

LIFTED SEARCH ENGINES FOR SATISFIABILITY

by

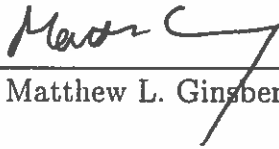
ANDREW JOHN PARKES

A DISSERTATION

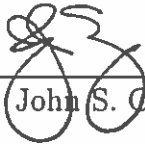
Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 1999

"Lifted Search Engines for Satisfiability," a dissertation prepared by Andrew John Parkes 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 L. Ginsberg, Co-chair of the Examining Committee



Dr. John S. Conery, Co-chair of the Examining Committee

6-1-99

Date

Committee in charge:

- Dr. Matthew L. Ginsberg, Co-chair
- Dr. John S. Conery, Co-chair
- Dr. Christopher B. Wilson
- Dr. David W. Etherington
- Dr. Roger Haydock

Accepted by:

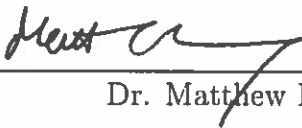



Dean of the Graduate School

©1999 Andrew John Parkes

An Abstract of the Dissertation of
Andrew John Parkes for the degree of Doctor of Philosophy
in the Department of Computer and Information Science
to be taken June 1999

Title: LIFTED SEARCH ENGINES FOR SATISFIABILITY

Approved: 
_____ Dr. Matthew L. Ginsberg, Co-chair


_____ Dr. John S. Conery, Co-chair

There are several powerful solvers for satisfiability (SAT), such as WSAT, Davis-Putnam, and RELSAT. However, in practice, the SAT encodings often have so many clauses that we exceed physical memory resources on attempting to solve them. This excessive size often arises because conversion to SAT, from a more natural encoding using quantifications over domains, requires expanding quantifiers.

This suggests that we should “lift” successful SAT solvers. That is, adapt the solvers to use quantified clauses instead of ground clauses. However, it was generally believed that such lifted solvers would be impractical: Partially, because of the overhead of handling the predicates and quantifiers, and partially because lifting would not allow essential indexing and caching schemes.

Here we show that, to the contrary, it is not only practical to handle quantified clauses directly, but that lifting can give exponential savings. We do this by identifying certain tasks that are central to the implementation of a SAT solver.

These tasks involve the extraction of information from the set of clauses (such as finding the set of unsatisfied clauses in the case of WSAT) and consume most of the running time. We demonstrate that these tasks are NP-hard with respect to their relevant size measure. Hence, they are themselves search problems, and so we call them “subsearch problems”.

Ground SAT solvers effectively solve these subsearch problems by naive enumeration of the search space. In contrast, a lifted solver can solve them using intelligent search methods. Consequently, lifting a solver will generally allow an exponential reduction in the cost of subsearch and so increase the speed of the search engine itself.

Experimental results are given for a lifted version of WSAT. We only use very simple backtracking for the subsearch, but we still find that cost savings in the subsearch can more than offset the overheads from lifting. The reduction in size of the formulas also allows us to solve problems that are too large for ground WSAT.

In summary, a lifted SAT solver not only uses far less memory than a ground solver, but it can also run faster.

CURRICULUM VITA

NAME OF AUTHOR: Andrew John Parkes

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
University of Edinburgh, United Kingdom
University of London, United Kingdom
University of Manchester, United Kingdom

DEGREES AWARDED:

Doctor of Philosophy in Computer Science, 1999, University of Oregon
Master of Science in Knowledge-Based Systems, 1994, University of
Edinburgh
Doctor of Philosophy in Mathematics, 1984, University of London
Bachelor of Science in Mathematics and Physics, 1981, University of
Manchester

AREAS OF SPECIAL INTEREST:

Artificial Intelligence and Combinatorial Optimization

PROFESSIONAL EXPERIENCE:

Research Assistant, Computational Intelligence Research Laboratory,
University of Oregon, Eugene, 1994-99
Research Associate, Federal Institute of Technology, Zürich, 1990-92
Research Associate, Department of Physics, University of California at
Davis, 1987-90
Research Associate, European Laboratory for Particle Physics, Geneva,
1985-87

Research Associate, Department of Physics, University of
Southampton, 1984-85

PUBLICATIONS:

Andrew J. Parkes and Joachim P. Walser. *Tuning Local Search for Satisfiability Testing*. Proceedings of AAAI-96, pages 356-362, 1996.

Andrew J. Parkes. *Clustering at the Phase Transition*. Proceedings of AAAI-97, pages 340-345, 1997.

Matthew L. Ginsberg, Andrew J. Parkes and Amitabha Roy. *Supermodels and Robustness*. Proceedings of AAAI-98, pages 334-339, 1998.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my advisor, Matt Ginsberg, for his excellent guidance, his insights into directions worth pursuing, and for providing examples of research done well. Many other insights and good examples have been provided by members of, and visitors to, CIRL past and present: Andrew Baker, Tania Bedrax-Weiss, Dave Clements, James Crawford, Heidi Dixon, David Etherington, Will Harvey, Alan Jaffray, Ari Jónsson, David Joslin, Bart Massey, Joe Pemberton, Amitabha Roy, and Joachim Walser. I would like to thank them all for the many discussions (technical and otherwise) that made the work a rewarding and fun experience. Thanks to Tania for being a good office companion, and to Laurie Buchanan for keeping the lab running smoothly.

Thanks to John Conery for co-chairing my committee, and also to the computer science department in general.

Special thanks to my family. Very special thanks to Maria for her great patience, encouragement, and support.

This work was sponsored in part by grants from Air Force Office of Scientific Research (AFOSR), number F49620-92-J-0384, and Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Rome, NY, under agreements numbered F30602-95-1-0023, F30602-97-1-0294 and F30602-98-2-0181. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be in-

terpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Laboratory, or the U.S. Government.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
1.1. Overview of the Thesis	4
II. LOGICAL REPRESENTATIONS, QPROP, AND PROBLEM DO- MAINS	7
2.1. Boolean Logic	8
2.2. Quantified Propositional Logic (QPROP)	11
2.3. Grounding QPROP to Boolean Logic	20
2.4. The Pigeonhole Problem (PHP)	22
2.5. SATPLAN	23
2.6. Summary	31
III. STANDARD INFERENCE METHODS IN SAT	33
3.1. Model Search	34
3.2. Iterative Repair	34
3.3. WSAT (WalkSAT)	36
3.4. Backtracking "Branch-and-Propagate" Methods	42
3.5. Unit Propagation	44
3.6. The Davis-Putnam Procedure	47
3.7. Proof-Search: Resolution	50
3.8. Mixed Methods	52
3.9. Pre-Processing and COMPACT	52
3.10. No-Good Learning in Intelligent Backtracking	57
3.11. Summary	57
IV. SUBSEARCH	59
4.1. Quantified Clausal Normal Form (QCNF)	60
4.2. The "Database of Clauses" Viewpoint	63
4.3. General Restrictions on the Database	65
4.4. Syntactic Restrictions: The Set $R(C, l)$	70
4.5. Semantic Restrictions: The Set $S(C, P, u, s)$	74
4.6. Subsearch Problems	75
4.7. Summary	82

	Page
V. REWRITING SEARCH ALGORITHMS IN TERMS OF SUBSEARCH	83
5.1. Splitting Subsearch and Search	84
5.2. Minimal vs. Non-Minimal Lifting	85
5.3. Model Checking	86
5.4. Exploiting Incremental Changes	88
5.5. Syntactic Restrictions and Static Indexing	89
5.6. WSAT	90
5.7. Unit Propagation	96
5.8. Davis-Putnam Procedure	97
5.9. Summary	99
VI. THE COMPUTATIONAL COMPLEXITY OF SUBSEARCH	100
6.1. Constraint Satisfaction Problems (CSPs)	101
6.2. Conversion of $S(c, P, 0, 0)$ to a CSP	103
6.3. Complexity of the Checking Problem for $S(C, P, 0, 0)$	107
6.4. Other Subsearch Problems for $S(C, P, 0, 0)$	111
6.5. General $S(c, P, u, s)$ and Counting-CSPs	112
6.6. Conversion of $S(c, P, u, s)$ to a CCSP	115
6.7. Complexity of the Checking Problem for $S(C, P, u, s)$	117
6.8. Summary	119
VII. INTELLIGENT SUBSEARCH SPEEDS SEARCH	120
7.1. Pruning the Subsearch for $S(C, P, 0, 0)$	121
7.2. General Subsearch Pruning: $S(C, P, u, s)$	126
7.3. Syntactic Incrementality	128
7.4. Expected Costs and Gains of Subsearch	130
7.5. Semantic Incrementality	140
7.6. Subsumption	142
7.7. Summary	144
VIII. RESULTS FROM IMPLEMENTING A LIFTED SOLVER	145
8.1. Implementation of a Simple QCNF Solver	146
8.2. The Test Domain: Logistics	150
8.3. Experimental Methods	152
8.4. Two Implementations of WSAT: WSAT(U) and WSAT(UB)	155
8.5. WSAT(U) Initialization Results	156
8.6. WSAT(U) Flip Rates	158
8.7. Pre-Calculating the Break Counts: WSAT(UB)	159

	Page
8.8. Experimental Fliprates for WSAT(UB)	165
8.9. Effects of Machine Cache on Scalings	166
8.10. Experimental Initialization Times for WSAT(UB)	171
8.11. Summary	173
IX. ADVANCED SUBSEARCH IN QCNF	175
9.1. Semantic and Syntactic Methods	176
9.2. Factored Clauses	178
9.3. Disjoint Sets Example	180
9.4. No-Circles Example	183
9.5. Exploiting Graph Structure	188
9.6. Summary	190
X. RELATED WORK	191
10.1. Global Constraints	191
10.2. Sebastiani's Approach to Non-Clausal Formulas	195
XI. OPEN QUESTIONS AND FUTURE WORK	197
11.1. Implications for Problem Encodings	197
11.2. Non-Minimal Lifting	198
11.3. Breaking out of QCNF	204
XII. CONCLUSION	209
12.1. Contributions	209
12.2. Closing Comment	210
BIBLIOGRAPHY	211

LIST OF FIGURES

Figure	Page
1. Logistics Example: Axioms Defining Auxiliary Predicates	29
2. Logistics Example: Axioms Expressing Consistency of the State.	29
3. Logistics Example: Axioms Defining the Allowed Actions	30
4. Pure Random Walk for SAT	36
5. The WSAT Algorithm	37
6. Two Variable Selection Methods in WSAT	38
7. Counting Makes and Breaks in WSAT	39
8. Updating the Unsatisfied Clause Set in WSAT	39
9. A Simple Recursive Way to Implement Unit Propagation.	45
10. Two Iterative Implementations of Unit Propagation	46
11. The Davis-Putnam Procedure	47
12. Selecting the Branch Variable in Davis-Putnam	48
13. Straightforward COMPACT-L	54
14. Utility Functions VALUE and TEST-VALUE	55
15. Splitting Subsearch from Search	84
16. Simple Lifted Model Check	87
17. WSAT in Terms of Subsearch	91
18. Lifted Updating of the Unsatisfied Clause Set in WSAT	93
19. Counting Makes and Breaks for WSAT Using Subsearch	96
20. Lifted Unit Propagation	97
21. Constraint Graphs of Sub-CSPs from Two Simple Clauses	106

	Page
22. Code to Search for Violated Clauses	125
23. Another Version of Lifted Unit Propagation	127
24. WSAT Initialization With a Simple Clause	138
25. Software Architecture of the Implemented Lifted Solver	147
26. Experimental Initialization Times for WSAT(U)	157
27. Experimental Performance of Ground and Lifted WSAT(U) Solvers .	158
28. Experimental Times per Flip for the Ground and Lifted WSAT(U) Solvers	160
29. Ratio of Fliprates for the Ground to Lifted WSAT(U) Solvers	161
30. Experimental Performance of Ground and Lifted WSAT(UB) Solvers	167
31. Ratio of Fliprates for the Ground to Lifted WSAT(UB) Solvers . . .	168
32. Experimental Times per Flip for the Ground and Lifted WSAT(UB) Solvers	169
33. Experimental Times per Flip, for WSAT(UB), With Physical Cache Disabled	170
34. Ratio of Fliprates for the Ground to Lifted WSAT(UB) Solvers With and Without Cache	171
35. Experimental Initialization Times for WSAT on a Large Axiom	173
36. Example of a Disconnected Sub-CSP Graph	179
37. Example of a Linear Sub-CSP Graph	181
38. Structure of the Sub-CSP for the “No-Circles” Constraint	184
39. Final Version of Unit Propagation	186

LIST OF TABLES

Table	Page
1. Examples of Sizes of Logistics Instances	153

CHAPTER I

INTRODUCTION

Propositional satisfiability (SAT) is the problem of determining whether or not there are satisfying assignments of a set of boolean clauses and is NP-complete [8, 22]. The NP-completeness, and the fact that it is a logical language, means that it is possible, and fairly natural, to express many combinatorial problems in SAT. Problems in circuit verification, scheduling, planning, and many other areas, have all been treated as SAT problems.

Not surprisingly, there has been a lot of work on finding practical algorithms for solving SAT. There are powerful systematic solvers based on the classical Davis-Putnam (DP) procedure [14, 15] but incorporating new heuristics, intelligent backtracking, and learning mechanisms. Recent examples are POSIT[21], NTAB [13], SATZ [42] and RELSAT [2]. There are also non-systematic algorithms based on local search: Some examples are the "Breakout" method of Morris [49], GSAT [66, 68] and WSAT [67].

A major problem with using such solvers is that encoding natural problems into SAT results in very large formulas. We will see examples in which the initial problem is of modest size but the encoding into a SAT problem gives formulas with many millions of clauses. With problems of such a large size it can be impractical to use existing SAT solvers because of a lack of physical memory.

A reason for this excessive size of SAT encodings is that propositional logic

does not have quantifiers. Many problems are most naturally expressed using a subset of first-order logic that includes quantification over elements within the domain. For example, it is natural to want to be able to express constraints using “quantified clauses,” such as

$$\forall i, j, k. \neg q(i, j) \vee r(j, k) \quad (1.1)$$

However, converting to SAT (grounding) means expanding such quantifiers. The resulting increase in size is exponential in the number of quantifiers.

This drawback suggests “lifting” successful SAT solvers: that is, adapting the solvers to use such quantified clauses. A lifted solver should be able to use such clauses directly, instead of having to first convert them into much larger boolean formulas. As a question of terminology we remark that the term “lifting” is somewhat overloaded within the search literature as a whole. Here, we use “lifted” to mean a solver that is based on propositional methods, but that can also handle expressions involving quantifiers.

Although such lifted solvers would solve the problem of excessive memory usage, it was generally believed they would be too slow to be practical. One reason for this belief is that the lifted solver has the overheads of handling the predicates and quantifiers. Another concern was that lifting would not allow the various indexing and caching schemes that are essential to the speed of a ground SAT solver.

Here we give evidence that, to the contrary, it is practical to handle quantified clauses directly. In fact we show a stronger result. We will see that lifting a solver

can lead to exponential savings (in both time and space) that are not available to the ground SAT equivalent.¹ A lifted solver can not only be competitive with a ground solver, but can even be faster (as well as using far less memory).

The core of our approach to successfully lifting a SAT solver lies in the observation that solvers spend most of their time in the collection and maintenance of information about sets of clauses. Relevant sets of clauses might consist of “those containing a literal l ” or “binary clauses”. The information might be quantities such as “the number of clauses that currently contain one true literal”. In fact, such “information collection” tasks typically dominate the runtime, but their very mundaneness means they are often ignored in the literature (though when coding it is crucial to implement them efficiently). We will call such tasks “subsearch problems”.

The approach we take to lifting is to identify and isolate these subsearch problems. Lifting a solver is converted into a two stage process:

1. Reformulate the SAT solver to access the set of clauses only by means of subsearch problems. Essentially the search algorithm itself will be shielded from whether the clauses themselves are ground or quantified.
2. Formulate the subsearch problems so that they can work with quantified clauses as well as the ground (non-quantified) clauses of SAT.

Note that subsearch is used to gather information about the clauses, and this information is then used by the search engine itself. It is important to realize that

¹That such savings are possible was initially observed by Ginsberg [26]. The “subsearch” viewpoint developed in the thesis arose as a generalization of these initial SAT-specific observations.

search and subsearch are otherwise quite distinct.

We will see that subsearch problems are NP-hard with respect to their relevant size measure, and therefore are search problems themselves – hence the name “subsearch”. Ground SAT solvers effectively solve these subsearch problems using enumeration of the search space, that is, generate-and-test. In a lifted solver we can solve these subsearch problems using more effective search techniques, and so obtain savings that are potentially exponential in the size of the quantified clauses. Based on the ideas in this thesis we implemented a lifted solver to run unit propagation (see Section 3.5) and WSAT. We will present experimental results (in Chapter VIII) to show that the advantages are real. We can also solve problems that are much too large to be solved by the pre-existing ground SAT solvers.

Although the main practical motivation for this work is to avoid the large boolean formulas encountered with SAT, we are equally motivated by exploitation of structure. The expansion of quantifiers on converting from quantified clauses to SAT not only causes a large increase in size but also totally obscures the problem structure. Instead, a solver should be able to exploit the structure of the problem. Of course, this is hard to do, but preserving the quantified structure does allow us to make a few steps in this direction.

1.1 Overview of the Thesis

The thesis starts with two chapters providing some necessary background on SAT and its solvers. Chapter II reviews SAT and first-order logic with finite sorts. It then defines “Quantified Propositional Logic” (QPROP), and the notion of grounding of QPROP to boolean functions. It also gives two examples of problem

domains and their logical representations. Most important of these is the logistics example that will be used throughout the thesis.

Chapter III describes standard inference methods for SAT. We cover search methods, both local search methods such as WSAT and systematic techniques such as Davis-Putnam.

Chapters IV-VIII describe the central theme of the thesis: that propositional solution methods can be lifted to provide a practical and useful way to handle quantified clauses. Chapter IV goes into more detail about our main representational language “Quantified Clausal-Normal-Form” (QCNF), and explains why it is likely to be useful and practical. We also define the various subsearch problems that arise in the implementation of search algorithms. Chapter V reformulates standard search methods in terms of these subsearch problems.

Chapter VI goes into the details of the subsearch problems themselves. We show that even the easiest problem is NP-hard (in a sense that will be described). This NP-hardness is the reason for the “search” in “subsearch”. The notions in Chapters V and VI are pulled together in Chapter VII where we show why the properties of subsearch should potentially enable search engines to run faster.

Of course, it could be that the expectations of theory are wrong, and so we implemented a solver based on the subsearch ideas. The implemented lifted solver is described in Chapter VIII, where we also present experimental results on its performance, confirming that such a solver is not only feasible, but can handle much larger problems than the ground solvers.

Chapter IX looks into using more advanced search techniques for the subsearch.

Before making our conclusions in Chapter XII, we cover some related work in Chapter X, and some open questions and ideas for future work in Chapter XI.

CHAPTER II

LOGICAL REPRESENTATIONS, QPROP, AND PROBLEM DOMAINS

In this chapter we consider the logical representation languages needed for this thesis. We will assume a reasonable familiarity with propositional and first-order logics, and so we will give just an informal and motivational review of relevant issues, rather than a presentation of standard formal logic.

One of the points of terminology we need to cover arises from the different meanings of the word “variable” in propositional and first order logic. Since we will need to work with both of these logics we need to be careful to distinguish the two meanings. Similar concerns will apply to the meaning of “quantifying over a variable”.

After a brief review of boolean logic, we will discuss a simple portion of first order logic that is commonly used in combinatorial and artificial intelligence (AI) problems. This portion has finite sorts and some restrictions on functions. We will call it “Quantified Propositional Logic” (QPROP), partly for lack of a good existing name, and partly to bring out the fact that it is essentially just propositional logic but with the addition of quantifiers. However, we emphasize that just the name is new, not the concept itself: QPROP has been used many times before, though possibly without comment and sometimes just as an intermediate language. Indeed, Jónsson has written a “constraint satisfaction problem compiler” (CSPC) for general use in AI [34], and QPROP is essentially the same as the input to CSPC.

We will refer to CSPC again in the experimental work in Chapter VIII.

As discussed in the introduction, we believe that QPROP is worthy of study in its own right because it allows quantifications that capture a form of structure common in real problems.

We will describe how QPROP expressions are “grounded” to obtain boolean formulas. We finish the chapter with examples of the usage of QPROP. We give the QPROP form of the pigeonhole problem and pigeonhole principle. More importantly we review some concepts from SATPLAN, a declarative approach to planning due to Kautz and Selman [38]. In particular, we take a simplified form of a logistics domain that has been used in the SATPLAN literature, and present an encoding of it in QPROP. This logistics domain and encoding are important, not because they contain anything new, but simply because they will be used as a running example throughout the thesis, and especially for the experimental results in Chapter VIII.

2.1 Boolean Logic

The fundamental object in boolean logic is a boolean variable. As far as boolean logic is concerned such a variable is atomic, it has no internal structure, merely a unique name. Logical expressions, i.e. boolean functions, are formed by combining variables using the connectives; negation \neg , conjunction \wedge , disjunction \vee , implication \longrightarrow , reverse implication \longleftarrow , and logical equivalence \longleftrightarrow . Parsing follows standard conventions, for example,

$$a \wedge b \vee c \longrightarrow c \vee \neg d \longleftrightarrow e$$

is parsed as

$$(((a \wedge b) \vee c) \longrightarrow (c \vee (\neg d))) \longleftrightarrow e$$

The most common case considered in work on combinatorial decision problems is that of a formula (often just called the “theory”) in conjunctive normal form (CNF). Such a formula is a conjunction of clauses. A clause is a disjunction of literals, and a literal is a boolean variable or its negation. A literal that is a variable is said to be a positive literal, a literal that is a negated variable is said to be a negative literal. The complement of a literal is defined as its negation. Usually we shall write formulas as sets of clauses, and leave the conjunctions implicit. For example,

$$x \vee y \vee z$$

$$x \vee \neg y$$

It is also common to refer to a “formula in CNF” as a “clausal formula” or “CNF-formula”, and say that it is in “clausal form.” Purely, for the sake of readability we will write clauses in various logically equivalent “implicational forms” such as

$$x \wedge w \longrightarrow \neg(y \wedge z)$$

$$x \wedge w \longrightarrow (\neg y \vee \neg z)$$

However, such rule-like forms will not themselves be used by the solvers we consider. Instead, the equivalent disjunctions of literals will always be used in practice.

So far we have covered only the syntax of boolean logic. The semantics of boolean logic is provided by interpretations. An interpretation is an assignment of a truth value, TRUE or FALSE, to each of the boolean variables. Expressions are evaluated using the standard truth tables for the connectives.

Expressions that evaluate to TRUE in all interpretations are called tautologies, for example $(a \longrightarrow b) \longleftarrow (b \longleftarrow a)$. Those that are FALSE in all interpretations are called contradictions. Otherwise a formula is said to be contingent: in this case some assignments, but not all, will cause the expression to evaluate to TRUE, such assignments are called “satisfying assignments” or “models”. (Note that in the literature, but *not* in this thesis, the term “model” is also used in the sense that a theory is a good model, or representation of some real problem.)

For a formula in CNF a satisfying assignment will make every clause in the list contain at least one literal that is assigned to TRUE. Deciding whether or not a CNF formula has a satisfying assignment is usually called SAT (for satisfiability):

Definition 2.1.1

SAT is the following decision problem:

INSTANCE: A set of clauses Γ , that is, a boolean formula in CNF.

QUESTION: Does Γ have a satisfying assignment? □

SAT is NP-complete [8], and hence (as far as we know) requires search methods to solve. We will discuss some of the best of the search methods in Chapter III. For a discussion of NP-completeness see, for example, the excellent books by Garey and Johnson [22] or Papadimitriou [51].

There are tractable classes of SAT problems [61], the best known being

1. 2-SAT: all clauses have length at most 2, i. e. binary or unary.
2. Horn: all clauses contain at most 1 positive literal.

Our results will not assume such restrictions on the clauses used. It might happen that an example is Horn or binary but, unless we note otherwise, this will not be relevant.

2.2 Quantified Propositional Logic (QPROP)

In practice, when representing a problem, we often want more than simple boolean variables. We might also want propositions having internal structure, quantifiers and functions. This naturally puts us into the area of first-order logic (FOL). However, we will also want a language that allows relatively direct use of the very effective methods developed to solve boolean problems. This suggests using a language that lies between full FOL and simple boolean logic.

Hence, in this section we will review the basics of FOL and then start to describe the subset that we wish to use. This subset will retain the important features of having predicates and quantification. This thesis is concerned with developing methods to handle the quantification. In fact, we shall further restrict the subset we use in Chapter IV, although in Chapter XI (see Section 11.3) we consider relaxing some of those restrictions.

2.2.1 First-Order Logic (FOL)

Firstly we briefly, and rather informally, review some of the essentials of first-order logic. We assume familiarity with FOL, but are aiming to pick out the

particular features that we shall need.

In first-order logic the role of boolean variables is replaced by *predicates*, the main feature of which is that they are no longer atomic but have internal structure: they consist of a predicate symbol followed by a fixed number of arguments. The number of arguments is called the *arity*. If the arity is k then we say the predicate is k -ary, the special cases of $k = 0, 1, 2$ are called nullary, unary and binary respectively.

Each argument of a predicate is a *term*. Terms are built out of constants, variables and function symbols in an inductive fashion. We use uppercase letters for constants and lowercase for variables. The number of arguments of a function is called the arity just as for predicates. For example, $f(i, g(H))$ is a term formed from the variable i , constant H , binary function f and unary function g . Note that functions can be nested arbitrarily, and hence there are an infinite number of syntactically different terms. A variable is merely a placeholder for a term, and ultimately will be bound to some term. A predicate or function containing no variables is said to be *ground*, for example $g(H)$.

A logical expression is formed from the same logical connectives as in the boolean case but, (assuming we have non-nullary predicates) logical expressions can also involve terms. Since we have an infinite number of terms, it follows that we cannot make a statement about “all bindings of a variable” by any finite combination of the boolean connectives. Accordingly, FOL contains quantification over variables: universal quantification means something is TRUE for all bindings, and existential quantification means that at least one binding of the variable causes satisfaction of the subexpression.

The semantics of the theory is given by its interpretations. An interpretation gives meanings to the various symbols. It consists of a universe, a mapping of function symbols to functions in the universe, and a mapping from constants in the logic to elements of the universe. These give a mapping from every ground term to an element of the universe. Finally, the interpretation also contains a truth assignment for every ground predicate. An interpretation satisfies an expression iff the expression evaluates to TRUE under the interpretation.

If we are given a set of axioms in FOL, then a theorem is an expression that is satisfied in *any* interpretation satisfying the axioms. Many different universes might be used. For example, the standard (Peano-based) axiomatization of arithmetic is to have a single constant '0', a unary function symbol s for successor, and a binary function plus for addition. We then have axioms such as $\forall x. s(x) \neq 0$. The natural numbers correspond to the terms $0, s(0), s(s(0)), \dots$ and this is the natural choice for the universe. However, it is well-known there are interpretations, and associated universes, that contain more than the natural numbers. We do not know, a priori, that variables will only be bound to natural numbers. One way to avoid such freedom of choice of universe is to inductively define the intended universe and make all candidate theorems conditional on the elements being within the desired universe. This corresponds to the addition of (second-order) induction rules to the theory. In the case of the Peano axioms these rules essentially make the statement that we do not care about anything that is outside of the natural numbers, and so can restrict our universe accordingly. However, such an approach is beyond what we need to solve the combinatorial problems with which we are concerned. Instead, it is better to restrict the universe by fiat, and a standard and

useful way to do this is to use finite sorts.

2.2.2 Finite Sorted Logic

In combinatorial problems we will typically know our universe in advance. This is clearly true in NP-complete problems such as graph coloring. It can also be true in harder problems. For example, if we are encoding planning problems then we can assume we know in advance all the objects in the domain. Accordingly, we can restrict interpretations to some fixed and known universe. We can also restrict the universe to be finite, so that we can describe it explicitly rather than having to use inductive descriptions.

In practice, it is standard to use a sorted logic: instead of just one finite domain, with quantification over that entire domain, we have multiple domains or “sorts”. All domains or sorts have a fixed finite size. We fix the sort of each argument of the predicates, and similarly for the arguments and “return sort” of functions. Interpretations will implicitly be restricted to being consistent with the sorts.

Quantification of a variable is then just quantification over the finite domain associated with a sort. Since we have at hand all possible elements of the domain, the quantifiers need only apply to the constants from the domain, and do not need to also bind to functions. When quantifying a variable x over a domain $D = \{d_1, d_2, \dots, d_n\}$ we can write $\forall x : D$ or $\exists x : D$ and then we have

$$\begin{aligned} \forall x : D. A[x] &\equiv \bigwedge_{d \in D} A[x/d] \equiv A[d_1] \wedge A[d_2] \wedge \dots \wedge A[d_n] \\ \exists x : D. A[x] &\equiv \bigvee_{d \in D} A[x/d] \equiv A[d_1] \vee A[d_2] \vee \dots \vee A[d_n] \end{aligned}$$

(2.1)

where A is any logical expression involving the variable x (and possibly other variables), and where x/d means the domain variable x is bound to (substituted with) the value d from the domain D . The big \wedge and \vee or are not symbols in the language but just shorthand for the actual expansions. Usually we drop the “: D ” from the quantifiers; it will be clear from the context.

Sorted logics, with their fixed finite domains and quantification over elements of a sort, have provided a natural way to express many AI problems. However, although they retain much of the notation of FOL they are really very different beasts. The restriction from FOL to finite sorts greatly reduces the expressiveness of the language, but in return we avoid the semi-decidability of full FOL. In particular, since the universe is finite, we have a finite number of interpretations, and everything is decidable: We could, in principle, test the truth of any statement by enumerating all possible interpretations.

Since the domains are finite, we can further assume that functions are “concrete” rather than acting as “constructors” meaning that any ground function will be interpreted by some existing domain element rather than as a new element of the universe. In the familiar arithmetic context (or modular arithmetic if we want finite sorts), instead of regarding $s(0)$ as constructing a new element, it is interpreted as the pre-existing constant, ‘1’.

2.2.3 Converting "Free" Functions to Predicates

So far, we still allow expressions such as

$$\forall i, j. p(j) \longrightarrow f(K, g(i)) \neq h(j) \quad (2.2)$$

which contains boolean quantities such as $p(j)$ but also functions such as $g(i)$. Finding an interpretation satisfying such axioms means finding values for the booleans, but can also mean finding values for the functions. However, we are aiming to use boolean based methods, and so any functions not having a known and fixed interpretation would seem to block this. Solving a problem should consist *only* of finding truth assignments for propositions. We will say that a function that does not have a known fixed interpretation is a "free" function. We will restrict ourselves to expressions that have no free functions. Predicates with fixed interpretations, such as the inequality in (2.2) are also allowed.

The absence of free functions is not a restriction on the expressiveness of the language, as we can convert any expressions with functions to ones with no functions. For example, given a function $f(i)$, we introduce a new predicate $p_f(i, k)$ with the intended meaning:

$$\forall i, k. [p_f(i, k) \longleftrightarrow f(i) = k] \quad (2.3)$$

We then add the axioms:

$$\forall i. \exists k. p_f(i, k)$$

$$\forall i, k_1, k_2. k_1 \neq k_2 \longrightarrow \neg (p_f(i, k_1) \wedge p_f(i, k_2)) \quad (2.4)$$

Any predicate satisfying these axioms defines a function from i to k . Given any logical expression A involving f we can remove an occurrence of f by using

$$A[f(i)] \longleftrightarrow \forall k. p_f(i, k) \longrightarrow A[k] \quad (2.5)$$

where k is a new variable. This allows conversion of all functions to predicates (a process called “relationizing the function”), at the cost of adding new predicates and axioms to constrain the new predicates.

Note, that at this point we have only demanded that “free” functions be relationized. If a function has a fixed interpretation (for example, a fixed permutation of elements in a domain) then we will see that it need not cause difficulties. We can also relationize such fixed functions, but then the new predicates are also totally known and hence need not be constrained by equations such as (2.4). Since fixed functions do not add new constraints they are fairly harmless.

However, even if we do allow such fixed functions there is one slight wrinkle in that we will often want to use partial functions as well. For example, it might be convenient to write $\forall i. p(i, i + 1)$ even though $i + 1$ might not be within the relevant domain. We allow such cases as long as the relevant quantifiers can be unambiguously restricted so that all terms are within their needed domains. In the case of $i + 1$ we would just exclude the largest element from the range of i . This is not essential but helps retain the clarity of expressions.

2.2.4 Definition of QPROP

We can now define Quantified Propositional Logic (QPROP)

Definition 2.2.1

QPROP is the subset of FOL with:

1. Finite sorts, and hence finite quantifications over domain variables. (We implicitly also impose closure in the sense that the finite sorts used are the only elements in the universe).
2. No free functions – any function not known exactly and fully in advance must have been relationized. Fully fixed functions are allowed.

□

As mentioned previously we will allow partial functions as long as the ranges of quantifiers are (possibly implicitly) restricted so that the functions exist whenever they are needed.

At this point we are *not* restricting the theory to be clausal. However, we will make such a restriction from Chapter IV onwards because we want to apply boolean methods designed for CNF formulas.

For reasons of simplicity of implementation, we will eventually demand that all functions are relationized. However, for the moment we still allow fixed functions because:

1. They aid in readability.

2. A future solver might quite reasonably have access to a fixed list of such known functions. It could exploit them to save extra reasoning required if they were converted to relations. Basically, we want to leave open the option for using simple procedural attachments [34].

We emphasize that we are not inventing a new language. QPROP has often been used before. We are merely giving it a name to bring out its strong connection to propositional logic, and to convey our view that it is a language worthy of study in its own right.

Before moving onto the connection of QPROP to boolean logic, we first clarify a potential source of confusion.

2.2.5 Different Orders of Quantifiers

The quantifiers used in QPROP are quite different from those used in Quantified Boolean Formulas (QBF) (see, for example, page 171 of Garey and Johnson [22]), such as

$$\forall p. \exists r. p \vee r \tag{2.6}$$

where the domains of the quantifiers are TRUE and FALSE for the *values* of x and y . To see the difference more clearly think of a boolean variable as a nullary predicate. Then the quantifications in QBF are second order quantifications over predicates:

$$\forall_2 p(). \exists_2 r(). p() \vee r() \tag{2.7}$$

where the subscript denotes the order of the quantifier. Of course, QBF does not have the undecidability of full second-order logic because we have only nullary predicates, and do not have any functions or first-order quantification. In contrast to the predicate quantification in QBF, the quantification in QPROP is over domain elements. Similarly, the “variables” in boolean logic are boolean variables whereas those in QPROP are domain variables.

2.3 Grounding QPROP to Boolean Logic

QPROP has only quantifiers over finite domains and no free functions. It is therefore straightforward to take a QPROP expression and convert it to a logically equivalent boolean formula. This process is called grounding and has two distinct stages:

1. Expand quantifiers, using (2.1), while simultaneously evaluating the fixed functions. The result of this is that all the arguments of predicates will reduce to constants.
2. “Linearize” the propositions. That is, define a list of boolean variables, one for every ground proposition remaining, and replace the propositions.

The restrictions on functions in QPROP were designed so that the first step results in a theory with no functions remaining.

Given a QPROP-formula ϕ we shall use $Gr[\phi]$ to refer to the results of just expanding quantifiers and evaluating function. It is a propositional logic theory: It has propositions, i.e. atoms (ground predicates such as $p(3,4)$) but not quantification. The final result after linearization will be called $GrL[\phi]$ and is just a

boolean formula.

The initial QPROP formula need not be clausal: grounding will just produce a non-clausal boolean formula. As a simple example, consider a case with a single domain $D = \{1, 2, 3\}$:

$$\phi = \forall i : D. p(i) \longrightarrow (p(i+1) \wedge r(i))$$

Expanding the quantifier, and dropping the case $i = 3$ because $i+1$ is just a partial function we get the conjunct of two expressions:

$$\begin{aligned} Gr[\phi] = & [p(1) \longrightarrow (p(2) \wedge r(1))] \wedge \\ & [p(2) \longrightarrow (p(3) \wedge r(2))] \end{aligned}$$

Finally, linearizing is just a relabelling of the remaining propositions, using a set of boolean variables v_i , by $p(1) = v_1$, $p(2) = v_2$, $r(1) = v_3$, $p(3) = v_4$ and $r(2) = v_5$.

This gives us

$$\begin{aligned} GrL[\phi] = & [v_1 \longrightarrow (v_2 \wedge v_3)] \wedge \\ & [v_2 \longrightarrow (v_4 \wedge v_5)] \end{aligned}$$

which is a (non-clausal) boolean formula.

