x_{49}$. In this problem there happens to be a subset of constraints involving only the variables $x_{50}, \ldots, x_{100}$ that together imply that $x_{50} =$ `true`. If we begin by setting $x_{50} =$ `false`, it will require some degree of searching to discover our mistake. When we finally do, we backtrack to x_{50}, set it to `true`, and continue on. Unfortunately, if later we need to backtrack to a variable set before x_{50}, for instance x_{49}, we are in danger of setting x_{50} to `false` again. We could potentially solve the same subproblem many times. The solution to this problem is to record the information that we discovered. In this example the pertinent information is that x_{50} must be `true`. We can record this information by adding the unary clause x_{50} to the set of clauses. Now we can immediately prune any subproblem with $\{x_{50} =$ `false`$\}$. This technique, called *learning*, was introduced by Stallman and Sussman in dependency directed backtracking [67] and will be described in detail below.

2.2.1 Resolution: The Inference of Learning

Learning occurs when a contradiction is encountered, and is essentially an analysis of what went wrong. The learning procedure determines the subset of the current assignment that caused the contradiction to occur. A new clause called a *nogood* is generated that prunes exactly the set of assignments that led to the contradiction. Generating a nogood serves two purposes: first, it is used in *backjumping* [32], a process that determines how far to backtrack; and second, it can be added to the constraint set to ensure the "bad" set of assignments will be avoided in the future, a technique called *learning* [67].

At the heart of the learning method is resolution. Resolution is the proof system generally associated with CNF representation. It has a single inference step defined as follows:

$$\frac{\begin{array}{c} a_1 \vee \cdots \vee a_k \vee l \\ b_1 \vee \cdots \vee b_m \vee \bar{l} \end{array}}{a_1 \vee \cdots \vee a_k \vee b_1 \vee \cdots \vee b_m}$$

Two clauses resolve if there is exactly one literal l that appears positively in one clause and negatively in the other. A new clause is derived by disjoining the two clauses and removing both l and $\neg l$. If a literal appears twice in the resulting clause, the clause can be rewritten with the literal appearing only once. This is known as factoring. The resolution inference step is ideal for analyzing contradictions in a DPLL search tree.

Analyzing the cause of a conflict requires that we maintain some record of why each variable assignment was made. This is achieved by annotating each assignment in our partial assignment with a *reason* (also referred to as an *asserting clause*).

Definition 2.2.1. *An* annotated partial assignment *is an ordered sequence*

$$\{(l_1, c_1), \ldots, (l_n, c_n)\}$$

of literals and clauses, where c_i is the reason *for literal l_i and either c_i =* `true` *(indicating the l_i was a branch point) or c_i is a clause such that l_i is a literal in c_i, and c_i is a unit clause under $\{l_1, l_2, \ldots, l_{i-1}\}$. An annotated partial assignment will be called* sound *with respect to a set of constraints, C, if and only if for each reason c_i, $c_i \in C$ or c_i is derived from C by resolution.*

After the literals $l_1, \ldots, l_{i-1}$ are all set to true, either l_i is a branch choice, or it is possible to conclude l_i from c_i by unit propagation.

Procedure 2.2.2 (DPLL-with-Learning). *Given a* SAT *problem* C *and a partial assignment* P *of values to variables, to compute* DPLL-WITH-LEARNING(C, P)*:*

```
1   while P is not a full assignment
2         do P ← UNIT-PROPAGATE(C, P)
3            if P contains a contradiction
4               then v ← contradiction variable
5                    c1 ← reason associated with v
6                    c2 ← reason associated with v̄
7                    if BACKJUMP(resolve(c1, c2), P) = FAILURE
8                       then return FAILURE
9               else l ← a literal not assigned a value by P
10                   P ← {P, (l, true)}
11  return SUCCESS
```

Procedure 2.2.3 (Backjump). *Given a nogood* c *that is unsatisfied under an annotated partial assignment* $P = \{(l_1, c_1), (l_2, c_2), \ldots, (l_m, c_m)\}$*, to compute* BACKJUMP$(c, P)$*:*

```
1  if c is the empty clause
2      then return FAILURE
3  l_i ← literal in P with maximum i such that l̄_i satisfies c
4  P ← {(l_1, c_1), (l_2, c_2), ..., (l_i, c_i)}
5  if c_i = true
6      then add c to the clause set
7           P ← {(l_1, c_1), (l_2, c_2), ..., (l_{i-1}, c_{i-1}), (l̄_i, c)}
8      else return BACKJUMP(resolve(c_i, c), P)
9  return SUCCESS
```

In Figure 2.1(a) we show an example of this process for the small set of clauses

$$a \vee \bar{b}$$

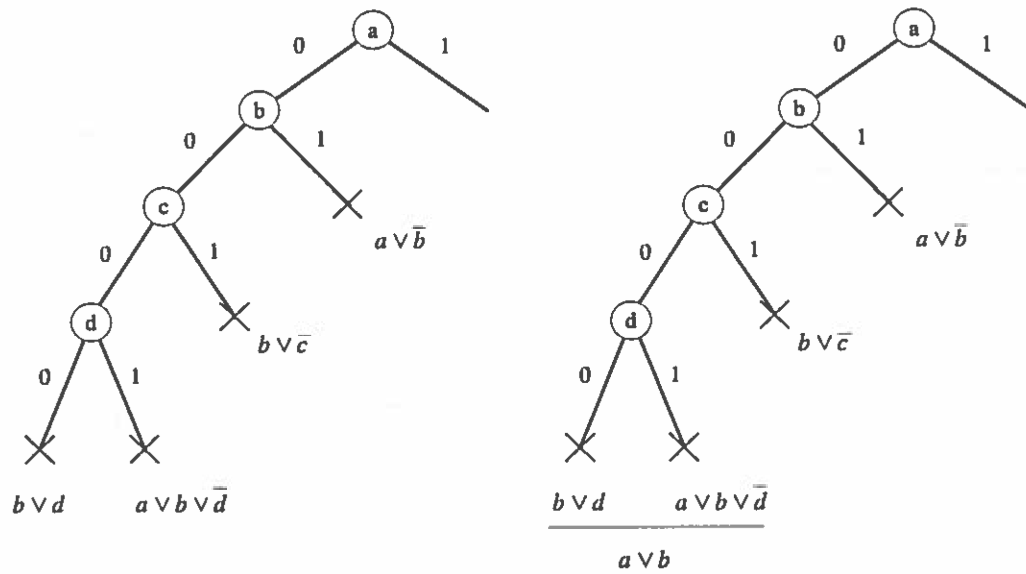
$$b \vee \bar{c}$$

$$a \vee b \vee \bar{d}$$

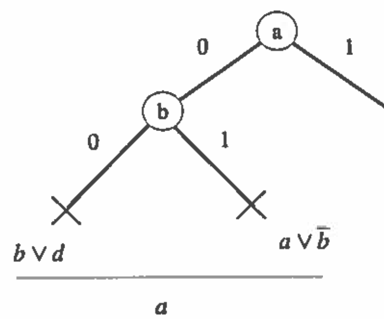
$$b \vee d$$

The original clause set contains no unit propagations under the empty partial assignment, so the search will begin at the top of the graph with a branch decision. The variable a is set to `false`, giving the partial assignment $P = \{(\bar{a}, \texttt{true})\}$. In our graph we represent `true` and `false` by 1 and 0 respectively and continue this format throughout our discussion. At this point, the clause $a \vee \bar{b}$ generates a unit propagation forcing the assignment $\bar{b}$. The partial assignment is extended to contain the new assignment, together with its motivating reason giving $P = \{(\bar{a}, \texttt{true}), (\bar{b}, a \vee \bar{b})\}$. The branch $b = 1$ is marked pruned and labeled with the pruning clause. Continuing with unit propagation, the partial assignment is extended with the additional assignments $\{(\bar{c}, b \vee \bar{c}), (\bar{d}, a \vee b \vee \bar{d}), (d, b \vee d)\}$ and pruned branches are labeled with their respective reasons. A contradiction occurs since there is no valid assignment for variable d.

Note that the reasons for assignments $\bar{d}$ and d resolve. This is a natural result of the way DPLL methods partition the search space. In Figure 2.1(b) we show the resolvent of these clauses. The resolvent $a \vee b$ is called a learned clause or a nogood. A nice property of the resolvent is that we can construct from it the exact set of assignments that need to be revised in order to avoid the immediate contradiction. We do this by simply creating a set containing the negation of each literal in the clause. In our example, this set would contain $\{\bar{a}, \bar{b}\}$. This set of assignments is sometimes called the *conflict set*. One of these assignments must be revised before we can progress forward again. The solver now backjumps to the nearest assignment in this set, in this case the assignment $\bar{b}$, giving partial assignment $P = \{(\bar{a}, \texttt{true}), (\bar{b}, a \vee \bar{b})\}$. Note that we skip over the assignment $\bar{c}$. The analysis tells us that the assignment $\bar{c}$ does not contribute to the cause of the conflict so changing its assignment will not remove the inconsistency. Continuing on in Figure 2.1(c), we find that we are unable to revise the assignment $\bar{b}$ because it has an associated reason. Here we repeat the



(a) (b)



(c)

FIGURE 2.1: Learning in a DPLL search tree

process, generating the unary clause a by resolving the reasons for $\bar{b}$ and b and calling backjump again. We backtrack to assignment $\bar{a}$ and find that this assignment has no associated reason because it was a branch decision. We now try the assignment a and proceed forward again.

We now add the unary clause a to the set of clauses. If our example were a subproblem within a larger search tree, the new clause a ensures that we never attempt to set $a = 0$ again and therefore we will never need to repeat this particular analysis. In this case we avoid re-solving a very small subproblem, but in practice, a single learned clause may prune a very large subtree and be derived through extensive branching and backtracking. These prunings add up over the entire search, leading to dramatically smaller search trees. Adding new learned clauses to the set of clauses is a form of inference, and more specifically, within the context of a DPLL search tree, learning is resolution. The resolution inference rule captures exactly the task of analyzing conflicts within a DPLL search tree.

2.2.2 Bounded Learning

Learning new constraints reduces the size of the search space by eliminating parts of the space that cannot contain solutions. Unfortunately, learning also indirectly increases the cost of propagation. The cost of unit propagation is a function of the size of the constraint database, and the number of constraints learned can be exponential in the size of the problem. The algorithm spends more time managing its large database of constraints and performance degrades. A solution to this problem is to limit the size of the constraint set in some way to prevent an unmanageable number of constraints from accumulating. It would be useful to have a way to ensure that the set of learned clauses is polynomially bounded in the size of the problem. To achieve this we need a method to determine when to cache a learned clause, and when to discard a clause from the existing cache. Ideally we'd like to keep the clauses that will be most useful for pruning the search space. In this section we review heuristical methods for choosing which learned clauses to retain so as to restrict the overall size of the learned clause set.

Length-bounded Learning

The first method proposed for restricting a learned clause set was length-bounded learning [23, 31]. In length-bounded learning we discard all clauses with length greater than some constant bound k. A clause of length l will prune $\frac{1}{2^l}$ of the possible assignments of values to variables. The size of the search space pruned by a clause is inversely proportional to its length. This is the motivation for retaining shorter clauses over longer clauses. DPLL with length-bounded learning improves substantially over DPLL with unrestricted learning [23, 31].

Relevance-bounded Learning

Length is not the best measure of the value of a learned clause. If we look at any particular subproblem within the search, we find long clauses that are useful for pruning the search space and much shorter clauses that are not useful at all for the task at hand. Imagine we have just expanded the node with the partial assignment $P = \{\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{e}\}$. All other variables are unvalued. Suppose also that we have learned the following two clauses in our search thus far:

$$a \vee b \vee c \vee d \vee e \vee f \tag{2.1}$$

$$\overline{a} \vee \overline{b} \vee g \tag{2.2}$$

The algorithm now solve the subproblem below the node given by the current partial assignment. The first clause (2.1) is unit and tells us to set f and to prune the branch with $\bar{f}$. The second clause (2.2) cannot be used to prune the subproblem's search space at all, since it is already satisfied by the current partial assignment. The longer clause is more useful than the shorter one in solving the immediate subproblem. Length-bounded learning would discard the long clause and retain the shorter one.

What is needed is a measure of length that considers the current context. The length of (2.1) should really only be one since the other literals in the clause are currently false. Relevance-bounded learning defines length relative to the current partial assignment. This idea originated in dynamic backtracking [33] and was generalized by Bayardo and Miranker who defined a general *irrelevance* measure [5].

Definition 2.2.4. *Given a clause c and partial assignment P, the* irrelevance *of c under P is defined as the number of satisfied literals in c plus the number of unvalued literals in c.*

Their implementation, called RELSAT, retained only those clauses whose irrelevance was less than a given bound. As the position in the search space changed, clauses that were no longer relevant were discarded to make room for more relevant ones.

Relevance-bounded learning, like length-bounded learning, also has the property that it retains all clauses with length less than the relevance bound k. This follows because the irrelevance of a clause can never exceed its length. Additionally, relevance-bounded learning also retains long clauses when they are relevant to the current search position.

Experiments show that relevance-bounded learning makes better use of space resources than does length-bounded learning, and even when relevance-bounded learning is restricted to linear space, the performance is comparable to that of unrestricted learning [6, 5]. Like length-bounded learning, relevance-bounded learning also maintains a set of learned clauses that is polynomially bounded in the size of the original problem [26].

Hybrid Approaches

Some solvers employ a hybrid approach. The ZCHAFF algorithm [55] uses a relevance bound and a larger length bound. The length bound is often significantly larger than the relevance bound. Clauses must meet both the relevance bound and the length bound to be retained. Another approach is taken by the BERKMIN [35] algorithm. Here, the set of nogoods is partitioned into two separate groups based on how recently the nogoods were acquired. The fractional sizes of these groups are $\frac{15}{16}$ and $\frac{1}{16}$ with recently learned clauses being the larger of the two groups. A relatively large length bound is used to cull the recently learned clauses while a smaller length bound is used to aggressively cull the smaller group of older clauses. There are currently no studies comparing these hybrid approaches to pure length-bounded or relevance-bounded methods.

Unique Implication Points

Unique implication points provide another way of restricting the number of clauses that are learned. UIPs were first introduced as part of the GRASP algorithm [52] and they occur in DPLL-style search trees. If we consider a node in the search tree, we can count the number of branch decisions between the root and the node. The branch decisions divide the path into decision levels with increasing index as we descend into the tree.

Definition 2.2.5. *Given a partial assignment* $P = \{(l_1, c_1), (l_2, c_2), \ldots, (l_m, c_m)\}$*, the set of branch decisions* $(l_i, true) \in P$ *partition* P *into* decision levels $\{d_0, d_1, d_2, \ldots, d_l\}$ *where a level* d_i *consists of the* i^{th} *branch decision* $(l_k, true)$ *together with the following assignments* $(l_{k+1}, c_{k+1}), \ldots, (l_j, c_j)$ *up to but not including branch decision number* $i + 1$.

Consider a clause that is learned in response to a conflict. When we learn a clause, it is unsatisfied with all its literals being valued unfavorably.

Definition 2.2.6. *Given a nogood* c *and a set of decision levels* $\{d_0, d_1, d_2, \ldots, d_j\}$*, the clause* c *is a* UIP clause *if it contains exactly one literal* l_i *such that* $l_i \in d_j$.

Experimental results show that UIP clauses are particularly useful for pruning the search space [76]. However, it is not fully understood why UIP clauses are so effective.

2.2.3 Non-Standard Backtracking

When a nogood is learned, the algorithm backs up, removing assignments from the partial assignment until the nogood becomes satisfiable again. The zCHAFF algorithm uses a non-standard backjump method. Instead of backing up to where the nogood is satisfied, it backs up further to the decision level where the nogood would have become unit had the nogood been around at the time.

ZCHAFF begins its backjump using standard backjumping, iteratively generating new nogoods until a UIP clause is generated. Once a UIP clause is generated, the

clause is added to the clause set and non-standard backjumping is applied, ending the backjump process.

Procedure 2.2.7 (Backjump). *Given a nogood c that is unsatisfied under annotated partial assignment* $P = \{(l_1, c_1), (l_2, c_2), \ldots, (l_m, c_m)\}$*, to compute* BACKJUMP$(c, P)$*:*

1 **if** c is the empty clause
2 **then return** FAILURE
3 **if** c is a UIP clause
4 **then return** NON-STANDARD-BACKJUMP(c, P)
5 **else**
6 $l_i \leftarrow$ literal in P with maximum index i such that $\bar{l}_i$ satisfies c
7 **return** BACKJUMP(`resolve`$(c_i, c), P)$

Procedure 2.2.8 (Non-Standard-Backjump). *Given a nogood c that is unsatisfied under annotated partial assignment* $P = \{(l_1, c_1), (l_2, c_2), \ldots, (l_m, c_m)\}$*, to compute* NON-STANDARD-BACKJUMP(c, P)*:*

1 add c to the clause set
2 $l_i \leftarrow$ literal in P with minimum index i such that c is unit under $\{(l_1, c_1), (l_2, c_2), \ldots, (l_i, c_i)\}$
3 $l \leftarrow$ the literal in c that is unit under $\{(l_1, c_1), (l_2, c_2), \ldots, (l_i, c_i)\}$
4 $d \leftarrow$ decision level such that $l_i \in d$
5 $l_k \leftarrow$ literal in d with maximum index k
6 $P \leftarrow \{(l_1, c_1), (l_2, c_2), \ldots, (l_k, c_k), (l, c)\}$
7 **return** SUCCESS

The advantage of procedure NON-STANDARD-BACKJUMP is that earlier decisions can be evaluated again in light of new information. Bad decisions made early in the search can be extremely costly. The size of the search tree can vary greatly depending on the order and choice of branch decisions. Consider a choice made early in the search process that creates a subproblem that has no solutions and requires a lengthy proof

of unsatisfiability. DPLL with standard backjumping is a form of depth-first search and therefore is forced to search the entire subtree.

Learning new clauses can improve the quality of branch decisions. This includes previously made decisions in addition to those the solver is about to make. Algorithms cannot fully take advantage of this aspect of learning in a depth-first search setting. Non-standard backtracking methods give algorithms more freedom in how they move about the search space and enable solvers to recover from early mistakes.

Non-standard backjumping can jump back over multiple branch decisions. At first glance, this appears to abandon what might be significant progress toward solving the current subproblem. However, if the clauses used to prune the search space along the abandoned path are retained, then the intermediate work is not really lost. Perhaps more concerning is that DPLL with non-standard backjumping is no longer a depth-first search algorithm. The completeness of this algorithm needs to be established.

Proposition 2.2.9. *Procedure 2.2.2 with non-standard backjumping Procedure 2.2.7 is complete.*

Proof. First, we discuss a few preliminaries. When a contradiction occurs, we say the contradiction occurred at decision level d_i where d_i is the decision level with the highest index. When a contradiction occurs, a UIP clause is always eventually learned. Let c be a learned clause that is not a UIP clause, having more than one literal assigned a value at the current decision level. Because c is unsatisfied under the current partial assignment P, the literals in c valued at the current decision level are falsified by assignment P. At least one of these assignments has an associated reason that resolves with c generating a new nogood allowing us to backjump further. This is seen in line 7 of Procedure 2.2.7. So a UIP clause is in fact necessary to terminate the backjump procedure.

Next we define the notion of progress for a decision level d_i. Each time the procedure backjumps to a decision level d_i (without backjumping over it), the decision level is appended with a new assignment implied by the learned nogood, as seen in line 6 of Procedure 2.2.8. Each time we backjump to a decision level and append a new assignment we make *progress* at that level. Assuming we never backjump over

decision level d_i to a decision level with lower index, the number of assignments that can be appended to decision level d_i is bounded by the number of variables because each variable can only be appended once. Consequently, we can only backjump to decision level d_i a finite number of times before a contradiction occurs at decision level d_i or a solution is found.

We prove that progress is made at every decision level by induction on the maximum number of decision levels seen during the solution of a problem. For the base case we show that if a contradiction occurs at level m, then progress occurs for some decision level with index less than m. For the inductive case, we assume that progress occurs for some decision level with index less than i and show that if no solution is found, then progress eventually occurs at some decision level with index less than $i - 1$. If a solution is not found, we can conclude that progress will occur at d_0 eventually leading to a contradiction at d_0 and the derivation of the empty clause.

For the base case, let a contradiction occur at decision level d_m. We learn a UIP clause and backjump to some decision level with index less than m and make progress at that level.

For the inductive case we assume that we make progress at some decision level d_j such that $j < i$. If $j < i - 1$ then we make progress at a decision level with index less than $i - 1$. If $j = i - 1$ then we make progress at decision level d_{i-1}. If we continue searching, we will either at some point backjump to a decision level with index less than $i - 1$ or we will continue to append assignments to decision level d_{i-1} until a contradiction occurs at decision level d_{i-1} or a solution is found. If a contradiction occurs we again learn a UIP clause and backjump to a decision level with index less than $i - 1$.

We can conclude that if no solution is found, progress is made at decision level d_0 and the empty clause is eventually derived. □

2.2.4 Branching Heuristics That Work with Learning

Branching choices and learning are deeply related to each other. The addition of learning to a DPLL style algorithm will have a significant effect on branching decisions

for all of the branching strategies we have discussed. As new clauses are learned and added to the clause database, they will change the literal counts used in approximations, and may also change the number of unit propagations an assignment will cause. The reverse is also true in that the choice of branch variables influences the clauses that are learned.

Very little is known about the relationship between branching and learning. Some newer heuristics begin to explore this relationship and are designed to work in concert with learning. The Variable State Independent Decaying Sum (VSIDS) heuristic [55] used by zCHAFF maintains a count of the number of times each literal occurs in the theory being solved. Each time a new clause is added, the counter associated with each literal in the clause is incremented by one. This is done for learned clauses as well as clauses contained in the original problem description. The heuristic prefers an unassigned variable with a high count value. Periodically all counts are divided by a constant factor. The routine reduction of the counts causes a bias toward branching on variables occurring in recently learned clauses. This rule, like the MOMS rule, is very inexpensive to calculate.

The BERKMIN heuristic builds on the zCHAFF idea, but is far more dynamic at responding to recently learned clauses. The BERKMIN [35] heuristic prefers to branch on variables that are unvalued in the most recently learned clause that is not yet satisfied. The heuristic also maintains VSIDS-like counts that are used to choose among unvalued variables in the clause. The domain value of the chosen variable l is selected so as to even out the occurrences of the literals l and $\bar{l}$ in the set of learned clauses. The BERKMIN algorithm compares favorably with zCHAFF on a number of benchmarks [35]. Unfortunately, it is difficult to determine whether the observed improvements in performance result from the BERKMIN branching heuristic because the algorithms also have other significant differences.

Branching heuristics for satisfiability solvers have become quite numerous. There is still much work to be done in evaluating their relative effectiveness. The studies comparing older heuristics were done with non-learning DPLL algorithms mainly on random 3-SAT instances [21, 28, 29, 30, 45, 47, 50, 58]. It is unclear how these heuristics fare in the presence of learning and on more structured real world problems.

Newer heuristics are designed to work with learning, but no rigorous experiments have been done comparing these heuristics to others.

2.3 The Benefits of Fast Propagation

2.3.1 Unit Propagation: The Main Loop

It is common knowledge within the SAT community that the bulk of computation time for DPLL-based solvers is spent in the unit propagation procedure. To verify this we profiled both zCHAFF and RELSAT on a variety of problems from the following benchmark suites:

- Microprocessor test and verification benchmarks
  http://www.ece.cmu.edu/~mvelev
- DIMACS benchmarks
  ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf
- SatLib benchmarks
  http://wwww.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html

Experiments were run on an Intel Pentium 4M running at 1.6GHz with a timeout occurring at 100 seconds. The percent of CPU time devoted to unit propagation for zCHAFF and RELSAT is shown in Figures 2.2 and 2.3 respectively.

The fraction of time that is spent in unit propagation increases with the difficulty of the problem and if we consider only those problems that timed out at 100 seconds (i.e. problems that required more processing time) the average time spent in unit propagation is 89.4% for zCHAFF and 81.1% for RELSAT. Because unit propagation is the major computational task of DPLL-style methods, it has been the focus of optimization efforts.

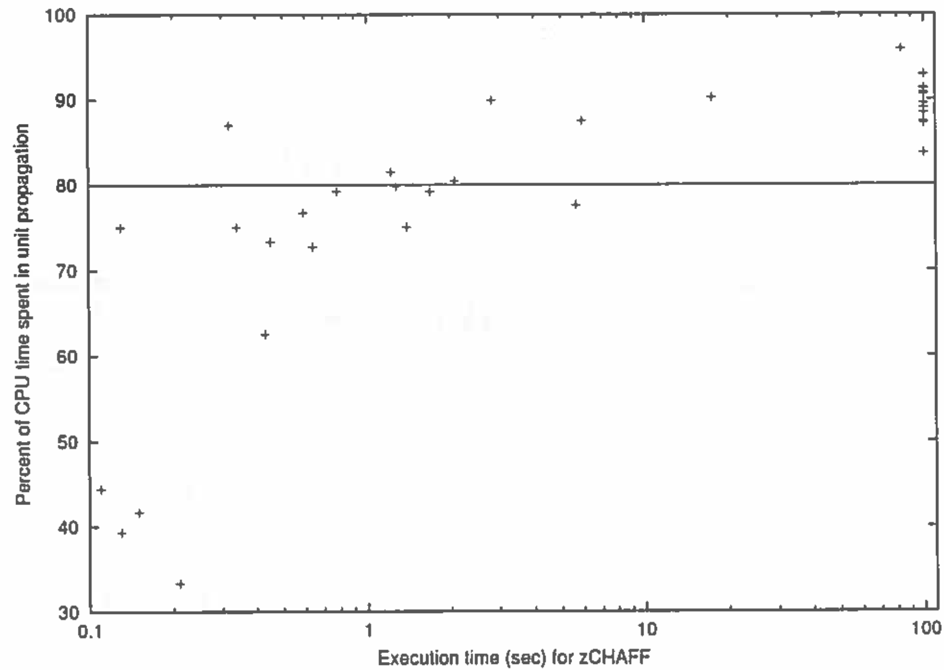


FIGURE 2.2: Percent of CPU time spent in unit propagation for ZCHAFF

2.3.2 Data Structures

Within the unit propagation procedure, the bulk of computation time is spent on the subtask of identifying clauses with no satisfied literals and exactly one unvalued literal. A naive approach is to examine every clause in the database. Each clause is walked to determine the number of unvalued literals and the number of satisfied literals.

Literal Indexes

The naive approach can be improved by creating a literal index. For each literal, we create a list of pointers to the set of clauses that contain that literal. When an assignment is made it is only necessary to search the set of clauses containing the negation of the assignment. For example, if we make the assignment x_1, we search only the set of clauses containing the literal $\overline{x}_1$ for new unit propagations.

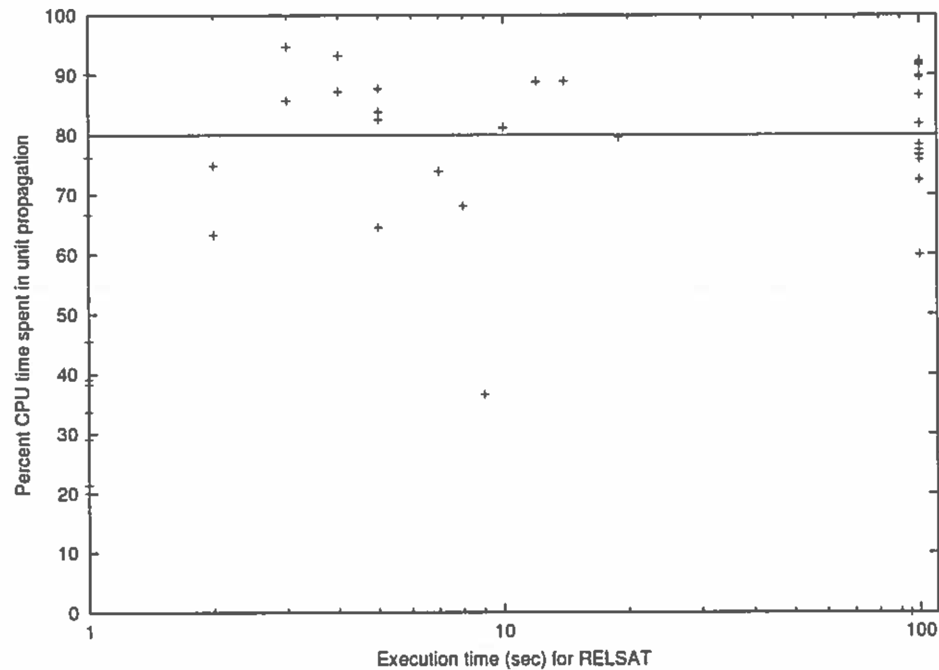


FIGURE 2.3: Percent of CPU time spent in unit propagation for RELSAT

Count-Based Methods

Count-based indexing improves on this further by maintaining for each clause a cached value of the number of unsatisfied literals and the number of satisfied literals in the clause. The number of unvalued literals can be calculated from these counts given the total number of literals in the clause. There is some overhead involved in incrementally maintaining these counts when variables are labeled and unlabeled. If we make the assignment x_1, we must increment the count of satisfied literals for all clauses containing x_1, and increment the count of unsatisfied literals for all clause containing $\overline{x}_1$. If we later unvalue x_1, we must decrement these counts accordingly. The advantage is that we can tell directly from the counts whether a clause is unit and avoid walking the length of the clause. The clause is only walked if the clause is unit, to identify which literal is unvalued. Count-based indexing improves significantly over naive indexing [21].

Watched Literals

The count-based method maintains a full literal index. For every literal in a given clause, there is a corresponding entry in that literal's index. The main idea of the watched literals [75, 76] method is that a full literal index is unnecessary. It is sufficient to have only two literals indexed per clause provided that they are both unvalued under the current partial assignment. A clause that is indexed in this way cannot be unit regardless of how other literals in the clause evaluate under the current partial assignment. This leads to smaller literal indexes and fewer clauses that need to be examined for unit propagations.

