Analysis of Algorithms for Monotonic and Non-Monotonic Reasoning

John Franco, John Schlipf, and Sean Weaver

Society for Industrial and Applied Mathematics
Dedication Page
Logic has always provided a rigorous foundation for mathematics and philosophy, and more recently it has been called upon to support formal concepts and problem-solving mechanisms for computer science, operations research, and artificial intelligence. Recent work emphasizes that, in addition to traditional concerns such as soundness and completeness of proof systems, it is important to study computational requirements. Notably, a system must produce relatively short proofs or else it will take too much time to be useful. What are the most powerful systems that will admit only short proofs? Even if short proofs are guaranteed by a system, short proofs obtained by a particular implementation of the system may not be guaranteed. Is there a method for finding short proofs when they exist? For a given proof system, a valid formula, and a function $f$, what is the probability that a system will produce a proof of length bounded by $f(n)$, where $n$ is the size of the input statement? If $f$ is a low degree polynomial and the probability in question is close to 1, then the system may be suitable for practical use. To answer questions such as those above, considerable research has focused on algorithm development and complexity analysis for proof systems. In classical logic, enlarging a set of axioms results in an enlarged set of conclusions that can be drawn from the axioms. In other words, existing assertions or conclusions cannot be retracted if the set of axioms is added to. For example, a proposition can be expressed in Conjunctive Normal Form (CNF) as follows:

$$(a \lor b \lor \bar{c}) \land (\bar{a} \lor \bar{c} \lor d) \land (b \lor c \lor \bar{d}).$$

Each disjunction, called a clause, is a constraint. Adding constraints (that is, enlarging the set of axioms) shrinks the set of satisfying interpretations (models) and enlarges the set of clauses that are implied (conclusions or implicants). Many systems for dealing with this more traditional, monotonic form of logic have been proposed and algorithms implemented. Most useful algorithms rely on monotonicity for determining whether satisfying interpretations exist. Since the question whether a propositional formula has a satisfying interpretation is $\mathcal{NP}$-complete, none of these algorithms has worst-case, polynomially bounded time complexity. However, if the structure of a given formula is restricted in certain ways, for example, by restricting the number of literals in a clause to no greater than two, the question can be settled in polynomial time. Exploiting such structures often yields algorithms that perform well with high probability on random formulas.

Twentieth century logicians have primarily studied monotonic logic. However, particularly in the last 20 years, the mathematical study of reasoning non-monotonically has developed in parallel to, and largely separate from, research on monotonic logic. The motivation for the study of non-monotonic
logic is the frequent difficulty one has trying to force deductive systems into AI applications where numerous default assumptions are necessary. In some applications, CNF expressions have been used successfully to classify data points based on the values of components of binary vectors. An example is oil exploration where a data point consists of binarized data taken from well logs and may be classified as likely or not likely to represent a ‘discovery.” However, CNF expressions may not be ideal for certain applications where, often, a conclusion must be made based on the absence of information. For example, one might conclude that a person is healthy based on the absence of symptoms, even though it has not been proven that the person is well. Later, additional evidence can cause that conclusion to be reversed. The notion of making conclusions based on current information plus default assumptions, with the possibility of reversing them later as more information becomes available, is desirable, even sometimes necessary, when implementing automated reasoning systems; the necessary size of the database under a classical logic in such an application could be infeasibly large and the generation of axioms infeasibly difficult. Partly for this reason, many researchers in logic programming view non-monotonic reasoning, and in particular negation as failure, as an essential component of common-sense reasoning. Non-monotonic reasoning has standard applications in areas such as diagnosis and logic programming.

There are a variety of approaches to non-monotonic reasoning falling into numeric and symbolic categories. Of most interest to us are the purely semantic approaches including the well-founded semantics and stable semantics, the former being polynomial-time solvable and the latter \( \mathcal{NP} \)-complete. A number of algorithms have been proposed to deal with both. They are different from the algorithms developed for monotonic logic but concerns about their complexity are similar to those arising in monotonic logic.

Research in algorithm development and complexity of monotonic and non-monotonic propositional logic has advanced considerably in the last decade. It is the aim of the proposed monograph to present this progress as an exposition of up-to-date mathematical tools for the analysis of algorithms for monotonic and non-monotonic logics, as well as the algorithms themselves. We will not be exhaustive. We are more interested in presenting approaches that reveal insights. We also hope that covering both forms of propositional logic in one volume will encourage communication between two separate groups currently engaged independently in research on one or the other field.
Contents