Finally, suppose that

$$\phi = \forall i_1, \dots \exists j_1, \dots A[i_1, \dots, j_1, \dots] \quad (2.8)$$

where $A[i_1, \dots, j_1, \dots]$ is a disjunction of literals. Then the grounding, $GrL[\phi]$, will be a CNF formula which we can refer to as the “ground theory of ϕ ”. We will say that a conjunction of such expressions is a clausal QPROP formula.

2.4 The Pigeonhole Problem (PHP)

Before looking at inference methods in the next chapter, we present examples of QPROP formulas that can arise when encoding a domain. The aim is to increase familiarity with QPROP and also to make sure that the language we eventually target has a reasonable chance of being useful.

In this section we consider a well-studied problem that is often used for illustrational purposes. Suppose that you have n pigeons and m pigeonholes with each pigeonhole being able to hold at most one pigeon. The “pigeonhole problem” is to find an assignment of pigeons to holes such that every pigeon gets a hole.

A natural representation is to use a unary function f with the meaning that pigeon i will go in hole $f(i)$, and then the only constraint is

$$\forall i, j. f(i) \neq f(j) \tag{2.9}$$

where $i, j \in \{1, \dots, n\}$. However, as we discussed earlier, we do not want to use such unknown functions, but instead their relational equivalents. Hence, we use a predicate $p(i, h)$, with $h \in \{1, \dots, m\}$, and with the intended meaning

$$\forall i, h. p(i, h) \iff f(i) = h \tag{2.10}$$

that is, pigeon i is in hole h . In this case we have constraints on $p(i, h)$ that a

pigeon is assigned to precisely one hole, and also a constraint that no two pigeons are assigned to the same hole. In total we have

$$\begin{aligned}
 & \forall i. \exists h. p(i, h) \\
 & \forall i, h_1, h_2. h_1 \neq h_2 \longrightarrow \neg [p(i, h_1) \wedge p(i, h_2)] \\
 & \forall i_1, i_2, h. i_1 \neq i_2 \longrightarrow \neg [p(i_1, h) \wedge p(i_2, h)]
 \end{aligned}
 \tag{2.11}$$

which we will refer to as the PHP(n, m) problem.

In fact, “PHP” often refers to the “propositional pigeonhole principle”: the fact that the case PHP($n, n - 1$) is manifestly unsatisfiable. It has mostly been of interest because any binary resolution proof based on the ground theory obtained from (2.11) has a size that is at least exponential in n [29] (a somewhat distressing reflection on the power of ground resolution).

2.5 SATPLAN

As we discussed in Chapter I, one of the initial motivations for this work arose from the satisfiability-based approach to declarative planning [37, 38, 39]. This approach, commonly referred to as SATPLAN, expresses a planning problem as a problem in SAT. This has the advantage that we can use an “off-the-shelf” SAT solver that does not have to contain planning specific knowledge. However, as we shall see, it has the difficulty that the formulas involved can get too large to be practical.

In this section, we will briefly summarize SATPLAN’s pertinent features.

We also give a detailed description of one particular problem domain, and give a representation of it in QPROP using a standard SATPLAN technique. The encoding presented will be used as a standard example throughout the rest of the thesis. We shall see that even a modestly sized instance of the encoding can give very large SAT formulas.

2.5.1 Encoding Planning Problems

SATPLAN has been directed mostly at problems representable in simple propositional STRIPS format [20]. We have a finite world in which a finite number of actions can be applied to change the world state. The state is described by “fluents”, which are just propositions that can be time dependent (see standard AI textbooks [23, 59, and others] for introductions to planning and further references).

In STRIPS format an action has the following components:

1. Preconditions – logical expressions that must hold immediately before the action is applied, otherwise the use of the action is illegal.
2. Effects – changes to the state as a result of the action:
 - (a) fluents added, i. e. fluents forced to hold immediately after the action
 - (b) fluents deleted, i. e. fluents forced not to hold after the action

The advantage of the STRIPS representation is that it just specifies changes in fluents. Fluents never affected by the action need not be mentioned in the action.

We are only going to use SATPLAN as an example of the use of QPROP and so we only need the basic idea: The problem of finding plans can be reduced to a problem of finding satisfying assignments for a boolean formula. Usually, but

not necessarily, the final formula is in CNF, so finding a plan means solving a SAT problem: hence the name SATPLAN. This declarative approach to planning relies on two key components:

1. By bounding the numbers of actions in a plan we can have a finite theory, and hence describe the space of possible plans in terms of a finite number of propositions.
2. The allowed actions, initial, and final states, and other consistency conditions on states, can all be translated into logical axioms. Satisfying assignments corresponding to legal plans.

The key point is that the relevant universe is fixed and finite. For example, in the logistics domain described in the next section, the numbers of planes, cities and timepoints are all fixed, finite, and known in advance. It is hence natural to use FOL with finite sorts, and so it should not be surprising that SATPLAN is also a rich source for QPROP instances that capture structures likely to be present in real problems.

The requirement of finite fixed domains does have the drawback that the SATPLAN approach requires specifying the maximum plan length or number of timepoints in advance, and this is rather unnatural for planning. Other planning systems would produce the number of timepoints or actions needed for an optimal (shortest) plan as part of the output. Hence, SATPLAN needs a secondary search on such parameters as the number of timepoints/actions, but this extra search is orthogonal to the issues in this thesis.

We briefly mention some of the different ways to encode a planning problem

(we refer the reader to the SATPLAN literature for explanations of the terms used):

1. Total-order state-based methods (see [38, 39]). These allow removal of direct reference to the actions by use of explanation closure: actions might be compiled out based on an approaches to the frame problem by Schubert [62] and Reiter [56]. Rather than giving a general description we will shortly present an example we will be using throughout the thesis.
2. Explicit actions, possibly partial order and with causal links (POCL) [4].
3. Encodings of GRAPHPLAN [5] based approaches to planning, e. g. BLACK-BOX [36].

Note that encodings do not necessarily directly produce CNF-formulas. For example, POCL has been encoded into a single “universal” axiom which is not in CNF [4]. It can easily be easily be converted to CNF but we do not know if ultimately that would be best. This axiom is one of the reasons we preferred to first describe general QPROP rather than moving directly to a clausal form.

As mentioned in the introduction, the term “lifted” is somewhat overloaded. In the context of SATPLAN, the term “lifting” has also been used to refer to a form of predicate splitting [37]. For example, if only one value of $p(x, y)$ should hold in a solution then we can split this into $p_1(x) \wedge p_2(y)$. This reduces the number of boolean variables from $|x||y|$ to $|x| + |y|$, and so has the same effect of reducing the size of the formulas. However, at some point we cannot split any further and also the result will still have quantifiers. Such predicate splitting is

quite distinct from the notion of lifting used in this thesis: The expressions before and after predicate splitting might well still be written in QPROP.

2.5.2 A Logistics Example

The simple example we use throughout the remainder of the thesis is derived from the “rockets problem” commonly used in SATPLAN [39], but in a rocket-less form used by Joslin and Roy [35]. The domain concerns moving objects around between cities by loading them onto planes. In this case, the relevant domains being finite means that we have fixed (finite) numbers of planes, packages and cities, and a finite time to get the necessary tasks completed.

We describe the fluents, that is the positions of the planes and objects, by means of the following predicates:

$$\text{in}(\text{object}, \text{airplane}, \text{time}) \quad (2.12)$$

$$\text{at}(\text{object}, \text{location}, \text{time}) \quad (2.13)$$

$$\text{planeAt}(\text{airplane}, \text{location}, \text{time}) \quad (2.14)$$

with the intended meanings

$$\text{in}(O, P, I) \iff \text{object } O \text{ is in plane } P \text{ at time } I$$

$$\text{at}(O, C, I) \iff \text{object } O \text{ is in city } C \text{ at time } I$$

$$\text{planeAt}(P, C, I) \iff \text{plane } P \text{ is in city } C \text{ at time } I$$

The allowed actions are

1. LOAD: put an object on a plane. This takes one time unit and the plane must be at the city both before and after the loading, that is, a minimum of two timepoints.
2. FLY: planes take just one time unit to move between any two cities.
3. UNLOAD: time-reversed version of LOAD.

Actions are allowed to take place in parallel. The domain is encoded using a state-based method in which actions are represented in terms of the allowed changes of state. Such a method seems well suited to cases that can have a lot of actions occurring in parallel.

The axioms of a theory should have no free variables. Also, typically, many of the variables will be universally quantified. Hence, we shall start using the convention that any domain variables not explicitly quantified are implicitly universally quantified over the relevant domain.

To simplify the encoding we use two auxiliary predicates defined by:

$$\text{inSomePlane}(o, i) \iff \exists a. \text{in}(o, a, i) \quad (2.15)$$

$$\text{atSomeCity}(o, i) \iff \exists c. \text{at}(o, c, i) \quad (2.16)$$

though we will (eventually) be using these constraints in the clausal form given in Figure 1.

State consistency axioms, such as the restriction that planes can only be in one place, are given in Figure 2. Finally, the actions lead to constraints on the changes of state that are given in Figure 3. For example, the fact that loading

$$\begin{aligned} \text{inSomePlane}(o, i) &\longrightarrow \exists a. \text{in}(o, a, i) & (2.17) \\ \text{inSomePlane}(o, i) &\longleftarrow \text{in}(o, a, i) & (2.18) \\ \text{atSomeCity}(o, i) &\longrightarrow \exists c. \text{at}(o, c, i) & (2.19) \\ \text{atSomeCity}(o, i) &\longleftarrow \text{at}(o, c, i) & (2.20) \end{aligned}$$

FIGURE 1. Logistics example: axioms defining the auxiliary predicates.

Planes are always in at least one city:

$$\exists c. \text{planeAt}(p, c, i) \quad (2.21)$$

Planes are never in two cities:

$$a < b \longrightarrow \neg \text{planeAt}(p, a, i) \vee \neg \text{planeAt}(p, b, i) \quad (2.22)$$

Objects are never in two cities:

$$a < b \longrightarrow \neg \text{at}(o, a, i) \vee \neg \text{at}(o, b, i) \quad (2.23)$$

Objects are never in two planes:

$$a < b \longrightarrow \neg \text{in}(o, a, i) \vee \neg \text{in}(o, b, i) \quad (2.24)$$

Objects are either in a city or a plane, but not both

$$\text{atSomeCity}(o, i) \vee \text{inSomePlane}(o, i) \quad (2.25)$$

$$\neg \text{atSomeCity}(o, i) \vee \neg \text{inSomePlane}(o, i) \quad (2.26)$$

FIGURE 2. Logistics Example: Axioms expressing consistency of the state.

Objects stay in location or are loaded:

$$\text{at}(o, c, i) \longrightarrow \text{at}(o, c, i + 1) \vee \text{inSomePlane}(o, i + 1) \quad (2.27)$$

Only load planes at same location:

$$\text{at}(o, c, i) \wedge \text{in}(o, a, i + 1) \longrightarrow \text{planeAt}(a, c, i) \quad (2.28)$$

$$\text{at}(o, c, i) \wedge \text{in}(o, a, i + 1) \longrightarrow \text{planeAt}(a, c, i + 1) \quad (2.29)$$

Objects stay in plane or are unloaded:

$$\text{in}(o, a, i) \longrightarrow \text{in}(o, a, i + 1) \vee \text{atSomeCity}(o, i + 1) \quad (2.30)$$

Only unload planes at same location:

$$\text{in}(o, a, i) \wedge \text{at}(o, c, i + 1) \longrightarrow \text{planeAt}(a, c, i) \quad (2.31)$$

$$\text{in}(o, a, i) \wedge \text{at}(o, c, i + 1) \longrightarrow \text{planeAt}(a, c, i + 1) \quad (2.32)$$

FIGURE 3. Logistics example: axioms defining the allowed actions. These arise from consideration of the allowed actions.

an object onto a plane requires the plane to be at the airport for two consecutive timepoints is expressed via (2.28) and (2.29).

Note that (2.28) has implicit quantifiers $\forall o, c, i, a$. Suppose that we use the same symbols for the domain size as for the domain itself: there are o objects, etc. Then, on grounding this axiom will give $oca(i - 1)$ different ground clauses. If we have a target of dealing with a few hundred objects, a hundred cities and planes and maybe 20 timepoints, we obtain about 40 million ground clauses from just the one axiom. Clearly, even a simple domain and modestly sized problems can generate very large numbers of ground clauses. On the other hand, we have just seen that the axioms in QPROP are very small: it is very natural to want to work with them directly.

2.6 Summary

After reviewing standard boolean logic and first-order logic, we observed that many combinatorial problems are likely to be best represented in some intermediate logic. We hence defined QPROP to mean first-order logic with finite sorts and only pre-defined totally-specified functions. Its main feature is that it allows quantification over domain variables, yet is logically equivalent to boolean logic, and furthermore occurs naturally in many problems.

We described the conversion process, “grounding”, that is typically used to convert problems in QPROP to standard propositional satisfiability problems. Grounding has the strong disadvantage of greatly increasing the size of the theory.

Most of the thesis will be concerned with a clausal subset of QPROP, and how lifted solvers that exploit its structure can gain advantages over ground solvers.

However, first we review algorithms used for ground solvers, in particular for the usual ground CNF-formulas of SAT.

CHAPTER III

STANDARD INFERENCE METHODS IN SAT

In the previous chapter, we reviewed boolean and first order logic. We defined QPROP, with the goal of merging the best boolean solution methods with the quantifiers that seem essential to reasonable representations. We also briefly discussed SATPLAN as an area that already naturally uses QPROP as an intermediate language, although the formulas are grounded out to SAT before usage.

We now review the methods used to solve problems expressed in ground clausal form, that is SAT. The ultimate goal is to take such methods and lift them, that is, to make them work for QPROP, or a subset of QPROP.

There are two basic approaches towards solving problems such as SAT: model search and proof search. In model search the focus is trying to build a satisfying assignment: The state of the search is given by candidate assignments of values to variables. Proof search is complementary: instead of looking for satisfying assignments we look for extra formulas or constraints that are entailed by the initial problem, and The current state of the search is primarily described by the database of constraints. We discuss model search and proof search in turn, and finish with some methods that are perhaps best described as a mix of these methods.

3.1 Model Search

In model search, the focus is on trying to build a satisfying assignment by searching through the space of candidate assignments. We shall typically use “ P ” to refer to the assignment currently under consideration. Assignments can be

1. Total: every variable is assigned TRUE or FALSE.
2. Partial: some variables might not be assigned a value, or equivalently can be regarded as having been assigned the value UNVALUED.

A total assignment is a model iff it satisfies all the constraints. In the case of SAT every clause must contain at least one literal that is assigned to TRUE.

The split of assignments into partial and total also corresponds to a split in the two main kinds of model-search algorithm. In iterative repair algorithms we work with total assignments that violate some constraints and iteratively try to repair them. In backtracking, or what we might call “branch-and-propagate” algorithms, we take a partial assignment and try to build up a satisfying total assignment.

Most of the thesis is concerned with lifting a particular iterative repair algorithm called WSAT, and hence we shall treat it first. Also, since the focus is on obtaining fast lifted *implementations* of the algorithms, we shall eventually have to go into implementational details that would normally be ignored.

3.2 Iterative Repair

The general form of iterative repair is a simple loop:

```
initialize P  
while P is not an acceptable solution  
    identify the deficiencies in P  
    select and apply a small modification to P to repair some deficiencies  
end while  
return best P seen
```

Of course the generic idea has been developed in many different ways. We will just mention a few, rather than attempting a survey. Generally, choices are made with a strong hill-climbing flavor, after all we do want to improve the solutions. However, pure hill-climbing tends to get trapped in local minima¹, so many other heuristic and stochastic methods are also used. In AI one success was MIN-CONFLICTS [45, 46]. There are also methods associated with operations research, such as simulated annealing [40], and Tabu search [28]. There are many other examples [54, and others]. Iterative repair algorithms have the deficiency of being incomplete: it is possible that they will not discover a solution even if one exists. However, they can work well on large problems for which complete search methods are ineffective in practice.

Many iterative repair methods have typically only been used in a rather domain-specific fashion, that is, for problems that are not used as a general-purpose representational scheme. Instead, we are concerned with domain-independent methods for SAT and the main iterative repair methods in this case are the GSAT

¹We use the convention that “smaller is better” (coincidentally, a rather appropriate theme for the thesis itself). Accordingly, hill-climbing is trying to go downhill.

```

proc Pure Random Walk
  P := random total assignment
  for i := 1 to MAX-FLIPS
    S := set of unsatisfied clauses
    if S =  $\emptyset$  return P                                Found a solution
    c := randomly selected clause from S
    l := randomly selected literal in c
    P := P[l :=  $\neg$  l]                                Flip value of l in P
  end for
  return solution-Not-Found
end

```

FIGURE 4. Pure random walk for SAT.

and WSAT family of algorithms. In fact we shall just concentrate on WSAT as it has proven to be one of the most effective methods for SAT.

3.3 WSAT (WalkSAT)

WSAT, and other algorithms in the GSAT family, work with a total truth assignment P and attempt to obtain a solution by repeatedly flipping values in P [66, 64]. That is, they heuristically select a variable and then flip (negate) its current value in P . Flips are made with the general intent of reducing the number of unsatisfied clauses.

3.3.1 Pure Random Walk

As a simple introduction to the ideas of WSAT consider the pure random walk in Figure 4. The first step is to create a random total assignment. Unless we are extremely lucky, this will leave some clauses unsatisfied – all of their literals

will be FALSE. We then randomly select one of these unsatisfied clauses and flip a randomly selected literal from within that clause. This guarantees that the selected clause will be become satisfied, that is, it will have been repaired. We repeat this for some specified number of flips, MAX-FLIPS. The name WSAT, or WalkSAT, arises because the selection of random clauses can be regarded as doing a random walk on the unsatisfied clauses. This simple algorithm works surprisingly well on 2SAT – it will find a solution, if one exists, in (expected) quadratic time [50]. Of course, 2SAT is tractable, and for this to work well on general intractable SAT we need to include powerful heuristics.

3.3.2 Heuristics in WSAT

The basic practical WSAT method [67] is illustrated in Figure 5, with supporting routines given in Figures 6, 7, and 8.

```

proc WSAT
  for t := 1 to MAX-TRIES                                Independent tries
    P := random assignment
    S := set of unsatisfied clauses
    for i := 1 to MAX-FLIPS
      if S =  $\emptyset$  return P
      c := randomly selected clause from S
      l := SELECT-LITERAL(c, S, P)                       Heuristic variable selection
      P := P[l :=  $\neg$  l]
      UPDATE(S, l)                                       Incrementally maintain S
    end for
  end for
  return solutionNotFound
end

```

FIGURE 5. The WSAT algorithm. The subroutines SELECT-LITERAL(c, S, P) and UPDATE(S, l) are defined in figures 6 and 8 respectively.

```

proc SELECT-LITERAL( $c, S, P$ )
  foreach  $l \in c$ 
     $b[l] := \text{NUM-BREAKS-IF-FLIP}(l)$ 
  end foreach
   $L_0 := \{l \mid b[l] = 0\}$ 
  if  $L_0 \neq \emptyset$  return random choice from  $L_0$ 
  with prob  $p$ :  $l :=$  randomly selected literal in  $c$ 
  else
     $l :=$  literal in  $c$  such that  $b[l]$  is a minimum
  return  $l$ 
end

```

Every such l will be FALSE in P

(a)

```

proc select-literal( $c, S, P$ )
  foreach  $l \in c$ 
     $b[l] := \text{NUM-BREAKS-IF-FLIP}(l)$ 
     $m[l] := \text{NUM-MAKES-IF-FLIP}(S, l)$ 
  end foreach
  with prob  $p$ :  $l :=$  randomly selected literal in  $c$ 
  else
     $l :=$  literal in  $c$  such that  $m[l] - b[l]$  is a maximum
  return  $l$ 
end

```

(b)

FIGURE 6. Two variable selection methods in WSAT. S is the current set of unsatisfied clauses, and c is one unsatisfied clause from S . p is a noise parameter supplied to the algorithm. The subroutines NUM-BREAKS-IF-FLIP(l) and NUM-MAKES-IF-FLIP(S, l) are defined in Figure 7. (a) Version based on minimizing the amount of damage (note that S is not used). (b) Greedier version that tries harder to move downhill: $m[l] - b[l]$ is the net decrease in the number of unsatisfied clauses if we should flip l .

Firstly, the random walk aspect is retained, that is, clauses are still selected randomly. However, the selection of a literal to be flipped is done using hill-climbing: that is, flips that reduce the number of unsatisfied clauses are encouraged.

Particular literal selection methods are given in Figure 6, with support from

```

proc NUM-BREAKS-IF-FLIP( $l$ )
   $count = 0$ 
  foreach clause  $c$  containing  $\neg l$ 
    if (number of true literals in  $c$ ) = 1 then  $count := count + 1$ 
  end foreach
  return  $count$ 
end

```

(a)

```

proc NUM-MAKES-IF-FLIP( $S, l$ )
   $count = 0$ 
  foreach  $c \in S$ 
    if  $l \in c$  then  $count := count + 1$ 
  end foreach
  return  $count$ 
end

```

(b)

FIGURE 7. Counting makes and breaks in WSAT. The input literal l will have been taken from an unsatisfied clause, and so both procedures assume that l is currently FALSE in P . (a) Counting breaks: Scans for clauses in which $\neg l$ is the only TRUE literal, and hence will break if l is flipped. (b) Counting makes: Scans the set S of currently unsatisfied clauses looking for those that contain l .

```

proc update( $S, l$ )
   $S := S - \{c \mid c \in S \text{ and } l \in c\}$ 
   $S := S \cup \{c \in C \mid l \in C \text{ and } \neg l \text{ was the only TRUE literal}\}$ 
end

```

Remove makes
Add breaks

FIGURE 8. Updating the unsatisfied clause set in WSAT in response to flipping l to TRUE.

the routines in Figure 7. We will be concerned with what happens when a literal is flipped, and it will be useful to think in terms of

1. "Makes": clauses that change from satisfied to unsatisfied.
2. "Breaks": clauses that change from unsatisfied to satisfied.

The heuristics for selection are then based on finding the numbers of makes and breaks that would occur if we were to flip the value of some literal. The net effect on the number of unsatisfied clauses is just the difference

$$(\text{number of makes}) - (\text{number of breaks}).$$

The most obvious heuristic, Figure 6(b), is to select variables that tend towards to maximizing this difference. The less obvious selection scheme of Figure 6(a) instead tries to avoid breaking any new clauses, and so selects on the basis of minimizing the number of breaks. In fact, from now on we shall just discuss and use the version of Figure 6(a), though extension of our work to (b), or many other selection schemes, should be straightforward.

Of course, we have the usual hill-climbing problem of getting stuck on local minima. One way to counteract this is for the variable selection to be noisy, and occasionally be allowed to increase the number of unsatisfied clauses. This is the purpose of the noise parameter p in the procedures of Figure 6. Again, we will not be concerned with this here (see the work of McAllester *et. al.* [44] for more information). A second way to counteract the problem of local minima is simply to restart the algorithm every so often. The restarts are controlled by the outer loop in Figure 5 over the "tries", and the associated parameter MAX-TRIES. There are

various issues concerned with restarts, such as selecting the best time to restart [30, 52, and others], but again we are not concerned with these here.

Eventually, we shall need to be concerned with the details of how these steps are really implemented. The vital features are to use indexing schemes, incremental methods, and caching of useful data. For example, typically, a large fraction of the runtime is spent in the NUM-BREAKS-IF-FLIP routine of Figure 7(a), and one way to support this is to store, *for every clause*, a count of how many literals in the clause are currently TRUE. Having this count means that the “if” test in Figure 7(a) can be done by an array lookup. Without this count, we would need to do a scan of the clause with the associated costs of looking up the values of literals in P . This count of TRUE-literals is updated after every flip. These methods will be described in more detail when we lift the algorithm in Chapters V and VIII.

It is also very important for efficiency that the set of unsatisfied clauses is not recalculated each time (though we did this for simplicity in Figure 4). Instead, it is updated incrementally after each flip (see Figure 8). On flipping l to TRUE there are two cases to consider: clauses that are made and clauses that are broken. The only clauses that can be made are those already in S and containing l : these can be removed by simply scanning the set S . Clauses that break are those which contain $\neg l$ and for which $\neg l$ is the only true literal. The search engine must find these and add them to the set S . Finding the broken clauses is a vital step in implementing WSAT and we shall revisit this issue for the lifted version in Section 5.6.

Historically, WSAT originated from the GSAT (Greedy SATisfiability) family of algorithms. These are similar except that, instead of restricting attention to literals from a single clause, GSAT algorithms consider all variables from all unsatisfied

clauses. These algorithms tend to be less effective, and so we shall not consider them further, however the similarity of their implementation means that the ideas in this thesis should extend to GSAT also.

3.3.3 Extensions to WSAT

The basic ideas in WSAT and GSAT have been extended to other languages than SAT. WSAT has been used for operations research representations such as pseudo-boolean [72] and integer programming [73]. GSAT has been applied to non-clausal formulas [63]; we will discuss this work in Section 10.2.

There are also various extensions designed to avoid local minima. It is possible to weight clauses so as to affect the probability of them being selected [65]; we will briefly consider ways to lift this in Section 11.2.2. Another way to escape local minima is to “fill them in” by adding new clauses [6, 74]. We shall also briefly consider lifting this in Section 11.2.2. Otherwise, we will just use the simple WSAT described above.

3.4 Backtracking “Branch-and-Propagate” Methods

Iterative repair algorithms are based on the use of total assignments, because they need to have a total assignment in order to have something to repair. In backtracking algorithms we are trying to build up a satisfying assignment. Starting from the empty assignment we extend it by valuing literals, i. e. changing their values from UNVALUED to TRUE or FALSE, with the goal of extending the assignment until we have a model. Such algorithms have two distinct components related to whether the extension is forced or speculative. Again we shall use P to denote the

assignment, but now regard it as a set of literals that are assigned TRUE.

1. Propagation: Given the current assignment P we might be able to deduce that some literal l must hold and so we can extend P to $P \cup \{l\}$. Propagation uses some limited reasoning method in order to find some such cases. Typically, only low-order polytime methods are used. Propagation can also discover a contradiction if it finds an unsatisfied clause, or that both l and $\neg l$ must hold.
2. Branches: Once propagation ceases, if we still have not detected a solution or a contradiction, then the search engine makes a speculative move. It selects a literal l not currently valued by P , and then extends P to $P \cup \{l\}$. It then propagates the effects of this extension.

The advantage of propagation is that it prevents looking at partial assignments that can easily be shown to be inextensible to a model. A backtracking search method must control the branches taken, and the remedies to be taken on detecting a contradiction.

In these algorithms, backtracking is just a matter of restoring some previous state. Most of the computational effort is spent in selecting branch variables and in doing propagation. Hence, as far as the issues in this thesis, are concerned it is perhaps better to think of them as “branch-and-propagate” algorithms. We shall soon look at a standard backtracking algorithm, the Davis-Putnam procedure, but firstly we look at propagation. In particular, we shall be concerned with the details of the simplest form of propagation: Unit propagation.

3.5 Unit Propagation

Conceptually, unit propagation works by using P to simplify the clauses by removing literals that are valued to FALSE. If a clause simplifies to a unit clause, i. e. just a literal, we can add that literal to P and continue to simplify. If a clause simplifies to an empty clause, then we have a contradiction. For example, given the clauses $\neg x \vee \neg y$ and $y \vee z$, and starting with $P = \{x\}$, then unit propagation will derive $\neg y$ and z , terminating with $P = \{x, \neg y, z\}$. We will use the symbol \Rightarrow to denote propagations, that is

$$P \Rightarrow z \tag{3.1}$$

will mean that z can be produced from P by unit propagation (and the relevant axioms). Usually, unit propagation is run to completion, that is, the process is repeated until no further literals can be found. Of course, unit propagation is sound and so also $P \models z$. However, it is not complete: we can have $P \models z$ but $P \not\Rightarrow z$.

In practice, unit propagation involves finding clauses in which there are no TRUE literals, and at most one UNVALUED literal. A clause with one UNVALUED literal, l , and the rest FALSE, means P should be extended by adding l : That is, P should be changed so that l is assigned to TRUE. If there are no UNVALUED literals, then P can never be extended to a model and a contradiction is reported.

In practice, unit propagation is done incrementally. We have a procedure $\text{UNIT-PROPAGATE}(l, P)$ that will set a literal l to TRUE in P and find all the resulting unit propagations. Such a method assumes that unit propagation to

```

proc UNIT-PROPAGATE( $l, P$ )           Propagate effects of  $l$  given  $P$ 
   $P := P \cup \{l = \text{TRUE}\}$ 
   $C_{\neg l} := \text{set of clauses containing } \neg l$ 
  foreach clause  $c \in C_{\neg l}$            Note that  $\neg l$  is false
    if  $c$  contains no TRUE literals by  $P$ 
      if  $c$  contains no unvalued literals by  $P$  return Contradiction
      if  $c$  contains precisely one unvalued literal  $l'$  by  $P$ 
         $P := \text{UNIT-PROPAGATE}(l', P)$ 
        if  $P = \text{Contradiction}$  return Contradiction
    end foreach
  return  $P$                            No contradiction was found
end

```

FIGURE 9. A simple recursive way to implement unit propagation.

completion has already been run on the theory. In this case, the only new propagation that can arise is from clauses that contain $\neg l$. Hence, we should scan such clauses, reporting any contradictions found, and adding any discovered forced literals to P , followed by unit propagating the effects of the new literals.

A straightforward recursive method to implement this is given in Figure 9. The recursive view is easiest to think about, but it should probably be implemented iteratively for efficiency. We give two such methods in Figure 10. In these, any literals discovered are stored in a container L and later extracted for propagation. In practice, the assignments can be made immediately, and the literals also stored for later propagation. It turns out that we need not care about the details of the algorithms except in the way that we scan the sets $C_{\neg l}$ for clauses that have no TRUE literals and at most one UNVALUED literal. We return to this issue in Chapter V.

```

proc UNIT-PROPAGATE( $l, P$ )
   $L := \{l\}$   $L$  is a container for literals awaiting propagation
  while  $L \neq \emptyset$ 
    select some  $l' \in L$  and remove it from  $L$ 
     $P := P \cup \{l' = \text{true}\}$ 
     $C_{l'} := \text{set of clauses containing } \neg l'$ 
    foreach unsatisfied clause  $c \in C_{l'}$ 
      if all literals in  $c$  are false return Contradiction
      if  $c$  contains exactly one unvalued literal
         $l'' := \text{the literal in } c \text{ that is not valued by } P$ 
         $L := L \cup \{l''\}$  Store  $l''$  for later propagation
      end foreach
    end while
  return  $P$ 
end

```

(a)

```

proc UNIT-PROPAGATE( $l, P$ )
   $L := \{l\}$ 
   $P := P \cup \{l = \text{true}\}$ 
  while  $L \neq \emptyset$ 
    select some  $l' \in L$  and remove it from  $L$ 
     $C_{l'} := \text{set of clauses containing } \neg l'$ 
    foreach unsatisfied clause  $c \in C_{l'}$ 
      if all literals in  $c$  are false return Contradiction
      if  $c$  contains exactly one unvalued literal
         $l'' := \text{the literal in } c \text{ that is not valued by } P$ 
         $P := P \cup \{l'' = \text{true}\}$ 
         $L := L \cup \{l''\}$ 
      end foreach
    end while
  return  $P$ 
end

```

(b)

FIGURE 10. Two iterative implementations of unit propagation arising from adding the literal l to the partial assignment P . (a) Eager: any propagations discovered lead to an immediate change in P . (b) Delayed: propagations are not enforced until all previous propagations have been completed.

3.6 The Davis-Putnam Procedure

The Davis-Putnam (DP) procedure [15, 14] is a simple backtracking algorithm that uses depth-first search to control the backtracking. It is systematic and complete: if there is a solution then it will be found, and the algorithm makes no (obvious) repetitions of work. The top-level control is simple and is given in Figure 11.

```

proc DP-Solve(C)                                Returns a model or else a contradiction
  return DP-Solve(C,  $\emptyset$ )
end

proc DP-Solve(C, P)
  l := Select-literal(C, P)

  P' := DP-Solve(C, UNIT-PROPAGATE(l, P))      Solve branch with l = TRUE
  if P' is a model return P'

  P' := DP-Solve(C, UNIT-PROPAGATE( $\neg$  l, P))  Solve branch with l = FALSE
  if P' is a model return P'

  return contradiction                            No model was found.
end

```

FIGURE 11. The Davis-Putnam procedure: Depth-first search with branching on selected literals.

Difficulties in creating fast implementations of the Davis-Putnam procedure arise from two main sources. Firstly, the algorithm is highly reliant on UNIT-PROPAGATE(*l*, *P*): typically a large fraction of the runtime is spent in this routine, and so it is important that the propagation is as fast as possible.

Secondly, we have to be careful in the selection of the branch variable. A

```

proc SELECT-LITERAL( $C, P$ )

    foreach variable  $v$  not already valued by  $P$                                 Pre-selection Phase
         $c_l(v) :=$  number of binary clauses in  $C$ , under  $P$ , and containing  $v$ 
         $c_f(v) :=$  number of binary clauses in  $C$  under  $P$ , and containing  $\neg v$ 
    end foreach
    restrict to the variables for which a specified function of  $c_l(v)$  and  $c_f(v)$  is largest

    foreach variable  $v$  still being considered                                Final Selection
         $c_l(v) :=$  length(unit-propagate( $v, P$ )) - length( $P$ )
         $c_f(v) :=$  length(unit-propagate( $\neg v, P$ )) - length( $P$ )
    end foreach
     $v :=$  variable for which a specified function of  $c_l(v)$  and  $c_f(v)$  is largest
     $l :=$  select  $v$  or  $\neg v$ 

    return  $l$ 
end

```

FIGURE 12. Selecting the branch variable in Davis-Putnam. The length of a partial assignment is just the number of literals that it values. By a clause being “binary under P ” we mean that it reduces to a binary clause after using the literal values contained in P .

good way to do this is sketched out in Figure 12 based on the approach of Crawford *et. al.* [12, 13]. The idea is that we should select branch variables that do the most to reduce the size of the search spaces in both the TRUE and FALSE branches. To estimate the reduction in the size of a search space from enforcing a literal we can unit-propagate that literal and see how many new literals are produced. The more propagation that occurs the better. This explains the final selection phase in Figure 12, however to do this for all literals would be too expensive. Instead there is a pre-selection phase that first eliminates some variables from further consideration.

We shall say that a clause is binary under P if simplifying it with P would reduce it to a binary clause, that is, it currently contains no TRUE literals and precisely two UNVALUED literals. The pre-selection phase relies on the observation that if a literal occurs in a clause that is binary under P , then if we were to set that literal to FALSE we would immediately get a propagation. Hence a fast (but inaccurate) estimate of the propagations that would result from a literal can be obtained by counting the number of binary clauses containing that literal. In a ground implementation, this information is stored and maintained so that these binary counts can be obtained quickly.

3.6.1 Sampling Methods

The Davis-Putnam procedure just does depth-first search, however there are reasons why this might be a bad idea. In cases where there are many models, we should be able to find one quickly; however, depth-first search is prone to making an early mistake, that is selecting a branch that happens to have no models.

Since it must explore all branches completely before backtracking this can mean, in practice, it will never recover from the mistake. There are extensive studies of this problem [31, 1], and there are also many algorithms available to combat the problem, for example Iterative Sampling [41], Iterative Broadening [27], and Limited Discrepancy Search (LDS) [32].

However, these are all still “branch-and-propagate” and use the same mechanisms for moving around the tree as simple depth-first search. Hence we need not talk about them explicitly. A lifted version of Davis-Putnam could easily be modified to give a lifted version of an algorithm such as LDS.

3.7 Proof-Search: Resolution

The goal of proof search is not to find a model but rather to find constraints entailed by the input axioms. The basic method is “combine-and-reduce”. We have inference rules that are a mixture of combining existing constraints and reducing them to simpler forms. In SAT we have simple binary boolean resolution. Clauses can be combined pairwise by the rule

$$x \vee A, \quad \neg x \vee B \quad \vdash \quad A \vee B$$

where x is a literal and A and B are disjunctions of literals. If A and B contain the same literal y then we should also “reduce” by using the factoring rule

$$y \vee y \vee C \quad \vdash \quad y \vee C$$

to merge two identical literals.

These rules are complete for refutation: If a theory is unsatisfiable then there is a resolution proof of the empty clause, that is, a contradiction. For efficiency, we can also use subsumption: If the literals of c_1 are a subset of those in c_2 then we can drop c_2 .