Definition 2.3.1. *Let c be a clause and let S be a subset of the literals in c of size $|S| = 2$. S is a* watching set *for c under partial assignment P if the literals in S are both unvalued or one of them is satisfied under P.*

Proposition 2.3.2. *If S is a watching set for a clause c under partial assignment P, then the clause c cannot be unit under P.*

Proof. A clause is unit when it has no satisfied literals and exactly one unvalued literal. The clause c has either two unvalued literals or at least one satisfied literal and therefore is not unit. □

The definition of watched literals assumes that clauses have length greater than one. We can assume that unary clauses are valued in the first decision level and retain their values for the duration of the search and therefore do not need watching sets. This is true for unary clauses in the original clause set. It is also true for learned unary clauses if non-standard-backjumping is applied.

If a watching set is maintained for each clause, the clause set cannot contain an unsatisfied clause under the current partial assignment. The challenge is to maintain these watching sets as the algorithm moves forward assigning values to unvalued variables, and as the algorithm backtracks.

Proposition 2.3.3. *Let S be a watching set for a clause c under partial assignment P. There exists a set S' that is a watching set for clause c under partial assignment $\{P, l\}$ if any one of the following conditions are true.*

1. *S does not contain l or $\bar{l}$*
2. *$l \in S$*
3. *$\bar{l} \in S$ and S contains a satisfied literal under P*
4. *$\bar{l} \in S$ and there exists an x such that $x \in c$, $x \notin S$, $\bar{x} \notin P$*

Proof. If condition (1) is true, then S remains a watching set under $\{P, l\}$. Under conditions (2) or (3), the set S will contain a satisfied literal under $\{P, l\}$ and will therefore be watching set for c under $\{P, l\}$. If condition (4) is true, then we construct a new watching set $S' = S \cup \{x\} - \{\bar{l}\}$ for c under $\{P, l\}$. □

Under the conditions stated in Proposition 2.3.3, it is easy to maintain watching sets when progressing forward assigning values to variables. As each new assignment is made, the watching sets are incrementally maintained. Any newly unsatisfied member of a watching set S for a clause c is replaced with a different satisfied or unvalued literal from the clause, generating a new watching set for c. The one case not covered by Proposition 2.3.3 is the case where both elements of S are unvalued under P, $\bar{l} \in S$, and the remaining literals in c are unsatisfied under P. In this case there is no literal x to replace $\bar{l}$.

Definition 2.3.4. *A partial assignment P for a set of C clauses will be called* C-closed *if no clause $c \in C$ is unit under P. A* C-closure *of P is any minimal extension of P that is C-closed. A C-closure of P will be denoted* $\texttt{closure}(P)_C$ *or simply* $\texttt{closure}(P)$ *if C is clear from context.*

A partial assignment P becomes closed when all unit clauses have been identified and all unit literals have been added to P. Any partial assignment returned by UNIT-PROPAGATE that does not contain a contradiction is a closed partial assignment. This is the termination condition for procedure UNIT-PROPAGATE.

Proposition 2.3.5. *Let* $\texttt{closure}(P) = \{P, l_1, l_2, \ldots, l_k\}$ *be a closure of partial assignment* P *for a set of clauses* C*. If* S *is a watching set for a clause* $c \in C$ *under partial assignment* P *such that both elements of* S *are unvalued under* P*,* $\bar{l}_1 \in S$*, and the remaining literals in* c *are unsatisfied under* P*, then* S *is a watching set for* c *under* $\texttt{closure}(P)$*.*

Proof. The clause c is unit under $\{P, l_1\}$ and the remaining unvalued literal in S, call it y, must be valued to $\texttt{true}$. Because $\texttt{closure}(P)$ is closed, $y \in \texttt{closure}(P)$. S contains a satisfied literal under $\texttt{closure}(P)$ and remains a watching set under $\texttt{closure}(P)$. ☐

Proposition 2.3.3 and Proposition 2.3.5 together ensure that watching sets can be maintained as the algorithm proceeds forward.

An advantage of the watching set approach is that watching sets remain valid during a backtrack without any incremental maintenance. If an element of a watching set S for a clause c is already unvalued, then unvaluing more assignments will not change this. If an element of S is satisfied, then backtracking could cause this literal to become unvalued. This is only a problem if the other watched literal in S is unsatisfied. Consider the following example.

Example 2.3.6. *Let* $P = \{(l_1, c_1), \ldots, (a, a \vee b)\}$ *be a partial assignment, and* $S = \{a, b\}$ *be a watching set for the clause* $a \vee b$*. The literal* a *is satisfied under* P*, so* S *is a valid watching set for* $a \vee b$*. However, the literal* b *is unsatisfied under* P *assuming that* P *is a sound partial assignment.* P *must have the form*

$$P = \{(l_1, c_1), \ldots, (\bar{b}, c_k), \ldots, (a, a \vee b)\}$$

Now imagine that there is a branch decision somewhere between the assignment of $\bar{b}$ *and* a*.*

$$P = \{(l_1, c_1), \ldots, (\bar{b}, c_k), \ldots, (l_i, \texttt{true}), \ldots, (a, a \vee b)\}$$

If we backtrack and unvalue a*, but do not back up far enough to unvalue* $\bar{b}$*, the set* S *won't be a valid watching set for the clause* $a \vee b$ *under the new partial assignment.*

Definition 2.3.7. *Given a partial assignment $P = \{(l_1, c_1), \ldots, (l_m, c_m)\}$ and assignment $(l_i, c_i) \in P$ such that $c_i \neq \texttt{true}$, let k be the minimum index in P such that l_i follows from $\{(l_1, c_1), \ldots, (l_k, c_k)\}$ by unit propagation. Assignment P will be called* unit complete *if for every non-branch assignment $(l_i, c_i) \in P$, l_i and l_k are valued at the same decision level.*

If the partial assignment P in the previous example were unit complete, then the assignments $\bar{b}$ and a would occur at the same decision level. The backtrack procedure must unvalue the entire decision level, leaving both watching set literals $\bar{b}$ and a unvalued under the new partial assignment. The watching set $\{\bar{b}, a\}$ remains a watching set under the retracted partial assignment.

Proposition 2.3.8. *If S is a watching set for a clause c under a unit complete partial assignment $P = \{l_1, \ldots, l_i\}$, then S is a watching set for c under any unit complete partial assignment $P' = \{l_1, \ldots, l_j\}$ such that $j < i$.*

Proof. If an element of S is unvalued under P, then it is also unvalued under P'. If an element of $l \in S$ is satisfied under P, then it may become unvalued under P'. This is only a problem if the other literal $l' \in S$ is unsatisfied under P, and has a lower index in P than l. Unvaluing l without unvaluing l' would leave S containing one unvalued literal and one unsatisfied literal. This situation cannot occur since l must follow from l' by unit propagation. Otherwise l' would have been removed from S and replaced by some other element of c that was unvalued or satisfied under P. Furthermore, because P' is unit complete, l and l' are valued at the same decision level. P' will either leave both l and l' valued, or unvalue them both. In either case S continues to be a watching set for c. □

Most DPLL style methods maintain a unit complete partial assignment; however, this has never been discussed in the literature. Because the unit propagation procedure terminates only when all unit clauses have been identified, DPLL-style algorithms maintain a unit complete partial assignments when proceeding forward. Maintaining a unit complete partial assignment during backtracking is trivial for a simple DPLL

algorithm. In the presence of learning, maintaining a unit complete partial assignment requires non-standard-backjumping. Standard-backjumping backs up only to the point where a learned clause c is satisfied. This can potentially split the point where c is unit and the point where c is satisfied between decision levels. Non-standard-backjumping backs up to the decision level in the partial assignment where c is unit, keeping the point where a clause becomes unit and the point where it is satisfied within the same decision level.

The watched literal method reduces the number of clauses that must be inspected for unit propagations when a variable is labeled. In addition, variables that are repeatedly assigned and reassigned tend to be watched in only a small number of clauses. This results in even shorter lists of clauses to inspect for unit propagations. When a variable is labeled, watchers are shifted away from that literal to other unvalued and satisfied literals. Watched literal data structures outperform count-based methods in almost all problem domains [51].

2.3.3 Notes on the Interplay of Propagation and Learning

An aspect of the relationship between propagation and learning was made clear with the introduction of the zCHAFF solver. The zCHAFF algorithm made a number of important contributions, but the most notable was the *watched literals* technique that greatly improved the speed of unit propagation. While the most significant contribution of zCHAFF was faster propagation, the indirect effect was to make zCHAFF a more effective learner.

The improved speed of propagation allows zCHAFF to manage a larger set of learned clauses efficiently. Before zCHAFF, an early rule of thumb put optimal relevance bounds around 3 to 4 for most problems. The default settings for zCHAFF are a relevance bound of 20 and a length bound of 100. ZCHAFF outperforms RELSAT not just because it is faster per node, but also because it learns more and builds smaller search trees.

All the methods described in this chapter emphasize keeping the cost of expanding a node low, favoring the fast traversal of a large search tree. This is in contrast to

the operations research techniques presented in Section 6.1 that use a more expensive computation at each node, but tend to generate small trees. While there is no clear argument in favor of either approach, it is clear that the choices made about speed of propagation will have profound effects on the type of learning that can be implemented.

2.4 Summary

In this chapter we have presented the family of systematic satisfiability methods that are based on the DPLL procedure. We have described the DPLL procedure and detailed the most significant improvements to this method, covering branching heuristics, bounded learning, and improved data structures for unit propagation.

Bounded learning has played an important role in the development of current methods because it has enabled solvers to begin solving structured problems. We will later show that this is only the first step in this direction; the power of learning is greatly increased when stronger representations are used. We saw that learning is a form of inference, and for DPLL-style trees, learning corresponds to resolution.

Fast propagation continues to be a requirement of DPLL-style methods and a focus of research. Unit propagation is the major computational task of DPLL-style solvers taking over eighty percent of execution time. The recent introduction of watched literals data structures has significantly reduced the cost of expanding a search node by increasing the speed of unit propagation. The result is solvers that can manage larger sets of clauses efficiently. These solvers can afford to increase their learning bounds and retain much larger sets of learned clauses and this in turn leads to smaller search trees and improved performance.

CHAPTER 3

Reconsidering Representation

In Chapter 2 we reviewed the history and development of modern systematic satisfiability engines. We saw that learning is an inference step, and that in the case of traditional satisfiability methods, learning is a resolution step. In this chapter, we develop this perspective further and show that the solvers discussed in Chapter 2 are automated proof systems.

When we view systematic satisfiability solvers as proof constructors, it makes sense to draw on results from the field of proof complexity. The major task of the field of proof complexity is to investigate the ability of propositional proof systems to generate polynomial-sized proofs. Much of this work takes the form of lower bounds on proof length for specific propositional proof systems. For example, there are known families of problems for which all resolution proofs are exponential in length [38, 69]. If we are in the business of constructing resolution proofs, then such a result is quite troubling. The underlying proof system a solver uses to construct proofs can place some very serious limitations on its performance. We will introduce the pigeonhole problem as a problem class that requires exponential length resolution proofs and show experimental results for many well known satisfiability solvers on pigeonhole problems. It is no surprise to see exponential runtime scaling on these instances.

It is important to note that the pigeonhole problem is not a rare or unrealistic problem instance. It is instead a pervasive subproblem in real-world problems including many of the primary problem domains of AI such as planning and scheduling. We

postpone a full discussion of the prevalence of pigeonhole problems until Chapter 5 when we present some experimental results showing the extent to which embedded pigeonhole problems occur within a simple logistics domain. What we hope to make clear in this chapter is that the limitations of the resolution proof system amount to a major roadblock for satisfiability research. Systematic methods that continue to follow the path of CNF and resolution will never provide efficient solutions in any domain that requires counting.

3.1 Systematic Solvers: A Special Kind of Proof System

Systematic methods are guaranteed to determine whether a problem instance is satisfiable or unsatisfiable. If an instance is satisfiable, the solver produces a solution as a witness. If the instance is unsatisfiable, the solver must construct some form of a proof that no solution exists. The depth-first backtracking methods we discussed in Chapter 2 spend the bulk of execution time constructing proofs of unsatisfiability, even on satisfiable instances. Consider that up until the point a solution is found, all time is spent proving that earlier encountered subproblems were unsatisfiable. We can conclude that the primary goal of the systematic methods we have discussed is to produce proofs of unsatisfiability.

Systematic satisfiability methods are really proof systems, and it is important to think of them in this way. If we think of a proof system as a set of syntactic inference rules for deriving new statements, then a systematic method is somewhat more than a proof system, since it also provides rules for how to combine inference steps to construct a proof given a problem instance. While systematic methods construct proofs of unsatisfiability, they typically aren't required to return those proofs as an answer. Usually a "no" answer affirming that no solution exists is sufficient. This is important because as we will see, many proofs are necessarily long, potentially having length exponential in the size of the original problem. In such cases, the runtime of solvers will be proportional to the time required to construct such a long proof, but

memory resources need not similarly suffer, since large parts of the proof can be discarded once they are no longer needed.

For any given solver, it would be useful to know what the underlying proof system is and what its properties are. For some systematic methods this is well known while for others very little is known. DPLL methods have the benefit of being well analyzed in this respect.

3.1.1 DPLL Descendants: Resolution-Based Methods

We now look specifically at the types of proofs constructed by the algorithms discussed in Chapter 2. All of these algorithms construct some form of resolution proof to determine unsatisfiability. Such methods are often referred to as *resolution-based* methods.

We look first at proofs constructed by the simple DPLL algorithm. The DPLL procedure constructs tree-style resolution proofs [7]. Every DPLL search tree can be converted to a resolution proof tree. Figure 3.1 shows a DPLL search tree and its corresponding tree-style resolution proof. In the search tree, pruned subtrees are represented by leaf nodes that are labeled with the clause that caused the subtree to be pruned. In the corresponding resolution proof, each interior node is labeled with the resolvent of its two child nodes. The empty clause Λ is derived at the root node. Similarly, resolution proof trees can be converted to DPLL search trees of equal or smaller size [7]. In a tree-style proof, each resolvent can be used only once. If it is needed again, it must be re-derived from the original clauses. This is exactly the thrashing phenomenon discussed in Section 2.2, from the perspective of proof construction.

This is the first indication of the role learning plays in determining the strength of the underlying proof system. Branching heuristics help DPLL methods find shorter proofs, but without learning, those proofs are still tree-style resolution proofs. The data-structures work presented in Section 2.3 has no direct effect on the underlying proof system. The addition of learning allows solutions from one part of the search space to be applied throughout the search. In terms of proof structure, this means

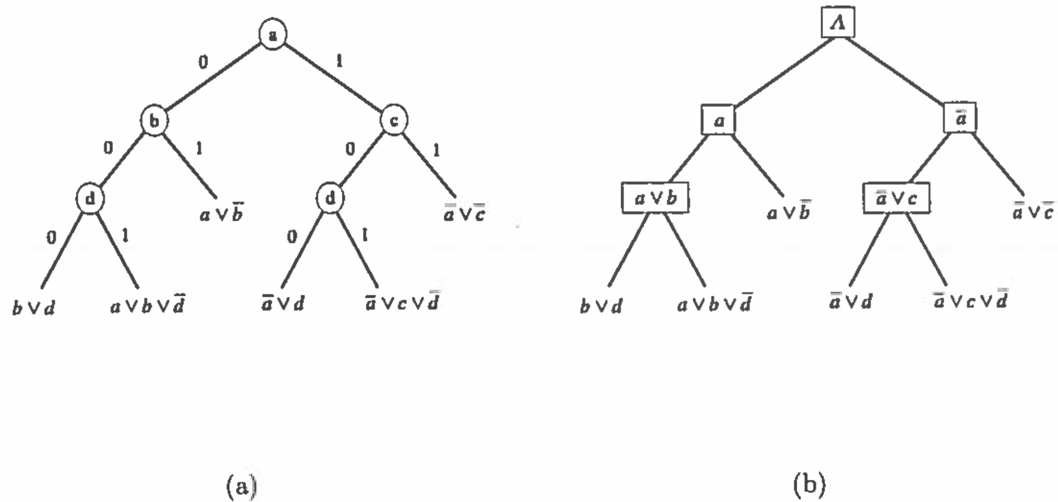


FIGURE 3.1: A DPLL search tree (a) and corresponding resolution proof of the empty clause Λ (b)

resolvents can be applied again without repeating their derivation. This leads to resolution proofs with the structure of a directed acyclic graph (DAG). DAG style proofs are significantly more efficient than tree-style proofs. For a thorough discussion of complexity results for conventional DPLL-style solvers, see the work of Beame, Kautz, and Sabharwal [8].

Observation 3.1.1. *Of all the improvements to the* DPLL *method discussed in Chapter 2, only learning improves the strength of the underlying proof system.*

3.2 Proof Complexity and Systematic Solvers

In this section we introduce the field of proof complexity and show how different propositional proof systems can be compared in terms of their strength. We then begin applying results from the field of proof complexity to the systematic satisfiability methods discussed in Chapter 2.

3.2.1 Proof Complexity

The field of proof complexity is concerned with determining the ability of specific proof systems to produce short proofs. Understanding the proof complexity of individual proof systems will ideally lead to deeper insights into the questions of theoretical computer science. A major question in the field of proof complexity concerns the existence of a polynomial-bounded propositional proof system. The non-existence of any proof system that has short proofs (polynomial bounded in the size of the problem) for all problem instances would imply that NP $\neq$ co-NP and by consequence P $\neq$ NP [18].

A proof system is formally defined as an algorithm.

Definition 3.2.1. [7] *A* proof system *for a language L is a polynomial time algorithm V such that for all inputs strings x, $x \in L$ if and only if there exists a string P such that V accepts input (x, P).*

Algorithm V determines whether proof P is a valid proof of x. If the language we are concerned with is the set of unsatisfiable propositional formulas then we can define a propositional proof system as follows.

Definition 3.2.2. [7] *A* propositional proof system *is a proof system for the set UNSAT of unsatisfiable propositional logic formulas, i.e. a polynomial time algorithm V such that for all formulas F, F is unsatisfiable if and only if there exists a string P such that V accepts input (F, P).*

Definition 3.2.3. [7] *The* complexity *of a proof system V is the function $f : N \rightarrow N$ defined by*

$$f(n) = \max_{x \in L, |x|=n} \left[\min_{P:V \text{ accepts } (x,P)} |P| \right]$$

For a given input length n, there are multiple inputs x. For each input x there may be multiple proofs P of input x, and the lengths of these proofs may vary. The function f returns the worst case proof length $|P|$ over all inputs x, assuming that P is the shortest possible proof of an input x.

A proof system V is called *polynomial bounded* if and only if $f(n)$ is bounded from above by a polynomial function of n.

Theorem 3.2.4. [18] *There is a polynomial bounded propositional proof system if and only if* NP = co-NP.

We can think of the strength of a propositional proof system as its ability to produce short proofs of unsatisfiability. It is useful to have a way of determining the relative strengths of different proofs systems.

Definition 3.2.5. *Given proof systems U and V that prove the same language L, the proof system U* polynomially simulates *(or p-simulates) a proof system V if and only if proofs in V can be efficiently converted into proofs in U, i.e. there is a polynomial-time computable function f such that*

$$V \text{ accepts } (x, P) \Longleftrightarrow U \text{ accepts } (x, f(P))$$

Definition 3.2.6. *Given proof systems U and V that prove the same language L, U is* exponentially separated *from V if there is an infinite sequence of inputs $x_1, x_2, \dots \in L$ such that for complexity functions f_U and f_V, $f_U(x_i)$ is polynomial in $|x_i|$ for all i and $f_V(x_i)$ is exponential in $|x_i|$.*

Two proof systems are said to be *polynomially equivalent* if each can polynomially simulate the other. A proof system U is *exponentially stronger* than proof system V, if U polynomially simulates V and is exponentially separated from it. Using these definitions we can define a hierarchy of proof systems. Notice that this definition of exponential separation requires finding a class of problems that are "hard" for a particular proof system. An example would be the pigeonhole problem mentioned earlier, which is hard for the resolution proof system. The proof complexity community has generated a suite of test problems used to separate out the different proof systems. These test problems can be used to test the strength of satisfiability solvers as well.

The framework used to study proof complexity has a lot to offer those interested in building satisfiability solvers. If we are in the business of constructing proofs, then we

care very much about proof length. If there are examples of problems that are known to be hard for a solver's underlying proof system, then this places a lower bound on solver performance. It is useful to know, for a given solver, what the underlying proof system it uses to construct proofs of unsatisfiability is, where this system falls in the hierarchy of proofs systems, and what classes of problems are known to require exponential length proofs in the proof system.

3.2.2 Exponential Lower Bounds for Resolution

The time it takes to construct a proof is obviously bounded from below by the length of the proof, so efficiently constructed proofs are necessarily short. We have established that the satisfiability solvers discussed in Chapter 2 construct some form of resolution proof as a witness to unsatisfiability. Unfortunately, it has been shown that many families of unsatisfiable problems do not have short resolution proofs of unsatisfiability. Consider for example the pigeonhole problem, that states that $n+1$ pigeons cannot be placed into n holes that each admit only one pigeon. The shortest resolution proof of unsatisfiability for the pigeonhole problem is exponential in the number of pigeons [38]. Any solver that constructs a resolution proof as a witness to unsatisfiability will necessarily have exponential scaling on pigeonhole problems.

We examine the performance of current resolution-based solvers on the pigeonhole problem in Figure 3.2. All problems were run on 1.5GHz Athlon processors. We report all times that did not exceed the time limit of 4000 seconds. As expected, for all solvers scaling is exponential, with problems of size $13-14$ pigeons becoming impractical. It is critical to understand that the poor performance seen here is a result of using resolution for the underlying proof system. If methods can only construct resolution proofs then they are guaranteed bad performance on these problem instances.

It is important to ask whether pigeonhole problems occur naturally as subproblems in domains of interest. If they do not occur in real world problems, then perhaps we need not be concerned. It has long been believed that pigeonhole problems do occur in many problem domains wherever counting or mapping is involved. Some examples of constraints that may induce pigeonhole problems are resource or capacity constraints

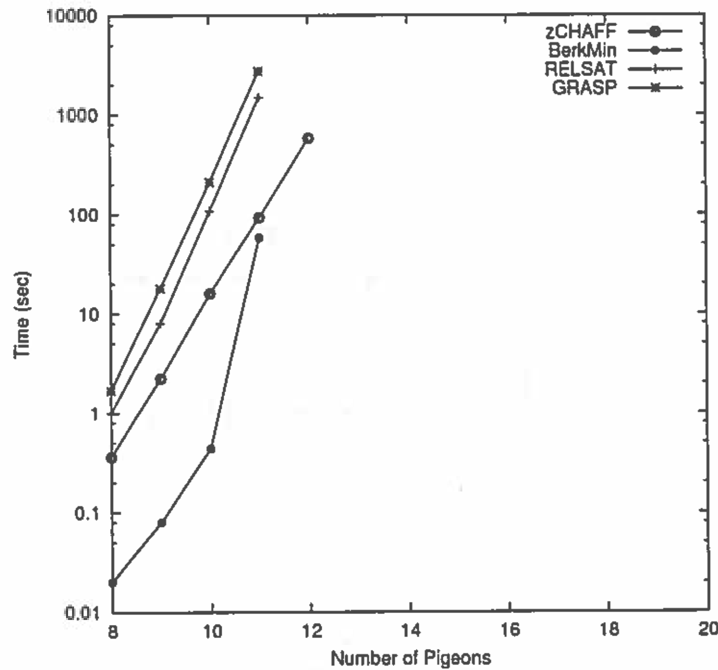


FIGURE 3.2: Performance on PHP_n^{n+1} for some well known solvers

that are integral in nature. There are also other problems that are hard for resolution. These include, but are not limited to, clique-coloring problems and certain parity problems. This is because resolution is a fairly weak proof system located near the bottom of the proof system hierarchy.

Improvements to current methods that do not move the underlying proof system beyond resolution can only lead to modest gains in performance. This sentiment was nicely summed up by David Mitchell [54], who said in reference to further resolution-based research:

> These are valuable methods, but it may turn out that we are arguing over the best way to build ladders when we need rocket-ships.

CHAPTER 4

A Pseudo-Boolean SAT Solver

If we want to solve problems like the pigeonhole problem efficiently, then we will need to work with a stronger proof system. Moving to a stronger proof system is clearly a necessary step toward improving performance; however, it does not guarantee that efficient solutions will be found. When automating strong proof systems a new set of problems can arise. Short proofs may exist but be difficult to find and ultimately inefficient to generate. We'd like to retain the substantial progress made within the CNF and resolution setting.

Pseudo-Boolean representation is a good first choice for this type of exploration. The proof system commonly associated with pseudo-Boolean representation is the cutting-plane proof system (CP). It is a relatively weak proof system within the proof system hierarchy, but it is stronger than the resolution inference system. Two important characteristics of strong representations and inference systems can be seen in pseudo-Boolean representation. First, a pseudo-Boolean constraint can be exponentially more concise than the logically equivalent CNF representation. Second, pseudo-Boolean inference can generate new, concise constraints from existing ones. As a result, a single pseudo-Boolean inference step may require an exponential number of resolution steps to derive the same information. Pseudo-Boolean representation together with cutting-plane inference allow polynomial-length proofs of the pigeonhole principle [19]. These aspects of pseudo-Boolean representation are explored in Section 4.1.

The remainder of this chapter describes PBCHAFF, our implementation of a DPLL-style algorithm adapted to use pseudo-Boolean representation. The goal of this implementation is to improve the strength of the underlying proof system without discarding the successful elements of traditional resolution-based methods.

4.1 Pseudo-Boolean Representation

Definition 4.1.1. *A* pseudo-Boolean constraint *is a linear inequality over Boolean variables x_i of the form*

$$\sum a_i x_i \geq k$$

with $x_i \in \{0,1\}$ and fixed integers a_i and k.

This representation comes from the operations research community and is a subset of the more general integer programming representation in which the x_i are allowed to take non-negative integer variables. Linear pseudo-Boolean inequalities typically have the more general form

$$a_0 x_0 + b_0 \overline{x}_0 + a_1 x_1 + b_1 \overline{x}_1 \cdots + a_n x_n + b_n \overline{x}_n \geq r$$

and allow for real coefficients a_i, b_i, and r [39]. For 0-1 problems integer coefficients are sufficient and we will assume that coefficients are integral from here on.

Definition 4.1.2. *A pseudo-Boolean constraint of the form*

$$\sum_i x_i \geq k$$

in which all coefficients are equal to 1 *is called a* cardinality *constraint.*

This constraint requires that at least k of the x_i are valued 1.

The cutting-plane proof system originated from an algorithm for general integer programming created by Gomory [36]. The algorithm was rarely used in practice because it converged slowly, but Chvátal recognized [15] that the method could function

as a proof system. The system has two rules of inference. First, we can derive a new inequality by taking a linear combination of a set of inequalities

$$\frac{\sum_i a_i x_i \geq k \qquad \sum_i b_i x_i \geq l}{\sum_i p a_i x_i + \sum_i q b_i x_i \geq pk + ql}$$

provided that multipliers p and q are non-negative integers. The second rule allows fractional rounding.

$$\frac{\sum_i a_i x_i \geq k}{\sum_i \lceil \frac{a_i}{d} \rceil x_i \geq \lceil \frac{k}{d} \rceil}$$

Here d is a positive integer, and the notation $\lceil q \rceil$ denotes the least integer greater than or equal to q. A derived inequality of this type is called a *cut*. If the inequality $0 \geq 1$ can be derived in this fashion, the original set of inequalities is inconsistent.

4.1.1 *P*-simulating Resolution

We can p-simulate the resolution proof system using the CP system [19]. Recall that a proof system U is said to p-simulate a proof system V if and only if proofs in V can be efficiently converted into proofs in U. Disjunctive clauses can be written as linear inequalities with propositional variables restricted to values of $0 = \texttt{false}$, and $1 = \texttt{true}$. A disjunction of literals:

$$x_0 \vee x_1 \vee \cdots \vee x_n$$

can be equivalently written as the linear pseudo-Boolean inequality:

$$x_0 + x_1 + \cdots + x_n \geq 1$$

The variable $\overline{x}$ refers to the negation of the variable x, so that for all literals x, $\overline{x} = 1 - x$.

The resolution rule will now have the following form: given two clauses $C_1 = p \vee \bigvee_i x_i$ and $C_2 = \overline{p} \vee \bigvee_i y_i$

$$\frac{\begin{array}{c} p \vee \bigvee_i x_i \\ \overline{p} \vee \bigvee_i y_i \end{array}}{\bigvee_i x_i \vee \bigvee_i y_i}$$

can be written as

$$\frac{\begin{array}{c} p + \sum_i x_i \geq 1 \\ \overline{p} + \sum_i y_i \geq 1 \end{array}}{\sum_i x_i + \sum_i y_i \geq 1}$$

The derived inequality follows because $p + \overline{p} = 1$. The factoring rule takes the form:

$$\begin{array}{rl} p + p + \sum_i x_i & \geq 1 \\ \sum_i x_i & \geq 0 \\ \hline 2p + 2\sum_i x_i & \geq 1 \\ p + \sum_i x_i & \geq \lceil \frac{1}{2} \rceil \end{array}$$

Rounding the fraction $\frac{1}{2}$ to 1 by virtue of the CP rule of integer rounding now gives the inequality $p + \sum_i x_i \geq 1$.

From a proof complexity point of view, the CP proof system is at least as strong as resolution. It can be seen by inspection that the CP proof system is a proper generalization of the resolution system.

4.1.2 Translating Pseudo-Boolean Constraints into CNF

We have seen how to translate CNF clauses into pseudo-Boolean. We can also translate a pseudo-Boolean constraint back into clause form, although multiple clauses may be required to capture the logically equivalent statement.