1 Applications  1
  1.1 Consistency Analysis in Scenario Projects . . . . . . . . . . . 1
  1.2 Testing of VLSI Circuits . . . . . . . . . . . . . . . . . . . . 5
  1.3 Diagnosis of Circuit Faults . . . . . . . . . . . . . . . . . . . 7
  1.4 Functional Verification of Hardware Design . . . . . . . . . . 9
  1.5 Bounded Model Checking . . . . . . . . . . . . . . . . . . . . 14
  1.6 Combinational Equivalence Checking . . . . . . . . . . . . . 15
  1.7 Transformations to Satisfiability . . . . . . . . . . . . . . . . 17
  1.8 Boolean Data Mining . . . . . . . . . . . . . . . . . . . . . . 22

2 Math and Logic Foundations  25
  2.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
    2.1.1 Logic . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
    2.1.2 Graphs . . . . . . . . . . . . . . . . . . . . . . . . . . 31
    2.1.3 Algorithmic Structures and Operations . . . . . . . . . 33
    2.1.4 Complexity . . . . . . . . . . . . . . . . . . . . . . . . 35
  2.2 Problems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
    2.2.1 Satisfiability . . . . . . . . . . . . . . . . . . . . . . . 36
    2.2.2 Finding Minimal Models . . . . . . . . . . . . . . . . . 37
    2.2.3 Finding Stable Models . . . . . . . . . . . . . . . . . . 37
    2.2.4 Well-Founded Models . . . . . . . . . . . . . . . . . . . 38
    2.2.5 Variable Weighted Satisfiability . . . . . . . . . . . . . 39
    2.2.6 Maximum Satisfiability . . . . . . . . . . . . . . . . . . 39
    2.2.7 Weighted Maximum Satisfiability . . . . . . . . . . . . . 39
    2.2.8 Equivalence of Boolean Formulas . . . . . . . . . . . . . 40
    2.2.9 Binary Decision Diagram . . . . . . . . . . . . . . . . 40
  2.3 Representations, Structures, and Measures . . . . . . . . . . . 40
    2.3.1 (0, ±1) Matrix . . . . . . . . . . . . . . . . . . . . . . . 41
    2.3.2 Binary Decision Diagram . . . . . . . . . . . . . . . . . 42
    2.3.3 Implication Graph . . . . . . . . . . . . . . . . . . . . . 44
    2.3.4 Propositional Connection Graph . . . . . . . . . . . . . . 44
    2.3.5 Variable-Clause Matching Graph . . . . . . . . . . . . . 45
    2.3.6 WFF Digraph . . . . . . . . . . . . . . . . . . . . . . . . 45
## CONTENTS

2.3.7 Satisfiability Index ........................................... 46  
2.4 Probability ......................................................... 46  
2.4.1 Identities and Inequalities ................................. 47  
2.4.2 Distributions and the Central Limit Theorem .......... 47  
2.4.3 First Moment Method ......................................... 48  
2.4.4 Second Moment Method ...................................... 48  
2.4.5 Distributions Over Input Formulas ....................... 50  
2.4.6 Formula density ............................................... 52  
2.4.7 Markovian approximation via differential equations .. 53  
2.4.8 Deferred Decisions ........................................... 58  
2.4.9 Martingale Processes ....................................... 58  
2.4.10 Jensen’s Inequality .......................................... 59  
2.5 Eigenvalues ......................................................... 59  

3 General Algorithms ................................................. 61  
3.1 Efficient Transformation to CNF Formulas ............... 61  
3.2 Resolution ......................................................... 67  
3.3 Extended Resolution ........................................... 71  
3.4 Davis-Putnam Resolution ....................................... 72  
3.5 Davis-Putnam Loveland Logemann Resolution ............. 73  
3.6 Decompositions .................................................... 77  
3.6.1 Monotone Decomposition .................................... 78  
3.6.2 Autarkies ....................................................... 83  
3.7 Branch-and-bound ................................................ 83  
3.8 Local Search ....................................................... 86  
3.8.1 Walksat ......................................................... 86  
3.8.2 Novelty ......................................................... 86  
3.9 Binary Decision Diagram ...................................... 86  
3.9.1 Existential Quantification ................................. 89  
3.9.2 Reductions and Inferences ................................. 90  
3.9.3 Restrict ........................................................ 91  
3.9.4 Generalized Co-factor ...................................... 92  
3.9.5 Strengthen ...................................................... 94  
3.10 Algebraic Methods ............................................... 94  
3.10.1 Gröbner Bases Applied to SAT ........................... 95  
3.10.2 Integer Programming ....................................... 100  
3.11 Cutting Planes ..................................................... 102  
3.12 Elliptical Cuts ..................................................... 102  
3.13 Satisfiability Modulo Theories .............................. 102  
3.14 Simulated Annealing .......................................... 102  
3.15 Genetic Algorithms ............................................ 102  
3.16 Constraint Programming ...................................... 102  
3.17 Lagrangian Duality ............................................. 102  
3.18 TABU Search ...................................................... 102  
3.19 Stable Models ..................................................... 102
3.20 Well-Founded Models ........................................... 102
3.21 Answer Set Programming ...................................... 102