If the theory is satisfiable then resolution will not directly produce a model. However, suppose that we resolve to completion: That is, we generate all resolvents until we find that the resolvent of any two clauses is subsumed by some other existing clause. After resolving to completion then we can produce models using just unit propagation and arbitrary branching and with no risk of having to backtrack. To see this, suppose that we did have to backtrack. This would mean that we must have set some literal l from an assignment P and discovered a contradiction, \perp . If Γ denotes the original set of clauses then we will have found that $\Gamma \wedge P \wedge l \models \perp$. But then $\Gamma \models (P \rightarrow \neg l)$ and so $(P \rightarrow \neg l)$ should have been a clause in the database after completion. In this case, when we reached P , unit propagation should have forced $\neg l$ and we would never have had the chance to branch on l . Resolution to completion will produce all such clauses in advance, and hence guarantee branching can never make a mistake.

Thus, in principle, resolution could be used to solve problems. However, in practice it is rarely used for SAT. The main problem is that many clauses can be generated, leading to memory problems, furthermore there is no guarantee that if a clause is generated it will ever be used. (These problems also occur with controlling resolution in general theorem proving.) In contrast, the depth-first search of the Davis-Putnam procedure is very efficient for memory use, and at least it guarantees that every subtree is used.

3.8 Mixed Methods

By mixed methods, we refer to methods that mix the approaches of backtracking model-search and proof-search. They attempt to find models by performing some branch-and-propagate search, but also attempt to find new constraints entailed by the initial axioms. We will consider two classes of mixed methods. Firstly, we will look at some pre-processing techniques that are designed to improve the properties of a SAT problem before it is given to a search engine such as WSAT or the Davis-Putnam procedure (DP). Secondly, we will very briefly discuss the addition of learning to DP-based algorithms.

Mixed algorithms tend to revolve around the generation of “No-Goods” from a model based search. A No-Good is a partial assignment that has been discovered to lead to a contradiction. That is, the corresponding subtree has been completely covered (via search or reasoning) and no models were found. If a partial assignment $P \models \perp$ then we know that the theory entails the clause $\neg P$ (remember the assignment is just a conjunction of literals). The negation of a No-Good is hence a clause that is entailed by the theory.

3.9 Pre-Processing and COMPACT

The aim of pre-processing is to simplify theories before trying to solve them. The most obvious simplification is to unit propagate all unit literals in the input, keep track of all variables whose values are fixed by the unit literals, and then remove all clauses that have become satisfied because of these fixed variables. For example, $(x) \wedge (x \rightarrow y) \wedge (\neg y \vee w \vee z)$ reduces to $(w \vee z)$. We will call this process COMPACT.

We will cover two other methods that seem to be most relevant to SAT; “failed literal tests” or COMPACT-L, and “failed binary tests” or COMPACT-B. The names are taken from Freeman [21]; an implementation of these methods due to Crawford [11]. The aim in these methods is to search for short No-Goods and add their negations to the theory, with the hope that this helps the subsequent search phase avoid some bad decisions and so reduces the search time. The catch is to find No-Goods that do lead to a useful gain. Similar preprocessing techniques are also used in operations research under the label of “probing” [60].

Many other simplification methods exist, for example

1. Subsumption: Remove a clause if it is subsumed by another clause.
2. Pure literal rule: If a literal l occurs only positively then we can fix it to TRUE without affecting the satisfiability of the theory. If there is a model with $l = \text{FALSE}$ then the absence of negative occurrences of l means this literal cannot be the reason for any clause being satisfied, and hence its value can always be flipped.

3.9.1 Failed Literals and COMPACT-L

COMPACT alone just unit resolves to completion – i. e., propagates the effects of any unit literals in the input problem. The next step up is COMPACT-L, which looks for literals that “fail”. That is, we search for literals l such that $l \implies \perp$. In such a case l is a No-Good and $\neg l$ can be imposed on the theory, and also unit propagated to find its effects. This removes l , possibly other variables, and typically many clauses, from the theory, resulting in a more compact theory. Pseudo-code to

```

proc COMPACT-L(C)
  foreach variable v ∈ C
    if v is not already valued
       $R_t := \text{TEST-VALUE}(v, \text{TRUE})$ 
      if ( $R_t = \perp$ ) Found failed literal v
         $R_f := \text{VALUE}(v, \text{FALSE})$ 
        if ( $R_f = \perp$ ) return contradiction Both values give  $\perp$ 
      else
        if ( $\text{TEST-VALUE}(v, \text{FALSE}) = \perp$ ) Found failed literal  $\neg v$ 
           $\text{VALUE}(v, \text{TRUE})$ 
      end foreach
    return contradiction-Not-Found
  end

```

FIGURE 13. Straightforward COMPACT-L.

search for such failed literals is given in Figure 13. Figure 14 gives some supporting functions.

If there are n variables then propagating the effect of a literal is $O(n)$ (because the propagation might cover many of the variables). Hence, COMPACT-L is $O(n^2)$, though in practice it seems to be quite usable.

3.9.2 COMPACT-B

As an introduction to COMPACT-B [11, 21] consider the following simple theory

$$\begin{array}{l}
 a \longrightarrow b \\
 a \longrightarrow c \\
 b \wedge c \longrightarrow d
 \end{array} \tag{3.2}$$

```

proc VALUE( $v,d$ )
  set variable  $v$  to (truth) value  $d$  and unit propagate
  return whether this gave a contradiction
end

```

```

proc TEST-VALUE( $i,v$ )
  set variable  $v$  to (truth) value  $v$  and unit propagate
  undo effects and return whether this gave a contradiction
end

```

FIGURE 14. Utility functions VALUE and TEST-VALUE

In this case $a \implies d$, however, the contrapositive of this propagation does not occur, i. e. $\neg d \not\implies \neg a$. Now, suppose that (3.2) were part of a larger theory, and that there were no other reason in the theory for the propagation $\neg d \implies \neg a$ to occur. If the search algorithm happened to branch on d first, it would miss this propagation, and could conceivably cause extra work for itself. It would be good in this case if we explicitly recognized that the clause $\neg a \vee d$ is entailed and added it to the theory. After this the propagation $\neg d \implies \neg a$ cannot fail to occur.

More generally, COMPACT-B is intended to look for failed binaries [21], that is, pairs of literals l_1, l_2 such that

$$l_1 \wedge l_2 \implies \perp \quad \text{but either} \quad l_1 \not\implies \neg l_2 \quad \text{or} \quad l_2 \not\implies \neg l_1 \quad (3.3)$$

In such a case the clause

$$\neg l_1 \vee \neg l_2 \quad (3.4)$$

is entailed, and will improve propagation by its presence. Hence, it is added to the set of clauses.

This can be implemented as a simple double loop over the variables. There are $O(n^2)$ pairs of literals to consider, so naive COMPACT-B is $O(n^3)$. COMPACT-B, in contrast to COMPACT-L, typically seems to be too expensive to run. It is also interesting that COMPACT-B adds new clauses to the theory, whereas COMPACT and COMPACT-L only result in the binding of variables to fixed values. We will return to this in Chapter XI (see Section 11.2). We could of course naively extend this to considering sets of k literals and testing for failure. However, this rapidly becomes impractical, at least without further insights into the problem.

A deeper and more general study of these issues has been given by Roussel and Mathieu [57] where the task of making sure that such propagations do occur as they ought to is called achievement. Detecting such cases is related to the existence of cycles in a graph built from the constraints. It is also related to the role that the factoring rule plays in resolution proofs. Consider (3.2) again: if we resolve clauses together with no factoring then we can obtain $a \wedge a \rightarrow d$ but in order to get the expected $\neg a \vee d$ we must factor on the $\neg a$. This merging of $\neg a$ but not of d explains the asymmetry between the two possible propagations. All this has been extensively studied and also extended to the first-order case [58]. We have discussed this more advanced pre-processing method to show that it is based on unit propagation, and hence lifting unit propagation can be useful even for this algorithm. We will also consider COMPACT-B again in Chapter XI (Section 11.2.1).

3.10 No-Good Learning in Intelligent Backtracking

So far the only backtracking method we have discussed is the Davis-Putnam procedure with its simple depth-first search. That is, simple chronological backtracking in which discovering a contradiction only leads to reconsideration of the most recent branch variable. For the sake of completeness we also mention some more intelligent backtracking methods. Techniques such as backjumping and dependency-directed backtracking are designed to pinpoint which decisions were responsible for the latest contradiction, and backtrack to them directly (see [1] for a review).

There are also algorithms that use truth maintenance methods to learn and store entailed clauses as they proceed. The main problem is that if we store all the clauses memory usage grows with linearly with runtime, and this gives us too many clauses. There have been two main proposals to limit the memory usage: k -order learning [16] and Relevance-Bounded Learning (RBL) [2, 3]. In k -order learning we only store clauses of length k or less. RBL uses a much more selective criterion that adapts to the portion of the search space currently under exploration. The key idea comes from dynamic backtracking [24, 25], and is to limit the number of literals in a clause that do not agree with the current partial assignment. Clauses that have too large a disagreement with the current assignment are presumed less likely to be of any future use and so are discarded.

3.11 Summary

In this chapter we have reviewed some of the methods used to solve instances of SAT. The main focus was on two state-of-the-art solution methods: WSAT and

the Davis-Putnam procedure. We now have enough background to move into the main portion of the thesis which is to apply such solution methods to problems expressed not in SAT, but in a lifted equivalent.

CHAPTER IV

SUBSEARCH

In Chapter II we defined quantified propositional logic (QPROP) and argued it is a natural language for many structured combinatorial problems. We also gave an example of a QPROP encoding of planning problems for a simple logistics domain. Normally such problems are grounded out to SAT. However, this involves expansion of the quantifiers in QPROP and results in a large increase in size of the theory. Instances of the logistics problem with just hundreds of objects could give many millions of clauses. Solvers that only take ground clauses will (almost) inevitably be swamped by such size, and so the sheer size of the ground theories provides a motivation to work *directly* with quantified expressions. In this chapter we lay the groundwork for lifting SAT algorithms to use QPROP representations directly.

This chapter has two main foci. Firstly, we describe a restriction of QPROP to a lifted form of SAT that we call “Quantified Clausal Normal Form” (QCNF): We need to take such an “intersection” between QPROP and CNF because we will be lifting algorithms designed to take clausal inputs.

The larger portion of this chapter is concerned with how SAT algorithms make use of their set of input clauses. The goal is to identify what information ground algorithms need to extract from the ground clauses, and write it in a way that allows extension to the case of extracting information from QCNF expressions.

We will identify various tasks, that we call “subsearch problems”, that provide the means for a search algorithm to extract the properties it needs from the clause database.

In the next chapter, we show that SAT search algorithms can indeed be rewritten in terms of these subsearch problems. Doing this shields the algorithms from having to deal directly with the clauses. We will see that this shielding results in lifting becoming a matter of using subsearch over quantified clauses, instead of having to change the search itself to deal with quantifiers. A primary goal of this thesis is to show that subsearch problems can be solved better in the lifted case than in the ground case.

4.1 Quantified Clausal Normal Form (QCNF)

Before proceeding to lifting SAT solvers, we first make a restriction on QPROP that mimics the restriction of general boolean logic to SAT. However, it is important to do this in a way that is motivated by real encodings of domains, and so we will be guided by the SATPLAN example of Chapter II (Section 2.5).

We want to do a minimal extension to CNF that will allow us to use quantifiers and in particular to handle sets of axioms such as those of the logistics example (Section 2.5). In particular, we want expressions that are clausal in the sense that on grounding they will directly convert to CNF. CNF just means that all conjunctions are outside all disjunctions (which are outside all negations), and in a finite domain universal quantifiers are just conjunctions and existentials are just disjunctions. This suggests demanding that all conjunctions and universals are outside all disjunctions and existentials, that is, of the form of (2.8). However,

this is still too general.

If we look at the axioms of the logistics example, there is very little use of existential quantifiers. Most of the blame for the size explosion from grounding lies with multiple universal quantifiers. Hence, for most of this thesis, we will consider clauses that have only universal quantifiers:

Definition 4.1.1

A *universally quantified clause* is a clause of the form

$$\forall i_1, \dots, i_q. A[i_1, \dots, i_q] \quad (4.1)$$

where A is a disjunction of literals built from the variables i_1, \dots, i_q and constants, and containing no further quantifiers. A ground clause is just the special case, $q = 0$, with no quantifiers or variables. \square

We will often just call these quantified clauses, the “universal” being implicit from the context. All the axioms of the examples in Chapter II (Section 2.5) are easily put into this form by expanding just the existential quantifiers. (In Chapter XI we will briefly look into reinstating the existential quantifiers.)

We can now define the language with which this thesis is primarily concerned:

Definition 4.1.2

If all constraints are universally quantified clauses we will call this *Quantified Clausal Normal Form* (QCNF). A QCNF expression is a conjunction (or set) of universally quantified clauses. \square

We use the term “clausal” rather than “conjunctive” as it is clauses that are

quantified, rather than conjunctions. Also, note that this does not restrict the expressive power any more than restricting to SAT.

As discussed in Chapter II, we will allow known pre-determined functions in the expressions, though in practice we will convert all functions to predicates. Also, axioms do not have free variables, and in QCNF all quantifiers are universal, so in some cases we will use the convention that all variables are implicitly universally quantified.

On page 20, we defined $Gr(C)$ to represent the ground formula obtained from a QPROP expression. Suppose that C is a quantified clause. What can we say about the relative sizes of C and $Gr(C)$? The number of quantifiers in a quantified clause is not bounded. If we ignore logarithmic factors (arising from issues in exactly how we encode the symbols) then the number of quantifiers can be linear in the size $|C|$ of the quantified clause. However, on grounding a quantified clause the number of ground clauses produced is exponential in the number of quantifiers. That is, we can expect

$$|GrL(C)| = O(\exp(|C|)) \quad (4.2)$$

and furthermore that there will be no (significantly) better bound.

This potentially exponential difference between the size of a quantified clause and the size of the ground equivalent will be very important in understanding the results in Chapter VI.

4.2 The “Database of Clauses” Viewpoint

We now move to the issue of how to set up algorithms that can take quantified clauses as input. We call such algorithms “lifted” because of their jump from boolean logic to a language with quantifiers.

The key observation we use is that many steps in SAT algorithms do little more than scan the set of clauses, or some subset of them, for clauses satisfying a certain property. We can think of this as a step in which the algorithm restricts its attention to a subset of the clauses, temporarily ignoring all the others.

There are two classes of restrictions that are applied:

1. Syntactic restrictions: these do not depend on the current truth value assignment P .
2. Semantic restrictions: these do depend on P .

An example of a syntactic restriction is to find all clauses containing a given literal. This restriction is very common in algorithms because they often proceed by making (or considering) changes to literals one at a time, and every time a literal is changed they want to find the effects of this change. It then makes sense that the effects of the change can only originate in clauses containing the changed literal. For example, in the Davis-Putnam procedure, when we value a literal l we call the unit-propagation procedure which first finds all clauses containing l and checks whether any new information can be extracted from those clauses.

Since, by definition, such syntactic restrictions do not depend on the current state of the solver, P , it follows that we can build all the required sets in advance. In the ground solvers, this is usually done by a simple indexing scheme into the

entire set of clauses.

An example of a semantic restriction is the set of currently unsatisfied clauses in WSAT: we need clauses in which all the literals are set to FALSE by the current assignment.

Semantic restrictions are handled quite differently from syntactic restrictions because they depend on the current state of the solver and so cannot be made before starting the search process. For example, in its initialization step, WSAT creates a new random assignment and scans all the clauses in the entire database looking for those that are unsatisfied. This must also be done after every restart; no static indexing scheme can take account of the changes in the assignment. Also, the set of unsatisfied clauses is constantly changing and must be updated after every flip of a variable. Again, the update is done by finding a subset of the clauses that satisfy a certain property – in this case that the clause contains the literal that was just flipped, and all the literals are now set to FALSE by the current assignment.

We dub all such processes that extract sets (or information about sets) of clauses satisfying semantic restrictions “subsearch”. The implications that such processes are search processes is deliberate. There are two reasons for considering them a form of search: Firstly, it is standard to refer to “searching a database”: we can view the set of clauses as a large database and then we search for all matches to our requirements. Secondly, and perhaps more importantly, in Chapter VI we shall see that the task is NP-hard with respect to the size of the quantified clauses. Hence we will indeed want to use search algorithms for this “subsearch”.

Syntactic restrictions are much simpler and will not be done by search; how-

ever they are still an important part of algorithms and so must also be lifted.

What does it mean to search the database of clauses when we have quantified clauses? Recall that the set of ground clauses arising from a quantified clause is just the set of possible sets of bindings for the universally quantified domain variables. Searching for ground clauses satisfying a certain property is equivalent to searching through sets of bindings for these variables.

For example, suppose we have the clause

$$\forall i, j. r(j) \vee p(i, j) \quad (4.3)$$

Then finding a ground clause satisfying some semantic restriction is the same as finding bindings for the variables i, j . Subsearch is a search for relevant bindings of domain variables. In contrast, syntactic restrictions on a quantified clauses will be done by a simple form of unification.

In the next chapter we shall cover the standard algorithms again and show how they can be rewritten in terms of subsearch. However, first it is convenient to formalize these ideas. We first formalize the concept of a restriction, and then treat the cases of syntactic restrictions and semantic restrictions separately.

4.3 General Restrictions on the Database

To formalize the idea of restricting the database of clauses we proceed in a fashion that can handle both ground and quantified clauses. For general restrictions we will need three components:

1. Some set of clauses, C , that can contain quantified clauses. For example C

could contain the clauses in Figure 3, or might consist of only ground clauses.

2. The current truth value assignment P for the ground predicates (propositions) in the theory. P can be a partial assignment: we might have $\tau(1) = \text{TRUE}$, $\tau(3) = \text{FALSE}$ and $\tau(2)$ left unassigned ($\tau(2) = \text{UNVALUED}$).
3. A boolean restriction function $f(c, P)$

The restriction function f takes a ground clause c and returns whether or not the clause is accepted according to the current semantics P . A common requirement is to restrict to violated (unsatisfied) clauses, in which case

$$f_v(c, P) \iff \forall l \in c. l = \text{FALSE in } P \quad (4.4)$$

that is, the assignment P definitely violates the clause c .

A syntactic restriction is the special case where the function f does not depend on P .

We use this notion of restriction function to make the following definition:

Definition 4.3.1

Given a clause set C , assignment P , and restriction function f , we define the term $S^Q(C, P, f)$ to refer to any set of (possibly quantified) clauses such that

$$c \in Gr(S^Q(C, P, f)) \iff c \in Gr(C) \wedge f(c, P) \quad (4.5)$$

That is, $S^Q(C, P, f)$ contains (perhaps implicitly) every ground clause

from C that satisfies the condition imposed by f . We will call $S^Q(C, P, f)$ an f -restriction of C under P . \square

The superscript “ Q ” is a reminder that $S^Q(C, P, f)$ can contain quantified clauses. If C is a single quantified clause then we can think of $S^Q(C, P, f)$ as specifying the minimal restriction on the bindings of the domain variables such that f holds.

As an example, consider the case where C consists of only (4.3):

$$\forall i, j. r(j) \vee p(i, j)$$

Let P assign all instances of $p(i, j)$ to FALSE, and all instances of $r(i)$ to TRUE, except $r(2)$ which is set to FALSE. Then

$$S^Q(C, P, f_v) = \forall i. r(2) \vee p(i, 2) \quad (4.6)$$

We now list some of the properties of an f -restriction that follow directly from the definition. Such properties will often underlie various discussions and algorithms, even though we will typically not refer to them explicitly.

4.3.1 Non-Uniqueness

There are certainly cases in which $S^Q(C, P, f)$ can contain quantified clauses. A trivial example would be the case that f accepts all clauses, in which case $S^Q(C, P, f) = C$ satisfies the definition. However, $Gr[C]$ also satisfies the definition. This means that an f -restriction is not unique. If we talk about equality of f -restrictions we will have to limit ourself to “equality up to representation”. We

will denote this by a subscript “ G ” with the meaning

$$C_1 =_G C_2 \iff Gr[C_1] = Gr[C_2] \quad (4.7)$$

We could make an f -restriction unique by demanding that it produces only ground theories, and in fact we will eventually do this for the case of semantic restrictions. However, for syntactic restrictions we will need to keep the option of having quantified clauses.

4.3.2 Idempotency

Since an f -restriction returns a set of clauses, we could f -restrict these again. If we use the same restriction then clearly the set will not change

$$S^Q(S^Q(C, P, f), P, f) =_G S^Q(C, P, f) \quad (4.8)$$

4.3.3 Distributivity

An f -restriction distributes across taking unions (conjunctions) of sets of clauses

$$S^Q(C_1 \cup C_2, P, f) =_G S^Q(C_1, P, f) \cup S^Q(C_2, P, f) \quad (4.9)$$

This essentially holds because the restriction is something that tests ground clauses individually and independently. An interesting property of the algorithms in Chapter III is that we do *not* need restrictions such as “there are precisely 6 unsatisfied ground clauses”, or nonmonotonic restrictions such as “accept a clause

iff there is no other accepted clause sharing a literal with it".

4.3.4 Commutativity

$$S^Q(S^Q(C, P, f_1), P, f_2) =_G S^Q(S^Q(C, P, f_2), P, f_1) \quad (4.10)$$

Again this is because the restriction tests only individual ground clauses. Both orders of restricting simply select all the clauses that satisfy both f_1 and f_2 .

In practice, the different usages of semantic and syntactic restrictions mean that they require two different instances of the notion of f -restriction. Algorithms typically take a syntactic restriction and then feed the result to a semantic restriction: for example, in order to detect a contradiction on setting a literal l to FALSE, we might want the set of clauses containing l and having no TRUE or UNVALUED literals.

The output of the syntactic restriction is a set of clauses on which we want to do further work, and it will turn out that we should leave it as lifted as possible. Since the restriction does not involve P we will see that this is feasible.

For semantic restrictions having the output in ground form is adequate (at least for the purposes of this thesis). Furthermore, in practice, P is likely to be much messier than the simple example above and so it is much less likely that a simple lifted form exists. The following two sections deal with syntactic and semantic restrictions separately in order to take account of these differences.

4.4 Syntactic Restrictions: The Set $R(C, l)$

By definition, syntactic restrictions are the case in which $f(c, P)$ does not depend on P , and so can be written as just $f(c)$. It turns out that all we need is the ability to restrict to clauses containing a given literal l , that is

$$f_l(c) \iff l \in c \quad (4.11)$$

In the ground case the f_l restriction of a set of ground clauses C will also be a set of ground clauses which we denote by $R^G(C, l)$, defined by

$$c \in R^G(C, l) \iff c \in C \wedge l \in c \quad (4.12)$$

However, in the lifted case we want the f_l -restriction to involve as little grounding as possible because the output of the restriction will be passed to a semantic restriction. The associated subsearch problem will be solved by exploiting the quantifiers in the clauses.

Thus, in the context of QCNF (or QPROP) we will use the notation $R(C, l)$ for the f_l -restriction. We will assume that l is ground – that is, a proposition or its negation, with no domain variables.

We require that $R(C, l)$ satisfies

$$Gr(R(C, l)) = R^G(Gr(C), l) \quad (4.13)$$

and also that $R(C, l)$ remains “as lifted as practical”.

We can, and will, obtain $R(C, l)$ via a one-way matching of the arguments

of l to the variables occurring in the clauses in C . For example, suppose that C is the clause of (4.3) and we want to find $R(C, r(2))$. The needed set of clauses corresponds to the binding $j = 2$ and hence

$$R(C, r(2)) = \forall i. r(2) \vee p(i, 2)$$

In the case of multiple matches we take the conjunction of the matches. For example, suppose we have the clause of (2.24):

$$C = \forall o, a, b, i. \neg(a < b) \vee \neg \text{in}(o, a, i) \vee \neg \text{in}(o, b, i) \quad (4.14)$$

and want clauses containing $\neg \text{in}(2, 3, 4)$. We can match on the second or third literals with different bindings. Then $R(C, \neg \text{in}(2, 3, 4))$ consists of two quantified clauses

$$\begin{aligned} \forall b. \neg(3 < b) \vee \neg \text{in}(2, 3, 4) \vee \neg \text{in}(2, b, 4) \quad \wedge \\ \forall a. \neg(a < 3) \vee \neg \text{in}(2, a, 4) \vee \neg \text{in}(2, 3, 4) \end{aligned} \quad (4.15)$$

In this case we can combine the clauses by changing the dummy variables and reordering the literals:

$$\forall a. (\neg(3 < a) \wedge \neg(a < 3)) \vee \neg \text{in}(2, 3, 4) \vee \neg \text{in}(2, a, 4)$$

and then using the properties of the fixed predicate $<$ to replace the conjunct with

the equality predicate:

$$\forall a. a = 3 \vee \neg \text{in}(2, 3, 4) \vee \neg \text{in}(2, a, 4) \quad (4.16)$$

In other words $R(C, \neg \text{in}(2, 3, 4))$ is just

$$\text{in}(2, 3, 4) \longrightarrow [\forall a. a \neq 3 \longrightarrow \neg \text{in}(2, a, 4)] \quad (4.17)$$

This also illustrates the way that $R(C, l)$ restrictions are used: If we set $\text{in}(2, 3, 4) = \text{TRUE}$ then we require that object 2 is not in any other plane at that time.

Note that (4.15) and (4.16) both ground out to the same SAT expressions, i.e. they give exactly the same set of ground clauses (and *without* first having to remove duplicate clauses). The fact that we can equally well represent the restriction by (4.15) or the arguably “more lifted” in (4.16) suggests that trying to give an exact definition to $R(C, l)$ being as “lifted as possible” is difficult. We shall take the practical view that an implementation should strive to avoid groundings, but need not attempt to do such combinations as above. The implementation we present in Chapter VIII knows the values of fixed predicates, such as arithmetic comparison operators, however, it does not do any reasoning that involves such combination of clauses. Such reasoning is not part of QCNF, or the lifted solver we consider, however, it might be useful to have a solver that tried to use such reasoning by means of calls to a theorem prover. In any case, the avoidance of grounding is just for efficiency reasons and does not affect the correctness of the implementations.

Since every clause in $R(C, l)$ contains the literal l we can also remove it from

every clause. The following definition will be convenient:

Definition 4.4.1

$R^-(C, l)$ is defined as the set of quantified clauses such that

$$c \in Gr[R^-(C, l)] \iff (c \vee l) \in Gr[R(C, l)] \quad (4.18)$$

or equivalently

$$c \in Gr[R^-(C, l)] \iff (c \vee l) \in Gr[C] \quad (4.19)$$

□

Note that $R(C, l) \equiv l \vee R^-(C, l)$. It can be helpful to think of $R^-(C, l)$ as the maximal set of subclauses of C that can be written as $\neg l \rightarrow R^-(C, l)$.

In the example given above, we have that

$$R^-(C, \neg \text{in}(2, 3, 4)) \equiv \forall a. [a = 3 \vee \neg \text{in}(2, a, 4)] \quad (4.20)$$

It is important that $R^-(C, \neg l)$ is also a QCNF formula. Implementations of algorithms will typically work by producing various $R^-(C, l)$ and then using the fact that they are just quantified clauses in order to solve their subsearch problems.

We could also extend the definition to the case that l is a literal with free variables (a “lifted literal”). Regarding l as defining a set L of ground literals, we can define

$$c \in R(C, L) \iff (c \in C) \wedge \exists l. [l \in L \wedge l \in c] \quad (4.21)$$

Finding $R(C, L)$ would then need unification, but we do not need such a case here, as we do not use lifted literals.

4.5 Semantic Restrictions: The Set $S(C, P, u, s)$

Semantic restrictions are those for which the function $f(c, P)$ does depend on P . It turns out that we only need a restriction function that simply tests the numbers of UNVALUED and TRUE literals in the clause

$$f_{u,s}(c, P) \iff \begin{aligned} &(\text{number of UNVALUED literals in } c \text{ is } u) \wedge \\ &(\text{number of TRUE literals in } c \text{ is } s) \end{aligned} \quad (4.22)$$

For example, to restrict to unsatisfied clauses we want $u = s = 0$, or for clauses that give rise to a propagation we want $u = 1$ and $s = 0$. (This will become clearer in Chapter V.) It will also be sufficient for the $f_{u,s}$ restriction to produce just ground clauses, and hence we use a special definition for this case:

Definition 4.5.1

The set $S(C, P, u, s)$, with C being a set (conjunction) of, possibly quantified, clauses, is the set of *ground* clauses with the properties that:

1. Precisely u of the literals are unvalued (assigned UNVALUED) by P .
2. Precisely s of the literals are assigned TRUE in P .

We will call this set the (u, s) -restriction of C under P . □

It is important to note that $S(C, P, u, s)$ is a conceptual tool. It is not necessarily the case that we need to find all of it. For example, if we just want to know whether or not a total assignment P satisfies the constraints C then we just need to know whether there are any unsatisfied ground clauses. Since we have a total assignment, all clauses will have $u = 0$ and so the clause will only be unsatisfied if it has no TRUE literals, hence we just need the case $u = s = 0$. If $S(C, P, 0, 0)$ is empty then the theory is satisfied and P is a model. To prove that it is not a model we do not need to produce the entire set $S(C, P, 0, 0)$, but merely to produce one element of it, any unsatisfied ground clause is enough to show an assignment is not a model.

As mentioned earlier, we will refer to the tasks of determining relevant properties or elements of $S(C, P, u, s)$ as subsearch problems.

4.6 Subsearch Problems

In this section, we define the various subsearch problems that we shall need. Lifted algorithms will be written in terms of solutions to these problems and so we will also give the relevant routines or fragments of pseudo-code to which they correspond.

4.6.1 The Checking Problem

Definition 4.6.1

The (u, s) -checking problem is the decision problem:

INSTANCE: A set of, possibly quantified, clauses C and a (possibly

partial) assignment P

QUESTION: Does $S(C, P, u, s) \neq \emptyset$? □

Note that u and s are fixed and not part of the input.

The inequality might seem strange but we shall see in Chapter VI that checking for non-emptiness is a somewhat more natural search problem. Also, we shall see in Section 6.3 that the problem written this way is NP-complete (in a sense that we shall describe in detail in that section).

4.6.2 The Counting Problem

Definition 4.6.2

The (u, s) -counting problem is the following problem:

INSTANCE: A set of, possibly quantified, clauses C and a (possibly partial) assignment P

TASK: Produce a binary representation of the number of elements in $S(C, P, u, s)$. That is, find $|S(C, P, u, s)|$. □

In pseudo-code we will refer to this by the procedure $\text{COUNT}(S(C, P, u, s))$.

We emphasize that although we will write all subsearch problems in this form, they are *not* to be thought of as functions that take the entire set $S(C, P, u, s)$ as input and only then solve the relevant problem. Although they could be implemented in such a fashion, it would be very inefficient. Instead, the problems are meant in the sense of functions such as $\text{COUNT}(S(C, P, 0, 0))$ that are implemented as directly as possible. However, we will retain the form with $S(C, P, u, s)$ to keep a link back to Definition 4.5.1.

4.6.3 The Iteration Problem

Definition 4.6.3

The (u, s) -iteration problem is the following problem:

INSTANCE: A set of, possibly quantified, clauses C and a (possibly partial) assignment P .

TASK: Produce a method to iterate over all elements of $S(C, P, u, s)$.

□

Note that we do not require all the elements to be available at once. It would be sufficient to have some procedure `NEXT()` that returns a new ground clause on each call, or a signal that no more ground clauses remain.

In pseudo-code we could refer to this by the procedure `NEXT(S(C, P, u, s))`. Alternatively, iteration could mean we have a way to implement a code fragment such as

```
foreach  $c \in S(C, P, u, s)$ 
  do something with  $c$ 
end foreach
```

We could use such an iterator in order to solve the counting problem by

```
count =: 0
foreach  $c \in S(C, P, u, s)$ 
  count := count + 1
end foreach
return count
```

The number of clauses could easily be exponential, but since the output of the checking problem is in binary it is possible that there are ways that are exponentially faster than just enumerating all the individuals. For example, if P values all literals to FALSE, and C contains no negative literals then $S(C, P, u, s)$ is just the entire set $Gr(C)$ and we can calculate this number directly rather than having to count it out.

4.6.4 The Enumeration Problem

Definition 4.6.4

The (u, s) -iteration problem is the following problem:

INSTANCE: A set of, possibly quantified, clauses C and a (possibly partial) assignment P .

TASK: Produce the set $S(C, P, u, s)$ itself. □

In general, $S(C, P, u, s)$ might be very large, but we shall actually use enumeration in WSAT because in that case the set typically stays at a manageable size.

4.6.5 Two Selection Problems

In some cases we merely want to select a clause satisfying a given property, i. e., to select an element of $S(C, P, u, s)$. We need two types of selection:

Definition 4.6.5

The *uniform selection problem* is the following problem:

INSTANCE: A set of, possibly quantified, clauses C and a (possibly partial) assignment P .

TASK: Produce a ground clause c that is uniformly and randomly selected from the set $S(C, P, u, s)$. \square

In pseudo-code we refer to this by `UNIFORMLY-SELECT($S(C, P, u, s)$)`. It will be used in `WSAT`.

Definition 4.6.6

The *free-selection problem* is the following problem:

INSTANCE: A set of, possibly quantified, clauses C and a (possibly partial) assignment P .

TASK: Produce any ground clause c in the set $S(C, P, u, s)$. The selection procedure need not satisfy any probability distribution, it is just required to return any element. \square

In pseudo-code we will refer to this by the function `FREELY-SELECT` with an intended meaning the same as

```
function FREELY-SELECT(  $S(C, P, u, s)$  )
    search through the clauses in  $S(C, P, u, s)$ 
    if find a clause  $c$  then return  $c$            return on finding first clause
    else return fail
```

In neither of these cases is it necessary to first explicitly produce $S(C, P, u, s)$.

4.6.6 "Union" Subsearch Problems

In the above, we only covered the case of subsearch over a single $S(C, P, u, s)$ set. In general, we could have subsearch over a union of such sets (for example with different u and s values). In one sense this adds nothing new, for example, we could FREELY-SELECT from a union of two sets using

```
proc FREELY-SELECT( $S_1 \cup S_2$ )
   $c :=$  FREELY-SELECT( $S_1$ )
  if  $c = fail$  return FREELY-SELECT( $S_2$ )
  return  $c$ 
end
```

However, this will restrict the subsearch too much. Instead the intention is that when possible the two sets are treated together. We shall see an example of the use of subsearch on a union in Figure 23.

4.6.7 Comments on the Problems

The free-selection and checking problem are essentially identical because, in practice (though not necessarily in principle), methods to solve the checking problem will explicitly produce an element if the set is non-empty.

Also, note that we do not claim that the list is exhaustive. There might well be other search algorithms that need subsearch problems not on this list. However, it is sufficient for WSAT and the Davis-Putnam procedure from Chapter III. In the next chapter, we illustrate how they are used.

We could have written all these subsearch problems in the more general terms

of $S(C, P, f)$, but we kept to the (u, s) -restrictions because it is these that give rise to interesting complexities and search problems.

4.6.8 Combining Clauses

Combining clauses is done in a trivial fashion. Write the set of all clauses C as a conjunction of single, possibly quantified, clauses

$$C = \bigwedge_i C_i \quad (4.23)$$

Then, directly from the definition, we have

$$S(\bigwedge_i C_i, P, u, s) = \bigcup_i S(C_i, P, u, s) \quad (4.24)$$

For the checking problem we just have

$$S(\bigwedge_i C_i, P, u, s) \neq \emptyset \iff \bigvee_i S(C_i, P, u, s) \neq \emptyset \quad (4.25)$$

and for the counting problem

$$\text{COUNT}(S(\bigwedge_i C_i, P, u, s)) = \sum_i \text{COUNT}(S(C_i, P, u, s)) \quad (4.26)$$

Enumeration and iteration just run over the separate clauses. Similarly, for the selection problems:

1. Free selection: Freely pick any C_i for which $S(C_i, P, u, s) \neq \emptyset$, and then just do free selection on C_i

2. Uniform selection: Pick a C_j randomly, but with appropriate weightings determined from all the $\text{COUNT}(S(C_i, P, u, s))$, and then call uniform selection on that C_j .

We have no method to solve the subsearch problems on $S(C, P, u, s)$, other than to just split it as above and solve the subsearch problems for individual clauses. Hence, we will often just take C to be a single quantified clause c , and leave it as implied that subsearch results are to be combined in the obvious fashion.

4.7 Summary

In this chapter we defined the language QCNF. Most of the thesis is concerned with solving problems written in QCNF. To provide a basis with which to describe such solvers we identified a task performed by current ground solvers: they have to scan the database of clauses for those satisfying given properties. This was formalized in terms of restrictions of the database and in particular the set $S(C, P, u, s)$. We finally gave a list of tasks, called subsearch problems, associated with $S(C, P, u, s)$. Ways to solve these tasks will form the heart of a lifted solver.