Proposition 4.1.3. [9] *Given a pseudo-Boolean constraint*

$$\sum_{i=1}^{n} a_i x_i \geq k \tag{4.1}$$

The constraint (4.1) is logically equivalent to the set of disjunctions

$$\bigvee_{x_i \in S} x_i \tag{4.2}$$

such that both $S \subseteq \{x_1, x_2, \ldots, x_n\}$ *and*

$$\sum_{i | x_i \notin S} a_i < k \tag{4.3}$$

Proof. Suppose that the full assignment ρ satisfies (4.1), and $R \subseteq \{x_1, x_2, \ldots, x_n\}$ is the subset of literals that are true under ρ so that

$$\sum_{i | x_i \in R} a_i x_i \geq k$$

If d is any disjunction in (4.2), then (4.3) says that the sum over the literals not in S is not enough to satisfy (4.1). It follows that $S \cap R \neq \emptyset$ and ρ must also satisfy d.

For the converse, suppose that assignment ρ satisfies all (4.2) but not (4.1). If $R \subseteq \{x_1, x_2, \ldots, x_n\}$ is the subset of literals that are true under ρ we have

$$\sum_{i | x_i \in R} a_i x_i < k$$

But this implies that the disjunction

$$\bigvee_{x_i \notin R} x_i$$

is an instance of (4.2) and is unsatisfied under ρ, contradicting our original assumptions. The assignment ρ must therefore satisfy (4.1). ☐

Proposition 4.1.4. *[27] There is no CNF encoding of a cardinality constraint*

$$x_1 + \cdots + x_m \geq k \tag{4.4}$$

that is more efficient than the encoding defined in Proposition 4.1.3.

Proof. By Proposition 4.1.3, (4.4) is equivalent to the set of $\binom{m}{m-k+1}$ disjunctions

$$\bigvee_{x_i \in S} x_i \tag{4.5}$$

where $S \subseteq \{x_1, \ldots, x_m\}$ and $|S| = m - k + 1$.

To show that this encoding is minimal we first note that the literals in any clause of our encoding will be a subset of $\{x_1, x_2, \ldots, x_m\}$. Otherwise, if an implied clause had the form

$$\bigvee_{x_i \in S} x_i \vee \bigvee_{x_i \in T} x_i \tag{4.6}$$

where $S \subseteq \{x_1, \ldots, x_m\}$ and $T \cap \{x_1, \ldots, x_m\} = \emptyset$ we could write it more concisely as

$$\bigvee_{x_i \in S} x_i \tag{4.7}$$

Any assignment satisfying (4.4) will also satisfy (4.6); however, it must also satisfy (4.7). Given any assignment ρ that satisfies (4.4) and (4.6) but not (4.7), we can construct another assignment ρ' by flipping all the literals in T that are satisfied. Assignment ρ' still satisfies (4.4) but not (4.6). This falsifies our assumption that (4.6) is implied by (4.4). We can conclude that (4.7) must also be implied by (4.4).

Next, note that any axiom c with length less than $m-k+1$ cannot be a consequence of (4.4). Let $S \subseteq \{x_1, \ldots, x_m\}$ be the set of literals in c. We can construct an assignment ρ that makes all of the literals in S false and all the literals in $\{x_1, \ldots, x_m\} - S$ true. Assignment ρ satisfies (4.4) but does not satisfy c.

Consider the set of clauses formed if we leave a single instance

$$\bigvee_{x_i \in S'} x_i$$

of (4.5) out of our encoding and include the rest of the instances of (4.5) together with every clause with length greater than $m-k+1$. We can construct an assignment ρ that satisfies all of our clauses but does not satisfy (4.4) by letting ρ make all the literals in S' false and the remaining literals true. We can conclude that (4.5) is the most efficient encoding of (4.4). □

Observation 4.1.5. *A pseudo-Boolean constraint can be exponentially more concise than the equivalent set of Boolean clauses.*

This potentially exponential blowup cannot be avoided unless we are willing to introduce new variables [72]. The ability to represent problems efficiently is an important characteristic of strong proof systems.

Translating pseudo-Boolean constraints to CNF is useful, but a question we are more likely to encounter in practice is how to rewrite a set of Boolean clauses as a logically equivalent yet smaller set of pseudo-Boolean constraints. Proposition 4.1.3 gives us some insight into when a set (or subset) of Boolean clauses can be rewritten as a single pseudo-Boolean constraint. Consider the set of clauses

$$\begin{gathered} a \vee b \\ a \vee c \\ b \vee c \end{gathered} \tag{4.8}$$

that are equivalent to the cardinality constraint

$$a + b + c \geq 2 \tag{4.9}$$

Here we simply note that the compression of (4.8) into the cardinality constraint (4.9) is possible because (4.8) contains a pattern. Informally we might say that cardinality constraints capture a particular type of problem structure. The constraint (4.9) can be generated from (4.8) through inference. We will discuss this type of inference further in the next section and give a method for automating it in Section 4.6.

4.1.3 Short Proofs of the Pigeonhole Principle

Unlike resolution, the CP proof system allows polynomial-length proofs of the pigeonhole problem [19]. This result, combined with the earlier p-simulation result, asserts that the CP proof system is exponentially stronger than resolution. A closer look at the pseudo-Boolean proof of the pigeonhole problem will help generate intuition about why the cutting-plane proof system is stronger than resolution, and

more generally what makes a proof system efficient. Recall from the previous section that a pseudo-Boolean constraint can concisely express a large number of Boolean clauses. We will call a pseudo-Boolean constraint whose equivalent CNF encoding requires more than one clause a *compound* constraint. If a pseudo-Boolean constraint is equivalent to a simple Boolean disjunction we will call it a *clausal* constraint. The power of pseudo-Boolean inference over resolution comes from two types of inference.

Parallel inference

Parallel inference is the ability to infer a new compound constraint from a set of compound constraints. A resolution step only allows the generation of a single Boolean clause. Consider what happens when a new compound constraint is generated in a single inference step. The new constraint will represent a numerous (possibly exponential) set of Boolean clauses. A single pseudo-Boolean inference step may require an exponential number of resolution steps to derive the same information. The pseudo-Boolean inference step performs multiple resolutions in parallel. Chatalic and Simon also discuss the idea of parallel resolution within the context of ZBDDs [13]. A discussion of their work and an introduction to ZBDDs is given in Section 6.2.3.

Building inference

The power of parallel inference is only possible if concise compound constraints exist in our problem description. We also need the ability to infer a new compound constraint from a set of clausal constraints or weaker compound constraints. The new compound constraint subsumes a subset of its parent constraints, providing a stronger, more concise representation of the set of parent constraints.

These are not intended to be formal definitions. An individual pseudo-Boolean inference step may contain both of these elements. What is important to note is that if either of these elements is removed from the proof system, then the strength of the system is reduced. Concise representations are of little value if there is no way

of exploiting that conciseness through inference. Similarly, a system that exploits concise constraints is of limited value if concise encodings are not known and there is no way of generating them.

A pseudo-Boolean encoding of the pigeonhole problem can be constructed by taking the CNF encoding and translating each disjunction to a linear inequality.

$$\sum_{k=1}^{n} p_{ik} \geq 1 \quad i = 1, \ldots, n+1 \tag{4.10}$$

$$\overline{p}_{ik} + \overline{p}_{jk} \geq 1 \quad i \neq j, k = 1, \ldots, n \tag{4.11}$$

The literal p_{ik} represents pigeon i being in hole k. The set of inequalities (4.10) asserts that every pigeon is in some hole. Inequalities (4.11) say that two different pigeons cannot occupy the same hole.

While this encoding is logically correct, pseudo-Boolean representation allows a more concise encoding. The set of inequalities (4.11) can be more concisely expressed by the set of cardinality constraints

$$\sum_{i=1}^{n+1} \overline{p}_{ik} \geq n \quad k = 1, \ldots, n \tag{4.12}$$

To better understand what is happening here, consider an instance of this set for $n = 3$.

$$\overline{p}_{11} + \overline{p}_{21} + \overline{p}_{31} + \overline{p}_{41} \geq 3 \tag{4.13}$$

which translates to the following set of clauses:

$$\begin{gathered} \overline{p}_{11} \vee \overline{p}_{21} \\ \overline{p}_{11} \vee \overline{p}_{31} \\ \overline{p}_{11} \vee \overline{p}_{41} \\ \overline{p}_{21} \vee \overline{p}_{31} \\ \overline{p}_{21} \vee \overline{p}_{41} \\ \overline{p}_{31} \vee \overline{p}_{41} \end{gathered} \tag{4.14}$$

If we instead started with the set of CNF clauses (4.14), the constraint (4.13) might be seen as capturing a pattern or structure in the CNF clause set. This structure

is present in the set of CNF clauses, but the resolution system has no way to keep track of this structure and make use of it. If we combine (4.10) with (4.12) we have a second more concise encoding of the pigeonhole problem.

The benefit of this second encoding is that the inequality $0 \geq 1$ can be derived simply by summing (4.10) and (4.12) over i and k respectively and adding the results giving a proof with length $O(n)$. The final inference step is equivalent to an exponential number of resolution steps. This is an example of parallel inference. Here is another.

Example 4.1.6. *Consider a single inference step from a pigeonhole problem of size 4. We resolve constraint*

$$\overline{p}_{11} + \overline{p}_{21} + \overline{p}_{31} + \overline{p}_{41} \geq 3$$

that says hole 1 can only have one pigeon, with the inequality

$$p_{11} + p_{12} + p_{13} \geq 1$$

that requires pigeon 1 *to be in a hole. The resulting constraint*

$$p_{12} + p_{13} + \overline{p}_{21} + \overline{p}_{31} + \overline{p}_{41} \geq 3$$

implies the new clauses

$$p_{12} \vee p_{13} \vee \overline{p}_{21}$$
$$p_{12} \vee p_{13} \vee \overline{p}_{31}$$
$$p_{12} \vee p_{13} \vee \overline{p}_{41}$$

Deriving these clauses with clausal resolution would require three separate resolution steps.

The pseudo-Boolean inference rule takes advantage of the stronger encoding in (4.13) and performs multiple clausal resolutions in parallel.

The proof presented above is not an acceptable proof from a proof complexity perspective. To truly provide a polynomial length proof of the pigeonhole principle

we must start from the first CNF encoding. Fortunately, the stronger encoding can be derived from the first in n^2 steps. We will derive the set of inequalities (4.12) by induction, generating all inequalities of the form

$$\sum_{i=1}^{j+1} \overline{p}_{ik} \geq j \tag{4.15}$$

for $j < n + 1$. The base case $j = 1$ is contained in the initial inequalities. For the inductive step, assume (4.15) is true for a given j.

$$\begin{array}{lllllll}
j(& \overline{p}_{1k}+ & \overline{p}_{2k}+ & \cdots & +\overline{p}_{j+1,k}) & & \geq j^2 \\
 & \overline{p}_{1k}+ & & & & \overline{p}_{j+2,k} & \geq 1 \\
 & & \overline{p}_{2k}+ & & & \overline{p}_{j+2,k} & \geq 1 \\
 & & & & \overline{p}_{j+1,k}+ & \overline{p}_{j+2,k} & \geq 1 \\
\hline
 & \overline{p}_{1k}+ & & \cdots & & +\overline{p}_{j+2,k} & \geq \frac{j^2+j+1}{j+1} = j + \frac{1}{j+1}
\end{array}$$

The right hand side of the final equation rounds up to $j + 1$. This is an example of building inference. Notice that this is not pseudo-Boolean resolution. Consider that no variables cancel out of derived constraints.

From a proof complexity perspective, building inference is an important component of the proof. From the perspective of a solver builder, this type of inference is important, but not as important as the ability to take advantage of strong constraints through parallel inference. Ideally one would like the ability to ignore any known structure, automatically discover it from scratch, then solve the problem, and do it all in polynomial time. While this may be a more elegant solution, solver builders often must take advantage of the structure that is available and use it the best they can. Currently, progress can be made with this approach since a large amount of known structure is typically thrown away or ignored when encodings in high-level representations are translated into CNF encodings. For this reason, our first priority is building solvers that take full advantage of known structure.

At some point we begin to care about identifying structure and building up structured constraints. It may be that a constraint set has structure that is not made explicit by a high-level description. One could also imagine that a problem contains an embedded structured problem that only becomes apparent after some inference

has taken place. Solvers that cannot recognize this structure will not be able to solve such problems efficiently. Fortunately, it is possible to automate cutting-plane versions of both types of inference.

4.2 Unit Propagation

In this section, we give a pseudo-Boolean implementation of unit propagation. We discuss how the data structures work of Section 2.3.2 can be implemented for pseudo-Boolean constraints. We present both count-based and watched literal indexing schemes.

4.2.1 Count-Based Methods

Recall from Chapter 2 the count-based method for identifying unit propagations. In addition to literal indexing, we maintain for each clause a count of the number of satisfied and the number of unvalued literals in that clause under the current partial assignment. These counts are incrementally maintained as assignments are made and revised. A clause is unit when the number of satisfied literals drops to 0 and the number of unvalued literals is one. A comparable count based implementation is possible for pseudo-Boolean constraints and was first implemented by Barth [3]. We define two new counts.

Definition 4.2.1. *Let c be a set of literals and P be a partial assignment. We define the sets*

$$S_P(c) = \{x_i | x_i \in c \text{ and } x_i \in P\}$$

$$V_P(c) = \{x_i | x_i \in c \text{ and } \bar{x}_i \notin P\}$$

where $S_P(c)$ denotes the literals in c that are satisfied under P and $V_P(c)$ denotes the literals in c that are either satisfied or unvalued under P.

Definition 4.2.2. *Let c be a pseudo-Boolean constraint of the form $\sum_i a_i x_i \geq k$, T be a subset of the literals in c, and P be a partial assignment of values to variables. The counts* `current` *and* `possible` *are defined as*

$$\texttt{current}(T, P) = \sum_{i | x_i \in S_P(T)} a_i - k$$

$$\texttt{possible}(T, P) = \sum_{i | x_i \in V_P(T)} a_i - k$$

The count `current` sums over the weights of satisfied literals in T and subtracts off k. If the value of $\texttt{current}(c, P)$ is greater than or equal to zero, then the constraint is satisfied. If the value is negative, the constraint is not yet satisfied. The count `possible` sums over the weights of both the satisfied and unvalued literals in T and subtracts off k. If the value of $\texttt{possible}(c, P)$ is greater than or equal to zero, then it is possible to satisfy the constraint. If the value becomes negative, it is no longer possible to satisfy the constraint c. The counts $\texttt{current}(c, P)$ and $\texttt{possible}(c, P)$ can be maintained for each constraint, and adjusted with each change in P.

Definition 4.2.3. *A pseudo-Boolean constraint c is a* unit constraint *under partial assignment P if there is a literal x_i in c such that x_i is unvalued and* $\texttt{possible}(c, P) < a_i$. *The literal x_i is called a* unit literal.

In this situation, the unit literal x_i must be valued to 1 if partial assignment P is to be extended to a full assignment. Any partial assignment P' that extends P and contains $\bar{x}_i$ is invalid because $\texttt{possible}(c, P') < 0$.

Example 4.2.4. *Consider the constraint*

$$2x_1 + x_2 + 2x_3 + x_4 \geq 4$$

under the partial assignment $P = \{x_1, \bar{x}_2\}$ and counts $\texttt{current}(c, P) = -2$ *and* $\texttt{possible}(c, P) = 1$. *We must assign a value of 1 to x_3 because the assignment $x_3 = 0$ will give* $\texttt{possible}(c, P) < 0$ *and the clause will be unsatisfiable.*

Given a set of pseudo-Boolean constraints and a partial assignment, the set of unit propagations generated by Definition 4.2.3 are exactly those generated by unit propagation performed on the set of equivalent CNF clauses.

Lemma 4.2.5. *Given pseudo-Boolean constraint c of the form $\sum_i a_i x_i \geq k$ and partial assignment P, a literal x_j in c is unit under P if and only if the set of CNF clauses equivalent to c as defined in Proposition 4.1.3 produces the unit propagation x_j under P.*

Proof. First we show that if c is unit under P and $x_j \in c$ is a unit literal, then there must be a clause v implied by c that has the unit literal x_j under P. By definition

$$\texttt{possible}(c, P) = \sum_{i|x_i \in V_P(c)} a_i - k < a_j. \tag{4.16}$$

We can construct v from the unsatisfied literals of c and literal x_j.

$$v = x_j \vee \bigvee_{x_i \notin V_P(c)} x_i$$

To show that clause v is implied by c, we manipulate (4.16) into $\sum_{i|x_i \in V_P(c)} a_i - a_j < k$ and apply Proposition 4.1.3.

To show the other direction, let

$$x_j \vee \bigvee_{x_i \in R} x_i \tag{4.17}$$

be a CNF clause implied by c with unit literal x_j under P. By Proposition 4.1.3 we have

$$\sum_{i|x_i \notin (R \cup \{x_j\})} a_i < k$$

Additionally,

$$\sum_{x_i \in R} a_i x_i = 0$$

since the literals in R contribute nothing toward satisfying c. Therefore the value of $\texttt{possible}(c, P)$ is at most $a_j - 1$, implying that x_j is unit in c under P. □

This shows the correctness of pseudo-Boolean unit propagation and that it is no more and no less powerful than its clausal counterpart.

Procedure 4.2.6 (PB-Unit-Propagate). *To compute* PB-UNIT-PROPAGATE(C, P)*:*

1 **while** no contradiction is found **and** there is a $c \in C$ that has an unvalued literal x_i with coefficient a_i such that `possible`$(c, P) < a_i$
2   **do**
3         $P \leftarrow \{P, x_i\}$
4 **return** P

Proposition 4.2.7. PB-UNIT-PROPAGATION *as computed by Procedure 4.2.6 is sound.*

Proof. Soundness follows from Lemma 4.2.5. □

We have shown a pseudo-Boolean version of unit propagation that is correct. We now need to show that it is also efficient to compute, in keeping with our second goal of maintaining the speed of unit propagation. We compare the cost of count-based pseudo-Boolean unit propagation with its clausal counterpart.

To begin, we examine the cost of maintaining the counts `current` and `possible`. Like the clausal case, these counts need to be incrementally maintained for each constraint. In the clausal case, counts are always incremented and decremented by 1. In the pseudo-Boolean case, the amount of the increment or decrement will depend on the size of the coefficient of the variable in the constraint being valued. For example, if we are extending the partial assignment with the literal $\overline{x}_1$, and we have a constraint c of the form

$$2x_1 + x_2 + x_3 + x_4 \geq 2$$

then we must decrement `possible`(c, P) by 2. Caching literal weights in a literal index allows easy access to the weight of the appropriate literal. If a constraint c contains the weighted literal $a_i x_i$, we add to the literal index for x_i a pair consisting of a pointer to c and the value a_i. Now if we value the variable associated with x_i, we can adjust counts `current` and `possible` without searching c to determine the weight associated with x_i. This allows counts to be maintained in time linear in the number of clauses, as in the clausal case.

Determining whether a clause is unit can be done in constant time in the clausal case by simply examining the values of the counts. A clause is unit exactly when the number of satisfied literals is 0 and the number of unvalued literals is 1. By Definition 4.2.3, a pseudo-Boolean constraint c is unit if $\texttt{possible}(c, P) < a_i$ for some literal $x_i \in c$. Determining whether c is unit will require scanning the literals in the constraint to determine if any literal has a coefficient strictly greater than $\texttt{possible}(c, P)$. This can be made more efficient by ordering the literals by decreasing weight. Now we can simply walk along the clause, stopping when we reach the first literal for which $a_i \leq \texttt{possible}(c, P)$. Determining whether a constraint is unit in the pseudo-Boolean case requires at most a partial walk of the constraint. The complexity of this task increases minimally over the clausal case; however, a single pseudo-Boolean constraint may be equivalent to an exponential number of Boolean clauses. Checking whether a single pseudo-Boolean constraint contains a unit literal may require a series of checks in the clausal case.

For cardinality constraints, where all weights are 1, there is no increase in complexity relative to the clausal case. If $\texttt{possible}(c, P) > 0$, we only need to inspect the first literal to determine that no unit propagation is possible. If $\texttt{possible}(c, P) = 0$, we walk the clause, setting every unvalued variable to 1. In general, once a constraint is determined to contain a unit literal, the constraint must be scanned to determine which literal is unit. This is the same in both the pseudo-Boolean and clausal cases with the exception that a pseudo-Boolean constraint may contain more than one unit literal.

4.2.2 Watched Literals

In Chapter 2 we discussed the watched literals method. Here we describe a watched literals method for pseudo-Boolean constraints. Recall that the watched literals method maintains a set of two pointers into each clause with the properties that either both literals are unvalued, or at least one of the literals is satisfied. As long as one of these properties is met, the clause cannot be unit. First, we generalize this rule for cardinality constraints.

Definition 4.2.8. *Let c be a cardinality constraint of the form $\sum_i x_i \geq k$ and let S be a subset of the literals in c. S is a* watching set *for c under a partial assignment P if and only if either of the following properties on S is true:*

1. $\texttt{possible}(S, P) \geq 1$

2. $\texttt{current}(S, P) \geq 0$

Proposition 4.2.9. *If S is a watching set for c under P then c cannot be unit.*

Proof. If S is a watching set for c under partial assignment P and the first property is true, then $\texttt{possible}(c, P) \geq 1$ and c is not unit. If the second property holds, then the clause is satisfied under P and again c is not unit. □

Let S be a watching set for a constraint c under partial assignment P. Consider what happens when a literal $x_i \in S$ is valued unfavorably by adding $\overline{x}_i$ to P to form a new assignment P'. There are two ways to obtain a watching set for c under P': create a new watching set $S' = S - \{x_i\} \cup \{x_j\}$ such that $x_j \in c$, $x_j \in V_{P'}(c)$ and $x_j \notin S$, or keep the same watching set S and extend partial assignment P' by setting the remaining literals in S to 1. If $k = 1$, then $|S| = 2$ and Definition 4.2.8 reduces to the clausal watching set definition.

A version of the watched literals rule for general pseudo-Boolean constraints is also possible.

Definition 4.2.10. *Let c be a pseudo-Boolean constraint of the form $\sum_i a_i x_i \geq k$ and let S be a subset of the literals in c. S is a* watching set *for c under a partial assignment P if either of the following properties on S is true:*

1. $\texttt{possible}(S, P) \geq \max_{x_i \in S}(a_i)$.

2. $\texttt{current}(S, P) \geq 0$

Proposition 4.2.11. *If S is a watching set for c under P then c is not unit.*

Proof. If the second property is true, then the constraint is satisfied under P and c is not unit. Now assume that the first property is true. We have

$$\texttt{possible}(c, P) \geq \texttt{possible}(S, P) \geq \max_{x_i \in S}(a_i)$$

Any unit literal in c must have a coefficient strictly greater than $\max_{x_i \in S}(a_i)$. This immediately excludes all of the literals in S as possible unit literals.

Let literal x_j with coefficient a_j be a unit literal in c such that $x_j \notin S$. The value of $\texttt{possible}(c, P)$ is defined as

$$\texttt{possible}(c, P) = \sum_{i | x_i \in V_P(c)} a_i - k \tag{4.18}$$

The set $V_P(c)$ can be broken down into the disjoint sets $V_P(S)$ and $V_P(c - S)$, so (4.18) can be rewritten as

$$\begin{aligned} \texttt{possible}(c, P) &= \textstyle\sum_{i | x_i \in V_P(c-S)} a_i + \sum_{i | x_i \in V_P(S)} a_i - k \\ &= \textstyle\sum_{i | x_i \in V_P(c-S)} a_i + \texttt{possible}(S, P) \end{aligned}$$

Because x_j is unvalued, it must be that $x_j \in V_P(c - S)$ and $\sum_{i | x_i \in V_P(c-S)} a_i \geq a_j$. Additionally we know that $\texttt{possible}(S, P) \geq \max_{x_i \in S}(a_i)$. So it must be that

$$a_j > \texttt{possible}(c, P) \geq a_j + \max_{x_i \in S}(a_i)$$

This is a contradiction. We can conclude that if the first property is true, c is not unit.

□

As before, if S is a watching set for a constraint c under partial assignment P, and we extend P to $P' = \{P, \overline{x}_j\}$ for some $x_j \in S$, then there is always a new watching set for c under P'. Either find a new set S' that satisfies Definition 4.2.10, or or keep the same watching set S and value a subset of the remaining literals in S to 1 satisfying the constraint. This is always possible since the coefficient a_j of x_j will be less than or equal to $\max_{x_i \in S}(a_i)$ assuring that the sum over the coefficients of the remaining

literals in S is at least k. Again we will require that partial assignments to be unit complete. This ensures that watching sets remain valid during backtracking.

We now compare the computational cost of maintaining watching sets for pseudo-Boolean constraints versus their clausal counterparts. Recall that, in the clausal case, when a watched literal is valued to `false`, the clause is searched for a new watching set. In the worst case, no watching set is found and the entire clause must be traversed. The remaining watched literal is added to the unit propagation list to satisfy the clause. The worst case complexity for maintaining a watching set is $O(n)$ where n is the number of variables in the theory. The pseudo-Boolean case is exactly the same. If a watched literal is valued to `false`, we again search the constraint for a new watching set, and again in the worst case no watching set is found and the entire constraint must be traversed, giving a worst case complexity of $O(n)$.

Where the two methods will differ is in the overall sizes of the literal indexes. Consider that a cardinality constraint

$$x_1 + x_2 + \cdots + x_m \geq k$$

requires only $k + 1$ watched literals. The equivalent Boolean clause set requires $\binom{m}{m-k+1}$ clauses with two watched literals per clause for a total of $2\binom{m}{m-k+1}$ watched literals. The overall number of literals that need to be watched for a pseudo-Boolean constraint set may be significantly smaller than the number required for the equivalent Boolean clause set.

4.2.3 Summary

The unit propagation procedure can be adapted to the pseudo-Boolean setting. Both count-based and watched literal methods are possible and both show minimal or no increase in cost per constraint. Additionally, the pseudo-Boolean case is likely to benefit from the added expressiveness of pseudo-Boolean constraints. Any of the tasks of maintaining counts, identifying unit propagations, or maintaining watching sets for a single pseudo-Boolean constraint will need to be carried out separately for each of the possibly large number of Boolean clauses required for the equivalent clausal encoding.

The original work on pseudo-Boolean count-based methods was done by Barth [3]. Our PBCHAFF implementation currently applies a watched literal method for all cardinality constraints (including simple disjunctions), and uses a count-based method for pseudo-Boolean constraints with coefficients strictly greater than 1. An equivalent implementation of watched literals for pseudo-Boolean constraints appears to have been done in parallel by Chai [12].

4.3 Learning

The choices we make in our pseudo-Boolean learning method will determine the ultimate strength of our underlying proof system. There are numerous choices to make for an implementation of pseudo-Boolean learning, and not all of them are good. The first bad choice is to not implement learning at all.

Lemma 4.3.1. *A pseudo-Boolean implementation of* DPLL *without learning is equivalent to tree style resolution.*

Proof. In Lemma 4.2.5, we saw that a pseudo-Boolean implementation of unit propagation produces exactly the same set of unit literals as unit propagation on the equivalent set of CNF clauses. It follows that for a given set of branch choices, a pseudo-Boolean version of DPLL will produce exactly the same search tree as the CNF version of DPLL that, in Section 3.1.1, was shown to be equivalent to tree-style resolution. □

Despite the stronger representation being used, the underlying inference system remains resolution-based. A simple DPLL implementation has no way to take advantage of the stronger representation through inference. What is missing is the parallel inference described in Section 4.1.3. Recall that this type of inference was essential for generating short cutting-plane proofs of the pigeonhole problem. In Chapter 5 we will see that pseudo-Boolean versions of DPLL show exponential runtime scaling on pigeonhole problems. Because learning is the primary inference method of DPLL-style algorithms, it is the key to determining the strength of the underlying proof system.

Learning should be implemented to take maximum advantage of the pseudo-Boolean representation through parallel inference.

4.3.1 Resolution Analog

A pseudo-Boolean implementation of learning requires a pseudo-Boolean version of resolution. Resolution is ideal for analyzing conflicts due to the partitioning scheme used by DPLL-style algorithms. We can define a pseudo-Boolean version of resolution in the following way.

Definition 4.3.2. *[40] Given pseudo-Boolean constraints c_1 and c_2 of the form*

$$ax + \sum_i a_i x_i \geq k$$

and

$$b\overline{x} + \sum_i b_i x_i \geq l$$

we call the derived constraint

$$\sum_i b(a_i x_i) + \sum_i a(b_i x_i) \geq bk + al - ab$$

the pseudo-Boolean resolvent *of the two constraints.*

We take a linear combination of the constraints c_1 and c_2 in a way that causes the variable x to cancel out of the resulting constraint.

4.3.2 Capturing Conflicts

The goal of learning methods is still to capture the cause of a conflict in the form of a new learned constraint. This learned clause can then be added to the constraint set to prevent the same set of bad assignments from recurring. The obvious implementation of learning for the pseudo-Boolean case is to replace the CNF resolution rule with the pseudo-Boolean analog. We will see that this approach has many of the benefits that we hoped for. Unfortunately, it is not sufficient to always capture the immediate conflict. This is best illustrated with some examples. Recall that a

contradiction occurs when a partial assignment together with two constraints causes a variable to be labeled both 1 and 0. Consider the following example.

Example 4.3.3. *Suppose we have a partial assignment $P = \{c, e, \bar{b}, \bar{d}\}$, and constraints*

$$a + d + e \geq 2 \tag{4.19}$$

$$\overline{a} + b + c \geq 2 \tag{4.20}$$

These constraints cause the variable a to be simultaneously forced to 1 and 0. Under inspection we can see that this conflict is caused by the assignments $\{\bar{b}, \bar{d}\}$ since the assignments $\{c, e\}$ both help toward satisfying the constraints. The pseudo-Boolean resolvent of (4.19) and (4.20)

$$d + e + b + c \geq 3 \tag{4.21}$$

eliminates this bad set of assignments since it implies the clause $b \vee d$.

The constraint (4.21) does more than eliminate the immediate conflict. It also eliminates additional bad assignments. Consider the logically equivalent set of CNF clauses.

$$\begin{array}{cc} d \vee e & e \vee b \\ d \vee b & e \vee c \\ d \vee c & b \vee c \end{array}$$

The constraint (4.21) also eliminates the assignments $\{\bar{c}, \bar{e}\}$ which is not part of the current partial assignment, in a sense eliminating a mistake we have not yet made.

Here we see the value of parallel inference from Section 4.1.3 within the context of a search algorithm.