4 Algorithms for Easy Classes of CNF Formulas 113
4.1 2-SAT ............................................................ 114
4.2 Horn Formulas ..................................................... 116
4.3 Renameable Horn Formulas ..................................... 117
4.4 Linear Programming Relaxations ................................. 117
4.5 q-Horn Formulas .................................................. 121
4.6 Matched Formulas ............................................... 124
4.7 Generalized Matched Formulas ................................. 124
4.8 Nested and Extended Nested Satisfiability .................... 124
4.9 Linear Autark Formulas ......................................... 130
4.10 Classes of Unsatisfiable Formulas .............................. 135
  4.10.1 Minimally Unsatisfiable Formulas ....................... 135
  4.10.2 Bounded Resolvent Length Resolution ................ 143
4.11 Comparison of Classes ......................................... 144

5 Assisting Search 147
5.1 Transformations ................................................... 147
5.2 Lookahead ......................................................... 147
  5.2.1 Depth-first Lookahead .................................... 147
  5.2.2 Breadth-first Lookahead ................................ 147
  5.2.3 Function-complete Lookahead ............................ 147
5.3 Learning: Conflict Resolution .................................. 147
5.4 Backjumping ...................................................... 147
5.5 Adding Uninferred Constraints ................................ 147
  5.5.1 Autarkies .................................................... 147
  5.5.2 Non-monotonic constraints ............................... 147
  5.5.3 Unsafe constraints: tunnels .............................. 147

6 Lower and Upper Bounds 149
6.1 Upper Bounds on Complexity .................................. 149
6.2 Exponential Lower Bounds on Complexity ..................... 150
6.3 Extended Resolution vs. Resolution .......................... 151

7 Probabilistic Analysis 153
7.1 Myopic algorithms for satisfiable formulas .................. 153
7.2 Non-myopic algorithms for satisfiable formulas ............. 163
7.3 Lower bounds on the satisfiability threshold ................ 165
7.4 Verifying unsatisfiability: resolution ....................... 166
7.5 Verifying unsatisfiability: a spectral analysis ............. 170
7.6 Polynomial time solvable classes ............................. 172
7.7 Randomized algorithms: upper bounds ....................... 176
7.8 Survey Propagation .............................................. 180
A Glossary 199
Chapter 1

Applications

In this chapter we present a sample of real-world problems that may be viewed as or transformed to monotonic or non-monotonic logic problems. We are primarily interested in exploring the thinking process required for setting up logic problems, and gauging the complexity of the resulting systems we will have to deal with. Since it is infeasible to meet this objective and thoroughly discuss a large number of known applications, we choose to focus deeply on a small but varied and interesting collection of problems.

Some terms and concepts which we believe are familiar to most readers are not defined nor discussed in this chapter while less familiar ones are given somewhat detailed treatment. We believe this will help the reader retain a high level of interest throughout. The reader is invited to check the glossary or Chapter 2 when encountering unknown terminology or concepts.

1.1 Consistency Analysis in Scenario Projects

This application, taken from the area of scenario management [10, 49, 63], is contributed by Feldmann and Sensen [52] of Burkhard Monien’s PC² group at Universität Paderborn, Germany. A scenario consists of (i) a progression of events from a known base situation to a possible terminal future situation, and (ii) a means to evaluate its likelihood. Scenarios are used by managers and politicians to strategically plan the use of resources needed for solutions to environmental, social, economic and other such problems.

A systematic approach to scenario management due to Gausemeier, Fink, and Schlake [61] involves the realization of scenario projects with the following properties:

1. There is a set $S$ of key factors. Let the number of key factors be $n$.

2. For each key factor $s_i \in S$ there is a set $D_i = \{d_{i,1}, d_{i,2}, \ldots, d_{i,m_i}\}$ of $m_i$ possible future developments. In the language of data bases, key
factors are attributes and future developments are attribute values.

3. For all $1 \leq i \leq n$, $1 \leq k \leq m_i$, denote by $(s_i, d_{i,k})$ a feasible projection of development $d_{i,k} \in D_i$ from key factor $s_i$. For each pair of projections $(s_i, d_{i,k}), (s_j, d_{j,l})$ a consistency value, usually an integer ranging from 0 to 4, is defined. A consistency value of 0 typically means two projections are completely inconsistent, a value of 4 typically means the two projections support each other strongly, and the other values account for intermediate levels of support. Consistency values may be organized in a $\sum_{i=1}^{n} m_i \times \sum_{i=1}^{n} m_i$ matrix with rows and columns indexed on projections.

4. Projections for all key factors may be bundled into a vector $x = (x_{s_1}, \ldots, x_{s_n})$ where $x_{s_i}$ is a future development of key factor $s_i$, $i = 1, 2, \ldots, n$. In the language of data bases, a bundle is a tuple which describes an assignment of values to each attribute.