In the next chapter we look at how search algorithms can indeed be rephrased so that access to the relevant QCNF formula is in terms of subsearch problems.

In Chapter VI, we will return to the subsearch problems themselves and study their computational complexity and how to rephrase them in terms of CSP-like problems.

CHAPTER V

REWRITING SEARCH ALGORITHMS IN TERMS OF SUBSEARCH

Our aim in this chapter is to show that the language of restrictions and subsearch problems from Chapter IV is useful. We will rewrite some SAT algorithms in terms of the syntactic restrictions such as $R(C, l)$ and the set $S(C, P, u, s)$ with its associated subsearch problems.

We suggest that on first reading of each algorithm it might be best to forget about QCNF and lifting and take the set C to be a set of ground clauses. Once it is clear that in this case the search algorithm has not been changed then lifting the algorithm is just a matter of allowing the set C to contain quantified clauses. In this chapter, the lifting is done in such a way that allowing the clause database to contain quantified clauses does not change the search algorithm itself. We will discuss this further in Section 5.2.

We are going to see that the subsearch problems give us an “abstract machine” with which we can build lifted search engines. We expand on this in the next section. Later in the chapter we will look at expressing WSAT, unit propagation, and the Davis-Putnam procedure in terms of subsearch problems. Hence, the essential point of this chapter is that we can express search algorithms in terms of subsearch problems rather than by direct access to the quantified clauses. In Chapter VII, we will see that subsearch can actually be done more efficiently for quantified clauses than for their ground equivalents.

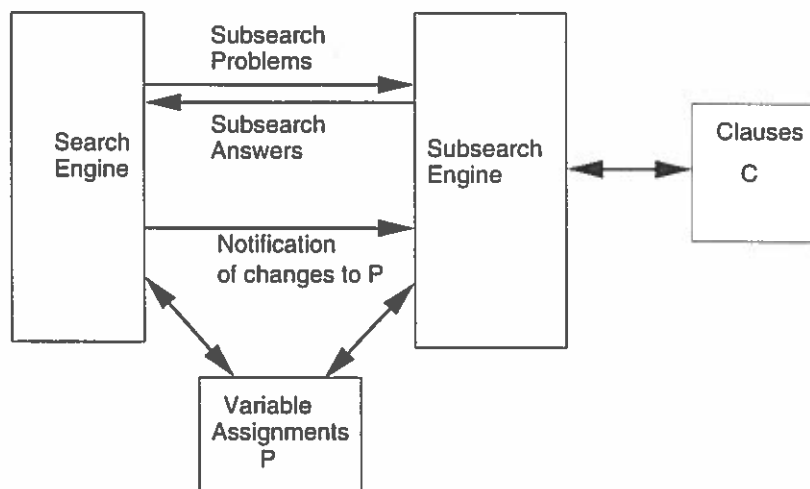


FIGURE 15. Splitting subsearch from search. The search engine only gets access to the set of clauses C by posing subsearch problems and having them answered by the subsearch engine. Both engines can access the current state P .

5.1 Splitting Subsearch and Search

This thesis has two major themes:

1. We can encapsulate the clauses in subsearch problems.
2. We can solve subsearch problems more efficiently if the clauses are not ground but are quantified.

The goal of this chapter is to demonstrate that the encapsulation is possible. We will do this by rewriting the algorithms to conform to a structure as given in Figure 15. The second theme will be treated in Chapter VII.

Similar splits have been useful in other areas. One might think of the database manager (DBM) in a database system. The DBM (c. f. subsearch engine) controls access to the raw data structures (tables, B-trees, etc) and provides a lan-

guage, such as SQL, that the user (c. f. search engine) can use to state problems. In fact, there is a close analogy with the DBM given the view taken in Section 4.2. The DBM shields the SQL programmer from the gory details of optimizing access to tables, so that, for example, how a join is actually implemented becomes irrelevant. In return the DBM can optimize many of the operations.

5.2 Minimal vs. Non-Minimal Lifting

Here we clarify whether or not lifting has an effect on the search algorithm itself. Our first liftings are, by construction, designed to have no essential effect. We will call this “minimal lifting”. It means that the course taken by the lifted search is the same as the ground search would take. In the case of Davis-Putnam we mean that the same set of branch variables and values would be taken; in local search we would go through the same set of states.

The concept of “sameness” is a little trickier in the case of algorithms involving random choices – in the Davis-Putnam procedure we might break tie breaks randomly, and in local search we have intrinsic use of stochastic steps. Such steps involve the random selection of an element from a set, and in practice the set is represented as specific data structure such as a list and different implementations might end up with a different order, so even if the sets are the same in two implementations, they might select different elements. We do not count this as a real difference as it is at the same level as changing the pseudo-random number generator, and will make no difference to the results in a stochastic sense. Even this difference could be avoided by imposing an order on all the literals, and always ordering the sets before selection. If this were done, then in minimal lifting

the sequences of states in the search would be exactly the same in the lifted and non-lifted cases.

The fact that we mostly consider minimal lifting has an important ramification: we do not really care much about the search itself. All the details of exactly which particular flavor of GSAT or WSAT we use, or parameter settings such as noise and so forth, are irrelevant to the issues discussed here. In minimal lifting we only care about handling quantified clauses in a way that looks to the search as if they were ground, but does not suffer the explosion of memory and also makes the search moves themselves faster. It is only important to us that the steps taken by the search engine can be done in less time and with less memory usage. It is not important (in minimal lifting) whether or not the search engine should be doing these steps at all.

We will use “non-minimal lifting” to mean cases when the lifting directly affects the course of the search itself. We will return to this in Section 11.2.

In this chapter we are doing only minimal liftings: we are not changing the algorithms at all, we are just rephrasing them in terms of subsearch. We will look at WSAT first as it is somewhat simpler than Davis-Putnam, but first we consider the simple task of checking whether or not an assignment is a model.

5.3 Model Checking

As a simple example to make an easy introduction and to become familiar with the notation, consider the task of checking whether a complete assignment satisfies the constraints. Consider the pseudo-code from Figure 16(a). We see that the most relevant subsearch problem in this case is the checking problem. The

checking problem determines for each quantified clause whether or not it contains a violated ground clause.

```

proc MODEL-CHECK( $P$ )
  if  $S(C, P, 0, 0) = \emptyset$  return  $P$  is a model of  $C$ 
   $c :=$  FREELY-SELECT( $S(C, P, 0, 0)$ )
  return  $P$  is not a model, a witness to this is  $c$ 
end

```

(a)

```

proc MODEL-CHECK( $P$ )
   $c :=$  FREELY-SELECT( $S(C, P, 0, 0)$ )
  if  $c = fail$  return  $P$  is a model of  $C$ 
  return  $P$  is not a model, a witness to this is  $c$ 
end

```

(b)

FIGURE 16. Simple lifted model check written in terms of subsearch problems from Section 4.6. P is assumed to be a total assignment. If P is not a model then a violated ground clause is returned. (a) Conceptual version. (b) Realistic version where FREELY-SELECT is used to also solve the checking problem.

In the case that P is not a satisfying assignment, we also added the ability to return a witness to this fact: we return a violated clause, obtained from the the set $S(C, P, 0, 0)$ by use of the free selection subsearch problem.

In practice, the two subsearch problems can be folded together as in Figure 16(b): we try to FREELY-SELECT a violated clause, and if this fails we deduce P is a model. In any case it is clear that the majority of the time will typically be spent in the subsearch subroutines rather than the routine itself.

We should also remember, as discussed in Section 4.6, that although we wrote selection in the form FREELY-SELECT($S(C, P, 0, 0)$) it is not to be thought of as a function that takes the entire set $S(C, P, 0, 0)$ as input and then selects one

at random, but as a function `FREELY-SELECT($C, P, 0, 0$)` that is implemented as directly as possible.

Of course, in the case of model-checking, P does not change. In contrast, in a search engine, P will constantly be changing, and it is vital to exploit the fact that changes are small and information can be changed incrementally. We now discuss some points of how to do this in the lifted case.

5.4 Exploiting Incremental Changes

An essential part of implementing ground solvers such as WSAT or the Davis-Putnam procedure is to exploit the fact that the changes made by the search engines to the assignment P are relatively small. Exploiting incremental changes will also be essential to the speed of lifted search engines. Typically, such incrementality means responding to the change of the value of a single literal.

There are two components of such incremental maintenance: syntactic and semantic. This split arises for the same reasons as discussed in Chapter IV. There are changes that depend only on the literal being changed, and not the rest of the assignment: these we will call syntactic. Other changes depend not only on the literal being changed but also on the other variable assignments. We shall call these semantic.

Handling the semantic case will consist of maintaining various $S(C, P, u, s)$ sets as P changes. This is a difficult issue that we will touch in this chapter, and also revisit later after having discussed more about implementing subsearch itself (Chapter VII).

5.5 Syntactic Restrictions and Static Indexing

By the “syntactic component of incrementality” we mean the steps in search algorithms that restrict attention to those clauses that can be affected by a change in a given literal. For example, in unit propagation, instead of redoing complete propagation every time a literal changes, we can look only at the clauses containing that literal. Such use of incremental maintenance is associated with the syntactic restrictions $R(C, l)$.

In ground solvers, we typically take care of this by a static indexing scheme. Before the search is started we can build, for every literal, a list of pointers to all clauses containing that literal. Similarly, in lifted solvers, we can build the sets of clauses $R^-(C, l)$ (see Definition 4.4.1) for every literal, in advance of the search itself.

The key point is that since these restrictions do not depend on P they can all be done in advance of the search. Lifting does not really have a strong effect (either good or bad) here.

Hence, the syntactic restriction mechanisms in Section 4.4 play the same role as the static indexing methods commonly used in ground solvers. Both are done in advance of the search itself and do not incur any significant runtime costs themselves. Therefore we do not need to optimize the calculation of the syntactic restrictions. We can essentially assume during the search that any syntactic restriction on the input clauses is available immediately.

It is important to keep this separate from the questions of semantic maintenance. This will have to be done dynamically, possibly via subsearch in the lifted case, and is not associated with indexing schemes even in the ground case.

5.6 WSAT

Now we return to the WSAT algorithm of Figure 5, to rewrite it, and its associated subroutines, in terms of subsearch problems. We remark that there is some relation between what we do here and work on converting GSAT to work with non-clausal formulas [63]. We will discuss this further in Section 10.2.

In converting an algorithm to use subsearch we have some freedom of choice as to how much work is delegated to the subsearch engine and how much is kept under the explicit control of the search engine itself.

In Figure 17(a) we give a version in which almost all control is passed off to the subsearch engine. The only access to the clauses is via the checking problem (to see whether we have solved the problem) and the uniform selection subsearch problem to pick a random element of $S(C, P, 0, 0)$. This version will be very inefficient if the subsearch engine solves the subsearch problems from scratch each time it is called. Instead, an efficient implementation should solve them incrementally, that is, it would keep track of the state P . After the search flips a variable the subsearch engine is informed so that it has a chance to do the needed incremental maintenance.

In WSAT, we shall be storing the set $S(C, P, 0, 0)$ explicitly.¹ This is feasible because WSAT is typically extremely effective at keeping to regions of the search space in which this set is much smaller than the entire set of clauses. In the lifted implementation discussed in Chapter VIII, the size of $S(C, P, 0, 0)$ has not proved

¹To simplify the discussions we will assume for now that only $S(C, P, 0, 0)$ is stored and maintained. In practice, more information can be, and should be, stored. See Chapter VIII, and in particular Section 8.7, for further discussion of this issue.

```

proc WSAT
  for  $t := 1$  to MAX-TRIES
     $P :=$  random total assignment
    for  $i := 1$  to MAX-FLIPS
      if  $S(C, P, 0, 0) = \emptyset$  return  $P$ 
       $c :=$  UNIFORM-SELECT( $S(C, P, 0, 0)$ )
       $l :=$  SELECT-LITERAL( $c, P$ )
       $P := P[l := \neg l]$ 
      inform subsearch engine that  $l$  has flipped
    end for
  end for
  return solutionNotFound
end

```

Subsearch
Subsearch

(a)

```

proc WSAT
  for  $t := 1$  to MAX-TRIES
     $P :=$  random total assignment
     $S := S(C, P, 0, 0)$ 
    for  $i := 1$  to MAX-FLIPS
      if  $S = \emptyset$  return  $P$ 
       $c :=$  select a random element from  $S$ 
       $l :=$  SELECT-LITERAL( $c, P$ )
       $P := P[l := \neg l]$ 
      UPDATE( $S, l$ )
    end for
  end for
  return solutionNotFound
end

```

The enumeration subsearch problem

Not subsearch!

Incrementally maintain S

(b)

FIGURE 17. WSAT in terms of subsearch: We give two ways to rewrite the WSAT of Figure 5 in terms of subsearch. (a) Passes almost all the work to the subsearch. The subsearch engine should handle UNIFORM-SELECT in an efficient incremental fashion. (b) The search itself takes control of the set of unsatisfied clauses, and their incremental maintenance.

to be a problem (however, see also the comments in Section 7.4.2).

In order to make this clearer we will also write WSAT so that the main search routine itself explicitly stores and maintains $S(C, P, 0, 0)$. This is illustrated in Figure 17(b). Note that since the routine now explicitly stores $S(C, P, 0, 0)$ the selection of a random clause is no longer a subsearch problem.

Versions (a) and (b) of WSAT use different sets of subsearch problems. It is quite common that a search can be written many different ways. This reflects different decisions about the exact dividing line between search and subsearch, and that many of the subsearch problems themselves are interrelated (as discussed in Section 4.6). Whether the maintenance of $S(C, P, 0, 0)$ is considered part of the search, as in (b), or as part of the subsearch, as in (a), is not so important. What is important is that something must maintain the set and so do the equivalent of the $\text{UPDATE}(S, l)$ of (b). Note that such freedoms do not affect the existence or utility of the conceptual split between search and subsearch: When making an assignment of tasks to the search or subsearch engines we do not change the structure as given in Figure 15.

5.6.1 Incremental Maintenance of $S(C, P, 0, 0)$

In WSAT we will rely on the set $S(C, P, 0, 0)$ being sufficiently small that it is possible to determine it explicitly. Hence, we start by solving the enumeration problem, and after that maintain the set incrementally using $\text{UPDATE}(S, l)$.

Note that the selected literal l is currently FALSE, since it is taken from a currently unsatisfied clause. Hence the flip will take l from FALSE to TRUE. As discussed in Section 3.3 we have two cases to consider, clauses containing l will

be “broken” (become unsatisfied), those containing $\neg l$ will be “made” (become satisfied).

The ground version of the update routine was given in Figure 8. The lifted, subsearch driven, version of this routine is given in Figure 18. The idea in both cases is the same.

5.6.1.1 Handling Makes

The only clauses that can be made are those already in S and containing l , hence we remove $R(S, l)$ from S . This is not a subsearch problem but is done with a simple scan of the set S .

5.6.1.2 Handling Breaks

The only clauses that can break are those containing $\neg l$ and with only one TRUE satisfying literal, where a “satisfying literal” is one that is TRUE and so causes the clause to be satisfied. Since $\neg l$ is currently TRUE then it must be the only satisfying literal. We assume that clauses are never tautologies, and hence the clause cannot also contain l . Hence, when l flips, the single satisfying literal flips and the clause breaks. Accordingly, all elements of $S(R(C, \neg l), P, 0, 1)$ will

```

proc UPDATE( $S, l$ )
   $S := S \setminus R(S, l)$                                 Remove clauses that are made
   $S := S \cup S(R(C, \neg l), P, 0, 1)$                 Add clauses that break
end

```

FIGURE 18. Lifted updating of the unsatisfied clause set in WSAT. This uses subsearch problems to update the set of unsatisfied clauses in WSAT in response to setting $l := \text{TRUE}$.

become elements of $S(C, P, 0, 0)$ after the flip and so should be added to the set.

More formally we might write

$$\begin{aligned}
 (l = \text{FALSE}) \in P & \quad \wedge \\
 P' = P[l := \text{TRUE}] & \quad \wedge \\
 l \in c \wedge c \in S(C, P, 0, 0) & \quad \longrightarrow \\
 S(C, P', 0, 0) & = [(S(C, P, 0, 0) \setminus R(S(C, P, 0, 0), l)) \cup \\
 & \quad S(R(C, \neg l), P, 0, 1)] \quad (5.1)
 \end{aligned}$$

The set of breaks can also be found by taking $R(C, l)$, removing l from each and finding the subset of these that have no TRUE literals, followed by taking the disjunction with l again. That is, the ground set of breaks can be written as

$$l \vee S(R^-(C, l), P, 0, 0) \quad (5.2)$$

where it is understood that the disjunction is distributed over the ground clauses obtained from the subsearch enumeration problem. In this case, even the incremental steps in WSAT involve finding sets $S(C, P, 0, 0)$ for an appropriate C , and so we will often focus on ways to find $S(C, P, 0, 0)$.

The order of handling the makes and breaks will not affect the correctness of the algorithm. However, it is better to do the makes first. If we did the reverse order we would be pointlessly checking the newly broken clauses to see if they were makes.

The ground implementation of WSAT directly exploits the fact that the set of interest is $S(R(C, \neg l), P, 0, 1)$. In the ground case, this set is obtained by a direct scan of *all* the elements of $R(C, \neg l)$ and selecting those cases in which there is just one satisfying literal. Since this task is very common (it also is used in the heuristics for variable selection), it is worthwhile to cache information about the number of satisfying literals in every ground clause. Hence, ground solvers store and maintain, for every clause, a count of the number of literals in the clause that are currently TRUE. This saves having to evaluate all the literals of a clause every time we want to know if there is just one TRUE literal. This is a very important savings for the ground solver, but it does not change the fact that the time to find all the breaks will still be $O(|R(C, l)|)$. In Section 7.5 we will see that, even without storing the literal count for each clause, the lifted version can improve on this.

5.6.2 Literal Selection in WSAT

Note that the literal selection schemes of Figure 6 make no direct reference to the clause database, and hence they need not be changed on lifting.

Counting the clauses that will be made (NUM-MAKES-IF-FLIP of Figure 7) is still a matter of scanning the explicitly stored set, and hence also does not refer to the clause database. However, for completeness we rewrite it in terms of the $R(C, l)$ notation. As far as subsearch goes, after the initialization the only interesting step is the num-breaks-if-flip routine in Figure 7. We give the subsearch version of these in Figure 19. (However, see Section 8.7 for a way to avoid subsearch on literal selection.).

```

proc NUM-BREAKS-IF-FLIP(l)
    Count the clauses in which l is the only TRUE literal
    return COUNT(S(R(C, ¬ l), P, 0, 1))
end

```

(a)

```

proc NUM-MAKES-IF-FLIP(S, l)
    Count the clauses that will be fixed by l := TRUE
    return COUNT(R(S, l))
end

```

(b)

FIGURE 19. Counting makes and breaks for WSAT using subsearch. (a) Counting breaks. (b) Counting makes within the unsatisfied set. Both (a) and (b) assume that *l* is currently FALSE (*l* will have been taken from a currently unsatisfied clause).

In summary, WSAT can be written in terms of subsearch by combining Figures 17, 18, 6, and 19. Although we only discussed WSAT we can expect that GSAT would be implemented in a similar fashion.

5.7 Unit Propagation

Converting unit propagation to rely on the subsearch problems is done much like the conversion of WSAT. In Figure 20 we give two conversions. Version (a) uses a recursive method, and relies heavily on the subsearch. Version (b) is iterative, keeping control of a container of literals yet to be propagated, and with eager assignment of any literals discovered. There are also other similar ways to implement the propagation.

In both cases it is worth noting that the sets $S(C, P, 1, 0)$ and $S(C, P, 0, 0)$ are constantly changing because *P* itself is changing as we discover propagations. The search and subsearch engine should be able to take account of this.

5.8 Davis-Putnam Procedure

The top-level control of the Davis-Putnam procedure, given in Figure 11, does not need to be changed because it makes no direct reference to the clauses

```

proc UNIT-PROPAGATE( $l, P$ )                               Propagate effects of  $l$  given  $P$ 
   $C_l := R(C, \neg l)$                                      Note that  $\neg l$  is false
  if  $S(C_l, P, 0, 0) = \emptyset$  return Contradiction
  while  $S(C_l, P, 1, 0) \neq \emptyset$                        Note that  $P$  is changing
     $c := \text{FREE-SELECT}(S(C_l, P, 1, 0))$ 
     $l' := \text{the literal in } c \text{ that is not valued by } P$ 
     $P := P \cup \{l' = \text{true}\}$                              Enforces unique way for  $c$  to be satisfied
     $P := \text{UNIT-PROPAGATE}(l', P)$ 
    if  $P = \text{Contradiction}$  return Contradiction
  end while
  return  $P$                                              No contradiction was found
end

```

(a)

```

proc UNIT-PROPAGATE( $l, P$ )
   $L := \{l\}$ 
  while  $L \neq \emptyset$ 
    select some  $l' \in L$  and remove it from  $L$ 
     $P := P \cup \{l' = \text{true}\}$ 
     $C_{\neg l'} := R(C, \neg l')$ 
    if  $C_{\neg l'} \neq \emptyset$ 
      if  $S(C_{\neg l'}, P, 0, 0) = \emptyset$  return Contradiction
      foreach  $c \in S(C_{\neg l'}, P, 1, 0)$ 
         $l'' := \text{the literal in } c \text{ that is not valued by } P$ 
         $L := L \cup \{l''\}$                                    Store  $l''$  for later propagation
      end foreach
    end while
  return  $P$ 
end

```

(b)

FIGURE 20. Lifted unit propagation. (a) A recursive way to implement unit propagation (c. f. Figure 9). (b) An iterative way to implement unit propagation (c. f. Figure 10(b)).

themselves. Instead access is only through unit propagation, which we have already covered, and the SELECT-LITERAL procedure given in Figure 12.

The final phase of SELECT-LITERAL also only uses unit propagation and so it only remains to lift the pre-selection phase, that is, the fragment:

```
foreach variable  $v$  not already valued by  $P$                                 Pre-selection phase
   $c_t(v) :=$  number of binary clauses in  $C$ , under  $P$ , and containing  $v$ 
   $c_f(v) :=$  number of binary clauses in  $C$  under  $P$ , and containing  $\neg v$ 
end foreach
```

The key observation is that, when we talk about binary clauses remaining after having enforced a partial assignment P , we are referring to ground clauses that have no TRUE literals, precisely two UNVALUED literals, and an arbitrary number of FALSE literals. That is, we are concerned with elements of sets $S(C, P, 2, 0)$. However, the heuristics want to know the number of binary clauses remaining that contain a given literal l . Thus, we can first restrict onto l , and, since l will be UNVALUED, we will have only one other UNVALUED literal in the clause. Thus, the subsearch problem needed by the heuristics is just $\text{COUNT}(S(R^-(C, l), P, 1, 0))$ and we can write

```
foreach variable  $v$  not already valued by  $P$                                 Pre-selection phase
   $c_t(v) := \text{COUNT}(S(R^-(C, v), P, 1, 0))$ 
   $c_f(v) := \text{COUNT}(S(R^-(C, \neg v), P, 1, 0))$ 
end foreach
```

5.9 Summary

In this chapter, we converted WSAT, unit propagation, and the Davis-Putnam procedure, to work via subsearch problems instead of via direct access to the clause sets. The parts of the algorithms that are left in the search code are fairly simple and will run much faster than the subsearch. It is clear² that the majority of the runtime will be spent solving subsearch problems (we include with this all incremental maintenance of sets such as $S(C, P, 0, 0)$). Hence, if we are to make the algorithms run efficiently with QCNF we will need to be able to handle the subsearch problems effectively.

In the next chapter we study the nature of the subsearch problems. In particular we look at their computational complexity, and find that they are NP-hard. This suggests it is appropriate to use search methods to solve the subsearch problems. In Chapter VII we will apply a simple backtracking method to the subsearch.

²We have also checked this by profiling the implementations. This will also become clear in Chapter VIII.

CHAPTER VI

THE COMPUTATIONAL COMPLEXITY OF SUBSEARCH

In this chapter, we look at the computational complexity of subsearch problems and see that they are indeed hard problems.

Unfortunately, we have no better method to solve subsearch problems on a set C of quantified clauses than just to treat each quantified clause separately. Hence, we will usually talk about subsearch in the context of a single quantified clause, c , and study subsearch problems on $S(c, P, u, s)$ rather than $S(C, P, u, s)$.

When writing a general clause we need to account for predicates having different arities. We do this by writing the arguments of a predicate as a tuple. We take

$$c = \forall v_1, \dots, v_n. \bigvee_a l_a(t_a) \quad (6.1)$$

where each tuple t_a is built from the domain variables v_1, \dots, v_n and possibly constants. Also, l_a is a predicate symbol or its negation. For example, $\neg p(i, 4)$ becomes $(\neg p)(t)$ with $t = \langle i, 4 \rangle$.

We will review the idea of a “Constraint Satisfaction Problem” (CSP), as we will need it extensively in our discussion of subsearch on $S(c, P, 0, 0)$. We then consider the checking problem for $S(c, P, 0, 0)$, and show it can be converted to a search problem in a CSP. We use the connection with CSPs to show that the checking problem is NP-complete.

We then introduce a slight extension of CSPs (called CCSPs) that we will define in such a way that they play the same role for $S(c, P, u, s)$ as CSPs play for $S(c, P, 0, 0)$. This allows us to extend the NP-completeness results to $S(C, P, u, s)$. (CCSPs themselves are not necessary to understanding the rest of the thesis: The definition and results are included for reasons of completeness, and because the CSP-based methods discussed in Chapter IX might, in future work, be extended to CCSP methods.)

Writing the subsearch problems in terms of CSPs will also be used to motivate algorithms to solve the subsearch.

6.1 Constraint Satisfaction Problems (CSPs)

CSPs have many applications in combinatorial optimization problems, and have been studied very extensively (see for example [70, 1]), and so we only review the most salient features here. Firstly, their definition:

Definition 6.1.1

A *constraint satisfaction problem* (CSP) has the following elements:

1. A set V of n finite domain variables v_1, \dots, v_n with domains D_i .
That is, the v_i take values in D_i by means of bindings such as $v_i = d_i$ with $d_i \in D_i$.
2. A set H of constraints. Each constraint h is a pair consisting of
 - (a) a tuple $t[h]$ of variables with $t \subseteq V$
 - (b) a set $G[h]$ of allowed tuples of bindings of variables to values, so-called "Goods", for the tuple t

A set of bindings, or domain variable assignments, B , for the domain variables is said to satisfy a constraint h iff the restriction of B to the variables in $t[h]$ gives a set of bindings that is equal to some element of $G[h]$.

An assignment is a satisfying assignment iff it satisfies all the constraints. □

The *arity* of a constraint h is the number of variables it restricts, that is $|t[h]|$. The maximum arity over all the constraints is said to be the arity of the CSP. A CSP with arity k will be called a k -ary CSP. A *binary CSP* (BCSP) is a 2-ary CSP, that is all tuples $t[h]$ have size of at most two.

The *constraint hypergraph* is the hypergraph formed by taking variables as nodes, and the constraints H as hyperedges (recall that a hyperedge can link more than two nodes). In the case of a BCSP we have a graph rather than a more general hypergraph. Note that the hypergraph depends only on the tuples $t[h]$ and not the sets $G[h]$.

As an example, consider graph coloring. The nodes correspond to variables v_i taking values in a set D of colors. Each edge (v_i, v_j) in the graph gives rise to a constraint h_{ij} with

$$\begin{aligned} t[h_{ij}] &= (v_i, v_j) \\ G[h_{ij}] &= \{(v_i = c_i, v_j = c_j) \mid c_i \in D \ \& \ c_j \in D \ \& \ c_i \neq c_j\} \end{aligned} \quad (6.2)$$

The constraint can be left as a functional definition of the set $G[v_i, v_j]$ or converted to an extensionally defined set by enumerating possibilities. Notice that, in this

case the constraint graph is the graph itself.

Consider a k -ary CSP for fixed k . For any constraint h the size of $G[h]$ is bounded by d^k where d is the size of the largest domain. Since this is independent of the number of domain variables n we can say that checking a constraint is constant time (with respect to n). Hence, checking whether a domain variable assignment satisfies the constraints will be polytime. That is, the satisfiability problem for a k -ary CSP is in NP.

However, we have just seen that even BCSPs contain graph coloring which is NP-complete. Hence BCSPs are NP-complete. Similarly k -ary CSPs contain BCSPs and so are also NP-complete.

Many of the search methods discussed for SAT in Chapter III are also used to solve CSPs (see [1] for a recent survey). We will mostly be concerned with simple backtracking: that is, we try to build up a satisfying assignment for the domain variables by checking the constraints as soon as all the variables involved in a constraint have been assigned values. However, in Chapter IX we will also discuss the potential implications for subsearch of some more advanced CSP search methods.

6.2 Conversion of $S(c, P, 0, 0)$ to a CSP

As discussed earlier, we are considering a single, but arbitrary, quantified clause c written in the form given in (6.1). If we are looking for the elements of $S(c, P, 0, 0)$ then we need to find bindings for the v_i such that every literal of the disjunction is FALSE under the truth value assignment P . The condition that every literal of a disjunction is FALSE suggests that it will be clearer if we negate the

clause. Doing so we obtain

$$\neg c = \exists v_1, \dots, v_n. \bigwedge_a \neg l_a(t_a) \quad (6.3)$$

If P is fixed then the set of bindings for t_a that cause the literal $\neg l_a(t_a)$ to be TRUE are determined entirely by P and hence are also fixed.

This looks like the question of whether there is a solution to a set of constraints. We will show that this view is correct by mapping the problem to an associated CSP. We will call this CSP the “sub-CSP” of c to emphasize that it is associated with the subsearch.

The set of variables of the sub-CSP are taken to be the same as the universally quantified domain variables $\{v_1, \dots, v_n\}$ of the clause c . Each literal $l_a(t_a)$ in c becomes a constraint h_a in the sub-CSP with

$$\begin{aligned} t[h_a] &= t_a \\ G[h_a] &= \{t_a = d_a \mid l_a(d_a) = \text{FALSE under } P\} \end{aligned} \quad (6.4)$$

That is, the variables in the constraint h_a are just the arguments of the literal l_a , and the allowed sets of values for these variables are defined to be exactly those for which the literal $l_a(t_a)$ is assigned to FALSE by the (possibly partial) assignment P .

It is important not to confuse the above assignment of values to the domain variables, with the assignment P of truth values for the predicates.

A value assignment to the domain variables that satisfies all the constraints of the sub-CSP iff it assigns all the negated literals in (6.3) to FALSE. In this case,

it is an element of $S(c, P, 0, 0)$.

Theorem 6.2.1

A set of domain variable assignments satisfies the sub-CSP from c iff it corresponds to an element (ground clause) in $S(c, P, 0, 0)$. \square

Proof: By construction. The domain variables satisfy a constraint iff the associated literal is assigned to FALSE under P . An assignment hence satisfies all the constraints iff it causes every literal of the clause to be set to FALSE, i. e., iff it corresponds to an element of $S(c, P, 0, 0)$. \square

Corollary 6.2.2

The sub-CSP of a quantified clause c is satisfiable iff $S(c, P, 0, 0) \neq \emptyset$.

\square

Proof: Immediate. \square

It is crucial to note that this CSP is related to the subsearch and *not* the search itself (which, after all, is over QCNF not a CSP). The two search spaces, and requirements and meanings of the searches are quite distinct.

Also, note that the constraint graph of the sub-CSP depends on the syntactic structure of the clause, but not on P . In Chapter IX, we will suggest ways to exploit this observation.

As an example, consider the clause

$$\forall i, j, k, m. p(i, j) \longrightarrow r(j, k, m) \quad (6.5)$$

The associated CSP-graph is shown in Figure 21(a). Similarly if we had

$$\forall i, j, k. q() \wedge p(i, j) \longrightarrow r(j, k, 4) \quad (6.6)$$

we get Figure 21(b). The nullary predicate does not involve any nodes and so is just represented in the diagrams as a “self loop”. (Strictly speaking, we no longer have a graph, but this does not matter as we the diagram is only for illustrational purposes and not formal reasoning.) Of course, a nullary predicate is either satisfied, in which case we can ignore it, or not satisfied in which case the problem is manifestly unsatisfiable.

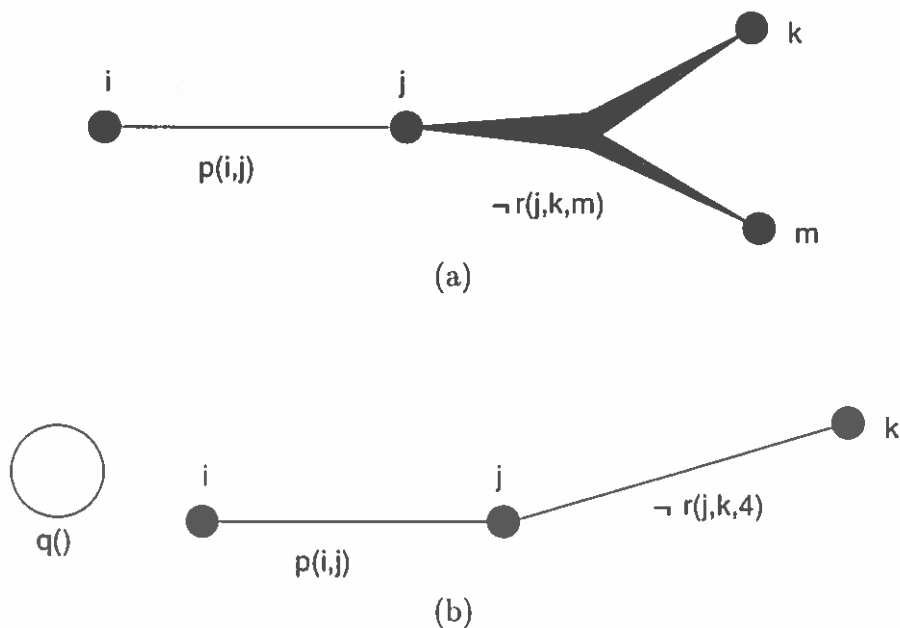


FIGURE 21. Constraint graphs of sub-CSPs from two simple clauses. (a) From (6.5). The predicate $r(j, k, m)$ becomes a hyperedge connecting the j , k and m nodes of the CSP. (b) From (6.6), the nullary predicate $q()$ becomes a constraint not attached to any nodes.

6.3 Complexity of the Checking Problem for $S(C, P, 0, 0)$

We have just seen that the question of whether a single quantified clause contains an unsatisfied ground clause can be converted to the question of whether an associated CSP, the sub-CSP, has a satisfying assignment. The conversion of a quantified clause to the sub-CSP is very direct, the syntactic structure of the clause converts directly to the constraint graph of the sub-CSP. The partial assignment P directly gives the constraints themselves. A model of the CSP maps directly to an element of $S(c, P, 0, 0)$.

Intuitively it seems clear that the sub-CSPs that are produced are not special CSPs but are fairly general, and since the CSP satisfaction problem is NP-complete it should not be surprising that finding an element of $S(c, P, 0, 0)$ will also be NP-complete. However, before proceeding, we first need to specify the size measure that we are using.

Definition 6.3.1

The size measure for the complexity results on subsearch is the space needed to specify the problem $S(C, P, u, s)$ using QCNF. \square

It is crucial to note that this is quite different from the size measure that we would use for the corresponding ground theory. The number of quantifiers in C can increase linearly with C . But the size of the ground theory is exponential in the number of quantifiers. Hence the size of $Gr[C]$ can be exponentially larger than the size of C .

We now have

Theorem 6.3.2

Determining whether there are any elements in $S(C, P, 0, 0)$ is NP-complete. That is, the checking problem is NP-complete. \square

Proof: it is straightforward to see that the problem is in NP. If the answer to the checking problem is “yes” then the witness to this is an unsatisfied ground clause in C . Assuming that $C = \bigwedge_i c_i$ then we can non-deterministically guess the appropriate clause c_i , and the appropriate set of domain variable bindings in c_i in order to give a ground clause. The resulting ground clause is certainly smaller than C itself, and hence checking that it is unsatisfied is polytime in the size of the problem.

To see that the problem is NP-hard we reduce an arbitrary BCSP to the case of a theory with a single clause c and specially constructed assignment P . The c and P that we construct will have the property that the associated sub-CSP is just the original BCSP.

For each constraint h_a of the BCSP we create a binary predicate p_a taking as arguments $t[h_a]$ i. e. the variables associated with the constraint. We take c to be

$$c = \forall v_1, \dots, v_n. \bigvee_a p_a(t_a) \quad (6.7)$$

For the sub-CSP from c to be the original BCSP we need to define P so as to satisfy

$$G[h_a] = \{t_a = d_a \mid l_a(d_a) = \text{FALSE under } P\} \quad (6.8)$$

Hence, we take P to be the total assignment such that $p_a(d_a)$ is FALSE if $d_a \in G[h_a]$ and is TRUE otherwise.