Observation 4.3.4. *A single pseudo-Boolean inference step may require multiple resolution steps on CNF clauses to generate the logically equivalent set of clauses. In the context of search, this corresponds to the elimination of multiple conflicts with a single learned constraint.*

Unfortunately it is possible to construct cases where the pseudo-Boolean resolvent does not exclude the set of assignments causing the contradiction.

Example 4.3.5. *Given the partial assignment $P = \{c, e, \bar{b}, \bar{d}\}$, and constraints*

$$2a + d + e \geq 2 \tag{4.22}$$

$$2\overline{a} + b + c \geq 2 \tag{4.23}$$

assignment P together with constraint (4.22) implies a, and assignment P together with constraint (4.23) implies $\bar{a}$. The pseudo-Boolean resolvent $d + e + b + c \geq 2$, allows the set of assignments in the conflict set $\{\bar{b}, \bar{d}\}$.

Note that variables c and e are valued favorably for the constraints, in both cases helping to satisfy the constraints. The assignments $\{c, e\}$ are not part of the conflict. Adding the pseudo-Boolean resolvent to the constraint set does not prevent a repetition of this mistake, nor does it give any direction to the backtrack since it is satisfied under the current partial assignment.

To solve this problem we need a learning method that is guaranteed to generate an unsatisfied constraint. Additionally we want a learned constraint that is as strong as possible, eliminating as many conflicts as possible. We begin by showing that the pseudo-Boolean resolvent will generate an unsatisfied learned constraint if the coefficient of the conflict variable has a value of 1 in at least one of reasons for the conflict.

Proposition 4.3.6. *Given constraints c_1 of form $qx_0 + \sum a_i x_i \geq j$ and c_2 of form $\overline{x}_0 + \sum b_i x_i \geq k$ that cause a conflict with variable x_0 under partial assignment P, the pseudo-Boolean resolvent*

$$\sum a_i x_i + \sum q(b_i x_i) \geq j + qk - q \tag{4.24}$$

is unsatisfied under P.

Proof. The constraint c_1 forces $x_0 = 1$ under P, so the sum over the satisfied and unvalued literals is less than j. It follows that

$$\sum_{i | x_i \in V_P(c_1)} a_i - j \leq -1 \tag{4.25}$$

Similarly, c_2 forces $x_0 = 0$ under P, so we have

$$\sum_{i|x_i \in V_P(c_2)} b_i - k \leq -1$$

which we multiply by q

$$\sum_{i|x_i \in V_P(c_2)} q(b_i) - kq \leq -q \tag{4.26}$$

Summing (4.25) and (4.26) and adding q to both sides yields the inequality

$$\sum_{i|x_i \in V_P(c_1)} a_i + \sum_{i|x_i \in V_P(c_2)} q(b_i) - j - kq + q \leq -1$$

in which the left hand side is the value of `possible` for the pseudo-Boolean resolvent (4.24). (4.24) is unsatisfied under P. □

If we have reasons c_1 and c_2 that together force assignments x_j and $\bar{x}_j$, and the coefficients for x_j and $\bar{x}_j$ respectively in c_1 and c_2 are both greater than 1, then we will reduce one of the constraints, say c_1, to a cardinality constraint. In the reduced cardinality constraint, the coefficient of x_j is now 1 and we will be guaranteed to generate an unsatisfied clause. There are two things we must consider when reducing c_1. First, we need to generate a cardinality constraint c_1' that is still a valid reason for valuing x_j to 1. Additionally, such a reduction will weaken the constraint so care must be taken to generate as strong a cardinality constraint as possible. We begin here with a method for generating a cardinality constraint from a pseudo-Boolean constraint.

Proposition 4.3.7. *Let c be a pseudo-Boolean constraint $\sum_i a_i x_i \geq k$ and L be a subset of literals x_i such that $\sum_{i|x_i \notin L} a_i < k$. Define the set S to be*

$$S = \{x_i | x_i \notin L \text{ and } a_i \geq \max_{x_j \in L}(a_j)\}$$

The cardinality constraint

$$\sum_{x_i \in L} x_i + \sum_{x_i \in S} x_i \geq 1 + |S|$$

is entailed by c.

Proof. First we break down the left hand side of constraint c into parts defined by L and S. Let $a_{max} = \max_{x_j \in L}(a_j)$. Now we take the following linear combination.

$$\begin{array}{rr} \sum_{x_i \in L} a_i x_i + \sum_{x_i \in S} a_{max} x_i + \sum_{x_i \in S}(a_i - a_{max})x_i + \sum_{x_i \in \overline{L \cup S}} a_i x_i & \geq k \\ \sum_{x_i \in S}(a_i - a_{max})\overline{x}_i + \sum_{x_i \in \overline{L \cup S}} a_i \overline{x}_i & \geq 0 \\ \hline \end{array}$$

$$\sum_{x_i \in L} a_i x_i + \sum_{x_i \in S} a_{max} x_i \geq k - \sum_{x_i \in S}(a_i - a_{max}) - \sum_{x_i \in \overline{L \cup S}} a_i$$

Rearranging the right hand side gives the inequality

$$\sum_{x_i \in L} a_i x_i + \sum_{x_i \in S} a_{max} x_i \geq k - \sum_{x_i \notin L} a_i + \sum_{x_i \in S} a_{max}$$

Applying the definition of the set L, it follows that

$$\sum_{x_i \in L} a_i x_i + \sum_{x_i \in S} a_{max} x_i \geq 1 + \sum_{x_i \in S} a_{max}.$$

Finally, dividing the inequality by a_{max} and rounding up the right hand side gives

$$\sum_{x_i \in L} x_i + \sum_{x_i \in S} x_i \geq 1 + |S|$$

□

Now we must ensure that the cardinality constraint c_1' we construct forces the assignment x_j just as our original reason c_1 did.

Example 4.3.8. *The constraint* $3a + 2b + 2d + e + f \geq 6$ *is a reason for* d *under partial assignment* $P = \{\bar{e}, \bar{f}\}$. *To reduce this constraint to a cardinality constraint, we build a disjunction corresponding to the set* L *in Proposition 4.3.7 that is implied by the original constraint and is also a reason for* d *under* P, *namely* $d \vee e \vee f$. *Next we construct the set* S *from literals in the constraint with coefficients greater than or equal to the maximum coefficient for a literal in* L. *Here we have* $S = \{a, b\}$. *Putting it all together, we sum over sets* L *and* S *and set the right hand side of the inequality to* $1+|S| = 3$ *producing the cardinality constraint* $d+e+f+a+b \geq 3$. *This constraint is also a reason for* d *under* P.

Procedure 4.3.9 (Reduce-To-Cardinality). *Given a constraint c_1 of the form $\sum a_i x_i \geq k$ such that c_1 is a reason for x_j under P, to compute* REDUCE-TO-CARDINALITY(c_1, P)*:*

1 $L \leftarrow \{x_j\} \cup T$ where T is a subset of the literals in c_1, for all $x_i \in T$, $x_i \notin V_P(c_1)$, and a disjunction over the literals in L is implied by c_1

2 $S \leftarrow x_i$ such that $x_i \in c_1$ and $a_i \geq \max_{x_j \in L}(a_j)$

3 **return** $\sum_{x_i \in L} x_i + \sum_{x_i \in S} x_i \geq 1 + |S|$

Proposition 4.3.10. *Given a constraint c_1 of the form $\sum a_i x_i \geq k$ that contains unit literal x_j under P,* REDUCE-TO-CARDINALITY(c_1, P) *as computed by Procedure 4.3.9 returns a constraint c_1' that is a valid reason for the assignment x_j under P.*

Proof. The existence of the set L follows from Lemma 4.2.5. Consider the value of possible(c_1', P). The sum over set L contributes at most 1 to the count since it has one satisfied literal x_j and the rest are unsatisfied under P. The sum over set S contributes at most $|S|$ to the count. This gives a count

$$\texttt{possible}(c_1', P) \leq 1 + |S| - (1 + |S|) = 0$$

implying that c_1' is in fact a valid reason for x_j under partial assignment P. □

Returning to Example 4.3.5, the constraint $2a + d + e \geq 2$ is reduced to the cardinality constraint $a + d \geq 1$ using the cardinality reduction described above, and then the pseudo-Boolean resolvent

$$2d + b + c \geq 2$$

is generated. This resolvent is unsatisfied under partial assignment $P = \{c, e, \bar{b}, \bar{d}\}$, and disallows the set of assignments in the conflict set $\{\bar{b}, \bar{d}\}$.

Procedure 4.3.11 (PB-Learn). *Given constraints* c_1 *of form* $qx_0 + \sum a_i x_i \geq j$ *and* c_2 *of form* $r\overline{x}_0 + \sum b_i x_i \geq k$ *that cause a conflict with variable* x_0 *under partial assignment* P*, to compute* PB-LEARN(c_1, c_2, P)*:*

1 **if** $q > 1$ **and** $r > 1$
2   **then** $c_1 \leftarrow$ REDUCE-TO-CARDINALITY(c_1, P)
3 **return** `PB-resolve`(c_1, c_2)

Proposition 4.3.12. *Given constraints* c_1 *of form* $qx_0 + \sum a_i x_i \geq j$ *and* c_2 *of form* $r\overline{x}_0 + \sum b_i x_i \geq k$ *that cause a conflict with variable* x_0 *under partial assignment* P*,* PB-LEARN(c_1, c_2, P) *as computed by Procedure 4.3.11 returns a constraint* c *that is entailed by* c_1 *and* c_2*, and unsatisfied under* P*.*

Proof. Entailment of c from c_1 and c_2 follows from Proposition 4.3.7 and the soundness of pseudo-Boolean resolution. That c is unsatisfied under P follows from Proposition 4.3.6 and Proposition 4.3.10. □

The current implementation of PBCHAFF has two learning modes. The first learns full pseudo-Boolean constraints applying the methods described above. The second learning mode learns only cardinality constraints. The previous methods are applied, and if the resulting learned constraint is not a cardinality constraint, it is reduced to one using Proposition 4.3.7. Full pseudo-Boolean representation is a stronger representation than cardinality constraints alone. Some learned information may be lost in the translation to a cardinality constraint. The benefit of cardinality constraints is that the unit propagation procedure for cardinality constraints is extremely fast. Both forms of learning add representational strength over CNF representation.

4.3.3 Bounded Learning

In the clausal case, the length of a clause was a useful metric in determining the pruning potential of a clause because it correlated indirectly with the number of assignments the clause eliminated from the overall search space. Relevance bounded learning builds on this idea and defines length relative to a partial assignment. Again

the relevant length corresponds to the number of assignments eliminated from a subproblem in the search space.

Within the pseudo-Boolean context, the number of literals in a clause no longer correlates with the number of assignments a constraint eliminates. Consider the constraints

$$a + b + c \geq 2 \tag{4.27}$$

$$a + b \geq 1 \tag{4.28}$$

$$a + b + c \geq 1 \tag{4.29}$$

If these constraints are part of a larger theory that has a total of 5 variables, then (4.27) has 3 literals and eliminates 16 assignments, (4.28) has 2 literals and eliminates 8 assignments, and (4.29) has 3 literals and eliminates 4 assignments. Clearly new definitions of *length* and *irrelevance* are needed.

For any cardinality constraint $x_1 + \cdots + x_m \geq k$ the number of assignments eliminated from the overall search space is

$$\left[\binom{m}{0} + \binom{m}{1} + \cdots + \binom{m}{k-1}\right] 2^{n-m}$$

where n is the total number of variables in the theory. Unfortunately, we do not know any efficient way to calculate the number of assignments eliminated by a general pseudo-Boolean constraint.

As the following example shows, it still makes sense to define the relevance of a constraint in relation to the current position in the search space. Consider the following nogoods given the partial assignment $P = \{a, b, c\}$:

$$a + b + c \geq 2 \tag{4.30}$$

$$a + b \geq 1 \tag{4.31}$$

$$\overline{a} + \overline{b} + e + f \geq 1 \tag{4.32}$$

Constraints (4.30) and (4.31) cannot be used for pruning anywhere in the subproblem below. The constraint (4.32), which is the weakest constraint, can be used to prune any node in the subproblem that contains $\{\overline{e}, \overline{f}\}$ as part of its partial assignment, making it the most useful to the immediate subproblem.

The PBCHAFF implementation employs a version of the hybrid bounded learning method used by the ZCHAFF algorithm. Recall that this method uses a relevance bound together with a larger length bound. Clauses that meet both the relevance and length bound are retained in the clause set. Adapting this heuristic to the pseudo-Boolean setting requires that we redefine our notion of *length* and *irrelevance*. The definitions used are simple and easy to calculate.

Definition 4.3.13. *The* length *of a pseudo-Boolean constraint* $\sum a_i x_i \geq k$ *is defined as* $\sum a_i - k + 1$.

Proposition 4.3.14. *If a pseudo-Boolean constraint is equivalent to a disjunction, then Definition 4.3.13 reduces to the clausal definition of length.*

Proof. Given a pseudo-Boolean constraint of the form $\sum_{i \in S} x_i \geq 1$, its length under definition Definition 4.3.13 is $|S| - 1 + 1 = |S|$. □

If the constraint is a cardinality constraint, the length of the constraint by Definition 4.3.13 is equivalent to the length of any clause in the equivalent set of Boolean clauses. If the constraint has coefficients strictly larger than 1, then the length by Definition 4.3.13 will be greater than or equal to the length of any individual clause in the equivalent Boolean clause set. This creates a somewhat biased policy that prefers cardinality constraints over constraints with larger coefficients.

Definition 4.3.15. *Given pseudo-Boolean constraint* c *and partial assignment* P, *the* irrelevance *of* c *under* P *is defined as* $\texttt{possible}(c, P) + 1$.

Proposition 4.3.16. *If a pseudo-Boolean constraint is equivalent to a disjunction, then Definition 4.3.15 reduces to the clausal definition of irrelevance.*

Proof. Given a pseudo-Boolean constraint c of the form $\sum_{i \in S} x_i \geq 1$ and partial assignment P, by definition

$$\texttt{possible}(c, P) = \sum_{i | x_i \in V_P(c)} 1 - 1 + 1$$

The value of possible is clearly equal to the number of satisfied literals plus the number of unvalued literals in the constraint. □

This policy generalizes irrelevance in a way similar to the above generalization of length. For cardinality constraints, the irrelevance of a constraint by Definition 4.3.15 will correspond to the irrelevance of the clause in the set of Boolean clauses that has the smallest clausal irrelevance measure. Again the policy is biased toward cardinality constraints over constraints with larger coefficients.

4.3.4 Summary

When we adapt a resolution-based method to use the stronger pseudo-Boolean representation, the strength of the underlying proof system will be largely determined by the learning method. The learning method defined in this section encourages parallel inference where a single inference step can correspond to multiple resolution steps. The pseudo-Boolean inference performs these resolutions in parallel. Within a search context, this corresponds to the ability to eliminate multiple conflicts with a single learned constraint. In addition to eliminating the immediate conflict, the solver can now avoid analogous conflicts in parts of the search space not yet explored. A resolution-based method is forced to discover these errors independently. Later in Section 4.6 we will see how building inference can be automated. This will increase the power of the underlying proof system further.

The main inference step used in pseudo-Boolean learning is the pseudo-Boolean analog of resolution. This inference is not sufficient to capture all conflicts and care must be taken to ensure that immediate conflict is always eliminated. When necessary one of the reasons for a conflict can be reduced to a cardinality constraint, which avoids generating a satisfied constraint. These reductions must be done carefully so that each learning step captures as much information as possible.

4.4 Non-Standard Backtracking and Unique Implication Points

Both the techniques of non-standard backtracking and unique implication points lift easily to the pseudo-Boolean setting. We begin with a discussion of pseudo-

Boolean UIP constraints, and follow with pseudo-Boolean versions of procedures BACKJUMP and NON-STANDARD-BACKJUMP.

In Section 2.2.2 we defined a UIP clause informally as having exactly one literal assigned a value at the current decision level. The remaining literals were valued earlier at lower decision levels. Consider the equivalent definition.

Definition 4.4.1. *Let c be a learned constraint such that l_1 is the decision level at which c is unit and l_2 is the decision level at which c is generated. Constraint c is a* UIP *constraint if and only if $l_1 < l_2$.*

Within the clausal framework these definitions create equivalent results. This is not the case in the pseudo-Boolean framework. Consider the following example. Assume that the assignments $\{a, b, \bar{c}\}$ were valued at the current decision level, and $\{\bar{d}\}$ was valued at some earlier decision level. The learned constraint

$$a + b + c + d \geq 3$$

has 3 literals valued at the current decision level, yet it is clearly unit at the earlier decision level that contains $\bar{d}$. PBCHAFF implements UIP constraints using Definition 4.4.1.

We chose to implement UIP constraint learning because ZCHAFF implements UIP clause learning. Many implementation choices were made to mirror implementation choices in ZCHAFF, with the goal of building a pseudo-Boolean version of ZCHAFF that differs only in its choice of representation. It is easier to examine the benefits of the representational change if all other factors are equal.

Procedure 4.4.2 (PB-DPLL-with-Learning). *Given a pseudo-Boolean problem C and a partial assignment P of values to variables, to compute* PB-DPLL-WITH-LEARNING(C, P) *:*

```
1   while P is not a full assignment
2       do P ← PB-UNIT-PROPAGATE(C, P)
3          if P contains a contradiction
4             then v ← contradiction variable
5                  c1 ← reason associated with v
6                  c2 ← reason associated with v̄
7                  if PB-BACKJUMP(PB-LEARN(c1, c2, P), P) = FAILURE
8                     then return FAILURE
9             else l ← a literal not assigned a value by P
10                 P ← {P, (l, true)}
11  return SUCCESS
```

Procedure 4.4.3 (PB-Backjump). *Given a nogood c that is unsatisfied under annotated partial assignment $P = \{(l_1, c_1), (l_2, c_2), \ldots, (l_m, c_m)\}$, to compute* PB-BACKJUMP(c, P)*:*

```
1  if c is the empty constraint
2     then return FAILURE
3  if c is a UIP constraint
4     then return PB-NON-STANDARD-BACKJUMP(c, P)
5     else
6          li ← literal in P with maximum i such that c is
                  satisfied under {l1, l2, ..., l(i-1), l̄i}
7          return PB-BACKJUMP(PB-LEARN(ci, c, P), P)
```

Procedure 4.4.4 (PB-Non-Standard-Backjump). *Given a nogood c that is unsatisfied under annotated partial assignment $P = \{(l_1, c_1), (l_2, c_2), \ldots, (l_m, c_m)\}$, to compute* PB-NON-STANDARD-BACKJUMP(c, P)*:*

1 add c to the constraint set
2 $l_i \leftarrow$ literal in P with minimum i such that c is unit under
 $\{(l_1, c_1), (l_2, c_2), \ldots, (l_j, c_i)\}$
3 $[j_1, \ldots, j_m] \leftarrow$ the set of literals in c that are unit under
 $\{(l_1, c_1), (l_2, c_2), \ldots, (l_j, c_i)\}$
4 $d \leftarrow$ decision level such that $l_i \in d$
5 $l_k \leftarrow$ literal in d with maximum k
6 $P \leftarrow \{(l_1, c_1), (l_2, c_2), \ldots, (l_k, c_k), (j_1, c), \ldots, (j_m, c)\}$
7 **return** SUCCESS

Once a constraint is determined to be a UIP constraint, non-standard backjumping is applied. The pseudo-Boolean versions of the BACKJUMP and NON-STANDARD-BACKJUMP procedures follow their clausal counterparts closely. There are two small differences. In PB-BACKJUMP, the simple clause resolution denoted by `resolve` in line 7 is replaced with the PB-LEARN procedure defined in Section 4.3.2. In PB-NON-STANDARD-BACKJUMP the number of unit propagations caused by a constraint may be more than 1. This set is identified in line 3 and appended to the partial assignment in line 6.

Proposition 4.4.5. PB-DPLL-WITH-LEARNING(C, P) *as computed by Procedure 4.4.2 is both sound and complete.*

Proof. Soundness follows directly from the soundness of pseudo-Boolean unit propagation, Proposition 4.2.7 and the soundness of pseudo-Boolean learning, Proposition 4.3.12.

For the proof of completeness, recall that the definition of progress at a decision level defined in Proposition 2.2.9 assumed that learned clauses are unsatisfiable under the current partial assignment. We need to apply Proposition 4.3.12 again, this time

to show that PB-LEARN returns a learned constraint unsatisfiable under the current partial assignment. Now we simply repeat the proof of Proposition 2.2.9. We prove that progress is made at every decision level by induction on the maximum number of decision levels seen during the solution of a problem.

For the base case, let a contradiction occur at decision level d_m. We learn a UIP constraint and backjump to some decision level with index less than m and make progress at that level.

For the inductive case we assume that we make progress at some decision level d_j such that $j < i$. If $j < i-1$ then we make progress at a decision level with index less than $i-1$. If $j = i-1$ then we make progress at decision level d_{i-1}. If we continue searching, we will either at some point backjump to a decision level with index less than $i-1$ or we will continue to append assignments to decision level d_{i-1} until a contradiction occurs at decision level d_{i-1} or a solution is found. If a contradiction occurs we again learn a UIP constraint and backjump to a decision level with index less than $i-1$.

We can conclude that if no solution is found, progress is made at decision level d_0 and the empty constraint $0 \geq 1$ is eventually derived. □

4.5 Branching Heuristics

The branching heuristic currently used by PBCHAFF is based on the VSIDS heuristic. Recall that this heuristic maintains literal counts based on the number of times each literal occurs in the theory being solved. Counts are incremented each time a clause is added to the clause set. This is done for learned clauses as well clauses in the original constraint set. The heuristic prefers to branch on an unassigned variable with a high count value. Periodically all counts are divided by a constant factor, creating a bias toward branching on variables occurring in recently learned constraints.

This heuristic is quite easy to adapt to the pseudo-Boolean setting. The policy for incrementing counts varies depending on the type of constraint being added to the constraint set. Cardinality constraints and pseudo-Boolean constraints with coefficients larger than one are handled differently.

Cardinality constraints that are part of the original constraint set are managed differently than learned cardinality constraints. If a cardinality constraint is one of the initial constraints, the literal counts are incremented as if the cardinality constraint were added in its equivalent CNF form. If it is added as a learned constraint, the literal counts are incremented by one for each literal in the constraint. For example, if we add the constraint

$$x_1 + x_2 + x_3 \geq 2$$

as one of the original constraints, then we would increment literal counts for x_1, x_2 and x_3 by two each. If instead the same constraint was added as a learned constraint, we would increment each of the literal counts x_1, x_2 and x_3 by only one.

When a constraint with coefficients greater than one is added to the constraint set, we increment the literal count by one for each literal in the constraint. The increment is always one and is not weighted by the weight of the literal.

The branch heuristic described above was chosen because it closely mirrors the zCHAFF heuristic and it performed well in informal experiments. Further study is needed to evaluate a larger range of branch heuristic for pseudo-Boolean solvers.

4.6 Strengthening

We now present a technique that automates building inference, which was discussed in Section 4.1.3. The goal of building inference is to infer new compound constraints from a set of clausal constraints, or weaker compound constraints. The effect is to build up a stronger more concise representation of a problem from a larger and weaker encoding.

The following method is a form of *coefficient reduction*. It is used in the operations research field most often as a preprocessing method for mixed-integer-programming problems [37, 61]. We have adapted it somewhat to the context of a DPLL-style algorithm.

Definition 4.6.1. *A constraint c of the form $\sum a_i x_i \geq k$ is* over satisfied *under partial assignment P if* $\texttt{current}(c, P) > 0$.

Procedure 4.6.2 (Strengthen). *Given a* SAT *problem C, to compute* STRENGTHEN(C)*:*

1 **for** each literal l in C
2 **do**
3 $P \leftarrow \{l\}$
4 $P \leftarrow$ UNIT-PROPAGATE(C, P)
5 **for** each constraint $c = \sum a_i x_i \geq k$ with $\texttt{current}(c, P) > 0$
6 **do**
7 $c \leftarrow \sum a_i x_i + \texttt{current}(c, P)\bar{l} \geq k + \texttt{current}(c, P)$

We begin with an empty partial assignment $P = \emptyset$. Now suppose we make the assignment l, adding it to P, and then apply unit propagation to our constraint set. We now discover that under the current partial assignment P, a constraint c is over satisfied with $\texttt{current}(c, P) = s$ for some $s > 0$. The over satisfied constraint can be replaced by a strengthened version of the constraint.

Proposition 4.6.3. *Given constraint set C and an assignment $P = \{l\}$, if $P' =$* UNIT-PROPAGATE(C, P)*, then any constraint c of the form $\sum a_i x_i \geq r$ with* $\texttt{current}(c, P') = s$ *such that $s > 0$, can be replaced by the constraint*

$$s\bar{l} + \sum a_i x_i \geq r + s \tag{4.33}$$

Proof. Let ρ be any full assignment that satisfies C. If $l \in \rho$, then $P' \subseteq \rho$ by unit propagation. This causes c to be over satisfied with $\texttt{current}(c, \rho) = s$, and $\sum a_i x_i \geq r + s$, so ρ satisfies (4.33). If $\bar{l} \in \rho$, then $s\bar{l} = s$. Since ρ satisfies every constraint in C, ρ must satisfy $\sum a_i x_i \geq r$. Again, ρ satisfies (4.33). It follows that (4.33) is implied by C.

To see that (4.33) implies c we take the linear combination and derive c from (4.33).

$$\begin{array}{ll} s\bar{l} + \sum a_i x_i & \geq r + s \\ sl & \geq 0 \\ \hline \sum a_i x_i & \geq r \end{array}$$

□

Example 4.6.4. *Consider the following set of clauses:*

$$a + b \geq 1 \tag{4.34}$$

$$a + c \geq 1 \tag{4.35}$$

$$b + c \geq 1 \tag{4.36}$$

If we set $P = \{\bar{a}\}$, we generate the unit propagations $\{b, c\}$. Constraint (4.36) is over satisfied and can thus be replaced by

$$a + b + c \geq 2$$

In fact, this new constraint subsumes all three original constraints, so (4.34) and (4.35) can be removed from the constraint set as well. The strengthened constraint will often subsume some or all of the constraints involved in generating it.

This rule can be generalized as follows.

Proposition 4.6.5. *Given constraint set C, partial assignment $P = \{l_1, l_2, \ldots, l_k\}$, assignment $P' =$ UNIT-PROPAGATE(C, P), and constraint $c \in C$ of form $\sum a_i x_i \geq r$ with* `current`$(c, P') = s$ *and $s > 0$, the constraint*

$$s \sum_{i=1}^{k} \bar{l}_i + \sum a_i x_i \geq r + s \tag{4.37}$$

is implied by C.

Proof. Let ρ be any full assignment that satisfies C. If $l_i \in \rho$ for every $l_i \in P$, then $P' \subseteq \rho$ by unit propagation. This causes c to be over satisfied with `current`$(c, \rho) = s$, and $\sum a_i x_i \geq r + s$, so ρ satisfies (4.33). If for any $l_i \in P$, $\bar{l}_i \in \rho$, then $s \sum_{i=1}^{k} \bar{l}_i \geq s$. Since ρ satisfies every constraint in C, ρ must satisfy $\sum a_i x_i \geq r$. Again, ρ satisfies (4.33). It follows that (4.33) is implied by C. □

The general rule can be automated as a preprocessing method as shown in Procedure 4.6.2 or applied during search. When a constraint becomes over satisfied under the current partial assignment, the set of assignments that caused the constraint to be over satisfied can be determined in time $O(n^2c)$ where n is the number of variables and c is the number of constraints. The procedure GET-ASSUMPTIONS can be applied to identify a set of assumptions $A \subseteq P$ that causes c to be over satisfied. The constraint is strengthened or a new constraint is learned.

Procedure 4.6.6 (Get-Assumptions). *Given a constraint c and an annotated partial assignment P such that* $\texttt{current}(c, P) = s$ *and* $s > 0$, *to compute* GET-ASSUMPTIONS(c, P):

1 $A \leftarrow \emptyset$
2 $L \leftarrow$ {satisfied literals in c}
3 $BEST \leftarrow L$
4 **while** $L \neq \emptyset$
5 **do**
6 $l \leftarrow$ a literal in L
7 remove l from L
8 $r \leftarrow$ reason for l
9 **if** $r = \texttt{true}$
10 **then** $A \leftarrow A \cup \{l\}$
11 **else** $L \leftarrow L \cup$ {unsatisfied literals in r}
12 **if** $|A \cup L| < |BEST|$
13 **then** $BEST \leftarrow |A \cup L|$
14 **return** $BEST$

Proposition 4.6.7. *Given a clause $c \in C$ and an annotated partial assignment P such that* $\texttt{current}(c, P) = s$ *and* $s > 0$, GET-ASSUMPTIONS(c, P) *as computed by Procedure 4.6.6 returns a set of assignments BEST such* $\texttt{current}(c, \texttt{closure}(BEST)) \geq s$.

Proof. We show that $\texttt{current}(c, \texttt{closure}(BEST)) \geq s$ by induction on the number of loop iterations. Let A_i and L_i be the elements of A and L respectively after i applications of the loop in Procedure 4.6.6. It is sufficient to show that $\texttt{current}(c, \texttt{closure}(A_i \cup L_i)) \geq s$ for all iterations i, because the set of returned assignments $BEST$ is the set $A_i \cup L_i$ for some i.

For the base case, $A_0 = \emptyset$ and $L_0 = P \cap \{c\}$. We know that $\texttt{current}(c, P) \geq s$ and L_0 is exactly the set of literals that are both in c and are satisfied by P. It follows that $\texttt{current}(c, L_0) \geq s$ and therefore $\texttt{current}(c, \texttt{closure}(A_0 \cup L_0)) \geq s$.