5. The consistency of bundle $x$ is the sum of the consistency values of all pairs $(s_i, x_{s_i}), (s_j, x_{s_j})$ of projections represented by $x$ if no pair has consistency value of 0, and is 0 otherwise.

Bundles with greatest (positive) consistency are determined and clustered. Each cluster is a scenario.

To illustrate, we simplify an example from [52] which is intended to develop likely scenarios for the German school system over the next 20 years. It was felt by experts that 20 key factors are needed for such forecasts; to keep the example small, we show only the first 10. The first five key factors and their associated future developments are: $[s_1]$ continuing education ($[d_{1,1}]$ lifelong learning); $[s_2]$ importance of education ($[d_{2,1}]$ important, $[d_{2,2}]$ unimportant); $[s_3]$ methods of learning ($[d_{3,1}]$ distance learning, $[d_{3,2}]$ classroom learning); $[s_4]$ organization and policies of universities ($[d_{4,1}]$ enrollment selectivity, $[d_{4,2}]$ semester schedules); $[s_5]$ adequacy of trained people ($[d_{5,1}]$ sufficiently many, $[d_{5,2}]$ not enough).

The table in Figure 1.1, called a consistency matrix, shows the consistency values for all possible pairs of future developments. The $s_i, s_j$ cell of the matrix shows the consistency values of pairs $\{(s_i, d_{i,x}), (s_j, d_{j,y}) : 1 \leq x \leq m_i, 1 \leq y \leq m_j\}$, with all pairs which include $(s_i, d_{i,x})$ on the $x^{th}$ row of the cell. For example, $|D_5| = 2$ and $|D_2| = 2$ so there are 4 numbers in cell $s_5, s_2$ and the consistency value of $(s_5, d_{5,2}), (s_2, d_{2,2})$ is the bottom right number of that cell. That number is 0 because experts have decided the combination of there being too few trained people ($d_{5,2}$) at the same time education is considered unimportant ($d_{2,2}$) is unlikely.

It is relatively easy to compute the consistency of a given bundle from this table. One possible bundle of future developments is $\{(s_i, d_{i,1}) : 1 \leq i \leq 10\}$. Since the consistency value of $\{(s_5, d_{5,1}), (s_2, d_{2,1})\}$ is 0, this bundle is
inconsistent. Another possible bundle is
\{(s_1, d_{1.1}), (s_2, d_{2.1}), (s_3, d_{3.1}), (s_4, d_{4.1}), (s_5, d_{5.2}), \\
(s_6, d_{6.3}), (s_7, d_{7.1}), (s_8, d_{8.2}), (s_9, d_{9.1}), (s_{10}, d_{10.2})\}.

Its consistency value is 120 (the sum of the 45 relevant consistency values from the table).

Finding good scenarios from a given consistency matrix requires efficient answers to the following questions:

1. Does a consistent bundle exist?
2. How many consistent bundles exist?
3. What is the bundle having the greatest consistency?

Finding answers to these questions is \(\mathcal{NP}\)-hard [52]. But transforming to CNF formulas,\(^1\) in some cases with weights on proposition letters, and solv-

\(^1\)A CNF formula is a conjunction of disjunctions of positive and negative literals. See Formula, CNF in the glossary for more information.
Clause of $\psi_{S,D}$  

<table>
<thead>
<tr>
<th>Clause</th>
<th>Subscript Range</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neg v_{i,j} \lor \neg v_{i,k}$</td>
<td>$1 \leq i \leq n, 1 \leq j &lt; k \leq m_i$</td>
<td>$\leq 1$ development/key factor</td>
</tr>
<tr>
<td>$v_{i,1} \lor \ldots \lor v_{i,m_i}$</td>
<td>$1 \leq i \leq n$</td>
<td>$\geq 1$ development/key factor</td>
</tr>
<tr>
<td>$\neg v_{i,k} \lor \neg v_{j,l}$</td>
<td>$i, j, k, l : C_{i,k,j,l} = 0$</td>
<td>Consistent developments only</td>
</tr>
</tbody>
</table>

Table 1.1: Formula to determine existence of a consistent bundle.

ing some variant of Satisfiability, is sometimes a reasonable way to tackle such problems. In this context we have the opportunity to use our vast knowledge of SAT structures and analysis to apply an algorithm that has a reasonable chance of solving the consistency problems efficiently. In fact, we next show how to construct representative formulas so that, often, a large subset of clauses is polynomial time solvable and, because of this, the whole formula is often relatively easy to solve.

Consider, first, the question whether a consistent bundle exists for a given consistency matrix of $n$ key factors with $m_i$ projections for factor $i$, $1 \leq i \leq n$. Let $C_{i,k,j,l}$ denote the consistency of the pair $(s_i, d_{i,k})(s_j, d_{j,l})$ and let $D = \bigcup_{i=1}^{n} D_i$. For each future development $d_{i,j}$, we define variable $v_{i,j}$ which is intended to take the value 1 if and only if $d_{i,j}$ is a future development for key attribute $s_i$. The CNF formula $\psi_{S,D}$ with the clauses described in Table 1.1 then “says” that there is a consistent bundle. That is, the formula is satisfiable if and only if there is a consistent bundle.