Then, by construction, there will be set of bindings for the v_i giving an unsatisfied ground clause iff every $p_a(t_a)$ is bound to FALSE by P , and hence iff the binding corresponds to a satisfying assignment of the BCSP. Thus, finding an element of $S(C, P, 0, 0)$ is equivalent to finding a solution of the arbitrary BCSP and therefore is NP-hard. \square

In fact we can make quite strong restrictions on the clause c and still stay NP-complete:

1. Binary predicates with all domains of size 3 is NP-complete by reduction from the BCSP corresponding to 3-coloring of a graph.
2. All domains of size 2, and with predicates limited to arity 3 is NP-complete due to an earlier proof of NP-completeness by reduction from 3SAT [26].

In the model checking considered in Figure 16(b) we could paraphrase the code as “we searched for a broken clause but failed and so P was a model”. This is very suggestive of co-NP, and indeed we have:

Corollary 6.3.3

Determining whether $S(C, P, 0, 0) = \emptyset$ is co-NP-complete. \square

Proof: By definition, as it is just the complement of the checking problem. \square

Finding a ground clause from the set C is just the free selection problem, hence, in summary:

1. FREELY-SELECT($S(C, P, u, s)$), and the checking problem are NP-complete.

2. Testing $S(C, P, 0, 0) = \emptyset$ is co-NP-complete.

6.3.1 Comparison with Complexities for SAT

At first sight these results might seem very counterintuitive, after all we “know” that showing $S(C, P, 0, 0) = \emptyset$ just means showing that P satisfies the constraints C , and this is “clearly polytime”.

The resolution of this paradox lies in our previous comments about the different size measures being used. The hardness of the sub-CSP is in terms of the number of its variables, but these variables are the quantified domain variables of C . Hence the expected exponential behavior is in terms of the number of quantifiers in C and *not* in terms of the number of ground literals.

Of course the disparity can be even more dramatic in some cases. As an extreme example consider the case of a quantified clause in which all the literals are positive, and take the assignment P to set all positive ground literals to TRUE. Then checking that the quantified clause is satisfied is just a matter of checking that all the literals in it are indeed positive, and hence polytime in the size of the quantified clause. In the ground case we would have to check every ground clause separately, but the size can increase exponentially on grounding, and hence even in this simple case checking the ground theory is not polytime.

Also, in practice, when we are given a sequence of search problems, the size parameter for the sequence will be the domain sizes, and the clause set C will otherwise remain constant. For example, in the logistics problem given in Chapter II the axioms are fixed, instead the numbers of planes and cities, etc, just affect the sizes of the domains. If the number of domain variables is n and the

maximum domain size is d then the search space for the sub-CSP is $O(d^n)$. If the axioms are fixed then n is fixed, and in this case we have just a polynomial in the domain sizes. Hence, solving the CSP is polytime with respect to the domain sizes. That is, if the axioms are fixed grounding the theory only has a polynomial effect on the size, and can be ignored for the purposes of NP measures of complexity.

Given the perhaps rather formal nature of this result one might wonder why it matters. As explained so nicely in Garey and Johnson's book [22] the practical impact of a complexity result is to suggest the types of algorithms that might be most effective, and change expectations about what is feasible. The same is true in our case: Since subsearch is NP-complete, we shall be approaching it using algorithms designed for such problems rather than expecting to find a fast general solution. Indeed, given the effort we have spent translating subsearch to search in a sub-CSP, it should not be too surprising that we will be concentrating on the application of CSP-search methods to the subsearch.

6.4 Other Subsearch Problems for $S(C, P, 0, 0)$

So far we have covered the free-selection and checking problems. What about the others? The counting problem is harder yet. The mapping from $S(c, P, 0, 0)$ to a CSP was one-to-one, and so counting the size of $S(c, P, 0, 0)$ means counting the models of a CSP. However, one case of such a CSP is 3SAT: just take all predicates to be arity 3, all domains to have size 2, and adjust the assignment P so that each predicate converts to a clause. It follows that the counting problem is at least as hard as counting the models of a 3SAT theory, which is #P-complete [22].

The enumeration and iteration problems can be potentially exponential time

for the simple reason that there are CSPs which have an exponential number of solutions. Storing $S(c, P, 0, 0)$ might require exponential space but for the iteration problem we do not need to store all the elements and so it will only require linear space.

Finally, we have the uniform selection problem. This requires us to randomly select a model from the sub-CSP and according to a uniform distribution. The obvious method is to first count the number of models, randomly chose a number up to this limit, and then use iteration until we reach that number. Clearly this is potentially exponential time. In some systems, there are methods based on Markov chains that allow us to do somewhat better than this naive iteration, but they require accepting approximations [33].

6.5 General $S(c, P, u, s)$ and Counting-CSPs

So far we have looked at just the case of $S(c, P, 0, 0)$, and converted to a CSP. This CSP viewpoint is helpful: The complexity of subsearch became manifest. We will also see that CSP search methods are useful for solving the subsearch problems. Hence, we would like to treat the general $S(C, P, u, s)$ problems in a similar fashion.

In the case of $S(c, P, 0, 0)$, we only needed to know whether a literal in c was FALSE, and this corresponded exactly with whether a constraint in the sub-CSP was satisfied. For general u, s we need to handle the three truth values TRUE, FALSE and UNVALUED and so for each truth value we will introduce corresponding types of constraints. In order to formalize this we make the following slight extension of CSP ideas:

Definition 6.5.1

A counting constraint satisfaction problem (CCSP) has the following elements:

1. a set V of finite domain variables v_1, \dots, v_n with domains D_i and $v_i \in D_i$
2. a set M of types of constraints
3. a set H of constraints. Each constraint h consists of
 - (a) a tuple $t[A]$ of variables (exactly as for a CSP)
 - (b) for each $m \in M$ a set $G[h, m]$ of allowed tuples of values for the tuple $t[h]$ in the type m . We demand that for every h the sets associated with all the different values of m form a partition of the set of all bindings for $t[h]$. That is, every set of possible bindings for the tuple $t[S]$ is a member of precisely one set $G[h, m]$, in which case we say that the binding is of type m .
4. A set of range restrictions R_m : one for each constraint type. A range restriction is just a sub-interval of $[0, \dots, |H|]$. In some cases the intervals will be tight, $[N_m, N_m]$ and then we call them count-restrictions, N_m . Conversely, if a range restriction R_m is the entire interval $[0, \dots, |H|]$ then we will say that type m is unrestricted.

An assignment B is said to satisfy a constraint h with type m iff the restriction of B to the variables in $t[h]$ gives a set of bindings contained in $G[h, m]$

An assignment is a satisfying assignment iff for every element m of M the number of constraints satisfied with type m is within the range R_m . □

(In constraint programming we can do something somewhat similar by reifying constraints. Reifying a constraint means creating a new boolean variable that says whether or not the constraint is satisfied. We can then place constraints on these new reifying variables, and obtain effects similar to the range restrictions in the CCSP.)

The standard CSP corresponds to the case of just two constraint types: “good” and “bad”, with the count restriction $N_{bad} = 0$. Since every edge has precisely one type this just means that every edge has to be “good”. In fact, any case with $|M| \geq 2$ but in which the counts are restricted to zero for all but one of the types is just a standard CSP again.

To verify that an assignment satisfies the CCSP we can classify the type of every constraint, and count the number of each type, and check these numbers against the restrictions. All this can be done in time polynomial in the size of the problem. However, we have just seen that CCSPs contains CSPs as a special case. Hence, satisfiability of a CCSP is also NP-complete.

Note that the tuple for each constraint is independent of types, and so the notion of constraint graph is not affected by the constraint types. Instead, each edge has a number of possible types, and we have overall requirements on numbers of edges in each type. In Chapter IX, we shall discuss some CSP methods that exploit the structure of the constraint graph, and so we hope that many CSP methods can be extended to CCSPs.

6.6 Conversion of $S(c, P, u, s)$ to a CCSP

We will convert the subsearch problems for $S(c, P, u, s)$ to a CCSP with three types S , U , and F corresponding to TRUE (satisfying), UNVALUED, and FALSE respectively.

Definition 6.6.1

A (u, s) -CCSP is the special case of a CCSP with types $M = \{F, U, S\}$ and the restrictions that $N_U = u$, and $N_S = s$, but N_F is unrestricted.

□

In the case $u = s = 0$ every edge must be satisfied with type F , and we just have a standard CSP. We can now directly generalize the mapping used for the case of $S(c, P, 0, 0)$. Take the set of variables of the sub-CSP to be the same as the universally quantified domain variables $\{v_1, \dots, v_n\}$ of the clause c . Each negated literal $\neg l_a(t_a)$ in the conjunction in (6.3) becomes a constraint h_a in the CCSP as follows:

$$\begin{aligned}
 t[h_a] &= t_a \\
 G[h_a, S] &= \{t_a = d_a \mid l_a(d_a) = \text{TRUE under } P \} \\
 G[h_a, U] &= \{t_a = d_a \mid l_a(d_a) = \text{UNVALUED under } P \} \\
 G[h_a, F] &= \{t_a = d_a \mid l_a(d_a) = \text{FALSE under } P \} \tag{6.9}
 \end{aligned}$$

Since every atom is set to precisely one of TRUE, UNVALUED, or FALSE it follows that the sets $G[h_a, S]$, $G[h_a, U]$, and $G[h_a, F]$ do form a partition of the possible bindings for the tuple t_a .

We then have:

Theorem 6.6.2

A set of domain variable assignments satisfies the (u, s) -CCSP iff it corresponds to an element (ground clause) in $S(c, P, u, s)$. \square

Proof: By construction. The numbers of U and S types are constrained in exactly the same way as the numbers of UNVALUED and TRUE literals in the clause. \square

Corollary 6.6.3

The (u, s) -CCSP for a clause c is satisfiable iff $S(c, P, u, s) \neq \emptyset$. \square

Proof: Immediate. \square

6.6.1 Why Have Ranges in the CCSP?

The reason for allowing ranges in the definition of a CCSP is that we can also use the union-subsearch problems of page 80.

For example, in unit propagation we wanted to find elements of $S(c, P, 0, 0) \cup S(c, P, 1, 0)$ which corresponds precisely to the range restriction $R_U = [0, 1]$ with the count restriction $N_S = 0$.

The case of ranges of the form $[0, \dots, N_m^{max}]$ might be particularly useful, especially as pruning in a backtracking search of the CCSP will then be straightforward. The usage of subsearch in unit propagation is a prime example of this: see Section 7.2.

We can also indicate such cases by just replacing the numbers u or s with

appropriate ranges. Thus, we can talk of

$$S(c, P, [0, 1], 0) = S(c, P, 0, 0) \cup S(c, P, 1, 0) \quad (6.10)$$

and the corresponding $([0, 1], 0)$ -CCSP.

6.7 Complexity of the Checking Problem for $S(C, P, u, s)$

We need just one complexity result for CCSPs. We present it for the case of just the types $\{F, U, S\}$, but it also extends to more general CCSPs.

Definition 6.7.1

Satisfiability of a (u, s) -CCSP

INSTANCE: An instance of a (u, s) -CCSP

QUESTION: Is there an assignment satisfying the restrictions on the U and S types? □

Note that the count-restrictions are not part of the input.

Theorem 6.7.2

Satisfiability of (u, s) -CCSP is NP-complete. □

Proof: Checking a solution in a CCSP is polytime, the particular range restrictions used do not affect this, and so the problem is in NP.

To show that it is NP hard, we will reduce from a CSP. The idea is to take the CSP and add extra predicates to absorb the u and s counts. Thus, take the CSP together with $u + s$ new constraints: take u of these to have

$G[h_{new}, S] = G[h_{new}, F] = \emptyset$, and s to have $G[h_{new}, U] = G[h_{new}, F] = \emptyset$. Then in any assignment these new constraints always give precisely u constraints of type U and s of type S . In this case an assignment is a solution of the entire (u, s) -CCSP iff the assignment restricted to the original constraints is a $(0, 0)$ -CCSP solution. But a $(0, 0)$ -CCSP is just a CSP: the original CSP, and hence NP-hard. \square

Note that it is necessary to prove the above result and not just rely on the NP-completeness of CCSPs, because it could have been the case that the special case used for counts caused us to lose the completeness. The essence of this problem is simply that all but one of the types have fixed count-restrictions, and NP-completeness can also be shown in this general case.

We now easily have

Theorem 6.7.3

Determining whether there are any elements in $S(C, P, u, s)$ is NP-complete. That is, the checking problem is NP-complete. \square

Proof: Clearly in NP. To show it is NP-hard just reduce from satisfiability of a (u, s) -CCSP in exactly the same way as we reduced from a CSP for the case of Theorem 6.3.2 \square

Also, of course:

Corollary 6.7.4

Determining whether $S(C, P, u, s) = \emptyset$ is co-NP-complete. \square

The complexity results were initially proved directly in QCNF itself by Ginsberg [26] by a reduction from 3SAT. However, the main aim here is not so much

to prove the complexity results, which are not surprising given the initial translation of the checking problem for $S(c, P, 0, 0)$ to a CSP, but rather to set up the formalization in terms of CSPs and CCSPs.

The goal is that formalizing a CSP-like search problem of the type needed for subsearch on $S(C, P, u, s)$, but not referring to the quantified clauses, will make it easier to reason about the (sub)-search processes. Also, we suspect many CSP search methods will transfer to CCSPs and we can then also transfer them to subsearch. Using some kind of CCSP as an intermediate step might ultimately help in clarifying ideas for subsearch. That complexity results can also be obtained directly from the CCSP is a bonus.

6.8 Summary

In this chapter we translated the checking and free selection subsearch problems for $S(C, P, 0, 0)$ into search problems in a CSP. We defined a CSP-like class of problems called counting-CSPs (CCSPs), and then were able to translate subsearch problems for $S(C, P, u, s)$ into search problems in a CCSP.

We also saw that the subsearch problems are NP-hard with respect to the size of the quantified clauses, and hence worthy of being called search problems. This hardness is quite distinct from the hardness of the equivalent ground SAT theories. Indeed, the obvious algorithms to solve the subsearch are polynomial with respect to the size of the ground theory. Instead, the hardness originates in the fact that the lifted theory might be exponentially smaller than the ground theory. However, the point is that subsearch problems are NP-hard in a meaningful sense and so should be solved using the best *search* methods.

CHAPTER VII

INTELLIGENT SUBSEARCH SPEEDS SEARCH

So far our motivation has been to show that search algorithms can be rephrased in terms of subsearch problems. In Chapter VI we saw that subsearch problems were indeed search problems, and could even be phrased in terms of search in a CSP (or CSP-like) formulation.

However, if these were merely formal tricks then subsearch would not be very interesting. In this chapter we shall exploit the formalization in terms of subsearch. We will see that existing algorithms that take the ground equivalents of QCNF-formulas, instead of the quantified clauses themselves, are effectively using generate-and-test for the subsearch. In general, we can expect to be able to do better than generate-and-test. This is equally true for the case of subsearch. We will see how using even the simplest intelligent search algorithms can lower the cost of subsearch. Lowering the cost of subsearch means that the search algorithms will run faster: they will follow the same sequence of states, but the “per-node” times will be reduced.

In considering the algorithms to be used for subsearch we note that we typically will need the subsearch problems to be answered with certainty, that is, the subsearch will need systematic and complete methods. Hence, in this chapter and for the implementation described in the next chapter, we consider the simplest backtracking method. That is, we use depth-first-search and check for potential

failure after every branch. In Chapter IX we will return to look at more advanced techniques.

It is vital to keep the search and subsearch conceptually quite distinct:

1. Search looks for truth assignments to the propositions in the theory, with the goal of satisfying all the quantified clauses.
2. Subsearch looks for assignments to domain variables in a quantified clause, with the goal of finding ground clauses that are needed by the search engine.

We first look at the simple case of $S(C, P, 0, 0)$ and the checking problem. We then look at more general $S(C, P, u, s)$. After looking at questions of “syntactic incrementality”, we ask the question of how much gain we might expect from intelligent subsearch. We then return to the important questions of how subsearch interacts with incrementality.

7.1 Pruning the Subsearch for $S(C, P, 0, 0)$

We first use simple model-checking as a starting example. We are given a set of clauses, and a total assignment P and just have to check whether there are any unsatisfied clauses. In Section 5.3, in particular in Figure 16, we saw that model checking is just checking that $S(C, P, 0, 0)$ is empty. But, by Corollary 6.3.3 we know that this is a co-NP problem, and hence we will need a complete search-based method to do this checking.

Consider a simple example with only one clause:

$$\forall v_1, \dots, v_Q. \neg r() \vee A[p_a, v_i] \tag{7.1}$$

where A is any disjunction involving the predicates of the theory with domain variables v_i , and suppose that P binds $r()$ to false. We immediately know that all the ground clauses are satisfied, because $r()$ will be a TRUE literal in every one of them. In this case, we can entirely prune the subsearch over the variables v_1, \dots, v_Q . In contrast, if the clause were first ground and presented to a ground solver it would have no choice but to look at all the ground clauses separately, because it has no way to know what they contain in advance.¹ But the number of ground clauses is exponential in the number of quantifiers Q . So, in this (admittedly extreme) example, an exponential amount of work in the ground case has been reduced to a constant amount of work in the lifted case.

Consider a slightly more complex example:

$$\forall v_1, \dots, v_Q. \neg r(v_1) \vee A[p_a, v_i] \quad (7.2)$$

with $r(2)$ set to TRUE and r assigned FALSE otherwise.

If we bind $v_1 = 1$ then the subsearch should immediately discover that the clause is satisfied independent of the other bindings. In this case it will prune further search with that binding, then backtrack and set $v_1 = 2$. At this point the subsearch will have to search through bindings for the variables v_2, \dots, v_Q , however, after doing so, it would soon discover that no other bindings for v_1 are relevant. In terms of the sub-CSP, we know that no model can have $v_1 = 1$ because it violates the unary constraint arising from $r(v_1)$ and so we should immediately

¹That savings of this kind are possible was initially observed by Ginsberg [26]. The subsearch framework arose as a way to generalize these initial WSAT-specific observations.

backtrack and try another binding for v_1 . Again, the ground solver looks at all the ground clauses, that is all the bindings of all the domain variables. At the least, we have reduced the work by a factor of the size of the domain of v_1 .

In terms of the associated sub-CSPs, in these cases, we would have nullary and unary constraints on the finite-domain variable v_1 , and any reasonable CSP algorithm would first enforce these constraints before proceeding to the other variables.

Hence, using even simple backtracking for the subsearch can lead to exponential reductions compared to the ground solvers.

More generally, we are trying to build a binding of the variables such that all literals are FALSE. We just want the subsearch to check that a literal passes this test as soon as we have bound enough variables that we can determine its value.

Normally when doing backtracking we would use a dynamic variable ordering, that is, the choice of the next variable to try binding would depend on the previous branch variables. For example in the Davis-Putnam procedure the result of the literal selection depends on the current state of the search. In trees that are deep this is an important way to reduce the search time: we want to select variables that have maximum propagation in order to reduce the number of variables remaining.

However, in our case the search tree is more likely to be shallow and bushy, that is, we have relatively few variables but each variable has many potential values. For example, in the logistics axioms we have at most four variables, but the variables are objects, planes, etc, and so might number in the hundreds. Furthermore, we are not doing propagation within the subsearch. This suggests that using a dynamic variable ordering is not likely to be worth the effort (at least not in our

first implementation). In any case we shall restrict ourselves to a static ordering of the variables.

If we have a static variable ordering then the search process can conveniently be represented as a set of nested for loops. To be concrete consider the simple example

$$\forall i, j. \neg r(j) \vee p(i, j) \quad (7.3)$$

Operationally the ground solver would evaluate $S(C, P, 0, 0)$ as shown in a way that is essentially equivalent to Figure 22(a). That is, it would take all the ground clauses that are generated and only then test them. We are equating a ground clause with a particular quantified clause and the set of domain variable bindings that generate it. A ground solver is doing the equivalent of generate-and-test. Generate-and-test is usually a bad way to do search, so it is not to surprising that it is relatively easy to do better.

Anyone writing the explicit for loops would be very likely to realize that there are many repeated and pointless tests of $r(j)$ and hoist this test out of the inner loop to obtain Figure 22(b). This simple hoisting of tests out off loops corresponds to the basic idea of backtracking, that is, to test constraints as soon as possible, and hence try to fail as soon as possible. In Section 6.1, we briefly discussed simple backtracking search to solve a CSP, and of course we are doing exactly the same process here. (Sub)search through bindings of domain variables of the quantified clause is directly matchable with searching the sub-CSP.

```

S := ∅
foreach i
  foreach j
    if ¬r(j)
      if p(i,j)
        insert < i,j > into S

```

At this point the clause is fully ground

(a)

```

S := ∅
foreach j
  if ¬r(j)
    foreach i
      if p(i,j)
        insert < i,j > into S

```

Tests a literal before the clause is fully ground

(b)

FIGURE 22. Code to search for violated clauses for the example of (7.3). A ground clause is equated with a tuple domain variable bindings. $S(C, P, 0, 0)$ is enumerated by means of simple nested sequence of loops and tests. (a) Naive form: Equivalent to what the ground theory would do. (b) Code after simple optimization by re-ordering the loops and then hoisting the test. Every time that the test on $\neg r(j)$ fails we save scanning the entire domain of i .

7.2 General Subsearch Pruning: $S(C, P, u, s)$

Backtracking search for $S(C, P, u, s)$ is also straightforward. We bind domain variables and keep track of literals that become ground and so have a value in P . We then keep count of the numbers of UNVALUED and TRUE literals according to the current bindings of the domain variables. We prune the subsearch whenever we have more than u UNVALUED literals or more than s TRUE literals.

The standard example, discussed in Section 5.7, is to find $S(C, P, 1, 0)$ when doing unit propagation. Actually, in this case we also want to simultaneously find $S(C, P, 0, 0)$ because if this has any elements we have a contradiction and the search engine will need to account for this.

Hence, rather than use the versions of unit propagation in Figure 20, we rewrite them so as to make it clearer that we can do a simultaneous subsearch on both $S(C, P, 1, 0)$ and $S(C, P, 0, 0)$ (for appropriate C). The new versions are given in Figure 23. Note they use the “union” form of the subsearch problems discussed in Section 4.6.6 (see also Section 6.6.1). The combined subsearch is more efficient, because we do not have to scan the same set of clauses ($C_{\neg t}$ or $C_{\neg t'}$) twice.

Hence, for unit propagation we really need to find $S(C, P, u, 0)$ with any $u \leq 1$. Alternatively, we might think of this as subsearch for $S(C, P, [0, 1], 0)$ where $[0, 1]$ denotes that either of these values is of interest (see also Section 6.6.1). In doing the associated subsearch, this gives the following pruning conditions:

1. If we find a literal set to TRUE, we can prune the subsearch.
2. Since we want $u \leq 1$, we just prune a branch on the subsearch if we have found two UNVALUED literals on that branch.

```

proc UNIT-PROPAGATE( $l, P$ )
   $C_{\neg l} := R(C, \neg l)$ 
   $c := \text{FREELY-SELECT}(S(C_{\neg l}, P, 0, 0) \cup S(C_{\neg l}, P, 1, 0))$ 
  if  $c = \text{fail}$  return  $P$  Propagation has completed
  if  $c$  is a contradiction return Contradiction
   $l' := \text{the literal in } c \text{ that is not valued by } P$ 
   $P := P \cup \{l' = \text{true}\}$  Enforces unique way for  $c$  to be satisfied
  return  $P := \text{UNIT-PROPAGATE}(l', P)$ 
end

```

(a)

```

proc UNIT-PROPAGATE( $l, P$ )
   $L := \{l\}$ 
  while  $L \neq \emptyset$ 
    select some  $l' \in L$  and remove it from  $L$ 
     $P := P \cup \{l' = \text{true}\}$ 
     $C_{\neg l'} := R(C, \neg l')$ 
    if  $C_{\neg l'} \neq \emptyset$ 
      foreach  $c \in [S(C_{\neg l'}, P, 0, 0) \cup S(C_{\neg l'}, P, 1, 0)]$ 
        if  $c$  is a contradiction return Contradiction
         $l'' := \text{the literal in } c \text{ that is not valued by } P$ 
         $L := L \cup \{l''\}$  Store  $l''$  for later propagation
      end foreach
    end while
  return  $P$ 
end

```

(b)

FIGURE 23. Another version of lifted unit propagation in terms of subsearch problems. These methods are very similar to Figure 20, but here we combine the subsearch on the relevant sets $S(C_{\neg l}, P, 1, 0)$ and $S(C_{\neg l}, P, 0, 0)$. (a) Recursive method. (b) Iterative method.

If we reach a leaf of the subsearch then we have found a ground clause with no TRUE literals, and $u = 0 \vee u = 1$ UNVALUED literals. Supposing the subsearch is being used by UNIT-PROPAGATE of Figure 23 then the two cases require different actions.

In the case of zero UNVALUED literals we have found a clause with only FALSE literals, that is a contradiction. This should be reported to the search engine. Also any further subsearch is pointless (because the search engine will just backtrack anyway) and so the subsearch can be terminated.

In the case of one UNVALUED literal we have discovered a chance to propagate. The literal that is UNVALUED should be reported to the search engine. In Figure 23(a), we enforce this literal in P , and then the recursive call restarts the subsearch. This restart of the subsearch is not very efficient. Instead it might be better to try to continue a subsearch once initiated. Hence, in Figure 23(b) if a literal is discovered, it is just stored for later propagation and the subsearch (the foreach loop) is continued.

7.3 Syntactic Incrementality

In Section 5.5 we discussed what we called “syntactic incrementality”, meaning the part of the vital incremental work in solvers that is concerned merely with the syntactic restrictions $R(C, l)$ (the projection onto the clauses containing l).

There are many places where we want to call a subsearch on a set of clauses that is not the total set of input clauses but a syntactic restriction. We already saw a case of this in unit propagation: in order to respond to a literal l changing, we want to do subsearch on $S(R(C, l), u, 0)$. Fortunately, as discussed in Section 5.5,

this is not a problem because the $R(C, l)$ sets can all be computed in advance. Also, as discussed in Section 4.4, they are just sets of quantified clauses themselves.

This also explains why we still usually discuss subsearch on a set C without necessarily referring to the fact that the relevant search might be done incrementally. If we make a point about subsearch with a set C then it could have been that the set arose as an $R^-(C, l)$ in an incremental portion of the search.

For example, if we had a set C , then it might have arisen from a change in a switch predicate $s()$ in the clause $\neg s() \vee C$:

$$R^-(\neg s() \vee C, \neg s()) = C \quad (7.4)$$

Hence, the set of clauses that can arise from using syntactic incrementality is essentially the same as QCNF itself, and so when discussing subsearch efficiency in general we might as well ignore whether the C is an input clause, or something arising from an $R^-(C, l)$ calculation.

It is important to note that the mere fact of doing incremental maintenance of this form does not replace intelligent subsearch.

7.3.1 Practical Usage of Syntactic Incrementality

In practice, the incrementality will be applied to predicates with arguments, and this will cause some of the variables to be fixed. For example, suppose we have a clause

$$c = \forall i, j, k. \neg s(k) \vee \neg r(j, k) \vee p(i, j) \quad (7.5)$$

and we wanted to find the effects of flipping $s(K)$ from FALSE to TRUE. We need to find unsatisfied clauses among those obtained from $\neg s(K)$. That is,

$$R^-(c, \neg s(K)) = \forall i, j. \neg r(j, K) \vee p(i, j) \quad (7.6)$$

One can also think of this in terms of rewriting the relevant portion of c as

$$s(K) \longrightarrow \forall i, j. \neg r(j, K) \vee p(i, j) \quad (7.7)$$

making it clearer that if $s(K)$ is suddenly turned on then we need to search the consequent for unsatisfied clauses.

The point here is that, in practice, we do allow l in $R^-(C, l)$ to be a literal with free variables and then build $R^-(C, l)$ with these free variables. When the subsearch engine is given a ground literal l , we just make the appropriate bindings in $R^-(C, l)$, obtaining a standard quantified clause with no free variables. We also index the sub-clauses on predicate symbols, and then use the binding for the arguments: e. g., for $R(C, s(2))$ we would look up (7.6) and bind K to 2.

The equivalent step in the ground case is build up static indexing from every ground literal to lists of ground clauses, i. e., the equivalent of binding the variables in the lookup of the correct list of clauses.

7.4 Expected Costs and Gains of Subsearch

So far we have seen some special cases in which backtracking subsearch can potentially give big gains over the generate-and-test style of subsearch inherent in a ground solver. But what kinds of gains might we expect in practice?

Firstly, we consider some cases when improving subsearch is not likely to lead to gains. The obvious case where subsearch pruning cannot be helpful is when there are no quantified clauses available. An extreme example is Random 3SAT, which by construction does not have any structure that a quantified clause could capture.

Also, intelligent subsearch is only likely to work if we have a large number of relevant clauses. Since most of the work in solvers is incremental we need to consider the typical size of $|Gr(R(C, l))|$. If this is small then we cannot expect to gain much advantage from an intelligent subsearch: for very small problems it can well be the case that the generate-and-test style of a ground solver is actually the best way. Again an extreme example here is Random 3SAT. Such theories are typically studied in regions where the number of clauses per variable is 4-5. Hence, the average number of occurrences of any given literal is only about 8-10, that is $GL(R(C, l)) \approx 9$ and it is hard to get much gain if the search space size is that small. Of course, in Random 3SAT we have no lifted clauses.

Another view of the advantages of backtracking subsearch is that it exploits common structure between sets of clauses. Pruning the subsearch before binding all the variables corresponds to finding a pattern of literals, that occurs in many ground clauses, but for which the pattern itself is sufficient to rule out the clause. For example, all the ground clauses of (7.1) have $r()$ in common but this shared (trivial) structure can be enough to eliminate them en masse.

In Random 3SAT, there is little chance that the clauses in $R(C, l)$ will have any literals in common, and hence will not have any shared structure that could be exploited.

Ideally, an implementation would recognize such cases where intelligent subsearch will give no gains and switch to a ground style in order to avoid the overhead associated with lifting.

Now return to truly lifted cases in which even $R(C, l)$ can be large. For example, in the logistics case the $R(C, l)$ will often still involve quantification over cities, planes, etc., and so could correspond to many ground clauses.

The first point to notice is that if we consider only worst-case behavior then we might not see much of a gain. For example, with $S(C, P, 0, 0)$ it could be that P satisfies all, or almost all, of the ground clauses but the TRUE literals are usually not discovered by the backtracking subsearch until it has bound all the variables. In this case we will end up enumerating all the clauses, and so do no better than a ground solver. (However, in Chapter IX we discuss some methods for subsearch that just rely on the syntactic structure and in some cases could beat the ground methods independently of P .)

This is a common facet of solving NP-complete problems. We can rarely say anything very interesting about the worst-case behavior of a search engine. However, the expected performance on instances taken from a realistic distribution can be much better than the worst case.

For concreteness, we will just talk in terms of the checking problem on $S(C, P, 0, 0)$. If the assignments of the values were random we could expect pruning to occur half the time we test a literal. If the clause is long then we will have many literals to test and it becomes likely that at least one of them will be TRUE and we can prune. Every time this happens we avoid the remaining subsearch problem and this could well be exponential in the size of the problem. We could probably

make an estimate of what the expected cost of subsearch for various classes of quantified clauses, but we do not pursue this for the simple reason that in practice P is not likely to be random. In practice, we are more concerned with the case in which P has “imbalanced” predicates.

7.4.1 Imbalanced Predicates

In practice we expect:

1. Literals in the clauses to be biased towards being negative.
2. The assignment P to be biased towards setting propositions to FALSE.

for reasons we will describe shortly.

The combination of these biases means that a literal from a clause is biased towards being TRUE. Typical subsearch problems require that there are no TRUE literals in the clause. It follows that we can expect to get more pruning in the subsearch than random choice would suggest.

The reason for expecting most of the literals to be set FALSE is firstly simple experience with such problems. The origin probably lies at least partially in the way that SAT represents problems. We often want to represent a quantity that is a function, or at least close to being a function. For example, in the logistics domain we have predicates encoding the positions of planes, but the position of a plane is a function (the plane cannot be in two places at once) and so the predicate `planeAt(p,c,i)` must be mostly FALSE whenever we are near to a solution of the problem.

A similar tendency towards relations being FALSE is also exploited by the closed-world assumption [55] in which it is convenient to assume that we just represent the TRUE predicates because we expect them to be far fewer in number than the FALSE ones. Similarly, in relational databases we store the relations that hold true, and expect them to be far fewer than the total number of possible relations.

If some predicate happened to have an imbalance towards TRUE then it seems likely that we would just negate the definition. In other words if there is an imbalance it often seems natural to represent the problem in such a way that FALSE is the default, and we use TRUE for the exceptions.

The second point is that the axioms themselves tend to be imbalanced, with a bias towards negative literals (e.g., inspect the axioms for the logistics domain). Part of the reason for this is probably that we often want to write down rules about conditions that apply in a specific context, that is a conditional statement about the domain. We can expect to often have quantified clauses of the form (or close to the form)

$$\bigwedge_i a_i(t_i) \longrightarrow a_C(t_C) \quad (7.8)$$

If at least some of the atoms $a_i(t_i)$ are biased towards being FALSE then we can expect to obtain pruning in the subsearch on such a clause.

Potentially, in cases when the imbalance is sufficiently strong, we might even hope that the cost of subsearch could become polynomial in the number of quantifiers, rather than the exponential behavior we would otherwise expect.

In contrast, the ground solvers are stuck with generate-and-test and its exponential cost, and have no way to exploit this imbalance.

7.4.2 Weighted Initialization in WSAT

So far we have just taken a passive approach to imbalanced predicates: If it happens, then we will exploit it. There is one case in which we need to take a more active approach and positively encourage an imbalance.

In the initialization stage of WSAT, as given in Figure 3.3 or Figure 5.6, we have to explicitly enumerate all the elements of $S(C, P, 0, 0)$. If the atoms are equally likely to be TRUE or FALSE then clauses with k literals will have a $1/2^k$ chance of being unsatisfied. If there are many relatively short clauses then the set $S(C, P, 0, 0)$ could easily become too large to manage. Also, the search engine will spend a lot of resources in an initial hill-climbing phase where it is just trying to move into a reasonable part of the search space.

We can expect that initialization with a bias towards FALSE will greatly reduce the size of $S(C, P, 0, 0)$ for the same reason we expected pruning in realistic situations: Most atoms will be set to FALSE but will occur negatively in quantified clauses.

This issue of wanting to start with an assignment that shares some statistical properties with real solutions is independent of whether we are lifting or not.

Even if we did not do weighted initialization we could still lift. One obvious way is for UNIFORM-SELECT to first count the elements, and then scan the clauses again with a suitably (randomly) chosen stopping point. In such a case, it would probably be better to give up on totally uniform selection of an element and just try

to approximate this via some sort of sampling scheme [33]. Doing approximately random selection until the size of $S(C, P, 0, 0)$ drops to a reasonable level, and then doing explicit enumeration would be a possible approach.

Thus, even if we did not bias the initial state we would probably be able to lift. However, it is more natural to bias the initial state. The usual preference for a random assignments is probably just a legacy from testing WSAT on Random 3SAT where, by construction, there is no bias towards either FALSE or TRUE.

Consider the pigeonhole problem described in Section 2.4. In particular, take the case of n pigeons and n holes. This is obviously a very easy problem and WSAT can go almost directly to a solution. However, if the initial state is not biased, then it can spend a long time making obvious removals of pigeons from holes. With a random assignment we start with about half the n^2 predicates $p(i, h)$ set to TRUE, but in a solution only n can be true, and we can only flip one predicate at a time. Thus, the hill-climbing phase must take time $O(n^2)$. However, if we weight the state so that $O(n)$ start as TRUE then finding a solution is $O(n)$ flips. As an example, we might just start with all atoms set to FALSE. Then WSAT will not make any mistakes when assigning pigeons and will take just n flips to find a solution.

In harder problems the time spent on initial hill-climbing is likely to be a small fraction of the total solution time and so such effects are less likely to be noticed.

7.4.3 Weighting for a Plane

To be concrete about the effects of weighting the initial assignment we return to the logistics example. Consider (2.22) again:

$$\forall p, a, b, i. \neg (a < b) \vee \neg \text{planeAt}(p, a, i) \vee \neg \text{planeAt}(p, b, i) \quad (7.9)$$

which says that the plane cannot be in two cities at the same time. In fact, this and (2.23) and (2.24) all have the same structure. They assert a property of the second argument of an arity three predicate and differ only in the predicate used. So let us consider the more general case

$$\forall v_1, v_2, v_3, v_4. \neg (v_3 < v_4) \vee \neg p(v_1, v_3, v_2) \vee \neg p(v_1, v_4, v_2) \quad (7.10)$$

We will use symbols N_i for domain sizes: $|v_i| \equiv N_i$, and we have $N_3 = N_4$ as v_3 and v_4 correspond to the same domain.