For the inductive case, assume that $\texttt{current}(c, \texttt{closure}(A_i \cup L_i)) \geq s$ for iteration i. If $r = \texttt{true}$, then $L_{i+1} = L_i - \{l\}$ and $A_{i+1} = A_i \cup \{l\}$, making $A_{i+1} \cup L_{i+1}$ equal to $A_i \cup L_i$, and $\texttt{current}(c, \texttt{closure}(A_{i+1} \cup L_{i+1})) \geq s$. If $r \neq \texttt{true}$, then $A_{i+1} = A_i$ and $L_{i+1} = L_i - \{l\} \cup Q$, where Q contains the literals in r that are unsatisfied by P. The literal l must be an element of $\texttt{closure}(A_{i+1} \cup L_{i+1})$ because l follows from Q by unit propagation. We have that $A_i \cup L_i \subset \texttt{closure}(A_{i+1} \cup L_{i+1})$ and therefore $\texttt{closure}(A_i \cup L_i) \subseteq \texttt{closure}(A_{i+1} \cup L_{i+1})$ so again

$$\texttt{current}(c, \texttt{closure}(A_{i+1} \cup L_{i+1})) \geq s$$

To prove termination of Procedure 4.6.6 we need to show that the condition $L = \emptyset$ is always reached. To do this we show that collecting the set of assumptions that cause a given set of assignments is equivalent to identifying for a set of nodes N in a directed acyclic graph (DAG) the set of all source nodes with a path to a node in N.

Given an annotated partial assignment P, we can construct a DAG as follows. We begin by adding the first annotated assignment (l_1, c_1) to the graph and continue through each assignment until all assignments have been added. For any assignment (l_i, c_i), if $c_i = \texttt{true}$ then l_i is a source node. If c_i is a constraint, then let S be the subset of $\{l_1, l_2, \ldots, l_{i-1}\}$ whose negations are in c_i. The set of assignments S force the assignment of l_i. Create a new node l_i, and add for each element s of S a directed edge from s to l_i. The graph formed is acyclic because edges always flow from nodes with low index in P to nodes with higher index in P. Set L is initialized with a subset I of nodes in the graph. The loop recursively removes a node from L and either replaces it with its parent nodes or if it is a source node, places it in the set

A. Because the graph is both finite and a DAG, termination follows, ending when all source nodes with a path to nodes in set I are found. □

Initial informal results suggest that the cost of applying strengthening for every occurrence of an over satisfied constraint is expensive. It is unclear whether an efficient implementation will provide benefits beyond those gained by preprocessing alone. However, excessive preprocessing can be expensive, so it may be valuable to let the search direct the strengthening process. This would also allow the possibility of strengthening constraints learned in response to contradictions.

CHAPTER 5

Solving the Pigeonhole Problem

In this chapter we explore the performance of the pseudo-Boolean satisfiability solver PBCHAFF described in Chapter 4. Before we begin, let us first review and integrate some of the results from earlier chapters with the goal of understanding what we should expect from PBCHAFF.

In Section 4.1.3 we discussed two categories of inference that make the cutting-plane proof system strong. The first, called building inference, makes it possible to build stronger compound constraints from a set of clausal or weak compound constraints. Recall that a compound constraint can be equivalent to an exponential number of clausal constraints. The second category of inference we defined was parallel inference. In parallel inference a new compound constraint is inferred from a set of compound constraints with the effect of performing multiple resolutions in parallel. Both types of inference were necessary for solving the pigeonhole problem. We used building inference to generate the concise encoding (4.12) from the clausal encoding (4.11). Then we were able to exploit the concise encoding with parallel inference to construct a short proof.

The situation is the same for an automated form of the cutting-plane proof system like PBCHAFF. The learning method used by PBCHAFF and defined in Section 4.3 automates a form of parallel inference. When applied to compound constraints, many resolutions occur in parallel and multiple conflicts are eliminated at once. When applied to clausal constraints, the method reduces to simulating resolution. The

power of the learning method is dependent on the conciseness of the encoding. If a concise pseudo-Boolean encoding of the problem is not known, then the strengthening method defined in Section 4.6 can be applied. This procedure automates a form of building inference and may be able to build the concise encodings needed.

If PBCHAFF is given a CNF encoding, it easily translates the problem into pseudo-Boolean by representing each clause as a clausal inequality. If strengthening is not applied, then the learning method reduces to resolution. Similarly, all other PBCHAFF methods such as bounded learning and branching heuristics reduce to the clausal versions when applied to clausal constraints. Although the solver uses pseudo-Boolean representation, in this situation the solver performs as a resolution based method. In Section 5.1.1 we examine how efficiently PBCHAFF performs as a resolution-based method. If PBCHAFF is able to efficiently simulate a resolution based solver, then there is little reason to prefer a traditional resolution-based solver over a pseudo-Boolean solver. In Section 5.2 we explore the potential benefits of using the stronger pseudo-Boolean representation by looking at the performance of PBCHAFF on pigeonhole problems. In this set of experiments, we apply strengthening to CNF encodings of the pigeonhole problem and then apply PBCHAFF to the stronger encoding. In Section 5.3 we go further still and look at performance on a set of planning problems with embedded pigeonhole problems, and in Section 5.4 we examine the performance of PBCHAFF on randomly generated planning problems.

5.1 CNF Encodings

5.1.1 Do No Harm

The resolution-based methods described in Chapter 2 have been very successful. As we stated earlier, the goal of our implementation is to improve the strength of the solver's proof system without abandoning these successes. To test whether the latter part of this goal has been achieved, we ran PBCHAFF as a resolution-based solver on a large number of problems from many different problem domains, and compared

its performance with zCHAFF. The goal of these experiments was to evaluate how efficiently PBCHAFF simulates a resolution-based solver.

The PBCHAFF algorithm, when functioning as a resolution-based method, follows the zCHAFF model so closely we would expect the two solvers to build search trees of similar size. The data structures used by PBCHAFF are necessarily different from those used by zCHAFF because PBCHAFF supports full pseudo-Boolean representation. We would expect the more expressive pseudo-Boolean representation to increase the cost of expanding a node.

We compared the performance of zCHAFF and PBCHAFF on a large and diverse set of problems encoded in CNF. We used a set of 488 problem instances from the Velev, DIMACS, and SatLib benchmark suites.

- Microprocessor test and verification benchmarks
  http://www.ece.cmu.edu/~mvelev

- DIMACS benchmarks
  ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf

- SatLib benchmarks
  http://wwww.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html

Some of the problem domains included are microprocessor verification, random satisfiability, planning, bounded model checking, parity, and pigeonhole instances. Both solvers used the same CNF encodings. Parameter settings for both solvers were set to the default values used by zCHAFF. Both solvers were set to time out after 1000 seconds. All experiments were run on 1.53MHz Athlon processors with 256K on-chip cache and 512MB of RAM. Code was compiled with Gnu g++ using full optimization and run under Linux.

5.1.2 Experimental Results

Figure 5.1 shows a comparison of solution time of zCHAFF and PBCHAFF on the set of CNF instances. A data point in the graph represents a problem instance. A

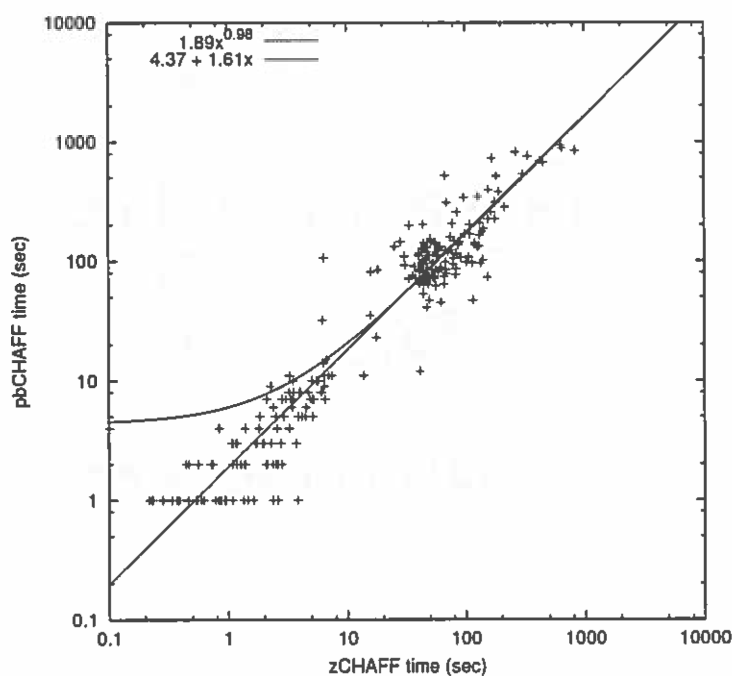


FIGURE 5.1: Comparison of execution time for PBCHAFF and ZCHAFF on CNF encodings. Each point corresponds to a CNF problem instance. The x coordinate corresponds to the execution time in seconds for ZCHAFF and the y coordinate corresponds to the execution time in seconds for PBCHAFF. The line $f(x) = 4.37 + 1.61x$ is the best linear fit to the data, and the curve $f(x) = 1.89x^{0.98}$ is the best log transformed linear fit.

point's x coordinate is the execution time for zCHAFF and the y coordinate is the execution time for PBCHAFF. Logscale is used on both axes. We fitted the data with both a linear model and a power function model. To fit the data with a power function model, both the x and y data are log transformed and then fit using a linear regression. A power function model is a good choice for this data set because the data has a strong linear trend when both axes are plotted using logscale and also because the variance or spread of the data tends to increase as solution time increases. Without the log transformation, the fit is biased toward the difficult instances. These points have very large residuals and dominate the regression. The log transformed linear regression weights the data more evenly and gives more insight into scaling properties. We used a Deming regression to fit the data instead of the more classic least squares regression because both the x and y data are subject to error. We made the assumption that the amount of error for both solvers was the same.

The best linear fit to the data was the line $f(x) = 4.37 + 1.61x$. As expected, the linear model does not fit the data in the lower left quadrant of the graph at all. The best fit to the log transformed data is the curve $f(x) = 1.89x^{0.98}$. It fits the data more closely and gives a more accurate prediction of how the data scales. For small problems PBCHAFF is slower by a factor of about 1.9. As problems get larger, this factor decreases somewhat to a factor of about 1.6.

We also calculated the cost of expanding a node for each solver on each problem and determined the average over the entire problem set. The average for PBCHAFF was 1.86×10^{-4} and the average for zCHAFF 1.28×10^{-4}. The ratio of these averages is again approximately 1.5.

Figure 5.2 shows a comparison of node counts for zCHAFF and PBCHAFF on the CNF encodings. The best fit for the log transformed data is the line $f(x) = 0.99x^{1.005}$, implying that node counts are approximately the same for both solvers on CNF encodings. Overall, the data suggests that the solvers build search trees of similar size, but zCHAFF is faster per node by a factor of about 1.6.

These experiments show that PBCHAFF functions quite well as a resolution-based method. The cost of supporting the more expressive pseudo-Boolean representation is a factor of around 1.6, allowing PBCHAFF to approximately match the performance

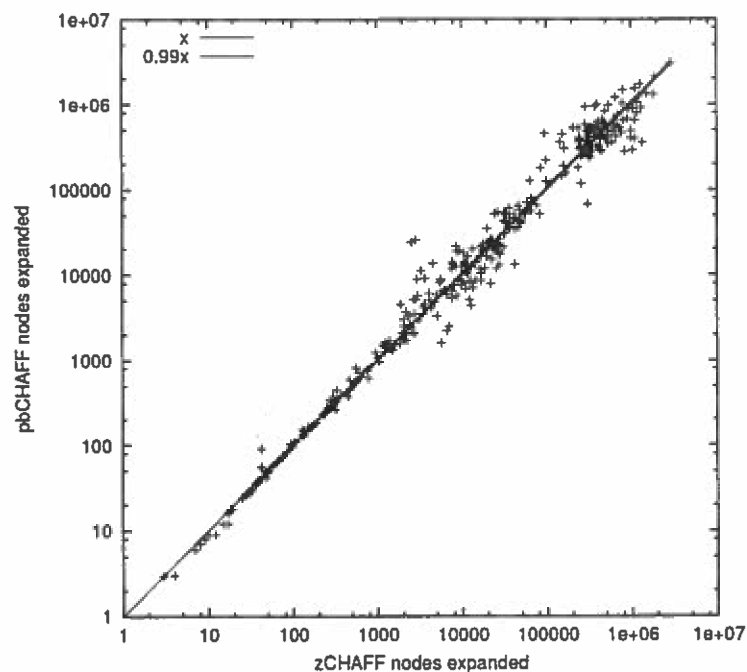


FIGURE 5.2: Comparison of node counts for PBCHAFF and ZCHAFF on CNF encodings. Each point corresponds to a CNF problem instance. The x coordinate corresponds to the number of nodes expanded for ZCHAFF and the y coordinate corresponds to the number of nodes expanded by PBCHAFF. The line $f(x) = x$ is plotted as a reference. The curve $f(x) = 0.99x$ is the best log transformed linear fit.

of zCHAFF on CNF encodings. This number might be improved if we spent additional time optimizing the PBCHAFF code. Both algorithms have been carefully optimized; however, zCHAFF is a more established algorithm that has gone through a number of revisions. It is likely that PBCHAFF still has some room for improvement. However, we cannot reduce this factor below 1.0 since PBCHAFF is really equivalent to zCHAFF with slightly more complex data structures when run as a resolution based method.

The results seen here are analogous to the proof complexity result of Section 4.1.1 that showed the cutting-plane proof system is a proper generalization of resolution. Similarly, these experiments show that PBCHAFF is a proper generalization of zCHAFF in that PBCHAFF captures the same functionality as zCHAFF with modest performance costs. In the worst case, we can always run PBCHAFF as a resolution-based solver and approximately match the performance of zCHAFF. Later we will see that the reverse is not true; zCHAFF is exponentially worse than PBCHAFF on important classes of problems. Clearly, pseudo-Boolean versions of DPLL style algorithms need not sacrifice the ability to quickly traverse search trees.

5.2 Pigeonhole Problems

In this section, we begin to explore the performance of PBCHAFF on the pigeonhole problem. We saw in Section 5.1.1 that PBCHAFF simulates a resolution based method when applied to CNF constraint sets. While PBCHAFF does perform quite nicely as a resolution-based method, the goal was to improve the strength of the underlying proof system beyond that of resolution.

We start by applying strengthening to CNF encodings of the pigeonhole problem as a preprocessing method. Then we compare the performance of PBCHAFF on the strengthened encoding to the performance of a number of resolution-based methods. In addition, we look at the performance of two other pseudo-Boolean methods. OPBDP is a pseudo-Boolean version of the DPLL method that does not apply learning [3]. The second is a version of PBCHAFF that learns only clausal constraints (simple disjunctions). Like PBCHAFF, both of these methods are given the encoding strengthened by preprocessing.

For all solvers, parameter settings were set to the default values and a timeout was set at 4000 seconds. Experiments were run on 1.8MHz Athlon processors with 256K on-chip cache and 512MB of RAM. Code was compiled with Gnu g++ using full optimization and run under Linux.

5.2.1 Experimental Results

By applying the strengthening method described in Section 4.6, we were able to generate a more concise encoding of the problem from the clausal encoding. In the clausal encoding

$$\begin{aligned} \sum_{k=1}^{n} p_{ik} \geq 1 \quad & i = 1, \ldots, n+1 \\ \overline{p}_{ik} + \overline{p}_{jk} \geq 1 \quad & i \neq j, k = 1, \ldots, n \end{aligned} \tag{5.1}$$

the set of constraints (5.1) is replaced with the constraints

$$\sum_{i=1}^{n+1} \overline{p}_{ik} \geq n \quad k = 1, \ldots, n$$

Table 5.1 shows node counts and execution time for both preprocessing and solution time. Resolution-based methods can solve problems of size 14 in a few days of execution time. The pseudo-Boolean solver PBCHAFF can solve a problem of size 90 in less than an hour.

The strengthened encoding generated by preprocessing the CNF encoding was strong enough to allow polynomial size search trees. A graph of the node count data is shown in Figure 5.3 and has been fitted with a polynomial curve $f(n) = 0.83n^{2.04}$ where n is the number of pigeons. The number of variables is $n^2 + n$, so the number of nodes scales linearly with the number of variables. A similar graph for both preprocessing and solution time is shown in Figure 5.4. Again we see polynomial scaling. Preprocessing scales at about n^6 in terms of the number of pigeons, or n^2 in the size of the CNF encoding. Solution time scales at around n^5 in terms of pigeons and $n^{2.5}$ in terms of the number of variables.

All of the other solvers we tested showed exponential scaling. Performance for all solvers is shown in Figure 5.5. With the exception of PBCHAFF, data points for all

Instance	Node count	Preprocessing (sec)	Solution (sec)
hole10.cnf	89	0	0.01
hole15.cnf	209	0.05	0
hole20.cnf	379	0.23	0.02
hole25.cnf	599	0.77	0.05
hole30.cnf	869	2.13	0.12
hole35.cnf	1189	5.19	0.36
hole40.cnf	1559	11.28	1.6
hole45.cnf	1979	23	2.82
hole50.cnf	2449	43.42	4.74
hole55.cnf	2969	78.11	7.69
hole60.cnf	3539	134.61	11.7
hole65.cnf	4159	252.42	17.38
hole70.cnf	4829	417.32	25.49
hole75.cnf	5549	686.22	35.64
hole80.cnf	6319	1119.01	49.15
hole85.cnf	7139	1922.51	66.55
hole90.cnf	8009	3174.47	88.54

TABLE 5.1: Results for PBCHAFF on pigeonhole problems. The table lists node counts, execution time in seconds for strengthening as a preprocessing method, and execution time in seconds for solution.

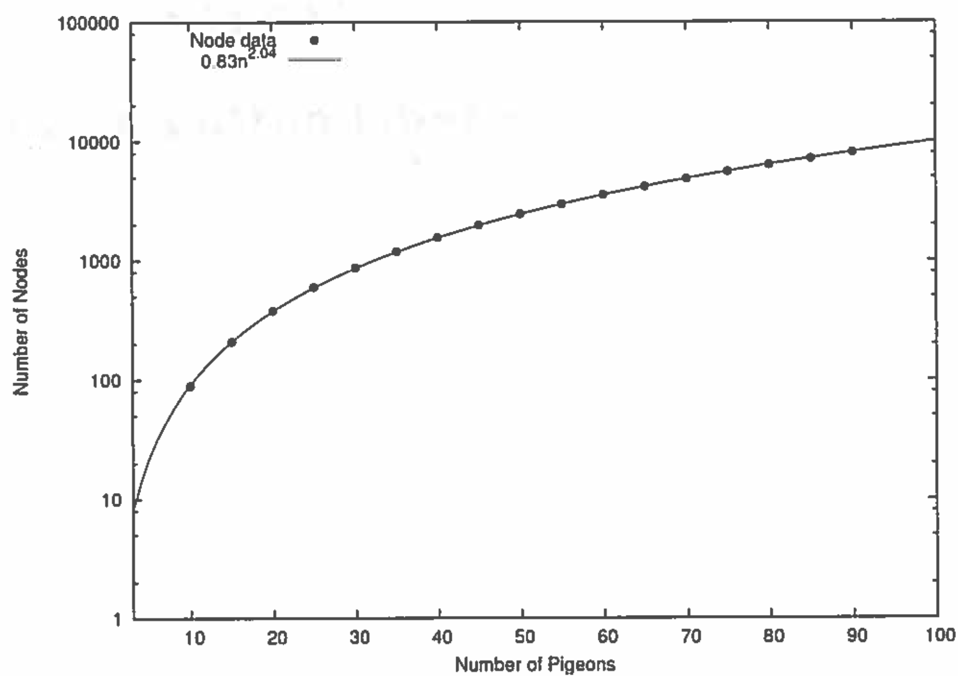


FIGURE 5.3: Size of search trees in terms of node counts for PBCHAFF on the pigeonhole problem. The curve shown, $f(n) = 0.83n^{2.04}$, is the best polynomial fit for the data.

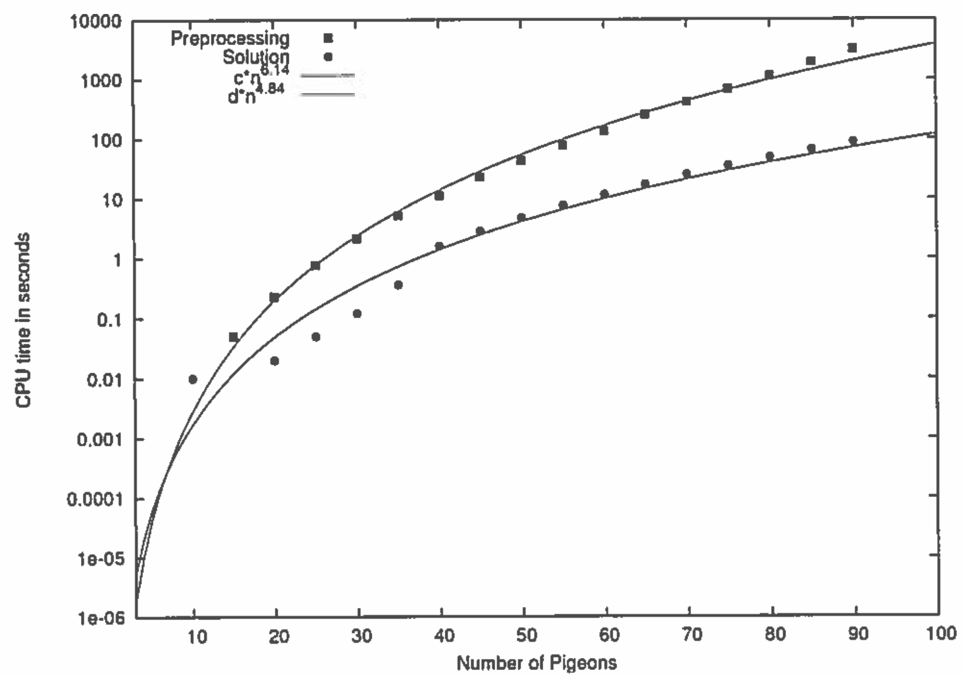


FIGURE 5.4: Preprocessing and solution time for PBCHAFF on the pigeonhole problem. The best polynomial fit for the preprocessing data is the curve $f(n) = cn^{6.14}$, and the curve $f(n) = dn^{4.84}$ is the best polynomial fit for the solution data.

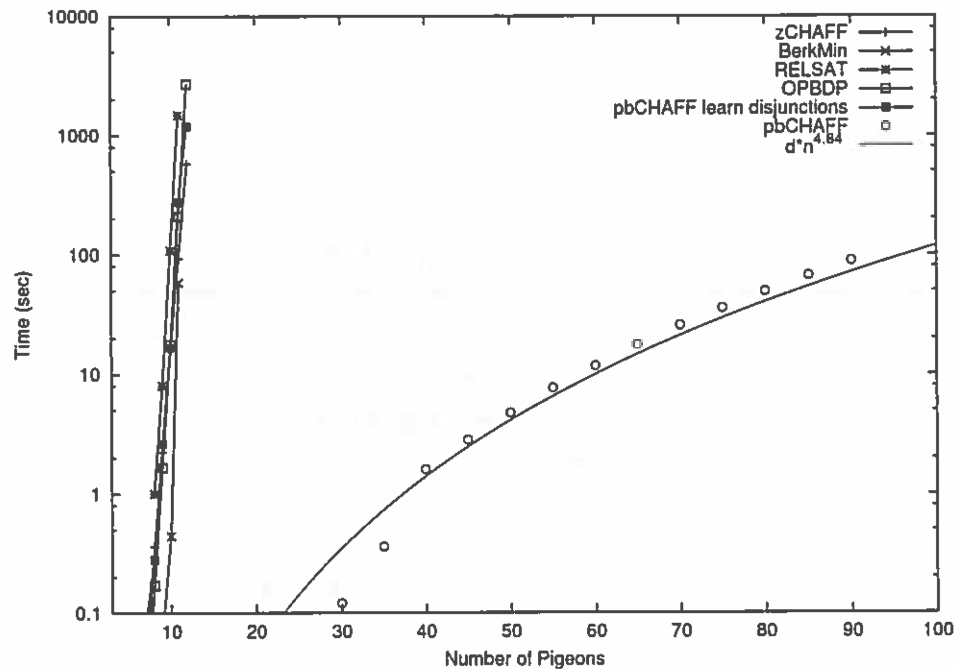


FIGURE 5.5: Comparison of solvers on the pigeonhole problem. Resolution based methods zCHAFF, RELSAT, and BERKMIN show exponential scaling. OPBDP, and the PBCHAFF version that learns only clausal constraints both show exponential scaling. PBCHAFF shows polynomial scaling. The best polynomial fit for the PBCHAFF solution data is the curve $f(n) = dn^{4.84}$.

the solvers are tightly clustered. Figure 5.6 provides a closer look at this cluster. The resolution-based solvers scale as expected, timing out at 12 to 13 pigeons.

Scaling is exponential in the number of pigeons for the pseudo-Boolean solver OPBDP. This also should be expected. Recall that Lemma 4.3.1 showed pseudo-Boolean DPLL without learning is still a resolution-based method. The version of PBCHAFF that learns only clausal constraints also scales exponentially on the pigeonhole problem. Although the solver learns, it lacks parallel inference. Compound constraints cannot be learned, so conflicts must be eliminated one at a time instead of in parallel. Note that both of these methods have the benefit of building inference used in the strengthening preprocessing method, but this is not helpful in reducing the size of proofs in the absence of parallel inference. Strengthening could be applied

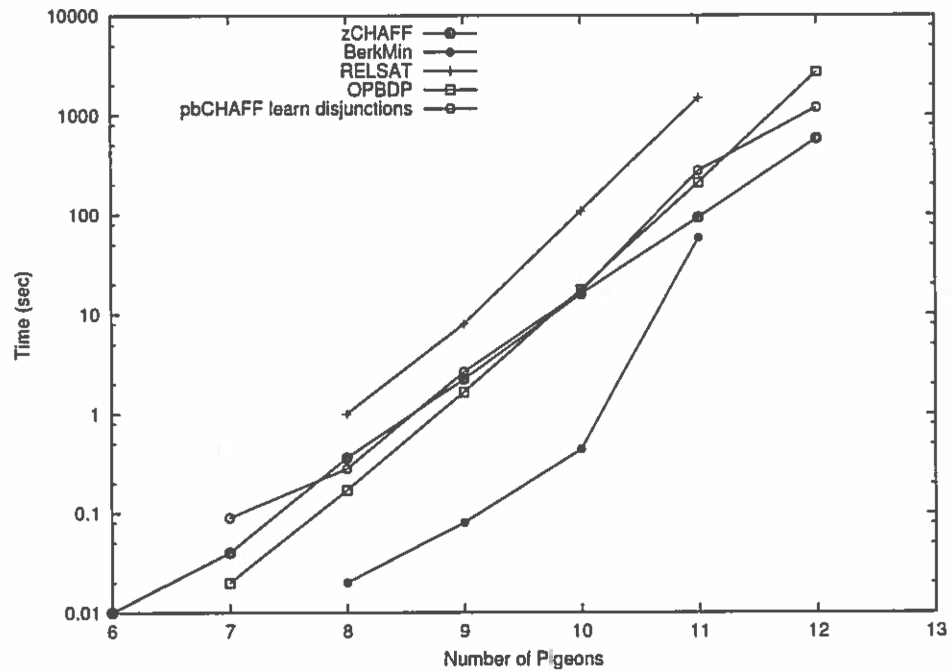


FIGURE 5.6: Comparison of solvers on the pigeonhole problem.

at every node, potentially building more compound constraints. Unfortunately, these constraints also cannot be used to reduce proof length without parallel inference.

These results show that we can automate proof systems stronger than resolution that easily find and efficiently generate short proofs of the pigeonhole problem. Given a weak CNF encoding of the pigeonhole problem, PBCHAFF can construct a stronger pseudo-Boolean encoding in polynomial time. PBCHAFF uses a learning method with parallel inference that allows it an exponential speedup over both resolution-based methods and pseudo-Boolean methods that lack parallel inference. We can conclude that moving beyond resolution requires more than just changing representation. Powerful inference rules must also be incorporated to shorten the length of constructed proofs.

5.3 Pigeonhole Problems Embedded in Planning Problems

Solving the pigeonhole problem is useful for exploring the proof strength of a solver, but in its pure form it is not a particularly interesting real world problem. However, many real world problems are believed to contain embedded pigeonhole problems. In this section we look at solving a pigeonhole problem embedded in a simple logistics problem. We compare the performance of zCHAFF with PBCHAFF when PBCHAFF is allowed a pseudo-Boolean encoding of the problem.

5.3.1 A Simple Logistics Problem

The problem domain we consider here and in the remainder of the thesis involves moving packages between cities using airplanes. We will first describe the problem in first-order logic and then show how it can be encoded as a satisfiability problem. The axioms defined below allow three actions: packages can be loaded onto planes, planes can fly between cities, and packages can be unloaded. The problem concerns objects, locations, planes, and time, and their respective domains O, L, P and T are finite in size. A problem instance is formed by adding an initial state describing the initial locations for planes and packages, and a goal state describing the destination cities for the packages. Then we can pose the question of whether a plan exists that delivers packages from their initial locations to their destination cities within a specified number of time steps.

The first-order encoding requires three predicates:

$$\begin{aligned}
\texttt{objectAt}(O, L, T) &\longleftrightarrow \text{object } O \text{ is in location } L \text{ at time } T \\
\texttt{planeAt}(P, L, T) &\longleftrightarrow \text{plane } P \text{ is in location } L \text{ at time } T \\
\texttt{inPlane}(O, P, T) &\longleftrightarrow \text{object } O \text{ is in plane } P \text{ at time } T
\end{aligned}$$

We will use numbers to denote constants. The context of the number will be sufficient to determine the domain of the constant. For instance, the literal $\texttt{objectAt}(O_3, l, t)$ where O_3 refers to the third element of O will be denoted $\texttt{objectAt}(3, l, t)$.

Frame Axioms:

$$\forall o, l, t. \exists p.\ \texttt{objectAt}(o, l, t) \rightarrow \texttt{objectAt}(o, l, t+1) \vee \texttt{inPlane}(o, p, t+1) \quad (5.2)$$

$$\forall o, p, t. \exists l.\ \texttt{inPlane}(o, p, t) \rightarrow \texttt{inPlane}(o, p, t+1) \vee \texttt{objectAt}(o, l, t+1) \quad (5.3)$$

Frame axioms require objects to maintain their current position unless they are moved. Objects stay at the same location or they are loaded onto planes (5.2), and they stay in the same plane or are unloaded (5.3).