The second question we consider is how many consistent bundles exist for a given consistency matrix? This is the same as asking how many satisfying truth assignments there are for $\psi_{S,D}$. A simple inclusion-exclusion algorithm exhibits very good performance for some actual problems of this sort [52].

The third question is which bundle has the greatest consistency value? This question can be transformed to an instance of the Variable Weighted Satisfiability problem, a variant of the Satisfiability problem in which numerical weights are assigned to the proposition letters and the weight of an assignment is the sum of the weights of the variables in the assignment of value 1. The transformed formula consists of $\psi_{S,D}$ plus some additional clauses as follows. For each pair $(s_i, d_{i,k})(s_j, d_{j,l})$, $i \neq j$, of projections such that $C_{i,k,j,l} > 0$, create a new Boolean variable $p_{i,k,j,l}$ of weight $C_{i,k,j,l}$ and add the following subexpression to $\psi_{S,D}$:

$$
(\neg v_{i,k} \lor \neg v_{j,l} \lor p_{i,k,j,l}) \land (v_{i,k} \lor \neg p_{i,k,j,l}) \land (v_{j,l} \lor \neg p_{i,k,j,l}).
$$

Observe that a satisfying assignment requires $p_{i,j,k,l}$ to have value 1 if and only if $v_{i,k}$ and $v_{j,l}$ both have value 1, i.e., if and only if the consistency value of $\{(s_i, d_{i,k}),(s_j, d_{j,l})\}$ is included in the calculation of the consistency value of the bundle.
1.2 TESTING OF VLSI CIRCUITS

If weight 0 is assigned to all variables other than the \( p_{i,k,j,l} \)'s, a maximum weight solution specifies a maximum consistency bundle. We point out that, although the number of clauses added to \( \psi^{S,D} \) might be significant compared to the number of clauses originally in \( \psi^{S,D} \), the total number of clauses will be linearly related to the size of the consistency matrix. Moreover, the set of additional clauses is a Horn subformula\(^2\). A maximum weight solution can be found by means of a branch-and-bound algorithm such as that discussed in Section 3.7.

1.2 Testing of VLSI Circuits

A classic application is the design of test vectors for VLSI circuits. At the specification level, a combinational VLSI circuit is regarded to be a function mapping \( n \) 0-1 inputs to \( m \) 0-1 outputs\(^3\). At the design level, the interconnection of numerous 0-1 logic gates are required to implement the function. Each connection entails an actual interconnect point that can fail during or soon after manufacture. Failure of an interconnect point usually causes the point to become “stuck-at” value 0 or value 1.

Traditionally, VLSI circuit testing includes testing all interconnect points for stuck-at faults. This task is made difficult because interconnect points are encased in plastic and are therefore inaccessible directly. The solution is to apply an input pattern to the circuit which excites the point under test and sensitizes a path through the circuit from the test point to some output so that the correct value at the test point can be determined at the output.

We illustrate how such an input pattern and output is found for one internal point of a 1-bit full adder: an elementary but ubiquitous functional hardware block that is depicted in Figure 1.2. For the sake of discussion, we assume a given circuit will have at most one stuck-at failure. The 1-bit full adder uses logic gates that behave according to the truth tables in Figure 1.3 where \( a \) and \( b \) are gate inputs and \( e \) is a gate output. Suppose none of the interconnect points enclosed by the dashed line of Figure 1.2 are directly accessible and suppose it is desired to develop a test pattern to determine whether point \( w \) is stuck at 0. Then inputs must be set to give point \( w \) the value 1. This is accomplished by means of the Boolean expression \( \psi_1 = (A \land B) \). The value of point \( w \) can only be observed at output \( Y \). But, this requires point \( v \) be set to 0. This can be accomplished if either the value of \( C \) is 0 or \( u \) is 0, and \( u \) is 0 if and only if \( A \) and \( B \) have the same value. Therefore, the Boolean expression representing sensitization of a path from \( w \) to \( Y \) is \( \psi_2 = (\neg C \lor (A \land B) \lor (\neg A \land \neg B)) \). The conjunction \( \psi_1 \land \psi_2 \) is the CNF expression \( (A \land B) \). This expression is satisfied if and only if \( A \) and \( B \) are set to 1. Such an input will cause output \( Y \) to have

\(^2\)Horn formulas are solved efficiently (see Formula, Horn in the glossary)

\(^3\)Actual circuit voltage levels are abstracted to the values 0 and 1
Figure 1.2: A 1-bit full adder circuit.

Figure 1.3: Truth tables for logic elements of a 1-bit full adder.

value 1 if $w$ is not stuck at 0 and value 0 if $w$ is stuck at 0, assuming no other interconnect point is stuck at some value.