Also note that `planeAt` encodes a function – the location of the plane, and hence in any solution will be `FALSE` for most value of the arguments. The same is true for `at(o, c, i)` and `in(o, p, i)` and so we expect p also to be usually `FALSE`.

We will study the cost of enumerating $S(C, P, 0, 0)$ using various forms of pseudo-code given in Figure 24.

The most naive enumeration would use the code of Figure 24(a). However, if we were dealing with a ground solver then in the grounding process we would always evaluate $\neg (v_3 < v_4)$ and over half the ground clauses would be satisfied


```

foreach  $v_1$ 
  foreach  $v_2$ 
    foreach  $v_3$ 
      foreach  $v_4$ 
        if ( $v_3 < v_4$ )
          if  $p(v_1, v_3, v_2)$ 
            if  $p(v_1, v_4, v_2)$ 
              insert  $\langle v_1, v_2, v_3, v_4 \rangle$  into  $S(C, P, 0, 0)$ 

```

Plane or object
Time point
City
City

(a)

```

foreach  $v_3$ 
  foreach  $v_4$ 
    if ( $v_3 < v_4$ )
      foreach  $v_1$ 
        foreach  $v_2$ 
          if  $p(v_1, v_3, v_2)$ 
            if  $p(v_1, v_4, v_2)$ 
              insert  $\langle v_1, v_2, v_3, v_4 \rangle$  into  $S(C, P, 0, 0)$ 

```

Evaluated away on grounding

(b)

```

foreach  $v_1$ 
  foreach  $v_2$ 
    foreach  $v_3$ 
      if  $p(v_1, v_3, v_2)$ 
        foreach  $v_4$ 
          if ( $v_3 < v_4$ )
            if  $p(v_1, v_4, v_2)$ 
              insert  $\langle v_1, v_2, v_3, v_4 \rangle$  into  $S(C, P, 0, 0)$ 

```

Test is aided by weighting

(c)

FIGURE 24. WSAT initialization with a simple clause. Pseudocode, equivalent to simple subsearch, to find $S(C, P, 0, 0)$ for (7.10). (a) Most naive form. (b) Equivalent of the ground theory: Reduces cost by about half. (c) Version to exploit more pruning in the case that p is almost always false: This can reduce cost by a factor of $|v_4|$.

and removed. Hence, the code of Figure 24(b) is a fair reflection of what a ground solver would effectively be doing.

Now, suppose that the initial assignment is controlled by a weight w : With probability w an atom is set to `TRUE`, otherwise it is set to `FALSE`.

In Figure 24(b) we always have to do all of each of the two outer loops, and (about) half the time we will also incur the cost of the inner loops. That is, we have a complexity of $N_3 \cdot N_4 \cdot (\frac{1}{2} + \frac{1}{2} \cdot N_1 \cdot N_2)$ or $O(\frac{1}{2}N_1N_2N_4^2)$. The result is independent of the weight, and this is just a reflection of the general inability of the ground solver to exploit the imbalanced assignment.

An alternative way to hoist the tests of literals out of the loops is given in Figure 24(c). In the language of backtracking search this, just corresponds to taking the static order of the branch variables to be $[v_1, v_2, v_3, v_4]$ rather than $[v_3, v_4, v_1, v_2]$ (this is related to general issues in controlling recursive inference [69]).

With Figure 24(c), the test of $p(v_1, v_3, v_2)$ is only expected to succeed a fraction w of the time, and so the expected complexity is $N_1N_2N_3((1-w) + wN_4)$. The worst case complexity is still $N_1N_2N_3N_4$ because the randomized assignment might happen to set all the atoms to `TRUE` despite the weighting (assuming $w \neq 0$).

The usual random assignment is just the case $w = 0.5$, and then the expected runtime for Figure 24(c) is $O(\frac{1}{2}N_1N_2N_4^2)$ which is the same as we expected from the ground solver. However, this does suggest that if we want to reduce the cost we should take w to scale as $1/N_4$. This is, in fact quite reasonable: in the `planeAt` case, it just means that for a given plane and time-point we would expect $O(1)$ values of the city to give `TRUE`, which matches the properties we expect of a solution. Note that we prefer not to set $w = 0$ because it gives unrealistic

configurations (the plane starts out nowhere), and in any case it does not improve the complexity any further. With w scaling as $1/N_4$ the cost of subsearch drops by a factor of N_4 to $O(N_1N_2N_4)$.

In summary, without weighting we get some savings from pruning (in this case a factor of $\frac{1}{2}$) though the ground solver does just as well because of the occurrence of the fixed known predicate. With a good weighting we reduce the cost by a factor of N_4 , and of course N_4 can be substantially larger than 2.

Here, we found a good weighting scheme by hand. It would be good to automate this selection of weights as far as possible. However, this is a separate issue and will not be pursued in this thesis.

7.5 Semantic Incrementality

In Section 5.6.1, we mentioned that a ground WSAT solver stores and maintains, for every ground clause, the number of literals in the clause that are currently TRUE. This is used in order to find $S(R(C, l), 0, 1)$, which is used for the heuristics and updating within WSAT.

Storing this number saves having to scan the ground clause each time we want to know the number of TRUE literals. This is a substantial savings and helps make ground WSAT run quickly. However, it does not change the fact that the ground solver has to scan $|R(C, l)|$ clauses: That is, it is still linear in the sizes of the relevant ground sets, and hence potentially exponential in the size of the quantified clauses.

Hence, even with the usual storage of a count for each ground clause, the ground solver does not obtain any of the advantages of subsearch pruning.

It might well be possible that a more complicated indexing scheme for the all ground clauses would be able to achieve some of the same advantages of the subsearch pruning. However, the indexing scheme would have to be dynamic, to account for the changes in value assignments. It would also probably require yet more memory rather than solve the memory problems. Hence we do not pursue such a scheme (though it might well be reasonable for theories that are small enough to fit in memory).

Furthermore, one point of the thesis has been that storing every ground clause takes too much memory. So, the philosophy we take here is that storage requirements should be at worst linear in number of atoms, and *not* in the number of ground clauses. That is, we can store P or anything linear in $|P|$, but not anything that might be as large as $O(Gr[C])$.

Instead, in this thesis we do a “partially incremental” maintenance and just rebuild the changes to the needed sets on demand.

In the case of WSAT, by (5.2), we will need to find $S(R^-(C, l), P, 0, 0)$ whenever we flip l to FALSE. We will do this using exactly the same mechanisms of backtracking subsearch that we already discussed for $S(C, P, 0, 0)$.

Another potential approach [26] is to store a count for every clause, but to do so in a fashion that exploits regularity in the problem in order to be able to store the counts in a lot less space than $|Gr[C]|$. In this case finding the sets $S(C, P, 0, s)$ can be done in a way similar to the ground case, but without the cost in memory usage. We do not pursue this approach here.

7.6 Subsumption

Subsumption is one issue in which a lifted solver does seem to suffer in comparison to the ground solver.

Normally, when simplifying a ground problem we will run some form of subsumption checking. At the very least, if the theory contains a unit literal l and another clause contains that literal then the clause will be removed because it will always be satisfied (we discussed this in Section 3.9).

However, if we have lifted clauses then we cannot so easily remove any ground clauses that might have been subsumed. For example, in the SATPLAN encodings we will have unit literals specifying the initial and final states of the planning problem. Presumably, these states will be consistent and so the axioms specifying consistency of the state will become redundant. In the ground case they would be removed. In the lifted case it is only some portions of the clause that might be removed.

The lifted solver could potentially suffer from this. In extreme cases, a large fraction of the ground clauses might be subsumed, but we cannot do the associated removal of clauses from within the quantified clause: We have to retain the whole quantified clause.

The lifted implementation in Chapter VIII does have this potential drawback, but, as we shall see, it still does well. Part of the reason for this might be that the clauses that would be subsumed are associated with the end points of the planning problem. Once we have finished initialization, all search and subsearch proceeds incrementally in response to changes. But since everything is fixed at the endpoints the redundant clauses will never be referred to, and need not contribute

significantly to runtime costs.

Another version of subsumption is the removal of duplicate clauses. Suppose we had (2.11) in the form

$$\forall i_1, i_2, h. \neg (i_1 = i_2) \vee \neg p(i_1, h) \vee \neg p(i_2, h) \quad (7.11)$$

then in creating the $R^-(C, \neg p(I, H))$ used in incremental maintenance we would get two matches for the two separate occurrences of p :

$$\begin{aligned} \forall i_2. \neg (I = i_2) \vee \neg p(i_2, H) \\ \forall i_1. \neg (i_1 = I) \vee \neg p(i_1, H) \end{aligned}$$

However, the two sets are identical and it would be inefficient to use both. This is just a reflection of the fact that grounding (7.11) produces two copies of every clause. In a ground solver we would remove these with pre-processing.

In the lifted case we shall assume that the axioms themselves are written so as to remove the need for subsumption, or removal of duplicates. For example, we would write

$$\forall i_1, i_2, h. \neg (i_1 < i_2) \vee \neg p(i_1, h) \vee \neg p(i_2, h) \quad (7.12)$$

This can be regarded as just part of the problem of encoding a domain. In any case even if it is not done the repetition just causes some extra work by the solver, but does not prevent it from working.

7.7 Summary

Even a very simple form of lifted subsearch – simple depth-first search that just checks for chances to backtrack whenever a variable is bound – can show significant gains over a ground solver. The essential reason is that a ground solver is effectively doing generate-and-test, and this is relatively easy to beat.

Of course, there are also potential losses: there is extra overhead for handling predicates rather than boolean variables. We lose the ability to store information with each clause separately. Instead we will often have to regenerate information on the clauses as needed. We also lose some chances for simplification.

To settle which of these wins in practice there is but one choice – implement it! Hence, we will now discuss the results from such an implementation.

CHAPTER VIII

RESULTS FROM IMPLEMENTING A LIFTED SOLVER

We have seen that allowing universally quantified clauses (QCNF-formulas) is potentially very useful. We can use the same algorithms as we used for SAT, but will be able to use intelligent search for the NP-hard subsearch problems, rather than the generate-and-test used by ground solvers. There is also the obvious potential advantage that the QCNF formulas needed are much smaller than their ground equivalents, and so memory resources are less likely to limit the problems we can even begin to solve.

However, using QCNF rather than SAT is likely to have various overheads. Working with predicates, of varying arities, requires mundane but time-consuming tasks such as looking up arities and binding arguments. The simple task of finding what value is assigned to a ground atom such as $r(3,4)$ can require significantly more time than looking up the value of the equivalent boolean variable in the ground theory.

It could be that the theoretical advantages would just be swamped by the practical difficulties of working with predicates and quantified clauses rather than boolean variables and simple ground clauses. We have also seen in Section 7.5 that we might have to sacrifice some of the data-caching that is used in the ground solvers and instead replace it with subsearch.

To investigate these issues we implemented a lifted solver. In this chapter we

describe the solver, and present some experimental results for its performance on the logistics domain.

We find that lifting does indeed induce an overhead, but that the overhead is not so large as to render lifting impractical. On the contrary, we find that the scaling of the lifted solver can be better than that of the ground solver. The lifted solver also manages to solve problems that the ground solver cannot handle at all because the latter runs out of physical (and even virtual) memory. We limit the scope of the experimental study because:

1. We only want to do “proof-of-concept” of use of QCNF. Exploring all the consequences for SATPLAN, etc., would take us too far off-track.
2. We hope to show in the remaining chapters that the implementation of subsearch could still be improved and so extensive experimental investigation would be premature at this stage (and is left to future work).

8.1 Implementation of a Simple QCNF Solver

The lifted solver we use is a result of a fairly direct implementation, in C++, of the ideas in Chapters V and VII. The solver is capable of doing unit propagation to completion and then running WSAT.

We followed an object-oriented approach, based on the overall idea of separating and encapsulating the search and subsearch, as portrayed in Figure 15. The overall architecture is given in Figure 25. We have a search engine controlling the current state P and talking to constraints that manage clauses and their associated subsearch. The search and subsearch are in entirely different spaces and can use

entirely different methods.

The input language is QCNF with no functions, that is, even the fixed functions have to be changed to relations, as described in Section 2.2. Although the language used by the solver is QCNF, (that is, universally quantified clauses) we do allow a single existential quantifier in the input. This is just for convenience in dealing with axioms such as those of Figure 1. Internally, the parser just converts the existential to an appropriate disjunction. The search and subsearch engines themselves do not use the existential. For convenience, the values of some fixed predicates such as equality and “less than” are predefined within the solver, and do not need to be defined within the input. (However, the solver does not do any reasoning about such fixed predicate: That is, it does not do any reasoning of the

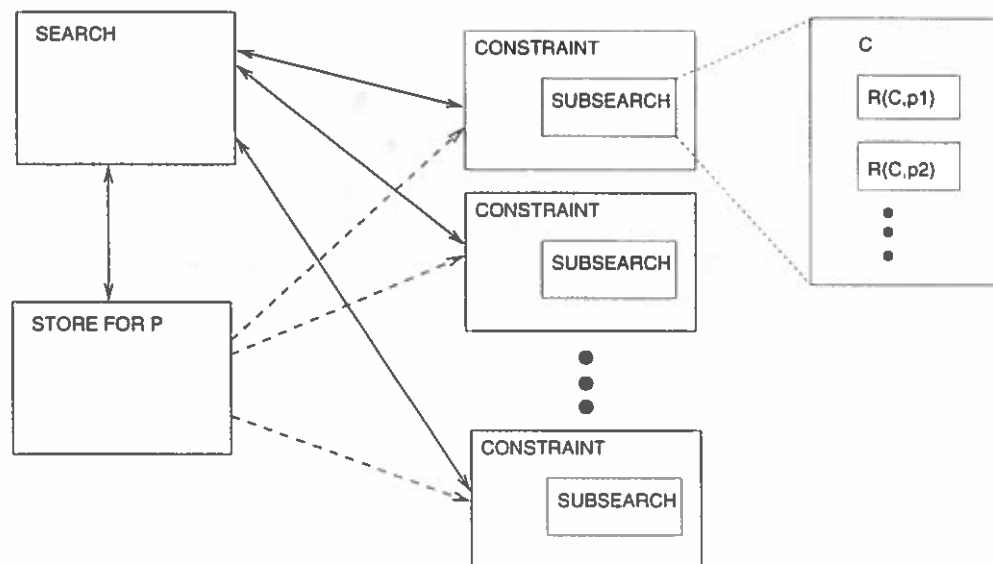


FIGURE 25. Software architecture of the implemented lifted solver. Each quantified clause becomes an object that does subsearch over the clause, or a precomputed syntactic restriction, in response to requests from the controlling search engine.

form used to convert (4.15) into (4.16).)

After parsing the input clauses, the solver instantiates an object for each quantified clause.

The class holding a quantified clause c stores the clause and also builds, and indexes, the quantified $R^-(c, l)$ and $R^-(c, \neg l)$ for every literal l appearing in c . These are used in order to handle incremental changes (see the discussion in Section 5.5). The equivalent step in a ground solver is the indexing of the set of clauses so that we can directly access all clauses containing a given literal. The only difference in the lifted case is that since the clauses themselves are so small we can make copies and use bindings of variables (Section 7.3) instead of having to index all the ground literals.

After the clauses and sub-clauses are built they are prepared for subsearch. This consists of organizing the literals into a structure allowing us to (sub)search over them using simple backtracking search. Basically we build the equivalent of a set of nested “for loops” (though using recursion instead of actual nesting). The literals are then hoisted out of the inner loops, and placed so that they will be tested as soon as they are ground.

The variable assignment P is stored in ground form with a byte for storing the value of each atom. Storage is accessed as a multi-dimensional array and hence is slower than the one-dimensional array that would be used in a ground solver.

8.1.1 Implementing WSAT

In WSAT, we will be storing (at least) the set of unsatisfied clauses, $S(C, P, 0, 0)$, explicitly. We will store these with the quantified clauses from which they arise,

and this is the reason that we use an object for each quantified clause. In fact, rather than store the ground clauses explicitly we just keep a set of lists of bindings (c.f. Figure 24).

The main search algorithm needs to know the size of $S(C, P, 0, 0)$ so that, in the random walk step, it can do a uniform selection over all the $S(C, P, 0, 0)$ from all the constraints. If a ground clause is selected for fixing then the literals from the clause need to be passed back to the main search routine so that their heuristics can be evaluated. A major fraction of the effort expended within the object goes into evaluating the breaks that result from flipping a literal.

8.1.2 Implementing Unit Propagation

Unit propagation is done by a fairly direct implementation of Figure 23(b). The method is stateless: it does not need to cache any information between calls.

8.1.3 Subsearch Variable Ordering

We mentioned that the subsearch is done by building data structures that enable us to simulate a set of nested for loops with appropriate pruning conditions, and actions on reaching the innermost statements (see, for example, Figure 24).

This means that we are doing backtracking search with a static variable order. The order used can sometimes make a difference to the subsearch. For example, in the WSAT initialization in Figure 24, the different procedures correspond to different orderings of the variables. However, picking a good order can be done for the domain as a whole rather than needing to be done for every problem instance separately. Hence, currently the solver just uses the order of variables as given in

the input clause. The responsibility to pick a good order is given to the system or person that produces the domain axioms. Of course, in the ground case we do not have even this option. Eventually the subsearch should be able to select its own ordering, perhaps using existing methods to select a static ordering [69]. This could be done by some probabilistic analysis of the clauses and expected assignments for the literals. However, the goal here is just to discover the overheads associated with lifting rather than do the best subsearch we can imagine. The implementation here is just a start on a potentially long (even open-ended) process of optimizing subsearch. We did enough to have preliminary but real results for the trade-off between the various costs and gains of subsearch.

8.2 The Test Domain: Logistics

The test domain we use is, of course, the logistics domain with the encoding given in Section 2.5. However, recall that existentials are not allowed and so, for example

$$\exists c. \text{planeAt}(p, c, i) \tag{8.1}$$

is used in the form

$$\text{planeAt}(p, C_1, i) \vee \text{planeAt}(p, C_2, i) \vee \dots \tag{8.2}$$

where the C_i are constants for the cities. Also, all functions or partial functions need to be converted to relations and so, for example,

$$(\text{at}(o, c, i) \wedge \text{in}(o, a, i + 1)) \longrightarrow \text{planeAt}(a, c, i)$$

would be entered as

$$\text{incTime}(i, j) \longrightarrow [(\text{at}(o, c, i) \wedge \text{in}(o, a, j)) \longrightarrow \text{planeAt}(a, c, i)] \quad (8.3)$$

where

$$\text{incTime}(i, j) \longleftrightarrow j = i + 1 \quad (8.4)$$

8.2.1 The Test Instances

One of our main underlying concerns has been that as the problems increase in size the ground solvers will no longer be able to cope. Hence, we are concerned with the scaling properties of the solvers as well as their performance on some fixed problem instance. Accordingly, we present results for a sequence of problem instances characterized by a size N . A problem instance, a planning problem, of size N from the logistics domain is created with

1. N planes.
2. $2N + 1$ objects.
3. $2N + 2$ cities: A distinguished “hub” and $2N + 1$ “satellites”.
4. 8 timepoints.
5. Initial state: Exactly one object per satellite city, and all the planes at the hub city.

6. Final state: one object per satellite, but a permutation of the initial assignment. (The permutation moves every object, but is otherwise random.) The planes do not need to return to the hub.

We need to move $2N + 1$ objects with N planes but the objects start and finish in different places, hence it follows that at least one plane must make three LOAD-FLY-UNLOAD sequences. With 9 timepoints there is always a plan (a satisfying assignment), but with 8 timepoints the problems are unsatisfiable.

8.3 Experimental Methods

The test instances are specified by the sizes of the domains, the axioms are given as quantified clauses, and the initial and final states as a set of forced literals. This information can be given directly to the lifted solver. The lifted solver runs unit propagation to completion itself before starting WSAT.

For the experiments with ground WSAT we must of course first convert the problem instance to a ground SAT problem. To do this, the instances are produced in a QCNF subset of the language used by CSPEC [34]. Running CSPEC produces an initial ground CNF, which is then fed through an implementation of COMPACT¹, in order to unit propagate to completion and remove all boolean variables that have a fixed value.

In this case, after unit propagation, and with 8 timepoints the resulting theory has

$$\# \text{ variables} = 19 + 65N + 40N^2 \approx 40N^2$$

¹Due to James Crawford. Available from <http://www.cirl.uoregon.edu/> in the NTAB package.

TABLE 1. Examples of sizes of the logistics test instances as a function of the number of objects N . We give the number of variables, the numbers of clauses, and the total numbers of literals in the sizes of the *ground* theories, *after* unit propagation and simplification. $N = 300$ is included as it is at the limit of what the current lifted implementation can handle with 256MB of memory.

N	variables	clauses	literals
10	4,669	112,357	296170
20	17,319	809,972	2128450
30	37,969	2,638,887	6924930
70	200,569	32,187,547	84,312,850
300	3,619,519	2,474,734,992	6,474,809,490

$$\# \text{ clauses} = 42 + 166.5N + 196.5N^2 + 91N^3 \approx 91N^3$$

$$\# \text{ literals} = 90 + 398N + 541N^2 + 238N^3 \approx 238N^3$$

In Table 1, we show some examples of sizes of the ground theory. In all cases the number of variables scales as $O(N^2)$ because we have arity three predicates, and arguments have domain size $O(N)$ except for the timepoint arguments which have fixed domain size. The number of clauses, and literals, scales as $O(N^3)$.

The lifted and ground cases are equivalent in that after unit propagation we are left with precisely the same set of UNVALUED variables. In fact, a large amount of time can be consumed in preparing the ground theories. A particular problem is running COMPACT: this tends to exhaust the RAM and then takes a long time to finish because of swapping to disk. The time taken to prepare the ground instances is not included in times for the ground solver (though it can be substantial), hence we are favoring the ground solver in this respect.

The problem of running out of RAM for ground WSAT is so severe that it limited us to $N \leq 40$ for the ground experiments.

The quantified clauses are small and so the memory usage of the lifted solver is driven primarily by the number of variables and various internal indexing schemes. $N = 300$ is about the maximum that we can handle in the current implementation with 256MB of RAM. (Of course, the mere fact that we can run WSAT does not mean that we will have a significant chance of finding a solution.) For the lifted solver we usually just give performance data up to $N = 70$.

WSAT is usually associated with a high variation in number of flips needed to find a solution, but the fliprates themselves are much more consistent. The instances themselves have a very uniform structure, also the variation in times between different runs on the same instance are small. Hence, it was quite adequate to just take one instance and one run at each data point. For example, fliprates are measured using a run of 10^6 flips, and then the variation of fliprate between runs is less than 1%. Also, when lifted and ground solvers are compared the same instances are used for each.

Satisfiable instances are solved quickly and so too much of the fraction of the runtime will be spent in the initial phase where the number of unsatisfiable clauses is relatively large. Hence, in order to get a better estimate of the speed that would be obtained in longer runs we used unsatisfiable instances. Remember that we only want to compare the speeds of the lifted and ground solvers, and do not care whether or not they actually succeed. Similarly, parameter settings for the noise are irrelevant (though we used $p=0.2$).

The primary experiments were run on a 400MHz Pentium II with 256MB of RAM, 512kB of cache, and running Linux. Code was compiled with Gnu gcc or g++ using full optimization.

8.4 Two Implementations of WSAT: WSAT(U) and WSAT(UB)

We have already discussed the ground WSAT algorithm at length in Section 3.3. We will be using two different ways to implement this algorithm. We label them WSAT(U) and WSAT(UB).

WSAT(U) based on an existing ground implementation due to Andrew Baker. It stores only the Unsatisfied clauses. The algorithm for creating and maintaining this set of clauses was described in previous chapters.

The ground WSAT(UB) is the 1996 (version 21) ground AT&T “walksat” as implemented by Cohen, Kautz and Selman². It is distinguished by storing not only the Unsatisfied clauses, but also information about the number of Breaks associated with each atom. We will describe this in more detail in Section 8.7. For ground WSAT(U) we took the original walksat (WSAT(UB)) and just removed the storage and maintenance of the extra information about the breaks. This was done for uniformity of coding, and also because the AT&T version is more efficient with respect to space usage; for example, it uses arrays rather than the linked lists of Baker’s original (un-optimized) implementation.

The lifted solvers are direct minimal lifted versions of each of these ground versions. Since, we did a minimal lifting, both the ground and lifted solvers will search in exactly the same fashion. In fact, we explicitly checked that their performances – in terms of flips needed to solve a satisfiable instance – were identical.

Firstly we look at results for the simpler version: WSAT(U).

²Available from <http://www.research.att.com/~kautz/blackbox/>

8.5 WSAT(U) Initialization Results

On initialization, the ground solver has to check every clause to see whether or not it is satisfied. However the number of clauses is $O(N^3)$, hence we expect initialization times for ground WSAT to scale as N^3 .

What about the lifted WSAT(U) solver?

In Section 7.4.3, we discussed some of the axioms expressing consistency of the state. In the language in that section, the set of instances we use N_1 , N_3 , and N_4 are all $O(N)$, whereas N_2 is a constant. Axioms such as (2.22) will produce $O(N^3)$. This means that the ground solver (which basically does Figure 24(b)) will also be $O(N^3)$. However, the lifted version (see Figure 24(c)) has complexity " $O(N^2) + wO(N^3)$." In our experiments we took $w = 0.01$ and this is small enough that the cubic term has little effect in the range of N values that we use. Hence, the effective scaling is reduced to $O(N^2)$. Other axioms such as those of Figure 3 undergo a similar reduction of complexity due to the backtracking subsearch in the lifted solver.

The experimental results on times taken by the ground and lifted solvers to enumerate and store all the unsatisfied clauses are given in lines (a) and (b) of Figure 26. The expected scaling behaviors are indeed observed at larger values of N . The behavior at small values of N is different, but this should not be too surprising as there might be many sub-leading terms.

In order to get an idea of just the overhead, and not the gains, associated with the lifting we also disabled the subsearch pruning in the lifted version. In this "non-pruning" version, the subsearch is arranged to mimic the ground solver. The results for this non-pruning version are shown in Figure 26(c). As expected, the

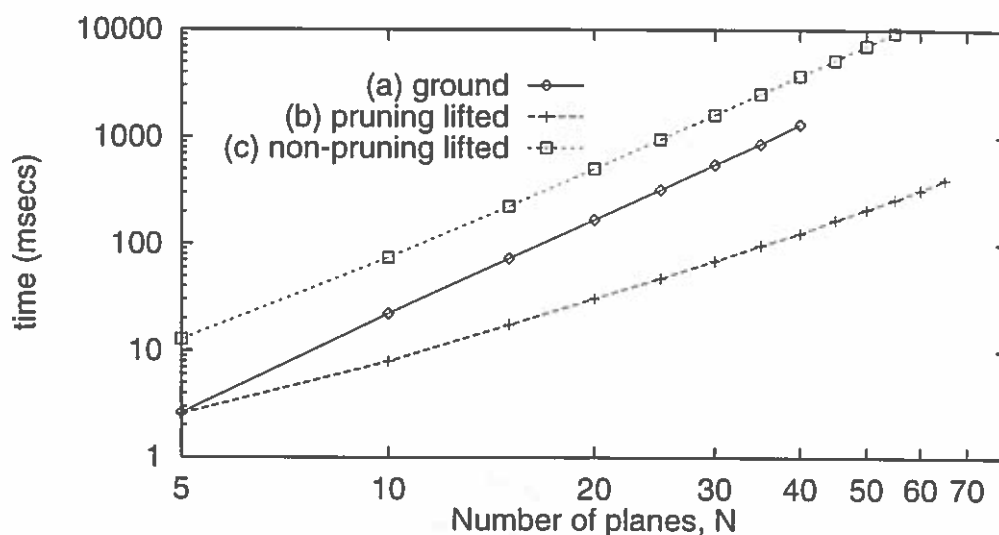


FIGURE 26. Experimental initialization times for $WSAT(U)$, on the logistics instances with N planes. (a) Ground $WSAT(U)$. (b) Lifted $WSAT(U)$, using simple backtracking for the subsearch. (c) Lifted $WSAT(U)$ but with pruning in the subsearch disabled so that it mimics the ground solver. Note that both axes have logarithmic scales: Lines (a) and (c) scale roughly as N^3 , whereas (b) scales as N^2 .

scaling behavior has returned to $O(N^3)$. At $N = 40$ (the largest size accessible to the ground solver) this non-pruning lifted version is running about 3 times slower.

Hence, in this case, it seems that the overhead of lifting roughly corresponds to a factor of about 3. This could easily be explained by extra costs such as looking up the value of an atom such as `planeAt(2, 3, 5)` which requires access to a multi-dimensional array (or equivalent) rather than just looking up a boolean variable in a one-dimensional array. However, we emphasize that when subsearch pruning is enabled we more than recover this loss: from Figure 26 we see that the lifted version with backtracking subsearch is a clear winner.

In practice, the time spent on initialization will usually be a small fraction of the total runtime, and so we now move on to look at how the lifted solver performs

when making repairs (flips).

8.6 WSAT(U) Flip Rates

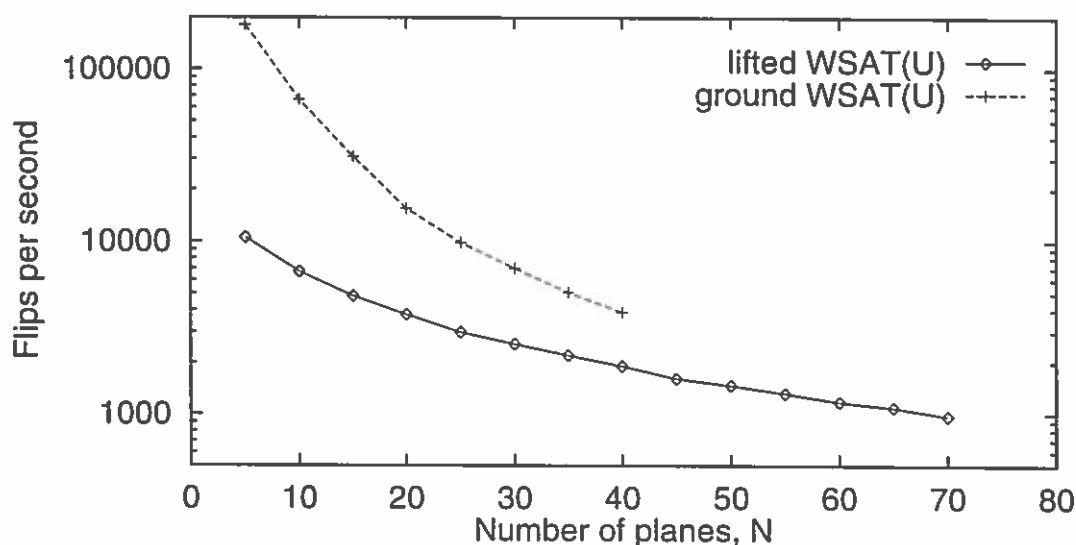


FIGURE 27. Experimental performance of ground and lifted WSAT(U) solvers on the Pentium II 400MHz machine. Performance is measured in terms of flips per second. The ground line terminates at $N = 40$ because beyond that point the available memory (256MB) is exhausted.

Figure 27 gives the experimentally observed fliprates for the ground and lifted versions of WSAT(U). To understand these results we need to consider the scaling of the time needed per flip. In WSAT, after having selected the clause to repair, and before making the flip itself, we have to select a literal from within the clause. This requires calculating a heuristic for each such candidate literal.

In WSAT(U) we have to calculate the number of breaks that would occur on flipping that literal and on inspecting the relevant sets of lifted clauses we find

that the cost of evaluating the heuristic for a candidate be expected to be $O(N)$ for both the ground and lifted solvers. (The nature of the relevant restricted clauses is such that the lifted solver has no opportunity for pruning with the particular axioms used.)

However, the logistics test instances contain clauses whose length is proportional to N , e. g. (2.21) says that every plane must be somewhere and so is a disjunction over the $2N + 2$ cities. Since we have $O(N)$ candidates and $O(N)$ work to evaluate each candidate, we can expect $O(N^2)$ time per flip.

Figure 28, gives scaling of time per flip for ground and lifted versions of WSAT(U). We see that a quadratic function of N is indeed a reasonable fit. The scaling is not quite quadratic due to small effects from the physical cache; we will return to this point in Section 8.9.

Finally, Figure 29 gives the ratio of fliprates of the ground and lifted solvers as a function of size. At small sizes the lifted solver does badly, but gets better as the sizes increase. The curve suggests that even if we could run the ground solver on very large problems it would only be about twice as fast.

8.7 Pre-Calculating the Break Counts: WSAT(UB)

As mentioned earlier, the original AT&T implementation of WSAT stores not only the set of currently unsatisfied clauses, but also information about the numbers of breaks associated with an atom. Specifically, it keeps a “Breakstore”, that is, for every atom in the theory it stores the number of clauses that would change from satisfied to unsatisfied if the current value of that atom were to be reversed. Consider an atom a in the theory and suppose that the current assignment P is

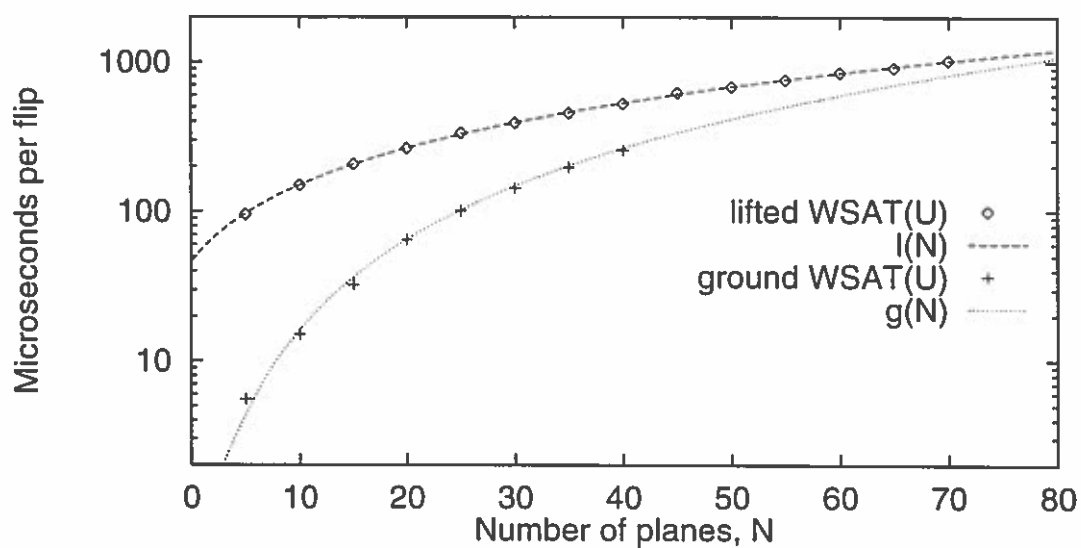


FIGURE 28. Experimental times per flip for the ground and lifted WSAT(U) solvers. The quadratic lines are $l(N) = 0.058N^2 + 9.85N + 45.9$ and $g(N) = 0.17N^2 - 0.21N + 1.17$, and are best fits to the data.

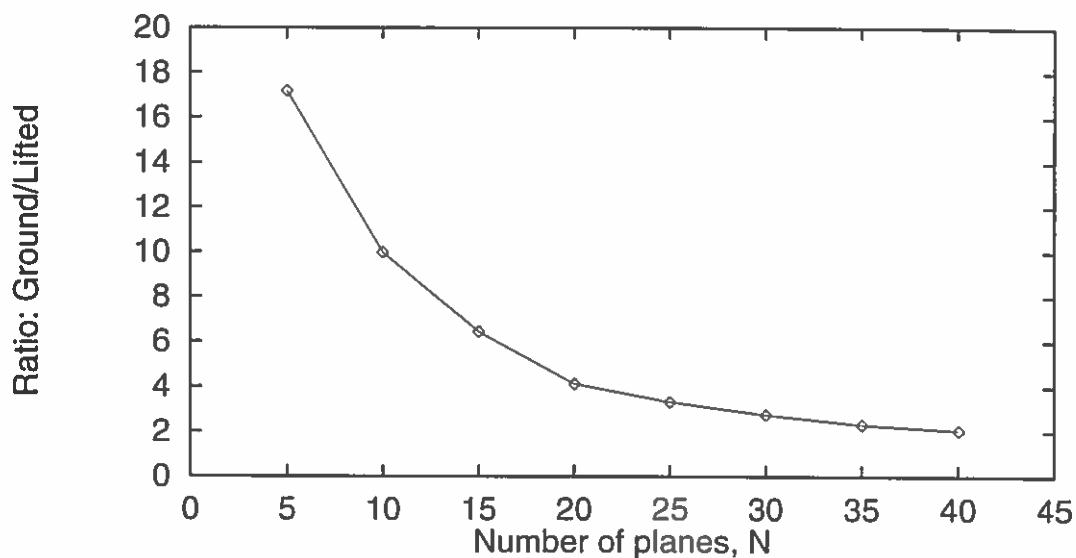


FIGURE 29. Ratio of fliprates for the ground to lifted WSAT(U) solvers.

such that $l = \text{TRUE}$ where $l = a$ or $l = \neg a$. Then, by definition, the entry in the breakstore associated with a must always be the number of clauses in which l is the *only* satisfying literal. We shall denote the relevant entry in the break store as $\text{BREAKSTORE}[l]$. In the ground case it really is stored as an array; in the lifted case it requires a multi-dimensional array to handle the arguments of the literal (we use the same storage system as for the values of the literals).