Change of State:

$$\forall o, l, t, p.\ \texttt{objectAt}(o, l, t) \wedge \texttt{inPlane}(o, p, t+1) \rightarrow \texttt{planeAt}(p, l, t) \quad (5.4)$$

$$\forall o, l, t, p.\ \texttt{objectAt}(o, l, t) \wedge \texttt{inPlane}(o, p, t+1) \rightarrow \texttt{planeAt}(p, l, t+1) \quad (5.5)$$

$$\forall o, l, t, p.\ \texttt{inPlane}(o, p, t) \wedge \texttt{objectAt}(o, l, t+1) \rightarrow \texttt{planeAt}(p, l, t) \quad (5.6)$$

$$\forall o, l, t, p.\ \texttt{inPlane}(o, p, t) \wedge \texttt{objectAt}(o, l, t+1) \rightarrow \texttt{planeAt}(p, l, t+1) \quad (5.7)$$

The change of state axioms define how packages are loaded onto and unloaded from planes. If an object is loaded onto a plane, then the plane is required to be at the same location as the object at the time when the object is loaded (5.5), as well as the preceding time step (5.4). Axioms for unloading an object are simply the reverse of loading. A loading or unloading action requires a total of two time-steps for completion.

Consistency of State:

$$\forall p, t, l_1, l_2.\ l_1 \neq l_2\ \texttt{planeAt}(p, l_1, t) \rightarrow \neg\texttt{planeAt}(p, l_2, t) \quad (5.8)$$

$$\forall o, t, l_1, l_2.\ l_1 \neq l_2\ \texttt{objectAt}(o, l_1, t) \rightarrow \neg\texttt{objectAt}(o, l_2, t) \quad (5.9)$$

$$\forall o, t, p_1, p_2.\ p_1 \neq p_2\ \texttt{inPlane}(o, p_1, t) \rightarrow \neg\texttt{inPlane}(o, p_2, t) \quad (5.10)$$

$$\forall o, t, l, p.\ \texttt{objectAt}(o, l, t) \rightarrow \neg\texttt{inPlane}(o, p, t) \quad (5.11)$$

$$\forall o, t. \exists p, l.\ \texttt{objectAt}(o, l, t) \vee \texttt{inPlane}(o, p, t) \quad (5.12)$$

Consistency of state axioms (5.8) and (5.9) require that planes and objects be in only one location at a time. Note that because planes cannot be at more than one location during any given time, it follows that a plane must take at least one time step to fly between cities. Axiom (5.10) requires that objects be in only one plane at a time. Axiom (5.11) keeps objects from being in both a plane and a location at the same time, and axiom (5.12) requires that each object always exists somewhere, either in a plane or at a location. We could also require planes to always be at some location, but these constraints are not actually necessary since planes disappearing can only impede the loading and unloading of objects.

5.3.2 The Logistics Pigeonhole Problem

To embed a pigeonhole problem in our instances, we create problems that have $n+1$ objects and only n planes. We need to do a bit more to force a pigeonhole problem since there is no limit on the number of objects a plane can carry. We could potentially put all $n+1$ objects in one plane. We create $n+1$ locations so we can map each object to a different starting location and give each object a destination city that is one hop over from its initial location. Objects must arrive at their destinations by time step 5. Each of the n planes are mapped initially to first n locations.

Initial State:

$$\texttt{objectAt}(i,i,1) \quad 1 \leq i \leq |O| \tag{5.13}$$

$$\texttt{planeAt}(i,i,1) \quad 1 \leq i \leq |P| \tag{5.14}$$

Goal State:

$$\texttt{objectAt}(i,i+1,5) \quad 1 \leq i \leq |O|-1 \tag{5.15}$$

$$\texttt{objectAt}(|O|,1,5) \tag{5.16}$$

The first n objects have planes immediately available for loading at time step 1. They can be loaded by time step 2, arrive at their destination at time step 3, and be unloaded at their destination by time step 4, well within the 5 time step deadline.

However, object number $n+1$ cannot be delivered on time. The fastest way to move object number $n+1$ without making any other deliveries late, is to have the nth plane pick it up when it delivers the nth object to its destination. The plane arrives at time step 3, so object number $n+1$ can be loaded by time step 4 (loading and unloading can occur in parallel). The plane now takes off for the destination city (back to city number 1) and arrives at time step 5. The object does not arrive officially until it is unloaded at time step 6, missing the deadline. Requiring all objects to arrive by time step 5 creates an embedded pigeonhole problem of size n within the simple logistics problem.

It was natural to consider the objects analogous to pigeons, and the planes analogous to holes, but it is actually the locations that are analogous to the pigeons. The distribution of the objects together with their deadlines force a mapping between planes and locations. Each location requires a plane at the same time (the first part of the pigeonhole problem), and planes can only be in one location at a time (the second part of the pigeonhole problem).

5.3.3 CNF Encoding

We create SAT encodings of our logistics pigeonhole problem for different values of n by grounding the first-order encoding over the finite domain values. For each axiom we generate a ground encoding by expanding the quantifiers. First, existential quantifiers are expanded to disjunctions. A literal with an existentially quantified variable is replaced with a disjunction over the set of literals formed by taking an instance of the literal for every possible assignment of domain values to the existential variables in the literal.

Example 5.3.1. *Given the formula*

$$\forall o. \quad \exists p. \quad \texttt{inPlane}(o, p, 3)$$

that says all objects are in some plane at time step 3, we create an instance of $\texttt{inPlane}(o, p, 3)$ *for every* $p \in \{1, 2, \ldots, |P|\}$ *and form the disjunction*

$$\forall o. \quad \texttt{inPlane}(o, 1, 3) \vee \texttt{inPlane}(o, 2, 3) \vee \cdots \vee \texttt{inPlane}(o, |P|, 3)$$

Once existential quantifiers are removed, universal quantifiers are expanded by generating an instance of the first-order formula for every possible assignment of domain values to variables. Finally, the propositional implications are converted to disjunctions.

Example 5.3.2. *If we consider a pigeonhole plan of size $n = 2$, the ground encoding for axiom (5.2) will require $|O||L||T| = 45$ clauses. Here we have generated the first 9 of those 45.*

$$\neg\texttt{objectAt}(1,1,1) \lor \texttt{objectAt}(1,1,2) \lor \texttt{inPlane}(1,1,2) \lor \texttt{inPlane}(1,2,2)$$
$$\neg\texttt{objectAt}(2,1,1) \lor \texttt{objectAt}(2,1,2) \lor \texttt{inPlane}(2,1,2) \lor \texttt{inPlane}(2,2,2)$$
$$\neg\texttt{objectAt}(3,1,1) \lor \texttt{objectAt}(3,1,2) \lor \texttt{inPlane}(3,1,2) \lor \texttt{inPlane}(3,2,2)$$
$$\neg\texttt{objectAt}(1,2,1) \lor \texttt{objectAt}(1,2,2) \lor \texttt{inPlane}(1,1,2) \lor \texttt{inPlane}(1,2,2)$$
$$\neg\texttt{objectAt}(2,2,1) \lor \texttt{objectAt}(2,2,2) \lor \texttt{inPlane}(2,1,2) \lor \texttt{inPlane}(2,2,2)$$
$$\neg\texttt{objectAt}(3,2,1) \lor \texttt{objectAt}(3,2,2) \lor \texttt{inPlane}(3,1,2) \lor \texttt{inPlane}(3,2,2)$$
$$\neg\texttt{objectAt}(1,3,1) \lor \texttt{objectAt}(1,3,2) \lor \texttt{inPlane}(1,1,2) \lor \texttt{inPlane}(1,2,2)$$
$$\neg\texttt{objectAt}(2,3,1) \lor \texttt{objectAt}(2,3,2) \lor \texttt{inPlane}(2,1,2) \lor \texttt{inPlane}(2,2,2)$$
$$\neg\texttt{objectAt}(3,3,1) \lor \texttt{objectAt}(3,3,2) \lor \texttt{inPlane}(3,1,2) \lor \texttt{inPlane}(3,2,2)$$
$$\vdots$$

5.3.4 Pseudo-Boolean Encoding

The pseudo-Boolean encoding of the logistics pigeonhole problem is very similar to the CNF encoding. For most axioms, we simply ground the first-order formula into CNF and then write each ground clause as a pseudo-Boolean clausal constraint.

Example 5.3.3. *The first ground clause from axiom (5.2)*

$$\neg\texttt{objectAt}(1,1,1) \lor \texttt{objectAt}(1,1,2) \lor \texttt{inPlane}(1,1,2) \lor \texttt{inPlane}(1,2,2)$$

is written as

$$\neg\texttt{objectAt}(1,1,1) + \texttt{objectAt}(1,1,2) + \texttt{inPlane}(1,1,2) + \texttt{inPlane}(1,2,2) \geq 1$$

The only axioms that are encoded with compound constraints are from the consistency of state axioms. We use a pseudo-Boolean encoding for axiom (5.8), stating that a plane can be in only one location at a time, that is more concise than the equivalent CNF encoding.

$$\sum_{l=1}^{|L|} \neg\texttt{planeAt}(p,l,t) \geq |L|-1 \qquad p = 1 \ldots |P|,\ t = 1 \ldots |T| \tag{5.17}$$

We can combine axioms (5.9), (5.10), and (5.11), that together require an object to be in only one place at a time (either a location or a plane) in the set of pseudo-Boolean constraints

$$\sum_{l=1}^{|L|} \neg\texttt{planeAt}(p,l,t) + \sum_{p=1}^{|P|} \neg\texttt{inPlane}(o,p,t) \geq |P|+|L|-1 \qquad p = 1,2,\ldots,|P| \quad t = 1,2,\ldots,|T| \tag{5.18}$$

5.3.5 Experimental Results

We compared the performance of ZCHAFF and PBCHAFF on the pigeonhole logistics problem. ZCHAFF was run on the CNF encodings using default settings. PBCHAFF was run on the pseudo-Boolean encoding using the same parameters. Both solvers were set to time out at 1000 seconds. Table 5.2 gives the number of variables and constraints for problem instances. Experiments were run on 1.53MHz Athlon processors with 256K on-chip cache and 512MB RAM.

The results for the pigeonhole logistics problem are shown in Figure 5.7. Both solvers struggled with the problem, showing what appears to be exponential scaling in the number of objects. This should be expected for solver ZCHAFF due to its resolution based approach. The results for PBCHAFF are more interesting. In Section 5.2, PBCHAFF was able to find short proofs of the explicitly stated pigeonhole problem: however, when the pigeonhole problem is obscured within a planning problem the solver fails to find a short proof, despite the stronger pseudo-Boolean representation.

N	variables	CNF clauses	PB constraints
6	480	5,216	3,246
7	665	8,483	5,188
8	880	12,891	7,786
9	1,125	18,611	11,136
10	1,400	25,814	15,334
11	1,705	34,671	20,476
12	2,040	45,353	26,658
20	5,800	217,029	125,074
30	13,200	744,644	425,214
40	23,600	1,779,659	1,011,754

TABLE 5.2: Size of the logistics pigeonhole problem as a function of the number of objects. The columns give the number of variables for each instance, the number of clauses in the CNF encoding of the instance, and the number of constraints in the pseudo-Boolean encoding of the instance.

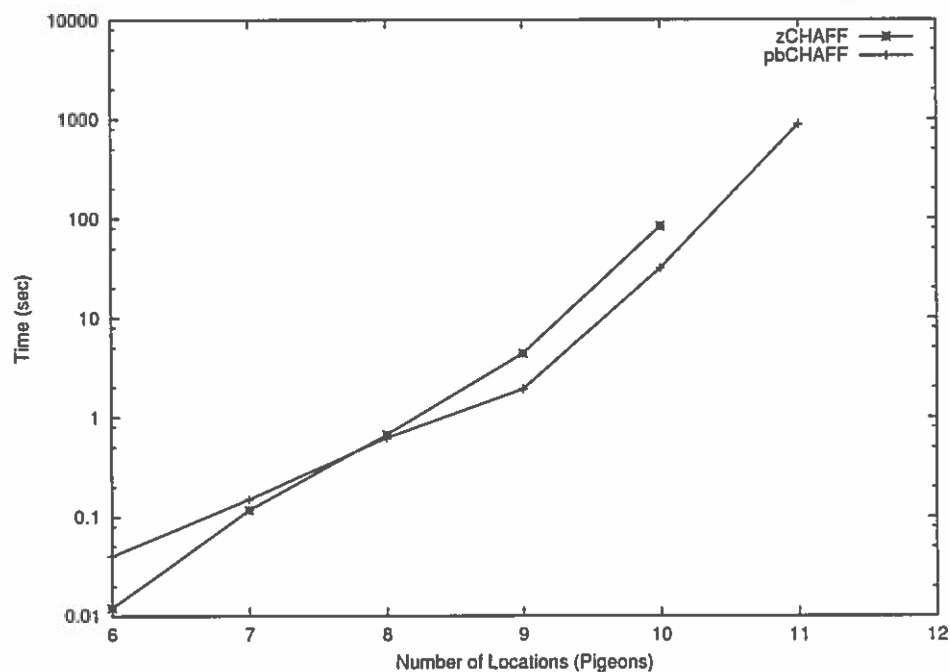


FIGURE 5.7: Comparison of solvers on the logistics pigeonhole problem.

This result prompted a closer look at the pseudo-Boolean encoding of the pigeonhole logistics problem. The consistency of state constraints

$$\sum_{l=1}^{|L|} \neg\texttt{planeAt}(p,l,t) \geq |L|-1 \qquad p = 1\ldots|P|,\ t = 1\ldots|T|$$

require planes to be in only one location at a time. Stating it another way, it allows only one location per plane. For any given time step t, these constraints are equivalent to the pigeonhole constraints

$$\sum_{i=1}^{n+1} \overline{p}_{ik} \geq n \quad k = 1,\ldots,n$$

allowing only one pigeon per hole. The pigeonhole constraints that require every pigeon to get a hole

$$\sum_{k=1}^{n} p_{ik} \geq 1 \quad i = 1,\ldots,n+1 \tag{5.19}$$

are not part of the original problem encoding. They need to be derived from the original constraint set through inference in order to make the embedded pigeonhole problem explicit. The necessary constraint has the form

$$\forall o,l,t.\ \exists p.\ \texttt{objectAt}(o,l,t) \wedge \neg\texttt{objectAt}(o,l,t+2) \longrightarrow \texttt{planeAt}(p,l,t+1) \tag{5.20}$$

The equivalent pseudo-Boolean encoding would be:

$$\neg\texttt{objectAt}(o,l,t) + \texttt{objectAt}(o,l,t+2) + \sum_{p=1}^{|P|} \texttt{planeAt}(p,l,t+1) \geq 1$$

$$o = 1,2,\ldots,|O| \tag{5.21}$$

$$l = 1,2,\ldots,|L|$$

$$t = 1,2,\ldots,|T|$$

If an object is at a location at time t and two time steps later it is gone, then there needs to be a plane at the same location at time $t+1$ to load the object and take it away. Under these conditions a location needs ("gets") a plane. In the pigeonhole logistics problem these conditions are met for all locations at time step 2 (each location

"gets" a plane). Note that packages may be loaded at time step 2 or time step 3 and still make the deadline of time step 5. In either case, a plane is required at time step 2 because a plane must be present at both the time step before and during loading. At time step 2, a subset of the the inequalities (5.21) reduce to constraints of the form (5.19) leading to the obvious contradiction.

The first-order constraint (5.20) can be inferred from the frame axiom (5.2) and change of state axiom (5.4) by first-order resolution. First, we transform axioms (5.2) and (5.4) to normal form. We remove implications by replacing them with disjunctions. The axiom (5.2) becomes

$$\forall o, l, t. \exists p.\ \neg\mathtt{objectAt}(o, l, t) \lor \mathtt{objectAt}(o, l, t+1) \lor \mathtt{inPlane}(o, p, t+1) \quad (5.22)$$

and axiom (5.4) becomes

$$\forall o, l, t, p.\ \neg\mathtt{objectAt}(o, l, t) \lor \neg\mathtt{inPlane}(o, p, t+1) \lor \mathtt{planeAt}(p, l, t) \quad (5.23)$$

Next we remove the existential quantifier from (5.22) by adding a *Skolem function* [65].

$$\forall o, l, t.\ \neg\mathtt{objectAt}(o, l, t) \lor \mathtt{objectAt}(o, l, t+1) \lor \mathtt{inPlane}(o, F(o, l, t), t+1) \quad (5.24)$$

The Skolem function $F(o, l, t)$, when applied to an object, location, and a time step, returns the plane that loads the object.

Now we are ready to resolve (5.24) and (5.23). We begin by determining the most general unifier for atomic sentences $\mathtt{inPlane}(o, F(o, l, t), t+1)$ and $\mathtt{inPlane}(o, p, t+1)$. These are the first-order literals we wish to resolve. A unifier is a substitution that would make both atomic sentences look the same. The most general unifier is the substitution that makes a minimal number of variable bindings. In this case, the most general unifier is $\{p/F(o, l, t)\}$. We now replace all occurrences of variable p in (5.23) with $F(o, l, t)$. We can now resolve the two axioms.

$$\forall o, l, t.\ \neg\mathtt{objectAt}(o, l, t) \lor \mathtt{objectAt}(o, l, t+1) \lor \mathtt{inPlane}(o, F(o, l, t), t+1)$$

$$\forall o, l, t, p.\ \neg\mathtt{objectAt}(o, l, t) \lor \neg\mathtt{inPlane}(o, F(o, l, t), t+1) \lor \mathtt{planeAt}(F(o, l, t), l, t)$$

$$\forall o, l, t.\ \neg\mathtt{objectAt}(o, l, t) \lor \mathtt{objectAt}(o, l, t+1) \lor \mathtt{planeAt}(F(o, l, t), l, t) \quad (5.25)$$

Next we do another resolution, resolving the resulting constraint (5.25) with itself. We unify $\texttt{objectAt}(o,l,t)$ and $\texttt{objectAt}(o,l,t+1)$ by making the substitution $\{t/t+1\}$ in one version of the constraint.

$$\forall o,l,t.\ \neg\texttt{objectAt}(o,l,t) \vee \texttt{objectAt}(o,l,t+1) \vee \texttt{planeAt}(F(o,l,t),l,t)$$
$$\forall o,l,t.\ \neg\texttt{objectAt}(o,l,t+1) \vee \texttt{objectAt}(o,l,t+2) \vee \texttt{planeAt}(F(o,l,t),l,t+1)$$

$$\forall o,l,t.\ \neg\texttt{objectAt}(o,l,t) \vee \texttt{objectAt}(o,l,t+2) \vee \texttt{planeAt}(F(o,l,t),l,t+1) \tag{5.26}$$

Recall that function F returns the plane that loads object o at location l and time t. The resolvent (5.26) is equivalent to the constraint we would produce by removing the existential quantifier from the formula (5.20). The two constraints are therefore logically equivalent.

The embedded pigeonhole problem can be made explicit by two first-order resolution steps. However, PBCHAFF knows nothing about first-order resolution. To make the embedded pigeonhole problem explicit, it must infer all the ground instances of (5.26) with $o = l$ and $t = 2$ by learning. The result seen in Figure 5.7 suggests that PBCHAFF is not able to quickly learn and collect these instances into the constraint database. This is not surprising when we consider that PBCHAFF has no way of preferentially deriving these instances over any of the many other constraints it might learn. For example, given a logistics planning problem of size 10 PBCHAFF learns over 24,000 new constraints. With so many possible constraints to learn, finding the 10 pigeonhole instances that collapse the problem is like searching for needles in a haystack. In fact, the problem is somewhat worse because **every** instance must be found before any benefit occurs. If any one of the instances is missing, the pigeonhole subproblem is satisfiable and a refutation proof is not possible.

A quick way to side-step this problem is to simply add all ground instances of (5.26) to our problem encoding. Certainly, these constraints are valid for all problem instances since they are derived from the basic problem axioms. We then repeated the experiment giving both solvers the new encoding. Results for both experiments are shown in Figure 5.8.

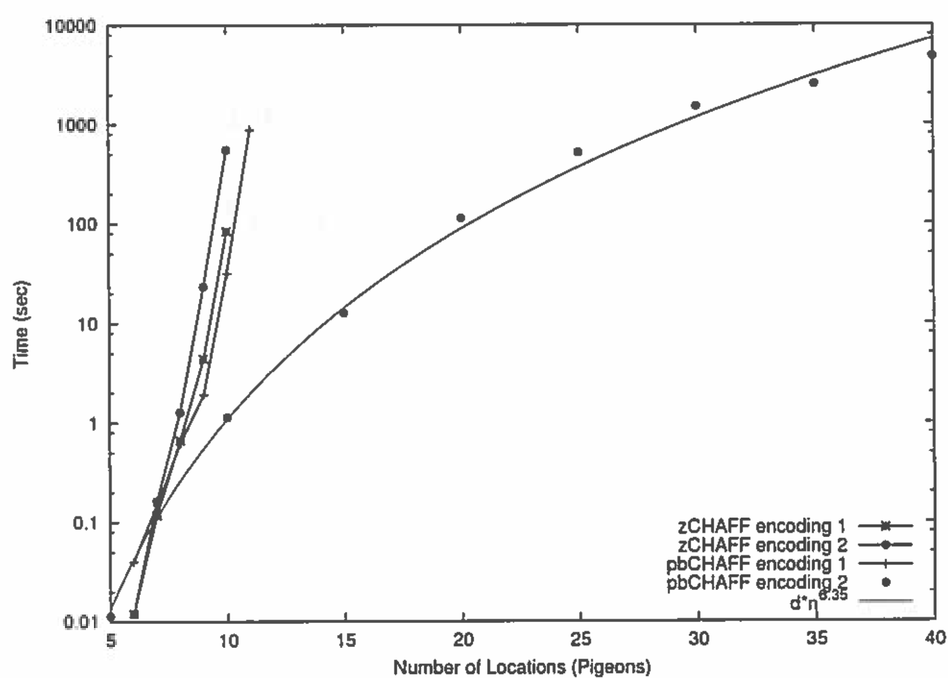


FIGURE 5.8: Comparison of zCHAFF and PBCHAFF on the logistics pigeonhole problem. Encoding 1 is the original encoding described in Section 5.3.1. Encoding 2 is the same encoding with the addition of ground instances of resolvent (5.26).

The new encoding did not improve the performance of zCHAFF over the old encoding. The performance of PBCHAFF was dramatically improved with the new encoding, allowing it to solve a logistics pigeonhole problem of size 40 in 4794 seconds.

5.3.6 Discussion

PBCHAFF was not able to solve the logistics pigeonhole problem efficiently because it lacked the necessary ability to reason about the problem's first-order structure. We were able to sidestep this problem by doing the first-order reasoning by hand and then adding the ground instances of the resolvent to the problem encoding.

This approach is not entirely infeasible. It was not difficult to identify the set of ground instances needed to enhance the encoding. We identified a resource that might become pigeonholed and added axioms that state explicitly the conditions in which the resource is required. If we needed to solve a large number of instances from a single planning domain, the investment of time needed to build and test the enhanced encoding might be warranted. The problem with this approach is that there may be many resources to consider and possibly multiple conditions that require a particular resource. The size of the encodings could grow substantially.

A heuristic approach might also be considered. Altering branching heuristics or relevance policies to favor learning, and retaining constraints that might be part of a pigeonhole problem, could increase the chances of an efficient solution. Unfortunately, as we have seen, the problem is extremely brittle since there is no benefit to finding any proper subset of the pigeonhole constraints. If any one instance is missed, the problem cannot be solved efficiently. A heuristic approach is unlikely to be strong enough to overcome the brittleness of the problem.

The results on the logistics pigeonhole problem are both surprising and interesting. On one hand, pseudo-Boolean representation and inference is ideal for capturing the structure of the pigeonhole problem and allowing an efficient solution. On the other hand, as soon as the problem is embedded in a real-world domain the representational benefits of pseudo-Boolean are easily lost. The embedded pigeonhole problem we consider here is about as simple as it can be and still be meaningful, yet

Domain	Range of sizes
Objects	10-19
Planes	2-12
Locations	10-19
Time steps	5-8

TABLE 5.3: Range of possible values for domain sizes for the randomly generated planning problems.

even this simple problem obscured the pigeonhole problem too much for PBCHAFF to recognize it. It appears that pseudo-Boolean solvers might be both ideal for solving pigeonhole problems and at the same time hopeless at solving any meaningful embedded pigeonhole problem. The solution we use here is not elegant. An ideal solution would capture and reason about the first-order structure of the problem as well as the pigeonhole structure. This suggests that a more general representation and inference system is needed to solve even simple embedded pigeonhole problems.

5.4 Random Planning Problems

In this section we revisit the logistics domain from Section 5.3, only now we consider randomly generated problems. These problems again use the axioms from Section 5.3.1. The domain size of each first-order domain is chosen randomly over a range of possible sizes listed in Table 5.4. Initial locations are selected randomly for objects and planes, and destination locations for objects are also selected randomly. We construct both a CNF encoding and a pseudo-Boolean encoding as described in Section 5.3. In addition to the basic axioms, initial and final conditions, we again augment the pseudo-Boolean encoding with the ground instances of (5.26). We generated 3000 randomly generated logistics problems and compared the performance of ZCHAFF on the CNF encoding with PBCHAFF on the pseudo-Boolean encoding.

Parameter values for both solvers were set to default values. Both solvers were set to time out after 1000 seconds. Experiments were run on 1.53MHz Athlon processors with 256K on-chip cache and 512MB RAM.

5.4.1 Experimental Results

A comparison of PBCHAFF and ZCHAFF on the random logistics problems is shown in Figure 5.9. Each data point represents a problem instance. A point's x coordinate is the execution time for ZCHAFF and the y coordinate is the execution time for PBCHAFF. The graph uses logscaling on both the x and y axes. We fit the data with a power function model by taking a log transformation of both axes and then fitting the transformed data with a linear regression. Again, we used a Deming regression instead of a least squares regression because both the x and y data contain error. We assumed the error associated with both x and y data was the same. Data points were discarded if either of the solvers timed out.

Unlike our earlier comparison of the solvers on CNF encodings, the data here does not have a strong linear trend. As a result, the fitted curve only gives us a rough quantitative measure of how the two solvers compare for this problem domain. There are a significant number of points in the lower right corner of the plot. These points are instances where PBCHAFF dramatically outperforms ZCHAFF. There are no similar points in the upper left hand corner for ZCHAFF. A majority of the data falls below the line $f(x) = x$. The best log transformed linear fit is the curve $f(x) = 0.49x^{0.77}$. For small problems that ZCHAFF solves in around 10 seconds we can expect PBCHAFF to be, on average, faster by a factor of 3.5. For larger problems that ZCHAFF solves in around 1000 seconds we can expect PBCHAFF to be, on average, 10 times faster. The data shows that PBCHAFF has a clear advantage over ZCHAFF on these problems.

Figure 5.10 shows the same graph with only the satisfiable instances. Here we see that on satisfiable instances, performance of the two solvers is fairly close, with ZCHAFF performing slightly better. The best log transformed linear fit to the data is the curve $f(x) = 2.53x^{0.91}$. Figure 5.11 gives a comparison of node counts for satisfiable instances. The best log transformed linear fit to the data is the curve $f(x) = 3.34x^{0.89}$. The data suggests that the size of search trees constructed by PBCHAFF and ZCHAFF are on average about the same for satisfiable instances. These two graphs are very similar to Figure 5.1 and Figure 5.2 from Section 5.1.1. PBCHAFF

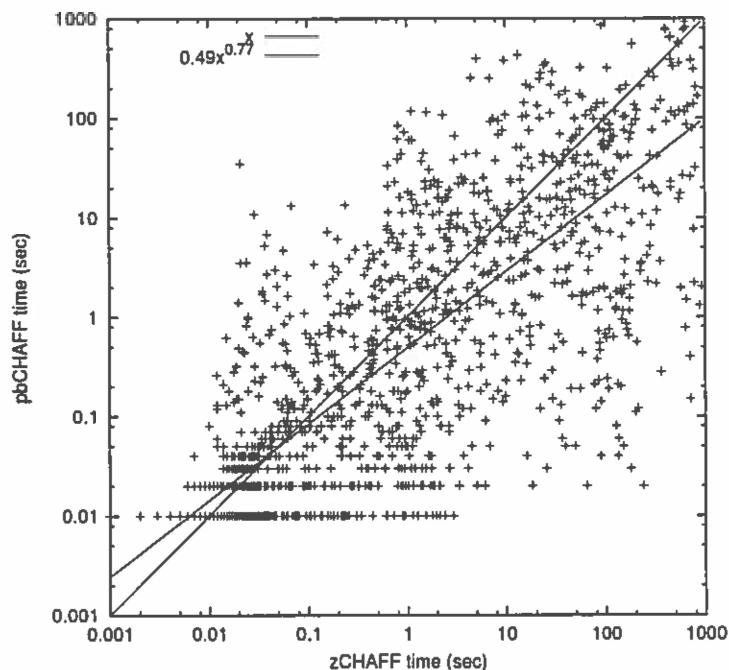


FIGURE 5.9: Comparison of PBCHAFF and ZCHAFF on random planning problems. Each point corresponds to a CNF problem instance. The x coordinate corresponds to the execution time in seconds for ZCHAFF and the y coordinate corresponds to execution time in seconds for PBCHAFF. The points above the line $f(x) = x$ represent instances where ZCHAFF outperformed PBCHAFF. Points below $f(x) = x$ represent instances where PBCHAFF outperformed ZCHAFF. The curve $f(x) = 0.49x^{0.77}$ is the best log transformed linear fit.