Test patterns must be generated for all internal interconnect points of a VLSI circuit. There could be millions of these and the corresponding expressions could be considerably more complex than that of the example above. Moreover, testing is complicated by the fact that most circuits are not combinational: that is, they contain feedback loops. This last case is mitigated by adding circuitry for testing purposes only. The magnitude of the testing problem, although seemingly daunting, is not great enough to cause major concern at this time because it seems that SAT problems arising in this domain are usually easy. Thus, at the moment, the VLSI testing problem is considered to be under control. However, this may change in the near future since the number of internal points in a dense circuit is expected to take major leaps upward.
1.3 Diagnosis of Circuit Faults

A natural extension of the test design discussed in Section 1.2 is finding a way to automate the diagnosis of, say, bad chips, starting with descriptions of their bad outputs. How can one reason backwards to identify likely causes of the malfunctioning? Also, given knowledge that some components are more likely to fail than others, how can the diagnosis system be tailored to suggest the most likely causes first? In this section we discuss the first of these questions.

The first step is to write the Boolean expression representing both the normal and abnormal behavior of the analyzed circuit. This is illustrated using the circuit of Figure 1.2. The expression is assembled in stages, one for each gate, starting with gate 3 of Figure 1.2, which is an and-gate. The behavior of a correctly functioning and-gate is specified in the right-hand truth table in Figure 1.3. The following formula expresses the truth table:

\[(a \land b \land c) \lor (\neg a \land \neg b \land c) \lor (\neg a \land b \land \neg c) \lor (\neg a \land b \land \neg c)\]

If gate 3 is possibly stuck at 0, its functionality can be described by adding variable \(Ab_3\) (for “gate 3 is abnormal”) and substituting \(A, B,\) and \(w\) for \(a, \bar{b},\) and \(c,\) to get the following abnormality expression:

\[(A \land B \land w \land \neg Ab_3) \lor (\neg A \land \neg B \land w \land \neg Ab_3) \lor (\neg A \land \neg B \land \neg w \land \neg Ab_3) \lor (\neg w \land \neg Ab_3)\]

which has value 1 if and only if gate 3 is functioning normally for inputs \(A\) and \(B\) or it is functioning abnormally and \(w\) is stuck at 0. The extra variable may be regarded as a switch which allows toggling between abnormal and normal states for gate 3. Similarly, switches \(Ab_1, Ab_2, Ab_4, Ab_5\) may be added for all the other gates in the circuit and corresponding expressions may be constructed using those switches. Then the set

\[\Delta = \{Ab_1, Ab_2, Ab_3, Ab_4, Ab_5\}\]

represents all possible explanations for stuck-at-0 malfunctions. The list of assignments to the variables of \(\Delta\) which satisfy the collection of abnormality expressions, given particular inputs and observations, determines all possible combinations of stuck-at-0 failures in the circuit. The next task is to choose the most likely failure combination from the list. This requires some assumptions which are motivated by the following examples.

Suppose that during some test inputs are set to \(A = 0\) and \(B, C = 1\) and observed output values are \(Y = 0\) and \(X = 1\) whereas \(Y = 1\) and \(X = 0\) are the normal outputs for those inputs. We can reason backwards to try to determine which gates are stuck at 0. Gate 4 cannot be stuck at 0 since its output is 1. We suppose gate 4 is working correctly. Since the only gate that gate 4 depends on is gate 1, that gate must be stuck. We cannot tell
Table 1.2: Possible stuck-at 0 failures of 1-bit adder gates (see Figure 1.2) assuming given inputs are $A = 0$ and $B, C = 1$ and observed outputs are $X, Y = 0$. This is the list of assignments to $\Delta$ which satisfy the abnormality predicates for the 1-bit adder. A ‘1’ in the column for $Ab_i$ means gate $i$ is stuck at 0. The symbol ‘*’ means either ‘0’ or ‘1’.

<table>
<thead>
<tr>
<th>IDs</th>
<th>$Ab_1$</th>
<th>$Ab_2$</th>
<th>$Ab_3$</th>
<th>$Ab_4$</th>
<th>$Ab_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-16</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>17-24</td>
<td>*</td>
<td>1</td>
<td>*</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>25-26</td>
<td>1</td>
<td>0</td>
<td>*</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Whether gates 2,3,5 are functioning; we would normally assume that they are functioning correctly until we have evidence to the contrary (just as we assumed, in designing tests in the previous section, that at most one gate is malfunctioning). Thus the natural diagnosis is gate 1 is defective (only $Ab_1$ has value 1).