Clearly, such a breakstore can lead to savings because now instead of having to evaluate the heuristic for each candidate flip, we can just look up the value in the breakstore. However, instead of just having to initialize and maintain the set of unsatisfied clauses we now also have to initialize and maintain the breakstore. The way that we do this is a straightforward conversion to subsearch language of the existing ground WSAT(UB) code.

We first deal with the somewhat simpler case of initialization.

8.7.1 Initializing the Breakstore

As just mentioned the entry in the breakstore associated with an atom a must always be the number of clauses in which the associated, currently true literal, l , is the *only* satisfying literal. That is, we need to find for every currently true literal l the size of $S(R(C, l), P, 0, 1)$, that is, to evaluate $|S(R^-(C, l), 0, 0)|$. However, rather than doing this for each literal in turn it is better to instead enumerate the set of all clauses with zero or one satisfying literals: $S(C, P, 0, [0, 1])$, using the notation of Section 6.6.1. On finding such a clause we have two possible actions depending on s the number of satisfying literals:

- $s=0$ The clause is unsatisfied and so should be added to the store of unsatisfied clauses (exactly as in $\text{WSAT}(U)$ and as discussed extensively in previous chapters).
- $s=1$ The clause has a single satisfying literal, call it l , and increment the relevant entry in the breakstore: that is, do $\text{BREAKSTORE}[l]++$.

The subsearch for the elements of $S(C, P, 0, [0, 1])$ can be lifted in the obvious fashion: we search over all clauses pruning whenever we find a clause with two satisfying literals. In fact, in the lifted case we can have literals that are fixed true but are not removed from the clauses. Such fixed literals can never be flipped by WSAT and so we can also prune the subsearch on finding just one such literal. We do not need to store or maintain values in the breakstore for such fixed literals.

Note that the required subsearch is very similar to that for propagation, and in fact the implementation re-used a lot of the propagation code. Such a commonality in the implementations of otherwise very different algorithms is not only a pleasing feature of the subsearch viewpoint, but also practically useful.

8.7.2 Maintaining the Breakstore

After we have flipped an atom then the number of true literals in clauses containing that atom will change and so we have to make appropriate changes to the breakstore. Suppose that we are setting $l = \text{TRUE}$, then there are two cases.

Case 1: Clauses containing l

In these clauses the number of satisfying literals will increase by one. We proceed by enumerating $S(R^-(C, l), P, 0, [0, 1])$. The relevant action to take depends on the number, s , of satisfying literals besides l itself.

$s=0$ This corresponds to clauses that became satisfied and so need to be removed from the store of unsatisfied clauses (exactly as in $\text{WSAT}(U)$). However, in this case l also becomes the only satisfying literal and so we also need to do $\text{BREAKSTORE}[l]++$ for each such clause.

$s=1$ These clauses used to have a single satisfying literal, call it l' . However, now that l is also true we have two satisfying literals and so the clause should no longer contribute to the breakstore for l' . That is, we need to do $\text{BREAKSTORE}[l']--$.

Case 2: Clauses containing $\neg l$

In these clauses the number of satisfying literals will decrease by one. We pro-

ceed by enumerating $S(R^-(C, -l), P, 0, [0, 1])$. Again, the relevant action depends on the number, s , of satisfying literals:

$s=0$ These clauses will become unsatisfied – they are exactly the clauses that would be discovered in the subsearch for breaks in $WSAT(U)$. They need to be added to the store of unsatisfied clauses. Also, l used to be their only satisfying literal and so we need to do $BREAKSTORE[l]--$ for each such clause.

$s=1$ Suppose the satisfying literal is l' , then both l and l' used to be satisfying literals but now l' will be alone. Accordingly we need to increment its count in the breakstore: $BREAKSTORE[l']++$

In both of these maintenance cases the lifted subsearch can be implemented in a similar fashion to the case of initialization: we prune the subsearch on reaching two satisfying literals, or a single satisfying fixed literal.

The standard “walksat” ground code uses indexing schemes to find the relevant sets $R^-(C, l)$ and $R^-(C, -l)$, and a stored value of the number of true literals in each clause to restrict attention to the relevant values of s .

Note that for us to be able to prune the subsearch in $WSAT(UB)$ we generally need to find *two* satisfying literals. In contrast, in $WSAT(U)$ we can prune on reaching just *one* satisfying literal. This means that the $WSAT(U)$ method has at least a potential of getting better gains from intelligent subsearch.

8.8 Experimental Fliprates for WSAT(UB)

We now return to the particular case of the logistics axioms, and look first at the times needed per flip (or flips per second).

In the logistics axioms there are clauses of length $O(N)$ and if one of these were selected for fixing then we will have $O(N)$ candidates. In the case of WSAT(U) we could expend $O(N)$ time on each of these candidates for a total of $O(N^2)$. In the case of WSAT(UB) the cost of evaluating the heuristic is now just $O(1)$ because we can just look up the value in the break store (this is the reason that the breakstore was used in the ground version from AT&T). This gives a total of $O(N)$.

However, we also need to consider the cost of updating data structures after making a flip. Potentially the weaker pruning in the subsearch needed to maintain the breakstore in WSAT(UB) could make it worse than WSAT(U). However, as we have already discussed, the logistics axioms are such that when combined with incremental maintenance there are no opportunities for pruning. In fact the worst case that we have is $O(N)$. Hence, we expect that the time per flip will be linear in N for WSAT(UB) in contrast to the quadratic time expected for WSAT(U). Accordingly, WSAT(UB) can be expected to have a much higher fliprate on the larger instances.

Note that if the search does not encounter many long clauses then WSAT(U) is indeed a reasonable implementation of WSAT: The gains from having the breakstore might be more than offset by the extra cost of having to maintain it. In fact, on random 3SAT instances, we found that the fliprate of ground WSAT(U) could be over twice that of the original ground WSAT(UB).

Figure 30 gives the experimentally observed fliprates for the ground and

lifted versions of WSAT(UB) and also repeats the results obtained for WSAT(U) so as to allow an easier comparison. As expected the WSAT(UB) versions beats the WSAT(U) versions (except that the ground version does well on the smallest instances). Figure 31 shows the relative performance of lifted and ground WSAT(UB) solvers. Again, lifting gives an overhead of a factor of about two on the largest theories.

We emphasize that a strong advantage of the lifted solver is that loading and running lifted propagation to completion takes seconds. In contrast, writing out the large ground theories and running them through COMPACT can take many minutes even in the cases that the theory is small enough for this to be possible. The lifted solver can handle, and solve, theories with many millions of clauses whereas the ground solver cannot even load such instances. In fact, we were able to load and solve an instance at $N = 300$ with 10 timepoints, corresponding to about 5 million variable and 2.5 billion clauses. However, at this size performance had dropped to about 240 flips/sec, and so solving took almost 10 hours despite the planning problem being very simple.

8.9 Effects of Machine Cache on Scalings

In the previous section we argued that the time-per-flip for WSAT(UB) should be linear in N . In Figure 32 we give the experimentally observed times for the lifted and ground versions of WSAT(UB) . Although close to a linear function there is clearly a discrepancy. Suspecting that this was an effect of the cache we repeated the experiments on a different machine: An Intel Celeron running at 374MHz and with 128k cache. This second machine had the helpful facility

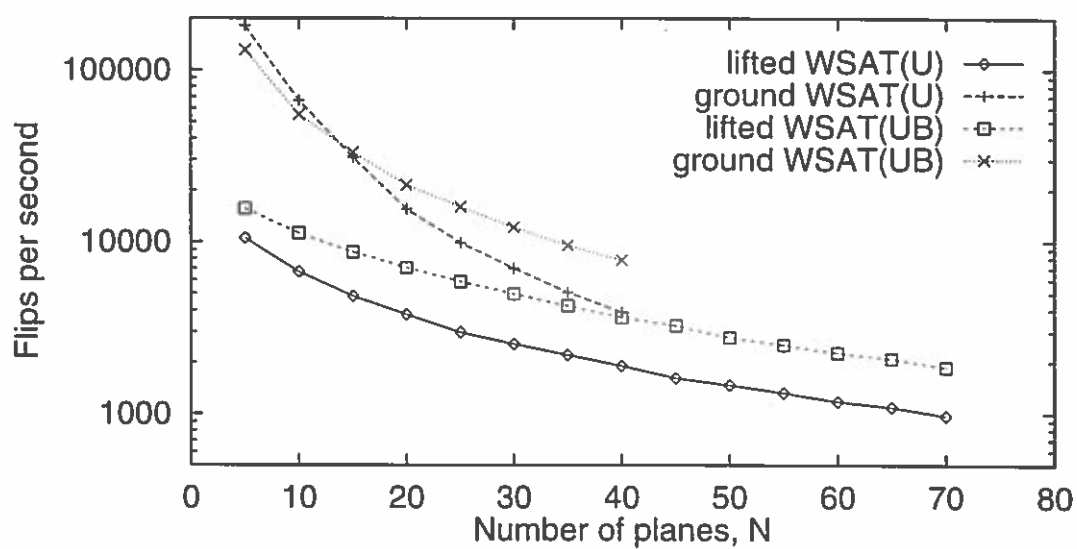


FIGURE 30. Experimental performance of ground and lifted WSAT(UB) solvers. As usual, the ground lines terminate at $N = 40$ because beyond that point the available memory (256MB) is exhausted.

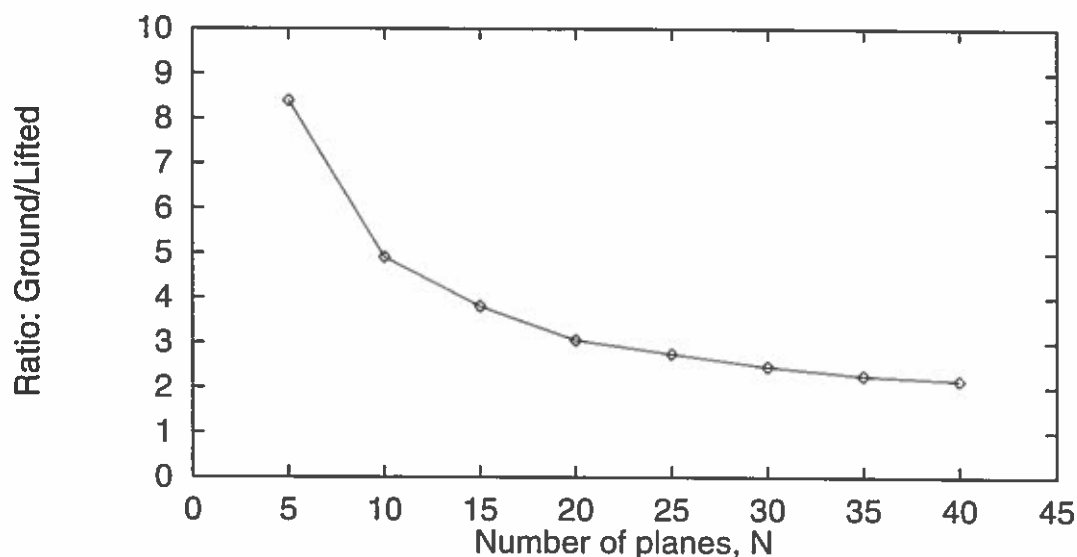


FIGURE 31. Ratio of fliprates for the ground to lifted WSAT(UB) solvers.

(supplied by the BIOS) to disable the cache. We use the same executables, but the Celeron machine had only 64MB of RAM and so experiments could only be performed up to $N = 25$.

The times per flip, running without cache, are given in Figure 33. Naturally the lack of cache slows down the solvers, however we also obtain a much closer fit to the expected linear time scaling for each solver. The observed variation of the ratio of fliprate for ground to lifted is a consequence of the ratio of such linear scalings. The constant terms in the linear fits are much worse for the lifted solver than for the ground solver. Hence, at small N , the ratio is high, however, as N increases it drops towards the value given by the leading, $O(N)$, terms. In this case the asymptotic value for the ratio is $10.2/4.82 = 2.12$, consistent with the behavior found in Figure 31.

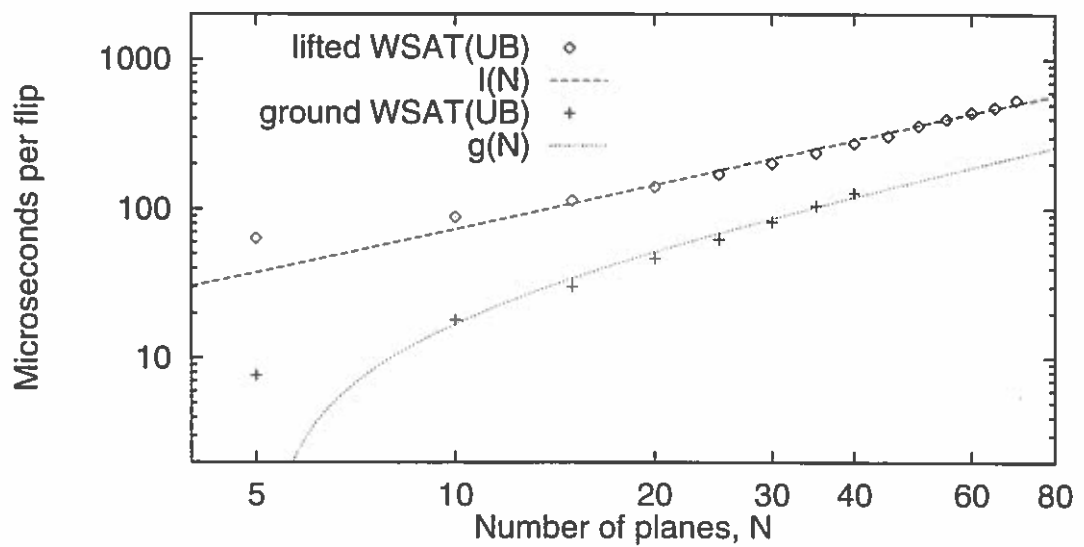


FIGURE 32. Experimental times per flip for the ground and lifted WSAT(UB) solvers. The lines $l(N) = 7.21N + 1.59$, and $g(N) = 3.46N - 17.6$ and are best linear fits to the data.

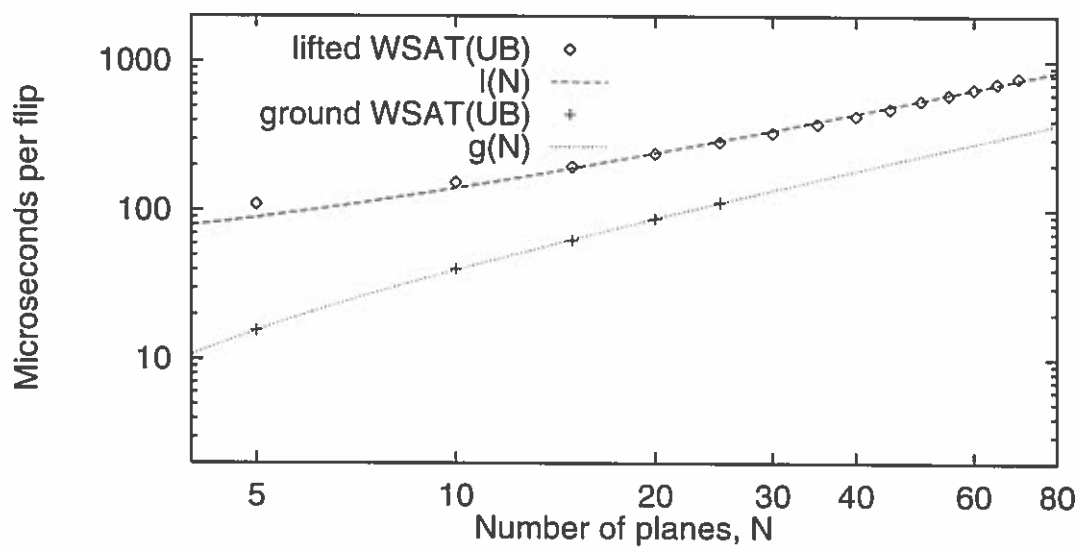


FIGURE 33. .

Experimental times per flip, for WSAT(UB), with physical cache disabled. The lines $l(N) = 10.2N + 38.2$, and $g(N) = 4.82N - 8.58$, and are best linear fits to the data.

Finally, in Figure 34, we give the ratio of ground to lifted fliprates, both with and without cache (on the Celeron machine). We see that the ratio is slightly more favorable, at larger sizes, to the lifted solver when cache is enabled. This seems reasonable as the lifted solver is using less memory and so can be expected to have a better cache miss rate once the ground solver has grown enough to use a substantial fraction of the RAM.

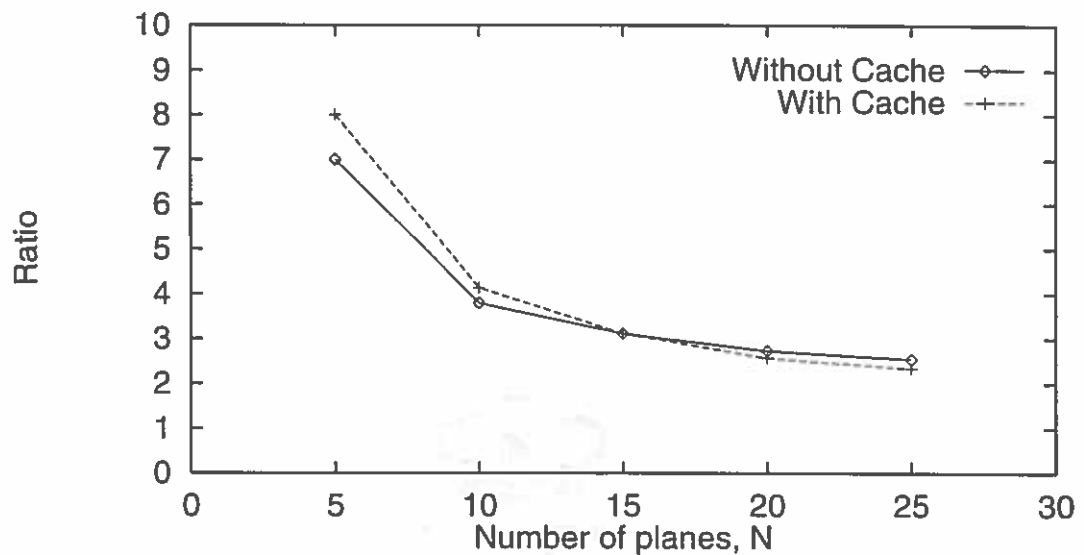


FIGURE 34. Ratio of fliprates for the ground to lifted WSAT(UB) solvers with and without cache.

8.10 Experimental Initialization Times for WSAT(UB)

So far we only gave experimental results for WSAT(UB) in cases in which it had no opportunity to prune. Unfortunately, the logistics domain axioms do not allow the lifted WSAT(UB) to obtain any pruning even during initialization. This is in contrast to the case of WSAT(U) (see Figure 26) and is just a reflection of

the somewhat weaker pruning rules in WSAT(UB).

To see a case in which pruning occurs even for WSAT(UB) we consider a new axiom

$$\begin{aligned}
 i + 1 < k \wedge c \neq d &\longrightarrow \\
 \text{at}(o, c, i) \wedge \text{planeAt}(a, c, i) \wedge \\
 \text{at}(o, c, k) \wedge \text{planeAt}(a, d, k) & \\
 \longrightarrow \text{in}(o, a, i + 1) & \qquad (8.5)
 \end{aligned}$$

which just says that if there is a plane going the same way as the object then we must immediately load the object onto that plane. It will also prevent two planes pointlessly duplicating routes. The axiom expands to $O(N^4)$ clauses and so the ground solvers can be expected to also have initialization costs that are $O(N^4)$.

If the subsearch values the variables i , k , a , c and o (but not d) then the literals $\text{at}(o, c, i)$, $\text{at}(o, c, k)$ and $\text{in}(o, a, i + 1)$ will be valued. Since the first two of these are very likely to be set FALSE by the initial weighting it follows that there is a reasonable chance that the subsearch will be able to prune at this point and not have to also value d . We can expect lifted WSAT(UB) to initialize in time roughly $O(N^3)$. In WSAT(U) we can do even better. After valuing just i , k , a and c then $\text{planeAt}(a, c, i)$ can be evaluated, and has a good chance of being found to be FALSE causing pruning and an expected cost that scales as $O(N^2)$.

Experiments are difficult because the ground theories rapidly become very large. accordingly we reduced the problem sizes by producing theories containing only this axiom. Since we are only initializing, and not doing flips, this does not

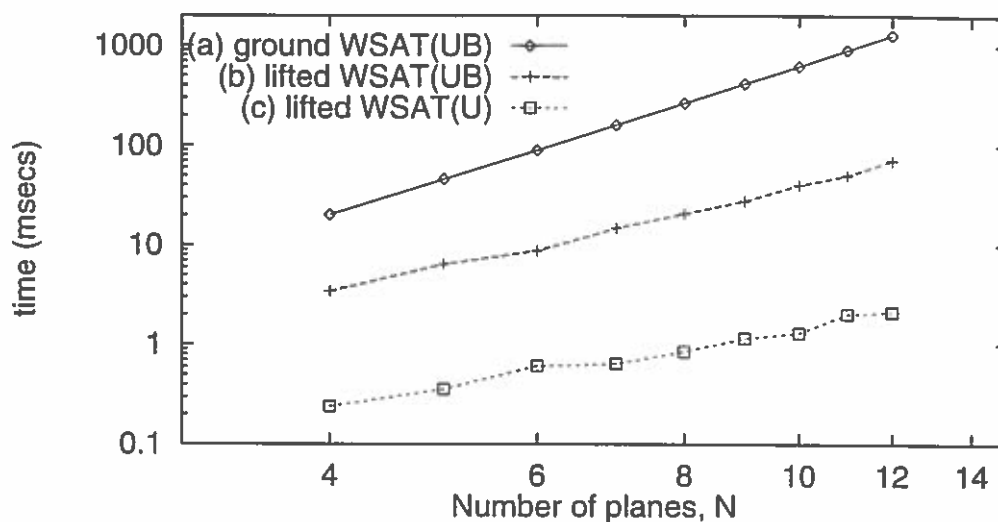


FIGURE 35. Experimental initialization times for WSAT on a large axiom (8.5). (a) Ground WSAT(UB). (b) Lifted WSAT(UB). (c) Lifted WSAT(U) Note that both axes have logarithmic scales: Line (a) scales as N^4 , (b) scales as N^3 , and (c) as N^2 .

affect WSAT. Even with 256MB the ground solvers can still only reach $N = 12$.³

Figure 35 gives the experimental initialization times and confirms that they behave much as expected. In particular doing lifting either of the solvers gives substantial savings.

8.11 Summary

We have seen evidence that the overhead from lifting is a constant factor of about 2 or 3. However, the ability of the lifted solver to prune in the subsearch can more than compensate. In one sense the simple ability of the lifted solver to

³Of course, the desire to handle such large axioms, in systems such as SATPLAN, provided much of the initial motivation for this thesis.

handle much larger theories is confirmation in itself of the utility of lifting. Even if the lifted solver had no gains from subsearch it would still be useful due to its ability to handle much larger theories. Often theories become significantly smaller on processing using COMPACT or COMPACT-L. So even a lifted solver that just did such pre-processing at the lifted level, and then ground out the resulting theory would still have utility.

CHAPTER IX

ADVANCED SUBSEARCH IN QCNF

So far we have only considered using simple backtracking for the subsearch, and found that even this simple method can show gains over the ground case. However, subsearch is just search in a CSP (or CCSP) and there are certainly far more effective search methods in CSPs. The topic of search in CSPs has been very extensively studied [1, 70, and many others]. A very partial list of approaches that one might consider includes:

1. Various levels of pre-processing such as node and arc consistency.
2. Dynamic variable ordering: when searching we generally allow the order of the branch variables to vary according to the current state of the subsearch.
3. Forwarding checking (FC): after having taken a branch, we check whether there is some reason that the subsearch can be terminated. That is, we use some form of propagation within the subsearch.
4. Many kinds of intelligent backtracking with backjumping or learning. For example, we might use Dynamic Backtracking [25] or one of its extensions such as Relevance Bounded Learning [2].
5. Exploitation of the graph structure: see Section 9.5.

Clearly, a lot of work would be needed to determine which of these work best for subsearch. We would also need to extend the methods to handle CCSPs as

far as possible. This is left to future work. Instead, here we shall focus on just a few ways that we might substantially reduce the cost of subsearch in the sub-CSPs. Firstly, we will distinguish between what we call semantic and syntactic approaches to solving the CSPs.

9.1 Semantic and Syntactic Methods

Here, the terminology follows our standard usage

1. Syntactic Methods: these do not refer to or need the current truth value assignment P .
2. Semantic Methods: properties of the current assignment P are exploited.

As defined in Section 6.2 and in particular (6.4), the constraint graph from the clause is syntactic, but the constraints associated with the edges are semantic.

Thus, we can distinguish between semantic methods directed at exploiting particular features of the constraints, and syntactic methods that exploit the structure of the constraint graph. In reality there is likely to be a lot of overlap. We will only briefly look at the semantic methods: this chapter will mostly concentrate on the syntactic case.

9.1.1 Semantic Methods

In SAT, if we restrict the clauses to be binary or Horn (at most one positive literal) then the theory is tractable (for a listing of such cases see [61]). Similarly in CSPs, restrictions can be placed on the constraint sets so that the theory becomes tractable. A well-studied case is that of “0/1/all constraints” in binary CSPs [9].

However, the constraints are controlled by P and hence under the control of the search, not the subsearch. Hence, it seems unlikely that only such tractable cases will occur. For example, WSAT might potentially hit any state, hence we cannot generally expect that there will be any limit on the nature of the constraints that arise.

Although the nature of the constraints seems unlikely to improve the *worst-case* behavior, it is quite possible that average-case behavior will be a lot better due to imbalanced predicates. As discussed in Section 7.4.1, it seems likely that literals within the clauses will tend towards being negative, and atoms will tend towards being FALSE. In terms of the constraints of the sub-CSP, this means that relatively few values for the domain variables will be allowed: the constraints will tend to be tight.

Whether or not the constraints will be tight enough for their expected complexity to be a lot better than the worst-case is a matter for implementation and experiment and will be left to future work.

Note that incrementality in this context will correspond to maintaining information about the sub-CSP as the assignment P changes, and hence as the constraints themselves change. Doing this efficiently is the topic of dynamic CSPs [18, 71]. It is an open question whether such dynamic CSP methods will prove useful for the subsearch.

9.1.2 Syntactic Methods

Instead of looking at the constraints themselves it is perhaps easier to just try to exploit the nature of the constraint graph to reduce the cost of subsearch. The

graph is independent of the assignment P . Hence, the remainder of this chapter will give a few examples and some discussion of how the graph structure of a constraint might be exploited.

It is worth bearing in mind that, although a particular clause c might not have a structure we can exploit, it could well be that some syntactic restriction $R^-(C, l)$ can be exploited. The effect of such a restriction on the graph is to remove some nodes and collapse some edges.

9.2 Factored Clauses

Consider the clause

$$c = \forall i, j. p(i) \vee r(j) \tag{9.1}$$

with $i, j \in D$ and $|D| = n$. This factors into

$$c = c_1 \vee c_2$$

with

$$c_1 = \forall i. p(i)$$

$$c_2 = \forall j. r(j)$$

The graph of the sub-CSP is given in Figure 36. Although this seems an unlikely constraint in itself, it might well arise from some syntactic restriction.

Suppose that we want to solve the checking problem for $S(c, P, 0, 0)$. The

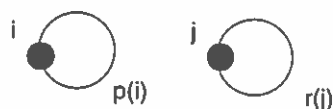


FIGURE 36. Example of a disconnected sub-CSP graph. The structure of the sub-CSP for the “factorizable” constraint of (9.1)

ground solver takes $O(n^2)$ as there are this many ground clauses. With simple backtracking we might still, in the worst case, take $O(n^2)$. However, we can do better by exploiting the factored form $c_1 \vee c_2$.

Define a cross-product \otimes on sets of clauses by

$$S_1 \otimes S_2 = \{(d_1 \vee d_2) \mid d_1 \in S_1 \text{ and } d_2 \in S_2\} \quad (9.2)$$

then

$$S(c, P, 0, 0) = S(c_1, P, 0, 0) \otimes S(c_2, P, 0, 0) \quad (9.3)$$

Now

$$S_1 \otimes S_2 \neq \emptyset \iff S_1 \neq \emptyset \wedge S_2 \neq \emptyset \quad (9.4)$$

So, of course, the checking problem becomes the sum of the checking problems on each subproblem, and hence $O(n)$ in the worst case. This beats both the ground and backtracking subsearch engines.

(The arguments here are also very suggestive of the motives behind dynamic backtracking [25], and the polynomial factorizability discussed by McAllester [43].)

Other subsearch problems can also exploit the factorization:

1. Counting: take the product of the sizes.
2. Iteration: a double loop of the iterator for each set.
3. Enumeration: just store the separate $S(c_i, P, 0, 0)$ to save space.
4. Selection: just select on each sub-clause separately.

If we have general u and s then we need CCSPs. We can split the allowed u and s between the two otherwise independent CCSPs.

$$S(c_1 \vee c_2, P, u, s) = \bigcup_{\substack{0 \leq u' \leq u \\ 0 \leq s' \leq s}} S(c_1, P, u', s') \otimes S(c_2, P, u - u', s - s') \quad (9.5)$$

For example

$$\begin{aligned} S(c_1 \vee c_2, P, 1, 0) &= S(c_1, P, 0, 0) \otimes S(c_2, P, 1, 0) \cup \\ &S(c_1, P, 1, 0) \otimes S(c_2, P, 0, 0) \end{aligned} \quad (9.6)$$

Compare these with (4.24) for conjunctions of clauses.

9.3 Disjoint Sets Example

Consider the clause

$$c = \forall i, j, h. \neg x(i, h) \vee \neg y(j, h) \quad (9.7)$$

with domains $i, j \in \{1, \dots, n\}$ and $h \in \{1, \dots, d\}$. The graph for the sub-CSP is given in Figure 37.

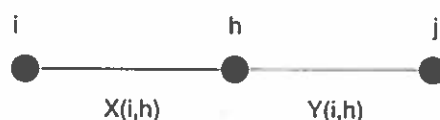


FIGURE 37. Example of a linear sub-CSP graph. Structure of the sub-CSP for the “disjoint” constraint of (9.7)

We can call this a “disjoint constraint” because if we define the sets

$$X = \{h \mid \exists i. x(i, h)\}$$

$$Y = \{h \mid \exists j. y(j, h)\}$$

then sets X and Y are constrained to be disjoint.

There are $O(n^2d)$ ground clauses, and this is the best we can expect from a ground solver.

A standard way to pre-process CSP problems is to enforce node consistency: that is, to make sure that we remove values from domains at nodes if they are inconsistent with their immediate neighbors. Doing this for node h means removing values of h that do not have a possible i value, and also if there is no possible j value. In this case the linear nature of the graph, and the associated lack of a constraint between i and j means that if there is a node-consistent assignment for h then there is a totally consistent assignment.

In terms of the clause we are effectively re-writing it as

$$c = \forall h. \neg [\exists i. x(i, h)] \vee \neg [\exists j. y(j, h)] \quad (9.8)$$

and the checking problem is just a matter of checking whether each set is nonempty for every h . Node-consistency is sufficient to solve the checking problem. This takes time $O(dn)$ even in the worst case, which improves on the ground solver.

If we want to enumerate all solutions to the CSP, we can handle each value of h separately. With $h = H$ we have the clause,

$$c = \forall i, j. \neg x(i, H) \vee \neg y(j, H) \quad (9.9)$$

which is effectively the same as (9.1) in the previous section, and can be solved in time $O(n)$. Thus we can expect a general reduction from $O(n^2d)$ to $O(nd)$ due to the linear nature of the graph.

Now consider simple backtracking for the checking problem. If we branch on variables in the order i, j, h we will also take time $O(n^2d)$ in the worst case. We can improve on this using the more natural order i, h, j and exploiting the fact that once we have reached j the value of i is irrelevant. This could be achieved by dependency maintenance techniques [1]. The difference of the complexities is just a reflection of the fact that in the order i, j, h the induced width (see [1, and others] for definitions and general discussion) of the graph is two, whereas with the order i, h, j the induced width drops to one. We shall return to this issue in Section 9.5.

An alternative view is that we should effectively define a new predicate $A(h)$

by

$$A(h) \iff \exists i. X(i, h) \tag{9.10}$$

$A(h)$ just remembers whether there was a successful value for i but doesn't need to remember the value itself. Then we can just check

$$\forall h, j. A(h) \longrightarrow \neg Y(j, h) \tag{9.11}$$

We can compute the new predicate $A(h)$ in time $O(nd)$ and then the final checking is again $O(nd)$. This is at the cost of $O(d)$ extra space usage.

A learning mechanism in the subsearch could well derive such $A(h)$ and such learning can be encoded in the axioms themselves by means of introducing new predicates. It is intriguing that issues in solving the subsearch might be used to suggest changes to the representation itself.

9.4 No-Circles Example

So far we have just given simple examples of reduction in cost arising from exploitation of the graph structure. In this example we want to compare subsearch with what we might do if we were given the freedom to directly implement the effects of an axiom with special-purpose code.

We take a constraint that we might reasonably add to the logistics system: we demand that objects are not moved in circles. That is, if an object is at a

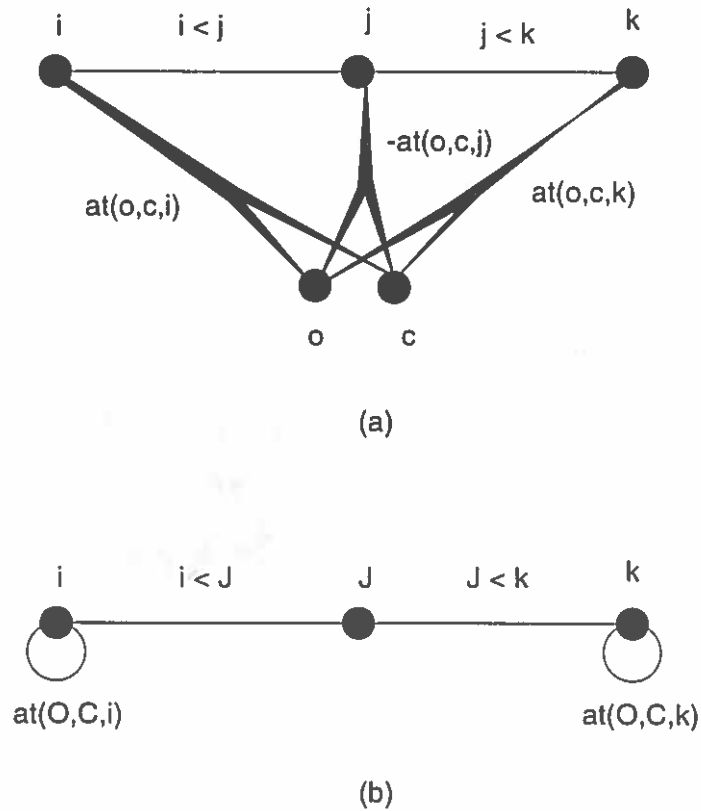


FIGURE 38. (a) Structure of the sub-CSP for the nocircles constraint c of (9.12). (b) Structure after binding $o = O$, $c = C$ and $j = J$ corresponding to $R^-(c, \neg \text{at}(O, C, J))$

location at two time-points then it is there at all intermediate times as well:

$$c = [i < j \wedge j < k \rightarrow \text{at}(o, c, i) \wedge \text{at}(o, c, k) \rightarrow \text{at}(o, c, j)] \quad (9.12)$$

The corresponding constraint hypergraph is given in Figure 38(a).