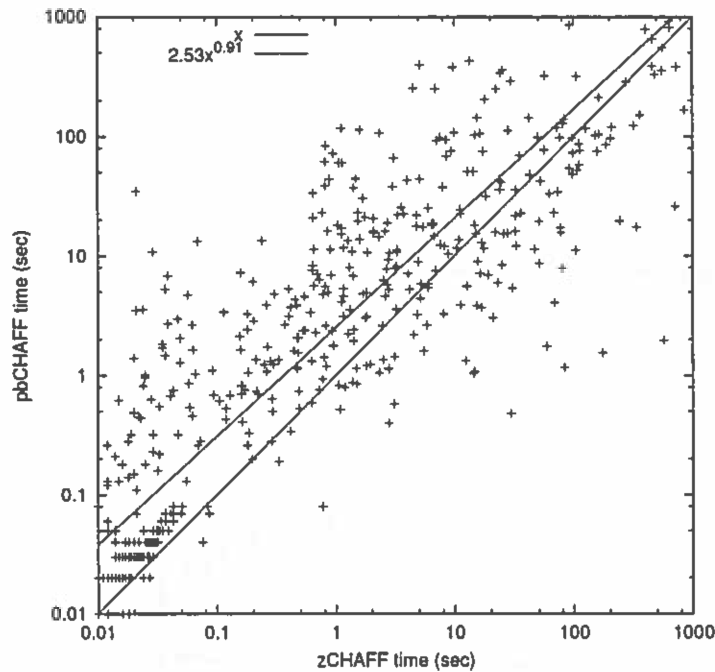


FIGURE 5.10: Comparison of solvers on satisfiable instances of random planning problems. The plot shows execution time in seconds for zCHAFF on the x axis versus PBCHAFF on the y axis. The best log transformed linear fit is the curve $f(x) = 2.53x^{0.91}$. The line $f(x) = x$ is plotted as a reference.

and zCHAFF build search trees of approximately the same size, but PBCHAFF is slightly slower per node.

When we look at the same comparisons for unsatisfiable instances the results are quite dramatic. Figure 5.13 shows a comparison of node counts on unsatisfiable instances and Figure 5.12 shows a comparison of execution time on unsatisfiable instances. In both graphs, almost all of the points are below the line $f(x) = x$. PBCHAFF almost always builds a smaller search tree and solves the problem in less time. The significant number of points in the lower right hand corner show that often PBCHAFF is dramatically better. PBCHAFF is clearly superior to zCHAFF on unsatisfiable instances, and by much more than small linear factor. It is common to find randomly generated problems that require hundreds of seconds for zCHAFF to solve and only a fraction of a second for PBCHAFF.

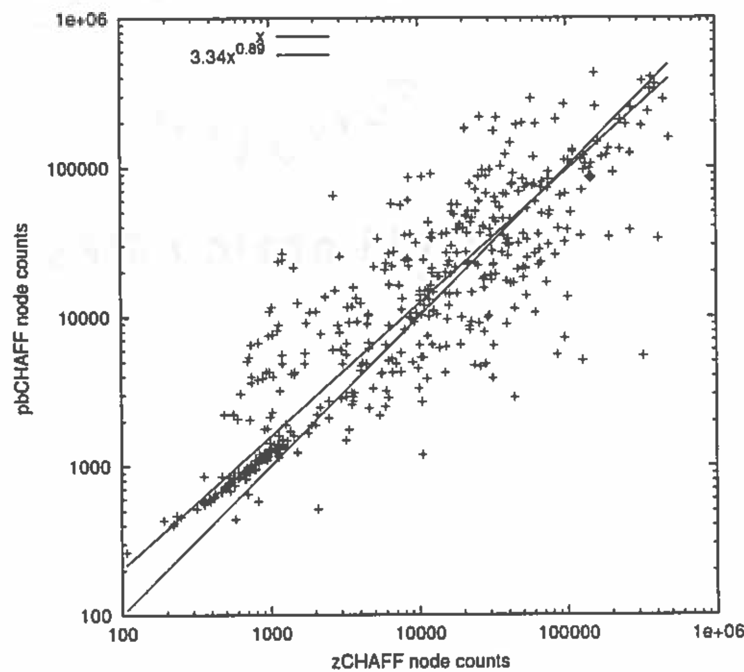


FIGURE 5.11: Comparison of solvers on satisfiable instances of random planning problems. The plot shows the number of nodes expanded by zCHAFF on the x axis versus nodes expanded by PBCHAFF on the y axis. The curve $f(x) = 3.34x^{0.89}$ is the best log transformed linear fit. The line $f(x) = x$ is plotted as a reference.

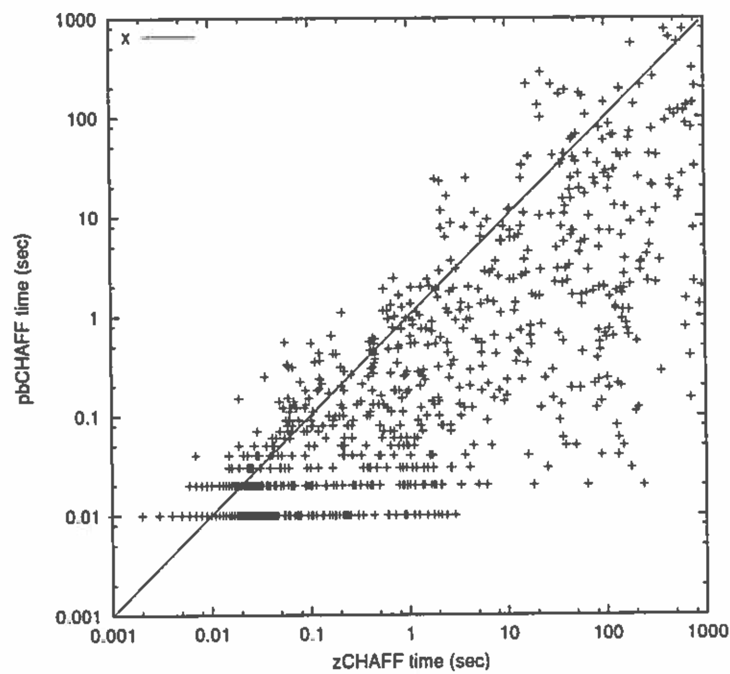


FIGURE 5.12: Comparison of solvers on unsatisfiable instances of random planning problems. The plot shows execution time in seconds for ZCHAFF on the x axis versus execution time for PBCHAFF on the y axis. The line $f(x) = x$ is plotted as a reference.

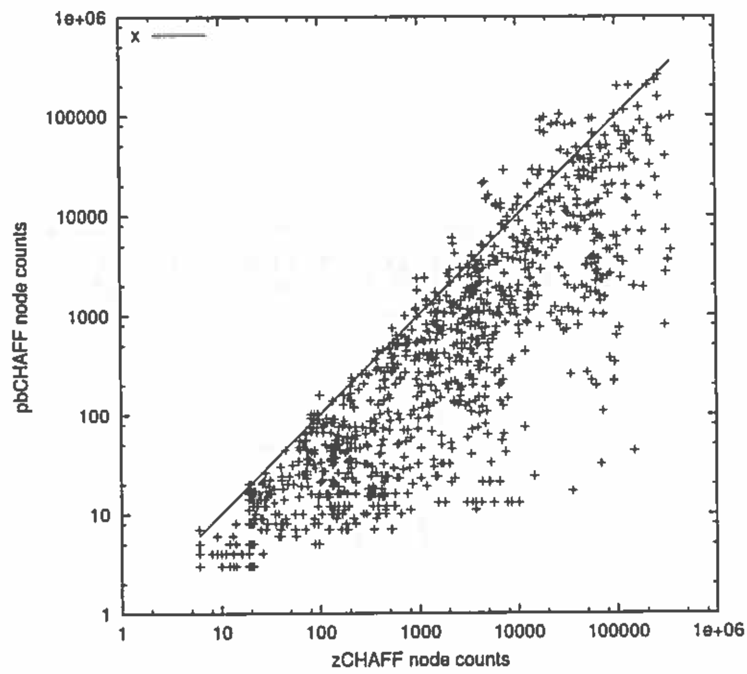


FIGURE 5.13: Comparison of solvers on unsatisfiable instances of random planning problems. The plot shows the number of nodes expanded for ZCHAFF on the x axis versus PBCHAFF on the y axis. The line $f(x) = x$ is plotted as a reference.

It has often been speculated that many problem domains such as planning and scheduling contain embedded pigeonhole problems. These experiments suggest that embedded pigeonhole problems do occur in random planning problems and may often be the cause of unsatisfiability. Further study is needed to determine whether the performance improvements seen here for the pseudo-Boolean methods can truly be attributed to better performance on embedded pigeonhole problems.

Traditional satisfiability solvers continue to provide competitive solutions on planning problems. However, these experiments show that despite their success, they are clearly unnecessarily slow. Lifted solvers that maintain the efficiency of traditional methods and at the same time improve the power of clause learning through stronger inference have the potential to provide highly competitive solutions on problem instances from planning and logistics domains.

CHAPTER 6

Related Work

6.1 Integer Programming Techniques

In this section we present techniques from the field of operations research (OR). Many important OR concepts and techniques have already been presented in Chapter 4. These include pseudo-Boolean representation, the cutting-plane proof system, and the coefficient reduction technique presented in Section 4.6. The methods presented in this chapter are some of the most well known integer programming techniques, and like PBCHAFF, they can be seen as automating some form of the cutting-plane proof system. However, the style of automation has some significant differences from PBCHAFF whose automation style is inspired by artificial intelligence (AI) satisfiability algorithms.

Until recently, methods from the fields of AI and OR differed in both choices of problem representations and algorithmic approaches. This made comparisons between methods difficult. The recent interest in integrated AI/OR techniques has made a new and more interesting set of experiments possible, leading to a deeper understanding of search and logical methods.

The operations research community is concerned with solving the *integer programming* problem. SAT problems are a subclass of *zero-one integer programing* problems. An integer programming problem is an optimization problem in which all variables are restricted to have nonnegative integer values. It can be described by a set of

constraints and an objective function of the form

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to:} \quad & Ax \leq b \\ & x \in Z_+^n \end{aligned}$$

where $A \in \mathcal{R}^{m \times n}$ is an $m \times n$ real-valued array, and $b \in \mathcal{R}^m$ and $c \in \mathcal{R}^n$ are real-valued vectors. The goal is to find a solution that does not violate any constraints and gives the optimal value of the objective function. We will outline the classic branch-and-bound and branch-and-cut methods and discuss how these methods have been applied to solve satisfiability problems.

6.1.1 Branch-and-bound

The classic approach to solving the integer programming problem is branch-and-bound. It is a systematic tree-style backtracking algorithm like the algorithms discussed in Chapter 2. Unlike satisfiability problems, the integer programming problem is an optimization problem. The main pruning techniques used by branch-and-bound are designed to eliminate feasible, but suboptimal solutions. The underlying idea of branch-and-bound is to find a feasible integer solution early in the search process, and use this solution to prune unproductive areas of the search space. Later we will discuss how satisfiability problems can be encoded as integer programming problems, and hence as optimization problems.

Branch-and-bound algorithms, like DPLL-style algorithms, use branch decisions to iteratively partition the search space into a search tree. A node n in the tree corresponds to a series of branch decisions, one for each of the nodes on the path from the root to n. Unlike branch decisions in DPLL-style trees, a branch decision in a branch-and-bound tree may not correspond to assigning a value to a variable. Branch decisions will be discussed in more detail later in this section.

At each node in the search tree, a relaxation of the problem is solved. A relaxation is an easier version of the problem that can generally be solved in polynomial time. The solution set of the relaxed problem is always a superset of the solution set to the original problem. For this reason the solution to the relaxed problem always provides

an upper (assuming a maximization problem) bound for the original problem. A good relaxation provides a good approximation of the true solution.

Within the context of a branch-and-bound tree, the solution to the relaxation can provide two types of bounds. If at any time during the search a relaxation identifies a feasible solution, this solution can be recorded as the *incumbent* solution. The incumbent solution records the best feasible solution encountered so far and is updated each time a better solution is found. Additionally, any solution to the relaxed problem at a given node provides an upper bound for all subproblems generated below that node. Hence, if the solution to the relaxation at a given node is less than the incumbent solution, then the subtree beneath that node can be pruned.

The most common relaxation used for IP problems is the linear relaxation which consists of removing or "relaxing" the integer constraints of the IP problem and then solving the associated linear programming problem (LP). An LP problem can be stated as:

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to:} & Ax \leq b \end{array}$$

where again, $A \in \mathcal{R}^{m \times n}$ is an $m \times n$ real-valued array, and $b \in \mathcal{R}^m$ and $c \in \mathcal{R}^n$ are real-valued vectors. The linear programming problem is solvable in polynomial time by the ellipsoid and interior point methods. The most commonly used simplex method is not a polynomial time algorithm, but is very fast in practice. The solution to the linear relaxation, as we discussed earlier, provides bounds for pruning the search tree.

Each linear inequality $ax \leq b$ in an linear programming problem defines a *half-space* in n-dimensional space. The intersection of the half-spaces is a *polyhedron*, or *polytope* if it is bounded. Solutions to the relaxed problem are points in this polytope, while integer solutions are a (possibly empty) subset of the polytope. The optimal solution to the relaxed problem will always be an *extreme* (*corner*) point of the polytope. Algorithms for solving the linear relaxation systematically search among the extreme points of the polytope for the optimal solution. There are many good references that discuss linear programming techniques [16, 56].

We return now to our discussion of the branch-and-bound algorithm. The description we give here might more accurately be referred to as a family of algorithms or an algorithm framework. This is because branch-and-bound allows for a wide variety of methods for partitioning subproblems, and subproblem selection. A high-level description of this framework follows.

1. *Initialization.* The original integer problem is added to the list of subproblems L to be solved. There is no incumbent solution.

2. *Termination.* If the list of subproblems L is empty, the algorithm terminates. The current incumbent solution is the optimal solution. If no incumbent solution has been found, the problem is infeasible.

3. *Problem Selection and Relaxation.* A subproblem is selected from the list L and a linear programming relaxation is run to find the optimal solution x^* to the continuous problem.

4. *Pruning and Fathoming.*

    - If the value of the objective function on x^* is less than or equal to the bound provided by the incumbent solution or there is no x^* because the problem is infeasible, then this subproblem can be pruned or *fathomed.* Go to step 2.

    - If the value of the objective function of the relaxed problem is greater than the minimum bound provided by the incumbent solution and the solution x^* is integer, then it becomes the new incumbent solution. The list of subproblems L is scanned for any problems that have objective function values less than the new incumbent. These problems are removed from the list. Go to step 2.

    - If the value of the objective function of the relaxed problem is better than that of the incumbent but is fractional, the current problem is partitioned into subproblems. These problems are added to the list L.

The framework of branch-and-bound is highly flexible; however, some methods are common enough to be considered standard. The most common subproblem partitioning method is to choose a variable in the relaxed solution with a fractional value. Two new subproblems can be generated by adding constraints that eliminate this fractional solution. For instance, if the variable x_i has a value of $\frac{1}{3}$ in the relaxed solution, then the problem can be partitioned into two new subproblems, one with the constraint $x_i \leq 0$ and one with $x_i \geq 1$, since clearly the range $0 < x_i < 1$ cannot contain any feasible solutions. The most common subproblem selection method is depth-first search. We will assume both of these methods from here on.

6.1.2 Branch-and-cut

Branch-and-cut is a variant of branch-and-bound with the additional ability to generate cutting-planes at each node. Given an integer programming problem, a *cut* or *cutting plane* is any linear inequality that is satisfied by the integral solutions of the problem. Within the context of a branch-and-cut algorithm, we are interested in generating a specific kind of cut called a *separating cut*. A separating cut is a cut that is satisfied by all integral solutions, but unsatisfied by the solution to the linear relaxation of the problem.

Example 6.1.1.

$$\begin{array}{llll} \max & x_1 & +x_2 & \\ \text{s.t.} & -x_1 & +2x_2 & \leq 4 \\ & 3x_1 & +x_2 & \leq 6 \\ & x_i \in Z_+^n & & \end{array}$$

Figure 6.1 gives a graphical representation of this problem. The solution to the linear relaxation $(x_1, x_2) = (\frac{8}{7}, \frac{18}{7})$ gives the optimal continuous solution to the set of inequalities. The solution is fractional and not a feasible integer solution. The cutting-planes

$$x_2 \leq 2$$

$$x_1 + x_2 \leq 3$$

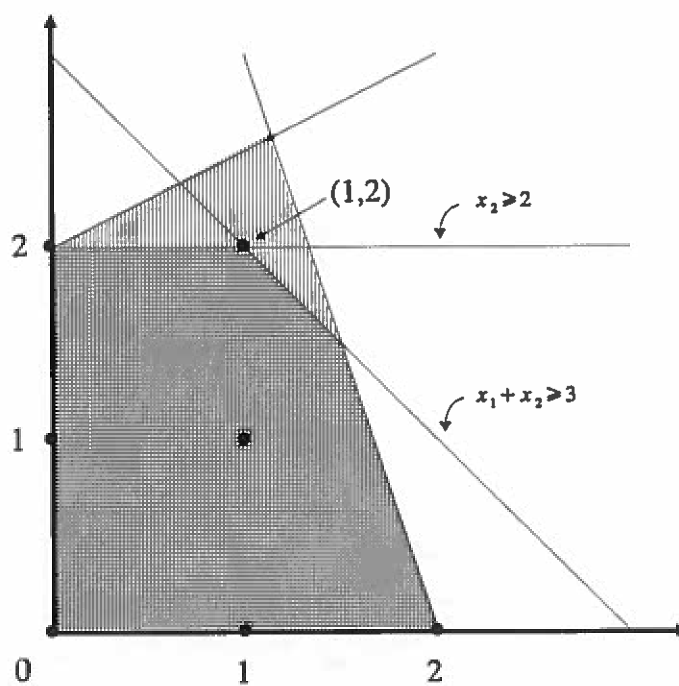


FIGURE 6.1: The non-integer solution $(\frac{8}{7}, \frac{18}{7})$ is eliminated by cutting-plane $x_2 \leq 2$ and $x_1 + x_2 \leq 3$.

are examples of separating cuts. They both eliminate the fractional solution $(x_1, x_2) = (\frac{8}{7}, \frac{18}{7})$ while satisfying the integral solutions of the problem. If these cuts are added to the linear relaxation, a new extreme point $(1, 2)$ is created giving an integer optimal solution.

Branch-and-cut proceeds as the branch-and-bound algorithm. A linear relaxation is solved at each node, but if the solution is fractional, there is the option of generating and adding separating cuts to the constraint set. The addition of separating cuts improves the quality of the linear relaxation by removing fractional extreme points.

Separating cuts can be seen as analogous to AI nogoods. Both are ways of inferring new constraints that help prune the search space. Both in their own way address an identified inconsistency. A nogood identifies a subset of the current partial assignment that is inconsistent and eliminates all assignments containing that subset. A separating cut eliminates an assignment that violates integer constraints. In the AI setting, the addition of learned or inferred constraints enables propagation to eliminate more assignments. Similarly, in the IP setting, cuts tighten the linear relaxation. The resulting relaxation approximates the integer programming problem more closely and can provide better direction for the search process.

There are many methods for generating cuts, and a full description of all methods is beyond the scope of this work. The most common general-purpose cut method is the Chvátal-Gomory cut. This method of cut generation was previously discussed in Chapter 4 in the context of the cutting-plane proof system. Given a set of linear inequalities, new implied constraints, or cuts are generated by taking linear combinations over the set of constraints and then applying integer rounding. Recall the separating cuts from our earlier example. If we take the following linear combination over the original problem constraints

$$\begin{array}{cccc} (3) & -x_1 & +2x_2 & \leq 4 \\ (1) & 3x_1 & +x_2 & \leq 6 \\ \hline \\ & & x_2 & \leq \frac{18}{7} \end{array}$$

and round down the fraction $\frac{18}{7}$, we generate the cut $x_2 \leq 2$. Note that here we are

dealing with $\leq$ constraints as opposed to the $\geq$ constraints used in Chapter 4. As a result, the right hand side of the generated inequality is rounded down instead of up. If we take the linear combination

$$\begin{array}{rrrl} (2) & -x_1 & +2x_2 & \leq 4 \\ (3) & 3x_1 & +x_2 & \leq 6 \\ \hline \\ & x_1 & +x_2 & \leq \frac{26}{7} \end{array}$$

and we round down the fraction $\frac{26}{7}$, we generate the cut $x_1 + x_2 \leq 3$. The result of the most recent linear relaxation can be used to suggest which constraints should be used to generate a cut. Variables assigned fractional values by the linear relaxation have corresponding constraints in the simplex tableau, and these are constraints are used to generate cuts.

The side effects of generating numerous cuts are analogous to generating numerous nogoods. Adding many constraints to the linear relaxation can dramatically reduce the efficiency of the procedure. Branch-and-cut methods maintain a set of constraints called the cut pool. Cut pools were first suggested by Padberg and Rinaldi [57]. Generated cuts are added to the linear relaxation and also stored for later use in the cut pool. At a given node, the cut pool can be searched for separating cuts. Reusing previously generated cuts can avoid a call to an expensive cut generating method. However, if the size of the cut pool becomes too large, the cost of searching the cut pool for separating cuts becomes more expensive.

One way of removing cuts from the cut pool is to maintain a count of the number of times the cut has been checked for violation since the last time it was actually found to be violated [59]. This metric is called the number of *touches* and provides a simple measure of "effectiveness" or usefulness of a cut. Cuts with a low number of touches are preferred over those with a higher number of touches. Systems may also allow the user to define their own metric to determine which constraints to keep and which to discard.

6.1.3 Solving Satisfiability Problems

A satisfiability problem can be encoded as an integer programming problem by introducing an *artificial* variable. Consider the small CNF clause set:

$$\begin{gathered} x_1 \vee x_2 \vee x_3 \\ x_1 \vee x_3 \vee \bar{x}_4 \\ x_2 \vee x_4 \vee \bar{x}_5 \\ \bar{x}_1 \vee \bar{x}_2 \vee x_5 \end{gathered} \tag{6.1}$$

It can be formulated as the integer programming problem:

$$\begin{array}{lllllllll} \min & x_0 & & & & & & & \\ \text{s.t.} & x_0 & +x_1 & +x_2 & +x_3 & & & \geq & 1 \\ & x_0 & +x_1 & & +x_3 & -x_4 & & \geq & 0 \\ & x_0 & & +x_2 & & +x_4 & -x_5 & \geq & 0 \\ & x_0 & -x_1 & -x_2 & & & +x_5 & \geq & -1 \\ & \multicolumn{4}{l}{x_j \in \{0,1\},} & \multicolumn{2}{l}{j = 0, \ldots, 5} & & \end{array}$$

First, each clause in (6.1) is translated to an equivalent pseudo-Boolean inequality. The artificial variable x_0 is then added to each clause. The problem is trivially satisfiable by setting $x_0 = 1$. This provides a feasible starting solution. The CNF clause set (6.1) is satisfiable if and only if the above integer programming problem has a feasible solution with $x_0 = 0$.

The pruning method based on the incumbent solution described earlier has no value in this context. Consider the trivial solution of setting $x_0 = 1$. The solution to the linear relaxation will never produce a value greater than 1 for x_0 so the trivial solution does not lead to pruning. An integer solution with $x_0 = 0$ is a valid solution to the satisfiability problem and no further pruning is necessary.

Branch-and-bound was first applied to satisfiability problems by Blair, Jeroslow, and Lowe [11]. An interesting result shows that when the linear relaxation is applied to an integer programming formulation of a satisfiability problem, the results are equivalent to applying unit propagation.

Theorem 6.1.2. [11] *The variables of a satisfiability problem T that are forced to truth values by unit propagation are fixed at the corresponding binary values by the linear relaxation. Any variable in T left unvalued by unit propagation is assigned a fractional value by the linear relaxation, and the linear relaxation is inconsistent if and only if unit propagation results in a contradiction.*

A consequence of Theorem 6.1.2 is that branch-and-bound and DPLL build identical search trees on satisfiability problems. An exception to this rule occurs when the linear relaxation turns up an integer solution by chance. The search trees generated by DPLL and branch-and-bound may have identical structure, however, DPLL provides far more efficient solutions. This is because the linear relaxation is significantly slower than unit propagation implementations [11, 47]. For this reason, branch-and-bound does not provide competitive solutions on satisfiability problems.

There has been significant work applying cutting-plane methods to solving satisfiability problems. Much of this work focuses on using clausal resolution as a way to generate cuts for cutting-plane methods [41, 42, 43]. This has led to branch-and-cut methods for satisfiability problems [44].

A primary goal to the branch-and-cut method of Hooker and Fedjki was to address the slowness of the linear relaxation for solving satisfiability problems [44]. Their method improves over the branch-and-bound method by adding cutting-plane to reduce the search space and replacing the linear relaxation in part by unit propagation. At any given search node, the unit propagation procedure is applied first. If a solution is found, the algorithm terminates. If a contradiction is encountered, the algorithm backtracks. Otherwise the linear relaxation of the problem is solved, and separating cuts are found and added to the constraint set. The cuts generated are always clausal inequalities and the separation algorithm is driven by clausal resolution. Separating cuts are not generated below a given depth in the search tree because the benefit of the cuts is not enough to offset the cost of the separation algorithm.

The branch-and-bound and branch-and-cut approaches to satisfiability problems we have presented here all use pseudo-Boolean representation and integer programming techniques. However, they don't improve over the traditional satisfiability meth-

ods presented in Chapter 2. This is not surprising if we consider that these methods take CNF encodings and translate them into clausal inequalities. On such encodings, the linear relaxation is equivalent to unit propagation, and the cut generating methods used are equivalent to clausal resolution. These methods are resolution-based, just like the traditional satisfiability algorithms. These studies shed light on the connections between IP and AI methods, but they do little to address the current challenges of satisfiability research.

6.1.4 Summary

Our review of integer programming methods for satisfiability problems is somewhat misleading. Operations research attempts to solve satisfiability problems fail to improve over the traditional CNF-based satisfiability solvers of Chapter 2 despite the increased representational power of pseudo-Boolean constraints. Almost all of the methods presented Section 6.1.3 would fail to solve a simple pigeonhole problem. It would be incorrect to conclude that integer programming techniques have little to offer for the solution of satisfiability problems. Consider that a pigeonhole problem is easily solved by a single linear relaxation if a stronger non-clausal encoding is used. General integer programming techniques are good at many other things including many 0-1 optimization problems like traveling salesman problems.

Integer programming methods have some important commonalities with DPLL-style solvers in their approaches. There is substantial literature discussing connections between AI and OR; we only intend to make a few general remarks. Branch-and-bound and branch-and-cut algorithms are tree based branching algorithms like DPLL-style methods. Both approaches infer new constraints when an infeasible region of the search space is identified to prune the infeasible region. An important connection between the approaches was discovered by Jeroslow when he showed that the result of applying the linear relaxation to an IP encoding of a satisfiability problem is equivalent to unit propagation [11].

Despite these similarities, integer programming methods and DPLL-style solvers have fundamentally different frameworks. In general, the concept of a relaxation is

very different from that of propagation. Propagation is a way to extend a partial solution by identifying assignments that are forced by the partial solution together with problem constraints. Unlike propagation, the linear relaxation does not identify forced variables. Variables assigned integer values by the linear relaxation at a search node may be assigned different values by the linear relaxation of a child node. A relaxation disregards some problem constraints and searches a superset of the solution space, finding solutions that may be infeasible, but hopefully are close to the optimal feasible solution. Outside of the result of Jeroslow, comparing the linear relaxation with AI propagation methods is a bit like comparing apples and oranges.

Running a linear relaxation at each search node is significantly more expensive than unit propagation. The cost of expanding a node in a branch-and-cut algorithm is much higher than the cost of expanding a DPLL search node, but the search trees tend to be much smaller than DPLL-style trees. The cost of a linear relaxation is also a function of the number of constraints in the relaxation, so similar to learning in DPLL-style algorithms, the addition of a large number of cutting-plane to the relaxation has a cost. For this reason, separation algorithms try to generate the strongest cut possible. Separation algorithms tend to be more complicated than the simple nogood learning schemes employed by satisfiability solvers. We noted in Section 2.3.3 the tight relationship between learning and propagation. The extremely fast propagation of traditional satisfiability methods allows them to be less particular about which clauses are learned and allows them to maintain a very large database of learned clauses.

PBCHAFF closely follows the DPLL search framework of fast propagation, quick traversals of large search trees, and large learned constraint databases. Although PBCHAFF shares a representation with integer programming techniques, the methods differ in their basic framework. Integer programming methods and traditional satisfiability solvers differ in representation, inference, and search framework, making comparisons of the methods difficult. A comparison of integer programming methods with PBCHAFF would be of particular interest because the differences are reduced to their frameworks. This opens the possibility of learning something fundamental about the frameworks themselves.

6.2 Lifted Solvers

6.2.1 Pseudo-Boolean

Since the publication of our work on pseudo-Boolean learning [24, 25] a number of other pseudo-Boolean DPLL-style solvers have emerged. The satisfiability solvers SATIRE [74] and PBS [2] are both DPLL-style solvers that allow pseudo-Boolean constraints in addition to clauses. These solvers differ from PBCHAFF mainly in their learning methods. Both solvers lack parallel inference in their learning methods, learning only simple clausal constraints. They therefore do not improve the strength of the underlying proof system beyond that of resolution.

The work of Chai [12] builds on one of our papers [25]. Chai describes a different method for ensuring that the pseudo-Boolean resolvent captures the conflict at hand. It uses the value of `possible` in each parent constraint to determine if one of the parents needs to be reduced before the pseudo-Boolean resolution is performed. Given constraints c_1 and c_2 that resolve and a partial assignment P that values all the literals in both c_1 and c_2, the value of $\texttt{possible}(c, P)$ for the pseudo-Boolean resolvent c is equal to the sum

$$\texttt{possible}(c_1', P) + \texttt{possible}(c_2', P)$$

where c_1' and c_2' are the parent constraints after they have been adjusted by linear multipliers.