Alternatively, suppose under the same inputs we observe $X, Y = 0$. Possibly, gate 5 and gate 4 are malfunctioning. If so, all other combinations of gate outputs will lead to the same observable outputs. If gate 5 is defective but gate 4 is good, then $u = 1$ so gate 1 is good, and any possible combinations of $w$ and $v$ lead to the same observable outputs. If gate 5 is good and gate 4 is defective the bad $Y$ value may be caused by a defective gate 2. In that case gate 1 and gate 3 conditions do not affect the observed outputs. But, if gate 2 is not defective, the culprit must be gate 1. If gate 4 and gate 5 are good then $u = 1$ so gate 2 is defective. Nothing can be determined about the condition of gate 3 through this test. The results of this paragraph lead to 26 abnormal $\Delta$ values that “witness” the observed outputs: these are summarized in Table 1.2, grouped into three cases. In the first two of these cases the minimum number of gates stuck at 0 is 1.

As before, we normally assume the set of malfunctioning gates is as small as possible, so we consider only those two diagnoses: that is, either (i) gate 5, or (ii) gate 2 is defective. In general, it is argued, commonsense leads us to consider only minimal sets of abnormalities: sets, like gate 2 above, where no proper subset is consistent with the observations. This is Reiter’s Principle of Parsimony[112]:

A diagnosis is a conjecture that some minimal set of components are faulty.

This sort of inference is called non-monotonic because we inferred, above, that gate 3 was functioning correctly, because there was no evidence it was not. Later evidence may cause us to withdraw that inference.
Yet a further feature of non-monotonic logic may be figured into such systems; the following illustrates the idea. Suppose it is known that one component, say gate 5, is the least likely to fail. Then, if there are any diagnoses in which gate 5 does not fail, it will report only such diagnoses. If gate 5 fails in all diagnoses, then it will report all the diagnoses. Essentially, this reflects a kind of preference relationship among diagnoses.

There is now software which automates this diagnosis process (e.g., [67]). Although worst-case performance of such systems is provably bad, such a system can be useful in many circumstances. Unfortunately, implementations of nonmonotonic inference are new enough that there is not yet a body of standard benchmark examples.

### 1.4 Functional Verification of Hardware Design

Proving correctness of the design of a given block of hardware has become a major concern due to the complexity of present day hardware systems and the economics of product delivery time. Prototyping is no longer feasible since it takes too much time and fabrication costs are high. Breadboarding no longer gives reliable results because of the electrical differences between integrated circuits and discrete components. Simulation based methodologies are generally fast but do not completely validate a design since there are many cases left unconsidered. Formal verification methods can give better results, where applicable, since they will catch design errors that may go undetected by a simulation.

Formal verification methods are used to check correctness by detecting errors in translation between abstract levels of the design hierarchy. Design hierarchies are used because it is impractical to design a VLSI circuit involving millions of components at the substrate, or lowest, level of abstraction. Instead, it is more reasonable to design at the specification or highest level of abstraction and use software tools to translate the design, through some intermediate stages such as the logic-gate level, to the substrate level. The functionality between a pair of levels may be compared. In this case, the more abstract level of the pair is said to be the specification and the other level is the implementation level of the pair. If functionality is equivalent between all adjacent pairs of levels, the design is said to be verified.

Determining functional equivalence between levels amounts to proving a theorem of the form the implementation $I$ realizes the specification $S$ in a particular, suitable formal proof system. For illustration purposes only, consider the 1-bit full adder of Figure 1.2. Inputs $A$ and $B$ represent a particular bit position of two different binary addends. Input $C$ is the carry due to the addition at the next lower valued bit position. Output $X$ is the value of the same bit position of the sum and output $Y$ is the carry to the next higher valued bit position. The output $X$ must have value 1 if and only if all inputs have value 1 or exactly one input has value 1. The output $Y$ has
value 1 if and only if at least two out of three inputs have value 1. Therefore, the following simple Boolean expression offers a reasonable specification of any 1-bit full adder:

\[(X \iff (A \land \neg B \land \neg C) \lor (\neg A \land B \land \neg C) \lor (\neg A \land \neg B \land C) \land (A \land B) \lor (A \land C) \lor (B \land C))).\]

A proposed implementation of this specification is given in the dotted region of Figure 1.2. Its behavior may be described by a Boolean expression that equates each gate output to the corresponding logical function applied to its inputs. The following is such an expression:

\[
(u \iff (A \land \neg B) \lor (\neg A \land B)) \land \\
(v \iff u \land C) \land \\
(w \iff A \land B) \land \\
(X \iff (u \land \neg C) \lor (\neg u \land C)) \land \\
(Y \iff w \lor v).
\]