We will consider just one optimization we would like to achieve. Suppose that we are doing backtracking search (not subsearch) and that the search engine

changes the value of $\text{at}(O, C, J)$ in P from UNVALUED to FALSE (where O, C, J are constants).

We need to look for propagations and contradictions arising from the change. Suppose we find that $\text{at}(O, C, I) = \text{TRUE}$ for some $I < J$, that is, the city C has already been visited by object O . In this case the search engine will want to ensure that O does not return to C at any time after J . That is, for all $k > J$ we set $\text{at}(O, C, k) = \text{FALSE}$, or report a contradiction if we find any such k for which it is already set to TRUE.

Suppose that the search engine did this and did not find a contradiction. If we were hand-coding this constraint we would know it would be pointless to continue to look for a different $I' < J$. Even if the city had been visited twice before J we will learn nothing new. We would optimize the subsearch and stop at the first I with $\text{at}(O, C, I) = \text{TRUE}$.

Is it possible for a general purpose subsearch method to capture such a seemingly special-purpose hand-crafted optimization?

On setting $\text{at}(O, C, I) = \text{TRUE}$, the usual incrementality means we will be concerned with subsearch on $R^-(c, \neg\text{at}(O, C, J))$ which is just

$$\forall i, k. \neg(i < J) \vee \neg(J < k) \vee \neg\text{at}(o, c, i) \vee \neg\text{at}(o, c, k) \quad (9.13)$$

The structure of this clause is given in Figure 38(b).

In pure subsearch terms, the linear structure, similar to the previous examples, suggest that we find all the solutions to i and k separately and then take the

cross-product. This would be appropriate if we were doing WSAT and wanted all unsatisfied clauses. However, if we are doing unit propagation then it would give us too much information. Consider, Figure 23(b) again. When we extract ground clauses from $[S(C_{\neg l'}, P, 0, 0) \cup S(C_{\neg l'}, P, 1, 0)]$ we only use the UNVALUED literal. If we get two different ground clauses but with the same UNVALUED literal then we will be repeating work.

```

proc UNIT-PROPAGATE( $l, P$ )
   $L := \{l\}$ 
  while  $L \neq \emptyset$ 
    select some  $l' \in L$  and remove it from  $L$ 
     $P := P \cup \{l' = \text{true}\}$ 
     $C_{\neg l'} := R(C, \neg l')$ 
    if  $C_{\neg l'} \neq \emptyset$ 
       $L' = \text{GET-ALL-LITERALS-IF}(S(C_{\neg l'}, P, 1, 0), S(C_{\neg l'}, P, 0, 0) = \emptyset)$ 
      if  $L' = \text{failure}$  return Contradiction
       $L := L \cup L'$            store the literals in  $L'$  for later propagation
    end while
  return  $P$ 
end

```

FIGURE 39. Final version of unit propagation for literal l . This method is similar to Figure 23(b), but uses a minimal extraction of information from the sets $S(C_{\neg l}, P, 1, 0)$ and $S(C_{\neg l}, P, 0, 0)$.

Instead we rewrite the unit propagation as in Figure 39 using a rather specialized subsearch problem.

Definition 9.4.1

“Conditional-false-stripped” subsearch

INSTANCE: A (u,s)-restriction S_1 and a conditional $\text{cond}(S_2)$ on a second set S_2

TASK: If the conditional fails then return failure. Otherwise take the clauses from S_1 and remove the FALSE literals in each, calling the results subclauses. Then remove duplicate subclauses, and return the resulting set of subclause. Hence, we are returning a *set* of clauses in which all literals are either TRUE or UNVALUED.

PSEUDO-CODE: GET-ALL-SUBCLAUSES-IF(S_1 , cond(S_2)) \square

Here we use the notation GET-ALL-LITERALS-IF for the special case of GET-ALL-SUBCLAUSES-IF with $S_1 = S(C, P, 1, 0)$, that is, when the subclauses will be just literals, and so we return either failure or a set of literals that are all currently UNVALUED.

The advantage of this approach is that once the subsearch returns a subclause it can prune subsearch that could only lead to returning the same subclause again. The conditional is just so that we can terminate the subsearch immediately if we detect a contradiction.

In the no-circles example we use such subsearch with $S_1(C, P, 1, 0)$ and C given by (9.13). But now note that (9.13) splits

$$\begin{aligned} & [\forall i. \neg (i < J) \vee \neg \text{at}(o, c, i)] \vee \\ & [\forall k. \neg (J < k) \vee \neg \text{at}(o, c, k)] \end{aligned} \quad (9.14)$$

and so we can use (9.6). For the first term in (9.6) the subsearch needs to check that

$$S([\forall i. \neg (i < J) \vee \neg \text{at}(o, c, i)], P, 0, 0) \quad (9.15)$$

is non-empty and then should evaluate

$$S([\forall k. \neg (J < k) \vee \neg \text{at}(o, c, k)], P, 1, 0) \quad (9.16)$$

The point is that subsearch no longer needs to do more than check whether the first set is empty, and so can stop on finding the first element. This is the subsearch equivalent of the special optimization we just described. However, it is now a result of the general-purpose language of subsearch.

Of course, it remains for future work to implement a subsearch engine that would achieve such optimizations automatically. It will be especially interesting to see to what extent domain-independent intelligent search mechanisms applied to the subsearch can take over from domain-specific approaches to optimizing the implementation of constraints.

9.5 Exploiting Graph Structure

If the induced width of the graph is k (that is, it is a partial k -tree) then there are algorithms that are exponential only in k rather than the number of nodes of the graph [19] (though it seems likely that such algorithms also have a space usage that is exponential in k [1]). For example, trees can be solved in linear time.

This suggests using the cycle-cutset method [17] to solve CSPs. The aim is to just branch on a set of “cutset” nodes. These nodes are chosen so that removing them from the graph gives a tree (or more generally a k -tree) which can then be solved directly rather than via search. For this to be effective we want to find a small, and preferably optimal, set of nodes. Unfortunately, finding the smallest

cutset is in itself NP-complete and so generally disfavors this approach.

However, in the case of subsearch the relevant size is the number of universally quantified domain variables which is likely to be a lot smaller than the total number of atoms in the problem. Also, in the case of subsearch the same clauses will be used many times with different states P , hence the effort spent to analyze the syntactic structure of the clause (that is, its graph) can be amortized over many uses of the same graph. In this sense sub-CSPs are an unusual case: normally we would expect the CSP graphs to change from instance to instance. It is also worth noting that in many cases the structure of a clause is independent of the domain sizes and hence of the size of the problem instance. For example, in the no-circles example that we just discussed it might be difficult to find the optimal way to handle the clauses, however we only need to do it once. It would also be possible to run tests with small problem instances to find the best ways to handle constraints and use the methods learned in order to handle the largest instances. For all these reasons it seems quite possible that analyzing the graph structure to optimize the way that a clause is handled will have very low amortized cost.

Finally, we remark that we have mostly ignored probabilistic issues concerned with the imbalance of predicates (see Section 9.1.1); however such semantic issues should be allowed to influence the way that the structure is exploited. Whether or not the constraints on an edge are almost always very tight or instead very loose will affect how it should be treated.

9.6 Summary

There are good reasons to believe that application of advanced search techniques to the subsearch would be beneficial. It could even be that the application of domain-independent techniques to the subsearch (not the search) will lead to optimizations that mimic what we would normally consider to be domain-specific implementation methods for specific constraints.

A subsearch system should be able to recognize various classes of graphs for the sub-CSP and make appropriate choices for the subsearch algorithm. The cost of such choices can be amortized between uses of the subsearch within an instance, and possibly even between problem instances as the problem instance just affects the size of the domain and not the structure of the graph. Note that such methods would not be conceivable in the ground approach because we have lost the structure that is a property of the general encoding and hence is repeated in different problem instances.

One difficulty with these ideas is that maybe we just never get to use clauses that are sufficiently complex for subsearch to show large gains. It seems unlikely when coding a domain that a human would write very large constraints. On the other hand, large clauses might well be generated automatically in some cases: perhaps, from conversion of non-clausal formulas, or perhaps from some learning process (see Section 11.2).

CHAPTER X

RELATED WORK

We are aware of just two related approaches: global constraints, and an extension of GSAT to work with non-clausal boolean formulas [63].

10.1 Global Constraints

As can be guessed from its name, a global constraint is a constraint that involves all (or at least many) of the variables and takes a global view, rather than involving just a small number of variables in the problem and some local property. Such constraints are often used in constraint programming (CP) and constraint satisfaction problems in general.

Clearly, quantified clauses will often qualify as global constraints. For example, in the logistics domain, many of the axioms will involve all timepoints and objects and so involve a large fraction of the atoms.

A standard example of a global constraint is the ALL-DIFFERENT constraint. This takes a set of finite domain variables X_i and ensures that they are given different values. That is, it has the meaning

$$\text{ALL-DIFFERENT}(\{X_i\}) \iff \forall i, j. X_i \neq X_j \quad (10.1)$$

In the area of scheduling a useful constraint is the following. For every interval (pair of timepoints) in the domain the cumulative resource usage of all the

tasks constrained to be done in that interval must not exceed the total resource availability in that interval. Roughly speaking we have

$$\forall t_1, t_2. \left(\sum_i \text{usage}(i) \times \text{within}(i, t_1, t_2) \right) \leq \text{available}(t_1, t_2) \quad (10.2)$$

where $\text{within}(i, t_1, t_2)$ is 1 if the task i has to be done in the interval and 0 otherwise, $\text{usage}(i)$ is the resource usage of task i , and the bound is obtained from the resources available within the interval. This is a global constraint for two reasons. It involves a universal quantification over pairs of timepoints. Also, the interval might be the large enough so that every task is within the interval, and hence it requires a global sum of all the resource usages.

We see that global constraints are also typically composites of simpler primitive constraints. For example, we really did not need ALL-DIFFERENT but could have just used the equivalent set of inequalities. Two reasons to use such composites are:

1. Better implementation. It may be more efficient to implement the composite as a whole, rather than with its components treated separately.
2. More powerful inference (propagation). It might allow propagations that would otherwise not occur (and would have to be discovered by search instead, and with a higher cost).

For example, ALL-DIFFERENT is composed of $O(n^2)$ primitive separate constraints, however it can be implemented to run in $O(n \log n)$ time [53] (since we can check for duplicates in a set by sorting them first).

The resource constraint for scheduling is a redundant constraint in the sense that it is not logically necessary. To check that a schedule is legal we do not need to check usages over all intervals but only at individual timepoints. It is not an axiom but rather a theorem derived from the axioms. It is used because it allows us to find propagations that we would otherwise not obtain directly from the axioms alone.

Formulas in QCNF can clearly be regarded as global constraints in the sense that they are a composite of all the equivalent ground clauses. Indeed our main point has been that a composite, the quantified clause, can be implemented (using subsearch) to run more efficiently than we can achieve by treating its components, the ground clauses, separately.

Thus, we can regard this work as falling within the general topic of global constraints. However, work on global constraints is usually focused on a single constraint, whereas the work here is much more general and considers a whole language: QCNF. Hence the immediate differences from general usage of global constraints are

1. QCNF is more general in the sense that we automatically handle more constraints without special purpose coding.
2. Subsearch and its associated complexity results: the fact that implementing a quantified clause as a constraint is NP-complete, and requires search, is new.

The generality of QCNF is a definite advantage. However, as usual, we can expect that a general method will also have cases in which it loses to a special purpose

encoding of a single global constraint.

One can make the conjecture that many of the specialized optimizations achieved when hand-coding global constraints might be achievable by using domain-independent search methods for the relevant subsearch. In practice, it is hard to believe¹ (but not impossible) that using domain-independent methods for subsearch will be able to reproduce all the considerable effort and ingenuity that has gone into implementing even such a simple constraint as the ALL-DIFFERENT constraint [53]. However, it may reproduce enough, and do so automatically, to be useful.

We have also not discussed the issue of proof-based methods, such as resolution, except in the general discussion of inference in SAT in Chapter III. The focus of this thesis has been entirely on model-search methods such as WSAT. However, QCNF is a subset of first order logic and so we can resolve together clauses, and do proof-based reasoning. This opportunity for reasoning does not seem to be practical for the global constraints in general. It would correspond to taking global constraints such as ALL-DIFFERENT and attempting to combine them to produce new constraints. Usage of global constraints, and CP itself, focuses on model-based search methods, and their extension to proof-based search would seem problematic. In contrast, in Section 11.2 we give some reasons to believe that using proof-based reasoning when using languages such as QCNF could lead to big advantages.

Finally, we note a drawback of the QCNF approach is that it only obtains gains when the primitive constraints can be collected together into a single quantified clause. We have not given methods that can handle collections of quantified

¹Of course, opinions differ substantially on this point!

clauses better than handling them all separately.

10.2 Sebastiani's Approach to Non-Clausal Formulas

Sebastiani has presented an interesting extension of GSAT to work with non-clausal boolean formulas [63]. The work was motivated by the observation that conversion of an arbitrary formula to clausal form can result in a large, and even exponential, increase in size.

For example, consider

$$(x \wedge y) \vee A \tag{10.3}$$

where A itself is a boolean formula. We convert to CNF by "multiplying out". We first get

$$(x \vee A) \wedge (y \vee A) \tag{10.4}$$

and then the formula A might in turn need to be expanded, so that eventually we can get exponential size increase because we have doubled the space needed by A by causing it to be repeated. Of course, the general idea of keeping formulas as small as possible was also an initial motivation for QCNF: conversion of a quantified clause to ground form can give an exponential increase in size. Sebastiani also implemented GSAT in terms of abstractions, such as procedures that return the effective number of unsatisfied clauses.

We differ on the following counts:

1. We handle quantifiers. Sebastiani was limited to boolean connectives. In

practice, it seems that quantifiers are more of a problem than non-clausal forms. After all, it is possible to convert to clausal form by introducing new predicates. In the example above, if we define the new variable z by $z = x \wedge y$ the size explosion can be avoided. (Such extra variables are not decision variables, but are dependent, and so the size of the search space is not necessarily increased). In practice, conversion to clausal form does not seem to cause difficulties (or they are avoidable), whereas conversion to ground form can easily make the problem too large to run on the machine.

2. Solving the relevant abstraction, subsearch, is recognized as being NP-hard and to be solved best by search. That is, the main difference is the recognition of the subsearch problems, and their solution via search methods.

At a deeper level it seems that use of quantified clauses captures more of the structure in a problem than the use of non-clausal formulas, and hence we can obtain more gains by exploiting such structure.

However, we did start by discussing QPROP rather than the special case QCNF and QPROP can contain non-clausal structures mixed in with quantifiers. It is conceivable that a combination of subsearch on quantifiers with Sebastiani's approach to non-clausality could prove useful.

CHAPTER XI

OPEN QUESTIONS AND FUTURE WORK

The concept of being able to run propositional algorithms such as Davis-Putnam and WSAT on quantified expressions via subsearch is new, and hence we have been able to do little more than sketch the basics. Many interesting questions and possibilities for future work arise.

There are likely to be many issues arising when trying to implement improvements to the subsearch, in particular in trying to implement the exploitation of sub-CSP structure as discussed in Chapter IX. Maybe, we could also use a CSP engine such as MULTI-TAC [47] that could take the many calls to the subsearch engines as learning examples, and so produce specialized methods to solve the sub-CSPs. However, here we just discuss a few of the issues arising.

11.1 Implications for Problem Encodings

In SATPLAN, a lot of effort has gone into manipulating the problem encoding so that the ground CNF-formulas produced are small [37]. If a lifted solver is available then it is no longer at all clear that the size of the ground CNF is a good criterion for selecting encodings. It is conceivable that there are encodings that are very large if ground, but for which the structure of the axioms happens to make the subsearch and search run very well.

An obvious example here is to explore the addition of “redundant constraints”

to the set of axioms: such constraints not being logically necessary, but designed to help the search engine (COMPACT-B is designed precisely to find some such constraints automatically). Such constraints can easily involve many quantifiers – being designed to help the solver in some special situation – but would tend to swamp a ground solver (unless the redundant constraints were themselves also carefully selected for size). However, a lifted solver could well be able to exploit them effectively.

11.2 Non-Minimal Lifting

Using quantified clauses instead of SAT preserves a lot of structure naturally present in problems. So far we have only exploited this in the subsearch. However, the search itself should also be able to exploit such structure. In such a case the search will itself be changed by the lifting. We will look at some ideas for non-minimal lifting of the backtracking propagation based methods, and also make some brief comments about possible non-minimal lifting in local search.

11.2.1 Lifted Backtracking

In Section 3.9.2 we considered the pre-processing method COMPACT-B, with the standard example

$$\begin{aligned}
 & \neg a \vee b \\
 & \neg a \vee c \\
 & \neg b \vee \neg c \vee d
 \end{aligned}
 \tag{11.1}$$

Here, simple resolution shows that $\neg a \vee d$ is entailed, and the idea was to add it to the theory because it improved the propagation (which we represented by “ \implies ”). COMPACT-B discovers such entailed clauses by speculative assignments to pairs of literals. In this example, it would try $d = \text{FALSE}$, find that $\neg d \not\implies \neg a$ but that $\neg d \wedge a \implies \perp$, and hence conclude that the clause is worth adding to the theory.

What happens when we try to lift this reasoning?

We could certainly do a minimal lifting of COMPACT-B. We would do all branching in terms of ground literals and just use the lifted form of unit propagation in order to derive the needed contradictions. However, this has the problem that we only ever find ground clauses. This does not seem right: instead, we should surely be learning quantified clauses. Consider the simple case where we just have multiple copies of the same example:

$$\begin{aligned}
 \forall i. \quad & \neg a(i) \vee b(i) \\
 \forall i. \quad & \neg a(i) \vee c(i) \\
 \forall i. \quad & \neg b(i) \vee \neg c(i) \vee d(i)
 \end{aligned} \tag{11.2}$$

If we did a minimal lifting we might branch on $d(1)$ and discover the entailed clause $\neg a(1) \vee d(1)$. However, the original problem was symmetric between all the elements of the domain of i . One can often regard the quantified forms as making such symmetries manifest, rather than hidden as they are in the ground formulations (see also the work by Joslin and Roy [35]). Such symmetries suggest that if we learn one ground clause then there is some chance that it could be

generalized to a lifted clause. In this case, a non-minimally lifted COMPACT-B should be able to directly deduce that $\forall i. \neg a(i) \vee d(i)$ is entailed.

An example very similar to this this actually occurs in the logistics axioms as given in Chapter II. It is clear from the axioms that moving an object means that it must be on a plane for at least two consecutive timepoints. If it is in a city at time i then at times $i + 1$ and $i + 2$ it cannot be in any other city. However, unit propagation does not capture this information, and to fix this we should add an extra lifted clause that is entailed by the axioms.

Suppose that we could indeed lift learning mechanisms so as to learn *lifted* clauses. This could have enormous advantages:

1. We can save a lot of work: in the example above we would hope to learn the lifted clause in constant time.
2. The lifted learning can jump ahead in the sense that we might learn clauses that will allow us to prune parts of the search space before reaching them. The lifted clause will contain many other clauses besides the ground clauses used to create it.
3. Learning lifted clauses rather than a set of ground clauses means we can use subsearch on them.
4. Often the clauses learned are properties of the domain rather than of the instance. In this case, the learned clauses might also carry over from one instance to the next. There is the potential to avoid learning everything from scratch on each new instance. In particular, we might learn new clauses

on small instances where extensive complete searches are possible, and then apply them to larger more challenging instances.

5. The entailed clauses could also be useful for theory validation: the user could inspect them to check that they are consistent with the intended interpretation [7].

Ginsberg has suggested a mechanism to do such lifted learning [26]. It is to lift the dependency maintenance techniques used within algorithms such as Relevance Bounded Learning (RBL) [3]. It is well-known that backtracking searches are equivalent to resolution proofs: given a search tree we can convert it to a resolution proof in polytime [48, and others]. Hence, the dependency maintenance techniques can also be thought of in terms of resolution proofs. In a ground solver all such resolution is ground resolution. However, when using QCNF we have the option of keeping the resolution lifted. Thus, in the example, when the solver derives $\neg a(1) \vee d(1)$ it would be tracing the steps with lifted resolution between the clauses. In this case lifted resolution between the clauses immediately yields the desired generalization of the ground clause.

We could, of course, have just tried lifted resolution between the clauses, but such resolution suffers from producing lots of clauses that are not used. Instead, using a backtracking search to guide the lifted resolution has a chance to ensure that the resolution is a lot more focused onto clauses that are really useful.

If such methods are applied to algorithms such as RBL, then we will potentially end up learning lifted clauses containing many literals. Part of the motivation for Chapter IX was to start to set up mechanisms that will be able to automatically exploit the subsearch needed with such clauses.

Finally, we recall that quantified clauses could be regarded as global constraints, one potential advantage of QCNF over the globals as used in constraint programming is that QCNF constraints can be combined to produce new constraints (with the potential advantages we just discussed). This would be hard to do in constraint programming in general: the language of constraints is much freer, and it is much harder to reason about them, rather than just using the propagation that they produce.

11.2.2 Lifted Learning for Local Search

We briefly consider two extensions to GSAT/WSAT algorithms that are used to combat the problem of getting trapped in a local minimum. Both are concerned with changing the cost surface so that the minimum gets filled in and the search will move on.

11.2.2.1 Adding-New-Clauses

We can add new entailed clauses that have the effect of filling in the minimum in which we are trapped [6, 74]. If such clauses were lifted then the effect would be to not only fill in the local minimum, but also to fill in other equivalent minima. This could be an advantage because then we would be filling in many minima simultaneously. However, the extra overhead from the lifted clauses could also slow down the search, and it is not clear whether there would be a net gain.

11.2.2.2 Clause Weights

If a clause is constantly being broken then it can be a good idea to adjust the search so that more emphasis is placed on fixing it. This is done in WSAT by weighting each clauses and adjusting the clause selection scheme accordingly [65].

If lifting, we cannot (excluding sparse storage mechanisms) store a weight for each ground clause. Instead we could just weight the quantified clauses. With such a mechanism ground clauses might end up receiving a high weight despite themselves never having been broken; the high weight having arisen from some other grounding of the quantified clause. It is conceivable that such weighting could be good, because in the case that the search should happen to move into the region where the ground clause is now broken then the weighting scheme needed to help us escape will already be in place. Experiments in this area would be interesting.

In both of the above areas, we again have the potential for inter-instance learning: clauses or weights might have a more general lesson than the particular instance being used. Again a ground solver without access to the lifted structure has no chance to do such inter-instance learning.

There are also interesting questions here as to whether the local minima met by a real search have enough common structure that clauses or weights to help escape one minimum will have much chance of helping some future minimum.

11.2.3 Subsearch Directed Search

We have seen that subsearch is a major component of search. Hence, it makes sense that the search engine might gain some advantages by modifying its decisions

to take account of the costs of subsearch. The search might want to avoid regions in the search space in which the subsearch becomes more costly: perhaps there are cases in which we can decide to search more nodes, but then search them so much faster that it is a net gain. If nothing else, then such considerations might be used as a tie-breaker between choices that are equal as far as the search itself is concerned.

Alternatively, the search now has access to the lifted structures and so might be able to exploit these in ways that are more effective or cheaper than the usual CSP heuristics such as “most-constraining first”. Perhaps the lifted structure of the clauses can be used to help pick the next branch variable in the Davis-Putnam procedure.

11.3 Breaking out of QCNF

There are many conceivable directions in which we might go in extending QCNF. Some directions are obvious:

1. Extend to various operations research representations:
 - (a) 0/1 variables with linear inequalities
 - (b) general FD domain variables (or finite domain variables in the sense of CSPs)

In such representations it is also reasonable to have global constraints that are naturally written with universal quantifiers.

2. Non-clausal form with quantifiers.

However, we will only look at one desirable extension which is to reinstate the existential quantifiers. After all, part of the motivation for using QPROP was that expanding quantifiers loses structure that we should be exploiting, and so always expanding existentials is surely obscuring some structure.

11.3.1 General QCNF and the Polynomial Hierarchy

Suppose we define “ k -Alternating-QCNF” as clauses of the form

$$c = \forall v_{11}, \dots \exists v_{21}, \dots \forall v_{31}, \dots \exists v_{41}, \dots \dots Qv_{k1}, \dots \bigvee_a l_a(t_a) \quad (11.3)$$

where Q is \exists if k is even and \forall if k is odd. That is, we have k alternations between existentials and universals. The case $k = 1$ is just QCNF as defined in Chapter IV. The case $k = 2$ still has the property of grounding directly to CNF, see (2.8).

If we now follow the same reasoning used in Section 6.2, then checking whether c is broken by the current assignment P becomes a matter of finding a solution to the negated clause

$$\exists v_{11}, \dots \forall v_{21}, \dots \exists v_{31}, \dots \forall v_{41}, \dots \dots \neg Qv_{k1}, \dots \bigwedge_a \neg l_a(t_a) \quad (11.4)$$

If we read each negated literal $\neg l_a(t_a)$ at fixed truth value assignment P as a relation on a subset of the variables v_{ij} then we have the same form as the problems B_k described in section 7.2 of Garey and Johnson [22].

Thus, the checking (subsearch) problem, that is, showing c is not satisfied, for such k -alternating problems are in the classes Σ_k^P of the polynomial hierarchy. The case $k = 1$ is just $\Sigma_1^P = \text{NP}$.

Note that even for the case $k = 2$, the checking problem is $\Sigma_2^P = \text{NP}^{\text{NP}}$. To show that a set of bindings for the universally quantified variables gives an unsatisfied clause, we have to show that no binding for the existentially quantified variables gives a satisfying literal.

Maybe such formulas have applications in game-like problems with the alternations correspond to alternating moves by the players. However, in SATPLAN, or similar systems, alternations do not seem to arise very often or be very deep. Hence, we will look at much simpler extension that is more directly usable.

11.3.2 An A*E Extension of QCNF

Instead of trying to handle arbitrary numbers of existentials and ending up in the rarefied heights of the polynomial hierarchy we will only consider a minimal usage of existentials. If we look at the logistics axioms of Chapter II, we observe that there are no formulas with more than one existential. Hence a good start to using existentials would be to limit ourselves to expressions of the form

$$\forall i_1, \dots. \exists j. A[i_1, \dots, j] \tag{11.5}$$

where A is a disjunction of literals with no further quantifiers. Even such clauses would be a useful extension to QCNF. For example, clauses such as

$$\forall i. p(i) \longrightarrow \exists j. r(i, j) \tag{11.6}$$

often arise from relationalizing some function or partial function. In this case, the “sub-CSP” is

$$p(i) \quad \wedge \quad \forall j. \neg r(i, j) \quad (11.7)$$

To find a satisfying assignment then means even with a fixed i we have to consider all j values. In this case using a sparse representation of P will probably be useful here because such a representation more naturally supports jumping directly to the relevant j values.

11.3.2.1 Relation to Simple Symmetries

One reason for wanting to allow explicit existentials is that they might allow better exploitation of symmetries. Quantifiers in QCNF naturally make some permutation symmetries manifest, and this is potentially useful as having to rediscover them is expensive. We can conjecture that an existential is often an indicator of potential symmetry that we should be exploiting; e. g. if you have $\exists h$ then the possibility of a symmetry over elements of the domain of h should be considered. As an example consider the pigeon-hole problem (PHP) described in Section 2.4. From (2.11) we have

$$\forall i. \exists h. p(i, h) \quad (11.8)$$

which seems associated with the symmetry between the holes. The PHP is hard to solve if written in propositional form [29], but when we exploit the symmetry it

becomes polytime [10].

Hence, existentials could well be used as by the solver to keep an eye out for symmetries in the quantified variables. However, exploiting symmetries within the context of a lifted SAT solver is generally an open question (though some previous work exists [35]).

CHAPTER XII

CONCLUSION

12.1 Contributions

We have considered “lifting” of propositional solvers, that is, converting them to work with quantified clauses rather than just the equivalent ground expressions. We found that such lifted solvers are not only possible with reasonable overheads, but also have distinct theoretical and practical advantages over the ground solvers.

Our primary technical contribution was to show that many apparently different tasks such as unit-propagation in the Davis-Putnam procedure and finding sets of unsatisfied clauses as WSAT actually have a lot in common. The common property is that they require extracting relevant information and clauses from the set of input clauses. We showed that such extraction is NP-hard in general (Chapter VI), and this fact, combined with the natural view of extracting the clauses as a form of database search, motivated us to label this process as subsearch.

We identified a number of different subsearch processes (Chapter IV), and showed (Chapter V) how they are used within standard algorithms. This allowed us to extend such algorithms to work with quantified clauses.

We showed that the standard way to deal with such quantified clauses, namely to ground them out first, corresponds to doing the subsearch using a method that is essentially generate-and-test. Using better search algorithms for the subsearch was shown, both theoretically (Chapter VII) and experimentally

(Chapter VIII), to enable the algorithms to run faster, and with considerably less memory usage. The difference in performance was sufficiently large that we could run problems inaccessible to usual ground solvers. The practical impact of this should be much greater freedom in encoding problems in areas such as SATPLAN.

We reformulated the subsearch problems on quantified clauses as search problems in an associated CSP (or CSP-like problem), that we called the sub-CSP (see Chapter VI). In Chapter IX, we also discussed various ways in which more advanced search techniques applied to the sub-CSP of a quantified clause have the potential to obtain some of the benefits that would normally require hand-coding of the corresponding global constraint.

We illuminated the natural overlaps and differences between quantified clauses and the concept of global constraints as used in constraint programming and constraint satisfaction. A potential advantage of quantified clauses is that they permit proof-based reasoning, and we discussed some potential advantages and methods for this in Section 11.2.1. It is possible to combine clauses by resolution, whereas global constraints in general do not have such a reasoning mechanism available.

12.2 Closing Comment

Quantifiers capture a structure that is present in many problems. Throwing away such structure away by conversion to SAT causes many difficulties. Instead we have seen that satisfiability algorithms can successfully exploit such structure. Many questions remain open, and there are many prospects for interesting future work in this area. Of particular interest is the possibility of lifting various learning schemes such as COMPACT-B and relevance-bounded learning.

BIBLIOGRAPHY

- [1] Andrew Baker. *Intelligent Backtracking on Constraint Satisfaction Problems*. PhD thesis, University of Oregon, 1995.
- [2] Roberto J. Bayardo and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 298–304, Portland, OR, 1996.
- [3] Roberto J. Bayardo and Robert C. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 203–208, Providence, RI, 1997.
- [4] Tania Bedrax-Weiss, Ari K. Jónsson, and Matthew L. Ginsberg. Unsolved problems in planning as constraint satisfaction. Available from <http://www.cirl.uoregon.edu/tania>, 1996.
- [5] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 2, pages 1636–1642, 1995.
- [6] Byungki Cha and Kazuo Iwama. Adding new clauses for faster local search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 332–337, Portland, OR, 1996. AAAI Press/The MIT Press.
- [7] John Conery. Personal Communication, 1999.
- [8] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.
- [9] Martin C. Cooper, David A. Cohen, and Peter G. Jeavons. Characterizing tractable constraints. *Artificial Intelligence*, 65:347–361, 1994.
- [10] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Exploiting symmetry by adding symmetry breaking predicates. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR-96)*, Boston, MA, 1996.

- [11] James M. Crawford. Code for NTAB and COMPACT. From URL <http://www.cirl.uoregon.edu>, 1996.
- [12] James M. Crawford and L. D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 21–27. AAAI Press/The MIT Press, 1993.
- [13] James M. Crawford and L. D. Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81:31–57, 1996.
- [14] Martin Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Journal of the ACM*, 5:394–397, 1962.
- [15] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
- [16] Rina Dechter. Enhancement schemes for constraint processing. *Artificial Intelligence*, 41:273–312, 1990.
- [17] Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.
- [18] Rina Dechter and Avi Dechter. Belief maintenance in dynamic constraint networks. In *Proc. of AAAI-88*, pages 37–42, 1988.
- [19] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.
- [20] R. E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
- [21] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.
- [22] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman and Company, 1979.
- [23] Matthew Ginsberg. *Essentials of Artificial Intelligence*. Morgan Kaufman, San Mateo, 1993.
- [24] Matthew Ginsberg and David McAllester. GSAT and dynamic backtracking. In Alan Borning, editor, *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, Seattle, May 1994.

- [25] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
- [26] Matthew L. Ginsberg. Personal Communications, 1996-99.
- [27] Matthew L. Ginsberg and William D. Harvey. Iterative broadening. *Artificial Intelligence*, 55:367–383, 1992.
- [28] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic, 1997.
- [29] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.
- [30] Steven Hampson and Dennis Kibler. Large plateaus and plateau search in boolean satisfiability problems: When to give up searching and start again. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring and Satisfiability*. American Mathematical Society, 1996. Proceedings of the second DIMACS Implementation Challenge, October 1993.
- [31] William D. Harvey. *Nonsystematic Backtracking Search*. PhD thesis, Stanford University, 1995.
- [32] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *Proceedings of IJCAI-95*, pages 607–613, 1995.
- [33] Mark Jerrum and Alistair Sinclair. The Markov chain Monte Carlo method: an approach to approximate counting and integration. In D. S. Hochbaum, editor, *Approximation Algorithms for NP-hard Problems*. PWS Publishing, 1996.
- [34] Ari K. Jónsson. *Procedural Reasoning in Constraint Satisfaction*. PhD thesis, Stanford University, Stanford, CA, 1997.
- [35] David Joslin and Amitabha Roy. Exploiting symmetry in Lifted CSPs. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 197–202, 1997.
- [36] H. Kautz and B. Selman. BLACKBOX: A new approach to the application of theorem proving to problem solving. In *Artificial Intelligence Planning Systems: Proceedings of the Fourth International Conference*, page (Unpublished submission to planning competition), 1998. Available from <http://www.research.att.com/~kautz/blackbox/index.html>.

- [49] Paul Morris. The breakout method for escaping from local minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 40–45, 1993.
- [50] C.H. Papadimitriou. On selecting a satisfying truth assignment. In *Proc. IEEE symposium on Foundations of Computer Science*, pages 163–169, 1991.
- [51] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
- [52] Andrew J. Parkes and Joachim P. Walser. Tuning Local Search for Satisfiability Testing. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 356–362, Portland, OR, 1996.
- [53] Jean-Francois Puget. A fast algorithm for the bound consistency of alldiff constraints. In *AAAI-98*, pages 359–366, 1998.
- [54] V. J. Rayward-Smith, I.H. Osman, C. R. Reeves, and G. D. Smith, editors. *Modern Heuristic Search Methods*. John Wiley & Sons Ltd, 1996.
- [55] R. Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 119–140. Plenum, 1978.
- [56] Raymond Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, Boston, 1991.
- [57] Olivier Roussel and Philippe Mathieu. A new method for knowledge compilation: the achievement by cycle search. In *Proceedings of CADE-13*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 493–507. Springer, 1996.
- [58] Olivier Roussel and Philippe Mathieu. Exact knowledge compilation in predicate calculus: The partial achievement case. In *Proceedings of CADE-14*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 161–175. Springer, 1997.
- [59] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
- [60] M.W.P. Savelsbergh. Preprocessing and probing for mixed integer programming problems. *ORSA J. on Computing*, 6:445–454, 1994.

- [61] T. Schaefer. The complexity of satisfiability problems. In *Proc. of the 10th ACM STOC*, 1978.
- [62] Lenhart K. Schubert. Monotonic solution of the frame problem in the situation calculus. In Henry E. Kyburg, Jr., Ronald P. Loui, and Greg N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67. Kluwer, Boston, 1990.
- [63] Roberto Sebastiani. Applying GSAT to non-clausal formulas. *JAIR*, 1:309–314, 1994.
- [64] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring and Satisfiability*, pages 521–531. American Mathematical Society, 1996. Proceedings of the second DIMACS Implementation Challenge, October 1993.
- [65] Bart Selman and Henry A. Kautz. Domain-independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of IJCAI-93*, 1993.
- [66] Bart Selman and Henry A. Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 46–51, 1993.
- [67] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings AAAI-94*, pages 337–343, 1994.
- [68] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, 1992.
- [69] D. E. Smith, M. R. Genesereth, and M. L. Ginsberg. Controlling Recursive Inference. *Artificial Intelligence*, 30:343–389, 1986.
- [70] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.
- [71] Gérard Verfaillie and Thomas Schiex. Solution reuse in Dynamic Constraint Satisfaction Problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 307–312, 1994.
- [72] Joachim P. Walser. Solving linear Pseudo-Boolean constraint problems with local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 269–274, 1997.

- [73] Joachim P. Walser, Ramesh Iyer, and Narayan Venkatasubramanian. An application of integer local search in production planning. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, 1998.
- [74] Makoto Yokoo. Why adding more constraints makes a problem easier for hill-climbing algorithms: Analysing landscapes of CSPs. In *CP97*, 1997. To appear in LNCS.