Example 6.2.1. *Given the partial assignment $P = \{\bar{b}, \bar{e}, a\}$ and constraints*

$$3a + b + c + d \geq 3$$
$$\bar{a} + e + f + g \geq 3$$

The value of $\texttt{possible}(c, P)$ *in the pseudo-Boolean resolvent c is the sum of the values*

of possible(c'_i, P) *in the adjusted parent constraints.*

possible(c'_i, P)			
$+2$	$3a + b + c + d$	$\geq$	3
-3	$3(\bar{a} + e + f + g)$	$\geq$	9
-1	$3e + 3f + 3g + b + c + d$	$\geq$	9

To ensure that a pseudo-Boolean resolvent is asserting, we simply reduce the parent constraints c_1 and c_2 until

$$m_1\texttt{possible}(c_1, P) + m_2\texttt{possible}(c_2, P) < 0$$

where m_1 and m_2 are the multipliers being used in the linear combination for c_1 and c_2 respectively. Constraints can be reduced by removing unvalued literals and then simplifying the constraint, or by reducing to a cardinality constraint by a method similar to the one we presented in Section 4.3.2. Chai goes on to show that a reduction to cardinality constraints outperforms the reduction to pseudo-Boolean constraints described above. This is because the additional overhead of managing the coefficients is greater than the benefit derived from the stronger representation.

Work by Hooker [40] generalizes the resolution procedure [60] for pseudo-Boolean constraints. This procedure is an inference-based procedure rather than a tree-style search algorithm like DPLL. It can also be viewed as a pure cutting-plane method. It uses pseudo-Boolean resolution together with a type of building inference called a diagonal sum. Work by Walser [71] adapts the local search method WSAT to use pseudo-Boolean constraints.

6.2.2 Parity Constraints

A parity constraint has the form

$$(x_1 + x_2 + \cdots + x_n)\texttt{mod}2 = 1$$

Parity constraints are sometimes referred to as XOR constraints and often written as

$$x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

A number of problem domains, including cryptanalysis and model checking, have constraints that can most naturally be expressed as parity constraints. Problems containing parity constraints can be translated into CNF and solved by traditional satisfiability solvers. However, satisfiability solvers perform poorly on such encodings. There are problems consisting entirely of parity constraints that have exponential proof complexity in resolution, and exponential time complexity for CNF based solvers. An example is the set of charged graph problems defined by Tseitin and shown to have exponentially sized resolution proofs [68]. These problems are sometimes referred to as Urquhart problems. Tseitin problems can be expressed with parity constraints and solved in quadratic time via a form of Gaussian elimination [62]. The translation to CNF clearly can make problems unnecessarily hard. Unfortunately, a pure parity representation is functionally incomplete. There are Boolean functions that cannot be expressed solely by parity constraints. More challenging problems arise when both parity and disjunctive clauses are present. In such cases an integrated approach is required if we are to avoid the unnecessary performance hit of a full translation to CNF.

Inference Rules for parity constraints

The methods we describe make use of parity inference rules. The first rule deals with manipulation of negation. Given a constraint with a negation, that negation can be transferred to any other literal in the constraint.

$$\begin{aligned}
\neg(x_1 \oplus x_2 \oplus \cdots \oplus x_n) &\equiv \overline{x}_1 \oplus x_2 \oplus \cdots \oplus x_n \\
&\equiv x_1 \oplus \overline{x}_2 \oplus \cdots \oplus x_n \\
&\ \ \vdots \\
&\equiv x_1 \oplus x_2 \oplus \cdots \oplus \overline{x}_n
\end{aligned}$$

There is a simplification or factoring rule.

$$\frac{l \oplus l \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n}{x_1 \oplus x_2 \oplus \cdots \oplus x_n}$$

We can define a resolution-like inference.

$$\frac{\begin{array}{c} l \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_n \\ \bar{l} \oplus y_1 \oplus y_2 \oplus \cdots \oplus y_n \end{array}}{x_1 \oplus x_2 \oplus \cdots \oplus x_n \oplus y_1 \oplus y_2 \oplus \cdots \oplus y_n}$$

We have presented this rule in a way that looks similar to resolution, however it has a number of additional important properties we now point out. Unlike resolution in CNF, here we place no restrictions on the literals $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$. Additionally, we note that if the literal l appears in the same phase in both constraints, resolution can still be applied after first applying the negation rule. A third important aspect of this inference rule is that one of the premises can be discarded at this point. If need be, it can be regenerated from the remaining premise and the resolvent. The combination of these properties allows us to use this inference rule in a form of Gaussian elimination for parity constraints. Given a parity constraint that contains a literal of the variable v in either phase, we can use that constraint to remove all literals of v from the remaining parity constraint set. This is similar to a Gaussian elimination row operation which removes a variable from a given equation.

Parity reasoning has been integrated into the DPLL algorithm [4, 49]. Such algorithms must also support a CNF clause set since parity constraints alone are not functionally complete. The earliest work that took this approach used a Gaussian elimination type procedure to preprocess the parity constraints and then used a traditional method to solve the reduced problem [73]. This works well when parity constraints dominate the constraint set. Unfortunately, variables that also occur in the CNF constraint set cannot be removed from the parity constraint set. This limits the preprocessing effectiveness for problems with significant CNF components.

In recent work a more integrated approach is taken [49]. Separate parity and CNF constraint sets are maintained, but propagation is applied to both during search. In this way propagations derived from one set can affect the propagation process of the other set. Propagation for parity constraints is generally some form of Gaussian elimination. In addition to identifying unit literals, these procedures also identify binary parity constraints. Such clauses assert the equivalence of two literals. Consider

the constraint $a \oplus \bar{b}$, which tells us that a is equal to b. When this type of equivalence is discovered, a substitution process is applied eliminating one of the variables from the constraint set. Then, both parity and CNF constraint sets can be simplified, and the size of the problem is reduced.

An integrated DPLL approach that adds parity reasoning dramatically outperforms traditional satisfiability solvers on the DIMACS 32-bit parity problem and certain types of bounded model checking problems [49]. On other problems tested, it appears to do no worse that the equivalent DPLL algorithm without parity reasoning. This implies that the overhead of adding parity functionality is minimal.

6.2.3 Zero Suppressed Binary Decision Diagrams

Zero suppressed binary decision diagrams (ZBDD) [53] are a promising propositional representation for automation. ZBDDs, like traditional BDDs, are directed acyclic graphs (DAGs) with two sink nodes and a set of internal nodes. The sink nodes are labeled the 1-sink and a 0-sink. Each internal node has two outgoing edges: a 0-edge and a 1-edge. Traditional BDDs are used to represent Boolean formulas. Internal nodes are labeled with Boolean variables and a path from source to the 1-sink represents a model of the Boolean function. ZBDDs are designed to represent sets of combinations. When used to represent sets of clauses, the internal nodes are labeled with literals and each path from the source node to the 1-sink node represents a clause. The clause contains the set of literals whose outgoing 1-edge is in the path. An example of ZBDD for the set of clauses

$$\begin{array}{cc} a \vee b & b \vee c \\ a \vee c & b \vee d \\ a \vee d & c \vee d \end{array} \tag{6.2}$$

is shown in Figure 6.2.

A resolution inference step was defined for ZBDDs and was shown to have a high degree of parallel inference [13]. This inference step was then incorporated into a Davis-Putnam (DP) style algorithm called ZRES [14]. The DP algorithm has been

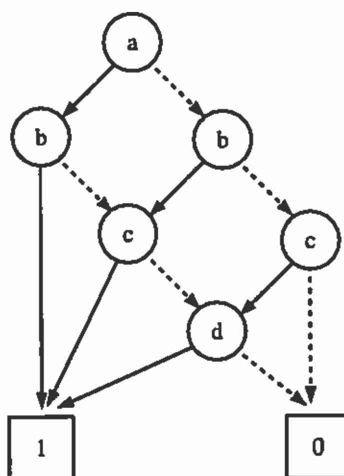


FIGURE 6.2: ZBDD for set of clauses (6.2). Solid edges represent 1-edges, and dashed edges represent 0-edges.

regularly confused with the DPLL, but is actually a purely inference based method and not a tree style search method.

Procedure 6.2.2 (Davis-Putnam). *Given a* SAT *problem C, to compute* DP(C)*:*

1 **while** empty clause $\notin C$ **and** $C \neq \emptyset$
2   **do**
3         $v \leftarrow$ a variable of C
4         $C_v \leftarrow$ all clauses in C that contain the literal v or $\bar{v}$
5         $C \leftarrow C - C_v$
6         $C \leftarrow C \cup \{$all binary resolvents on v from $C_v\}$
7         $C \leftarrow C - \{$subsumed clauses$\}$
8 **if** $C = \emptyset$ **return** SAT
9   **else** **return** UNSAT

Variables are incrementally removed from the constraint set by selecting a variable, removing all clauses that contain the variable from the constraint set, and replacing them with the set of all resolvents on the variable. The problem is unsatisfiable if the empty clause is generated and satisfiable if the set of constraints becomes empty.

The DP algorithm is impractical for general satisfiability problems because the set of resolvents generated is usually too large to manage efficiently. However, ZRES, the ZBDD version of DP, provides polynomial solutions to both the pigeonhole problem and Urquhart problems. This shows that resolution defined for ZBDDs is strong enough to provide short proofs for both pigeonhole problems and Urquhart problems, providing efficient solutions for problems with very different types of structure. Unfortunately, ZRES performs poorly on general satisfiability problems from the DIMACS benchmark set. It is possible that ZRES suffers from the same inefficiencies as clausal DP when applied to weak CNF encodings. There is work on adapting DPLL-style solvers to use ZBDDs [1], but it does not incorporate learning methods and does not take advantage of the stronger inference possible for ZBDDs.

6.2.4 Finitely Quantified Clauses

Ginsberg and Parkes extended the local search method WSAT [63] to use finitely quantified constraints (QPROP) [34]. QPROP constraints allow universal quantification over finite domains. They introduced the concept of *subsearch*, showing that common solver subprocedures for managing a database of clauses are actually search problems in their own right. The WSAT procedure spends a majority of its time searching the clause database for clauses that are unsatisfied under a full assignment. When constraints are represented as QPROP constraints, the structure of the constraint can be used to improve the performance of the subsearch task. This was shown to be exponentially more efficient than using traditional methods for searching through the set of equivalent CNF instances. This result has implications for DPLL-style solvers. The task of searching the clause database for unit clauses is also a subsearch task and could potentially be improved by applying knowledge of the problem structure. This however, has not yet been tried.

CHAPTER 7

Conclusion

7.1 Contributions

We have built a pseudo-Boolean SAT solver based on the DPLL framework that automates a proof system properly stronger than the proof system of resolution based methods. We found that the stronger proof system allowed exponential reductions in both the size of search trees and execution times when compared with resolution-based methods on pigeonhole problems and planning problems. In fact, not only did we see dramatic advantages to using a stronger representation and proof system, we showed that there are no significant disadvantages to this approach over the weaker resolution-based methods since the pseudo-Boolean solver comes close to matching the performance of its resolution-based counterpart on CNF encodings, differing only by a factor of around 1.6 over a large variety of problem domains and sizes.

We introduced the idea that strong representations capture problem structure. If we consider the set of disjunctive clauses that equivalently represents a pseudo-Boolean constraint, the pseudo-Boolean constraint captures a pattern or symmetry of the equivalent clause set. We define two types of inference that are particularly useful for solvers: building inference that builds up a stronger, more structured constraint from a set of weaker constraints; and parallel inference that takes advantage of strong representations performing multiple symmetric inferences in parallel. The

combination of these two inference types allows short proofs of the pigeonhole principle in the cutting-plane proof system.

Our primary technical contribution was to show that the implementation of the learning method is the key to improving the strength of the underlying proof system of a pseudo-Boolean solver. Methods that use stronger representations, but fail to incorporate strong inference rules, do not gain the extra pruning power available when strong representations are combined with strong inference. Learning is the primary inference rule of DPLL-style solvers, so it is a natural place to incorporate a strong inference rule. We implemented a pseudo-Boolean version of learning that uses parallel inference. Within the context of a search tree, the multiple inferences of a parallel inference correspond to the elimination of multiple symmetric conflicts at once. This creates a very powerful pruning rule that leads to exponential speedups over clausal learning. The method we describe uses a pseudo-Boolean analog of resolution. A resolution analog is necessary because the partitioning scheme of DPLL-style solvers recursively partitions the search based on the domain values of a Boolean variable.

Unfortunately, the pseudo-Boolean resolvent does not always capture the conflict at hand. We proved conditions sufficient to ensure that the pseudo-Boolean resolvent will capture a conflict and gave a method for reducing two conflict reasons to satisfy these conditions. This may require weakening one of the conflict reasons to a cardinality constraint before the pseudo-Boolean resolvent is generated. We gave a method for reducing a pseudo-Boolean reason to a cardinality reason with care given to maintaining a high degree of parallel inference in the pseudo-Boolean resolvent. Experimentally, we saw that the learning rule that uses parallel inference achieved polynomial time scaling on pigeonhole problems while pseudo-Boolean solvers that don't learn, or learn only simple disjunctions, still showed exponential scaling just like resolution-based methods.

We applied the operations research technique of coefficient reduction within the DPLL framework. Coefficient reduction automates a form of building inference and therefore can be used to identify certain kinds of problem structure. The stronger pseudo-Boolean encoding of the pigeonhole principle can be derived from the weaker

clausal encoding in polynomial time using our preprocessing method. We also described an implementation that applies coefficient reduction during search although it appears that such an implementation is too expensive to be applied at every node.

Although our method is able to solve the explicit pigeonhole problem efficiently, we saw that it struggled to solve a pigeonhole problem embedded in a simple logistics domain. The difficulty was caused by the solver's inability to reason about the problem's first-order structure. We avoided this problem by doing the first-order reasoning by hand and adding the results to the problem encoding. This approach may work well if we need to solve a large number of problem instances from a single domain and time can be given to building and testing new problem encodings, but it is not likely to be feasible in general. Solving pigeonhole problems embedded in real world domains efficiently will likely require the ability to reason about more than one type of problem structure.

We can draw two important high-level conclusions from this work. First, DPLL-style solvers that automate proof systems stronger than resolution are not impractical. In fact, contrary to general sentiment, there appears to be little advantage to resolution-based methods over their pseudo-Boolean counterparts. Second, in order for an automated inference system to solve embedded pigeonhole problems efficiently, it must use a representation and set of inference rules that are general enough to capture a large range of problem structure. We believe the framework for building lifted solvers outlined here can serve as a guide for adapting DPLL-style solvers to use other representations and inference systems.

7.2 Future Work

There are a number of things we might do to extend this work within the pseudo-Boolean framework. It would be interesting to see if the work on solving random logistics problem could be extended to other planning domains. How difficult is it in general to find encodings that enable PBCHAFF to identify the embedded pigeonhole problems? It is unclear whether this approach will be practical for more complex planning domains.

A comparison of PBCHAFF with branch-and-bound or branch-and-cut methods might help illuminate the relative strengths and weaknesses of the integer programming framework as compared to the DPLL framework. Unfortunately, because 0-1 integer programming methods also use pseudo-Boolean representation and cutting planes inference, one could make a theoretical argument that 0-1 integer programming techniques will suffer from the same inability to reason about multiple types of structure.

The most compelling direction of future work is to implement a DPLL-style solver that uses a representation and inference system general enough to capture a variety of different types of problem structure and reason about structured constraints uniformly. The augmented clause representation [26] is promising in this respect. An augmented clause consists of a propositional clause together with a permutation group on the set of literals. The augmented clause is logically equivalent to the conjunction over the set of clauses that can be formed by applying elements of the group to the representative clause. This representation is known to generalize cardinality constraints, parity constraints, and finitely quantified clauses, and allows uniform reasoning between all augmented clauses regardless of their origin. The resolution procedure that has been defined for augmented clauses fosters a high degree of parallel inference allowing short proofs of pigeonhole problems, parity problems and clique coloring problems. The significant body of existing work from the field of computational group theory provides a library of routines for manipulating and searching through the resulting groups.

INDEX

annotated partial assignment, 13
  sound, 13
asserting clause, 13

BACKJUMP, 14, 21
backjumping, 12
BERKMIN, 19, 24
branch heuristics, 10
  and learning, 23
  BERKMIN, 24
  MOMS, 11
  pseudo-Boolean, 75
  VSIDS, 24
branching, 8
building inference, 49

cardinality constraint, 43
coefficient reduction, 76
conflict set, 15
count-based indexing, 27, 53
`current`, 54
cutting-plane
  p-simulate resolution, 44
  proof system, 43

decision levels, 20
DP, 132
DPLL, 10
DPLL-WITH-LEARNING, 14
dynamic backtracking, 18

exponentially seperated, 39
exponentially stronger, 39

factoring, 13

GET-ASSUMPTIONS, 79
GRASP, 20

irrelevance, 18, 19, 70

learning, 11, 12, 61
  bounded, 17
  length-bounded, 18
  relevance-bounded, 18
length, 70
literal index, 26

nogood, 12
NON-STANDARD-BACKJUMP, 21

parallel inference, 49
partial assignment, 9
  closed, 29
  closure, 29
  extends, 9
  invalid, 9
  unit complete, 31
  valid, 9
PB-BACKJUMP, 73
PB-DPLL-WITH-LEARNING, 73
PB-NON-STANDARD-BACKJUMP, 74
PB-UNIT-PROPAGATE, 56
pigeonhole problem, 40, 48, 88
polynomially equivalent, 39
polynomially simulates, 39
`possible`, 54
proof complexity, 38
proof system, 38
  polynomial bounded, 39
propositional proof system, 38
pseudo-Boolean
  bounded learning, 68
  branch heuristics, 75
  constraint, 43
  irrelevance, 70
  learning, 61
  length, 70
  resolvent, 62
  watching set, 58

reason, 13

RELSAT, 19, 25
resolution, 12, 13, 44
  DAG-style, 37
  lower bounds, 40
  tree-style, 36
resolution-based, 36

SAT, 7
satisfiability, 7
satisfied, 7
Skolem function, 103
solution, 7
STRENGTHEN, 77
strengthening, 76
subsearch, 133

thrashing, 11

UIP clause, 20
UIP constraint, 72
unique implication points, 20
unit clause, 10
unit complete, 31
unit constraint, 54
unit literal, 54
UNIT-PROPAGATE, 10
unit propagation, 10, 25
  data structures, 26
unsatisfiable, 7

watched literals, 28, 32
watching set, 28
  cardinality, 57
  pseudo-Boolean, 58
WSAT, 133

ZCHAFF, 19, 20, 24, 25, 32

BIBLIOGRAPHY

[1] ALOUL, F. A., MNEIMNEH, M., AND SAKALLAH, K. A. ZBDD-based backtrack search SAT solver. In *International Workshop on Logic Synthesis (IWLS)* (2002), pp. 131–136.

[2] ALOUL, F. A., RAMANI, A., MARKOV, I. L., AND SAKALLAH, K. A. PBS: A backtrack-search pseudo-Boolean solver and optimizer. In *Symposium on the Theory and Application of Satisfiability Testing* (2002), pp. 346–353.

[3] BARTH, P. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Tech. Rep. MPI-I-95-2-003, Max Planck Institut für Informatik, Saarbrücken, Germany, 1995.

[4] BAUMGARTNER, P., AND MASSACCI, F. The taming of the (X)OR. In *Computational Logic*, Lecture Notes in Computer Science 1861. Springer, 2000, pp. 508–522.

[5] BAYARDO, R. J., AND MIRANKER, D. P. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (1996), pp. 298–304.

[6] BAYARDO, R. J., AND SCHRAG, R. C. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence* (1997), pp. 203–208.

[7] BEAME, P. Proof complexity. http://www.cs.toronto.edu/~toni/Courses /Proofcomplexity/Papers/paul-lectures.ps. accessed August 2004.

[8] BEAME, P., KAUTZ, H., AND SABHARWAL, A. Understanding the power of clause learning. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence IJCAI'03* (2003), pp. 1194–1201.

[9] BENHAMOU, B., SAIS, L., AND SIEGEL, P. Two proof procedures for a cardinality based language in propositional calculus. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science* (1994), Springer-Verlag, pp. 71–82.

[10] BIERE, A., CLARKE, E. M., RAIMI, R., AND ZHU, Y. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. *Lecture Notes in Computer Science 1633* (1999), 60–71.

[11] BLAIR, C. E., JEROSLOW, R. G., AND LOWE, J. K. Some results and experiments in programming techniques for propositional logic. *Computers and Operations Research 13*, 5 (1986), 633–645.

[12] CHAI, D., AND KUEHLMANN, A. A fast pseudo-Boolean constraint solver. In *Proceedings of the 40th Design Automation Conference* (2003), pp. 830–835.

[13] CHATALIC, P., AND SIMON, L. Multi-resolution on compressed sets of clauses. In *Twelth International Conference on Tools with Artificial Intelligence (ICTAI'00)* (2000), pp. 449–454.

[14] CHATALIC, P., AND SIMON, L. Zres: the old Davis-Putnam meets ZBDDs. In *17th International Confernece on Automated Deduction (CADE'17)* (2000), D. McAllester, Ed., no. 1831 in Lecture Notes in Artificial Intelligence (LNAI), pp. 449–454.

[15] CHVÁTAL, V. Edmonds polytopes and weakly Hamiltonian graphs. *Mathematical Programming 5* (1973), 29–40.

[16] CHVÁTAL, V. *Linear Programming.* W.H. Freeman Co., 1983.

[17] COOK, S. A. The complexity of theorem-proving procedures. In *Procedings of the Third Annual ACM Symposium on the Theory of Computing* (1971), pp. 151–158.

[18] COOK, S. A., AND RECKHOW, R. A. The relative efficiency of propositional proof systems. *Journal of Symbolic Logic 44*, 1 (1977), 36–50.

[19] COOK, W., COULLARD, C., AND TURÁN, G. On the complexity of cutting plane proofs. *Journal of Discrete Applied Math 18* (1987), 25–38.

[20] COPTY, F., FIX, L., GIUNCHIGLIA, E., KAMHI, G., TACCHELLA, A., AND VARDI, M. Benefits of bounded model checking in an industrial setting. In *13th Conference on Computer Aided Verification, CAV'01* (Paris, France, July 2001), pp. 436–453.

[21] CRAWFORD, J. M., AND AUTON, L. D. Experimental results on the crossover point in random 3SAT. *Artificial Intelligence 81* (1996), 31–57.

[22] DAVIS, M., LOGEMANN, G., AND LOVELAND, D. A machine program for theorem proving. *Communications of the ACM 5*, 7 (1962), 394–397.

[23] DECHTER, R. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence 41*, 3 (1990), 273–312.

[24] DIXON, H. E., AND GINSBERG, M. L. Combining satisfiability techniques from AI and OR. *The Knowledge Engineering Review 15*, 1 (2000), 31–45.

[25] DIXON, H. E., AND GINSBERG, M. L. Inference methods for a pseudo-Boolean satisfiability solver. In *The Eighteenth National Conference on Artificial Intelligence (AAAI-2002)* (2002), pp. 635–640.

[26] DIXON, H. E., GINSBERG, M. L., LUKS, E. M., AND PARKES, A. J. Generalizing Boolean satisfiability II: Theory. Tech. rep., CIRL, University of Oregon, Eugene Oregon, 2004.

[27] DIXON, H. E., GINSBERG, M. L., AND PARKES, A. J. Generalizing Boolean satisfiability I: Background and survey of existing work. *Journal of Artificial Intelligence Research 21* (2004), 193–243.

[28] DUBOIS, O., ANDRE, P., BOUFKHAD, Y., AND CARLIER, J. SAT versus UNSAT. In *Second DIMACS Challenge: Cliques, Colorings and Satisfiability* (Rutgers University, NJ, 1993).

[29] DUBOIS, O., AND DEQUEN, G. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence* (2001), pp. 248–253.

[30] FREEMAN, J. W. *Improvements to propsitional satisfiability search algorithms.* PhD thesis, University of Pennsylvania, PA, 1995.

[31] FROST, D., AND DECHTER, R. Dead-end driven learning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994), pp. 294–300.

[32] GASCHNIG, J. Performance measurement and analysis of certain search algorithms. Tech. Rep. CMU-CS-79-124, Carnegie-Mellon University, 1979.

[33] GINSBERG, M. L. Dynamic backtracking. *Journal of Artificial Intelligence Research 1* (1993), 25–46.

[34] GINSBERG, M. L., AND PARKES, A. J. Satisfiability algorithms and finite quantification. In *(KR)2000: Principles of Knowledge Representation and Reasoning* (Breckenridge, Colorado, 2000), Morgan Kaufmann, pp. 690–701.

[35] GOLDBERG, E., AND NOVIKOV, Y. Berkmin: A fast and robust SAT solver. In *Design Automation and Test in Europe (DATE) 2002* (2002), pp. 142–149.

[36] Gomory, R. An algorithm for integer solutions to linear programs. In *Recent Advances in Mathematical Programming*. McGraw-Hill, New York, 1963, pp. 269–302.

[37] Guignard, M., and Spielberg, K. Logical reduction methods in zero-one programming. *Operations Research 29* (1981).

[38] Haken, A. The intractability of resolution. *Theoretical Computer Science 39* (1985), 297–308.

[39] Hammer, P., and Rudeanu, S. *Boolean Methods in Operations Research and Related Areas.* Springer, 1968.

[40] Hooker, J. N. Generalized resolution and cutting planes. *Annals of Operations Research 12* (1988), 217–239.

[41] Hooker, J. N. Resolution vs. cutting plane solution of inference problems: Some computational experience. *Operations Research Letters 7*, 1 (1988), 1–7.

[42] Hooker, J. N. Input proofs and rank one cutting planes. *ORSA Journal on Computing 1* (1989), 137–145.

[43] Hooker, J. N. Constraint satisfaction methods for generating valid cuts. In *Advances in Computational and Stochastic Optimization, Logic Programming, and Heuristic Search*, D. L. Woodruff, Ed. Kluwer, 1997, pp. 1–30.

[44] Hooker, J. N., and Fedjki, C. Branch-and-cut solution of inference problems in propositional logic. *Annals of Mathematics and Artificial Intelligence 1* (1990), 123–139.

[45] Hooker, J. N., and Vinay, V. Branching rules for satisfiability. *Journal of Automated Reasoning 15* (1995), 359–383.

[46] Jackson, D., Schechter, I., and Shlyakhter, I. Alcoa: the alloy constraint analyzer. In *Proceedings of the International Conference on Software Engineering* (Limerick, Ireland, 2000), pp. 730–733.

[47] Jeroslow, R., and Wang, J. Solving the propositional satisfiability problem. *Annals of Mathematics and Artificial Intelligence 1* (1990), 167–187.

[48] Kautz, H. A., and Selman, B. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)* (1992), pp. 359–363.

[49] Li, C. M. Integrating equivalency reasoning into davis-putnam procedure. In *AAAI00* (2000), pp. 291–296.

[50] Li, C. M., and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (1997), pp. 366–371.

[51] Lynce, I., and Marques-Silva, J. Efficient data structures for fast SAT solvers. Tech. rep., Instituto de Engenharia de Sistemas e Computadores, 2001.

[52] Marques-Silva, J. P., and Sakallah, K. A. GRASP: A search algorithm for propositional satisfiability. In *IEEE Transactions on Computers* (1999), vol. 48, pp. 506–521.

[53] Minato, S. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. 30th ACM/IEEE Design Automation Conf.* (1993), pp. 272–277.

[54] Mitchell, D. G. Hard problems for CSP algorithms. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence* (1998), pp. 398–405.

[55] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference* (2001), pp. 530–535.

[56] Nemhauser, G. L., and Wolsey, L. A. *Integer and Combinatorial Optimization.* Wiley, New York, 1988.

[57] Padberg, M., and Rinaldi, G. A branch and cut algorithm for the resolution of large-scale symmetric traveling salesman problems. *SIAM Review 33* (1991), 60–100.

[58] Pretolani, D. *Satisfiability and hypergraphs.* PhD thesis, Universita di Pisa, 1993.

[59] Ralphs, T., Ladanyi, L., Trotter, L., and Jr. Branch, cut, and price: Sequential and parallel. citeseer.nj.nec.com/486881.html. accessed November 2004.

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

[61] Savelsbergh, M. W. P. Preprocessing and probing for mixed integer programming problems. *ORSA Journal on Computing 6* (1994), 445–454.

[62] Schaefer, T. The complexity of satisfiability problems. *Proceedings of the 10th ACM Symposium on Theory of Computing (STOC-78)* (1978), 216–226.

[63] Selman, B., Kautz, H., and Cohen, B. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence* (1994), pp. 337–343.

[64] Selman, B., Kautz, H., and McAllester, D. Ten challenges in propositional reasoning and search. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence* (1997), pp. 50–54.

[65] Skolem, T. Über die mathematische logik. *Norsk matematisk tidsskrift 10* (1928), 125–142.

[66] Smith, T. *Window-Based Project Scheduling Algorithms.* PhD thesis, University of Oregon, 2004.

[67] Stallman, R. M., and Sussman, G. J. Forward reasoning and dependency directed backtracking in a system for computer aided circuit analysis. *Artificial Intelligence 9*, 2 (1977), 135–196.

[68] Tseitin, G. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic, Part 2*, A. Slisenko, Ed. Consultants Bureau, 1970, pp. 466–483.

[69] Urquhart, A. Hard examples for resolution. *Journal of the ACM 34* (1987), 209–219.

[70] Velev, M. N., and Bryant, R. E. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW. In *Proceedings of the 38th Conference on Design Automation Conference 2001* (New York, NY, USA, 2001), ACM Press, pp. 226–231.

[71] Walser, J. P. Solving linear pseudo-Boolean constraint problems with local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence* (1997), pp. 269–274.

[72] Warners, J. P. A linear-time transformation of linear inequalities into conjunctive normal form. *Information Processing Letters 68*, 2 (1998), 63–69.

[73] Warners, J. P., and Maaren, H. V. A two phase algorithm for solving a class of hard satisfiability problems. *Operations Research Letters 23* (1998), 81–88.

[74] Whittemore, J., Kim, J., and Sakallah, K. A. SATIRE: A new incremental satisfiability engine. In *Proceedings of the Design Automation Conference* (2001), pp. 542–545.

[75] Zhang, H., and Stickel, M. E. Implementing the Davis-Putnam method. *Journal of Automated Reasoning 24*, 1-2 (2000), 277–296.

[76] ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. H., AND MALIK, S. Efficient conflict driven learning in a Boolean satisfiability solver. *International Conference on Computer-Aided Design* (2001).