Designate these formulas \(\psi_S(A, B, C, X, Y)\) and \(\psi_I(A, B, C, X, Y, u, v, w)\), respectively. The adder correctly implements the specification if, for all possible inputs \(A, B, C \in \{0, 1\}\), the output of the adder matches the specified output. For any individual inputs, \(A, B, C\), that entails checking whether \(\psi_I(A, B, C, X, Y, u, v, w)\) has value 1 for the appropriate \(u, v, w\):

\[\psi_S(A, B, C, X, Y) \iff \exists u, v, w \ \psi_I(A, B, C, X, Y, u, v, w),\]

where the quantification \(\exists u, v, w\) is over Boolean values \(u, v, w\). For specific \(A, B, C\), the problem is a satisfiability problem: can we find values for \(u, v, w\) which make the formula have value 1? (Of course, it’s an easy satisfiability problem in this case.) Thus, the question of whether the adder correctly implements the specification for all 32 possible input sequences is answered using the formula:

\[\forall A, B, C, X, Y(\psi_S(A, B, C, X, Y) \iff \exists u, v, w \ \psi_I(A, B, C, X, Y, u, v, w)),\]

which has a second level of Boolean quantification. Such formulas are called quantified Boolean formulas.

The example of the 1-bit full adder shows how combinational circuits can be verified. A characteristic of combinational circuits is that output behavior is strictly a function of the current values of inputs and does not depend on past history. However, circuits frequently contain components, such as registers, which exhibit some form of time dependency. Such effects may be modeled by some form of propositional temporal logic.
1.4 FUNCTIONAL VERIFICATION OF HARDWARE DESIGN

Table 1.3: The operators of temporal logic

<table>
<thead>
<tr>
<th>Op. name</th>
<th>$(S, s_i) \models$</th>
<th>if and only if</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$ (p a variable)</td>
<td>$s_i(p) = 1$.</td>
<td>$\neg \psi_1$</td>
</tr>
<tr>
<td>$\neg$</td>
<td>$(S, s_i) \not\models \psi_1$</td>
<td>$\psi_1 \land \psi_2$</td>
</tr>
<tr>
<td>$\land$</td>
<td>$(S, s_i) \models \psi_1$ and $(S, s_i) \models \psi_2$</td>
<td>$\psi_1 \lor \psi_2$</td>
</tr>
<tr>
<td>$\lor$</td>
<td>$(S, s_i) \models \psi_1$ or $(S, s_i) \models \psi_2$</td>
<td>$\Box \psi_1$</td>
</tr>
<tr>
<td>$\Box$</td>
<td>$(S, s_j) \models \psi_1$ for all states $s_j, j \geq i$.</td>
<td>$\Diamond \psi_1$</td>
</tr>
<tr>
<td>$\Diamond$</td>
<td>$(S, s_j) \models \psi_1$ for some state $s_j, j \geq i$.</td>
<td>$\psi_1$</td>
</tr>
<tr>
<td>$\psi_1$</td>
<td>$(S, s_{i+1}) \models \psi_1$.</td>
<td>$\psi_1 \mathcal{U} \psi_2$</td>
</tr>
<tr>
<td>$\psi_1 \mathcal{U} \psi_2$</td>
<td>$(S, s_i), (S, s_{i+1}), \ldots, (S, s_{j-1}) \models \psi_1$, and $(S, s_j) \models \psi_2$.</td>
<td></td>
</tr>
</tbody>
</table>

Systems of temporal logic have been applied successfully to the verification of some sequential circuits including microprocessors. One may think of a sequential circuit as possessing one of a finite number of valid states at any one time. The current state of such a circuit embodies the complete electrical signal history of the circuit beginning with some distinguished initial state. A change in the electrical properties of a sequential circuit at a particular moment in time is represented as a fully deterministic movement from one state to another based on the current state and a change in some subset of input values only. Such a change in state is accompanied by a change in output values.

A description of several temporal logics can be found in [133, ?]. For illustrative purposes we discuss one of these, the Linear Time Temporal Logic (LTTL), here. LTTL formulas take value 1 with respect to an infinite sequence of states $S = \{s_0, s_1, s_2, \ldots\}$. States of $S$ obey the following: state $s_0$ is a legal initial state of the system; state $s_i$ is a legal state of the system at time step $i$; every pair $s_i, s_{i+1}$ must be a legal pair of states. Legal pairs of states are forced by some of the components of the formula itself (the latch example at the end of this section illustrates this). Each state is just an interpretation of Boolean variables.

LTTL is an extension of the propositional calculus that adds one binary and three unary temporal operators which are described below and whose semantics are outlined in Table 1.3 along with the standard propositional operators $\neg$, $\land$, and $\lor$ (the definition of $(S, s_i) \models$ is given below). The syntax of LTTL formulas is the same as for propositional logic except for the additional operators.
Let $\psi$ be a LTL expression that includes a component representing satisfaction of some desired property in a given circuit. An example of a property for a potential JK flip-flop might be that it is eventually possible to have a value of 1 for output $Q$ while input $J$ has value 1 which has a corresponding formula $\Diamond(J \land Q)$. If $\psi$ has value 1 for state $s_i$ in $S$ we say $(S, s_i) \models \psi$.

We say $S \models \psi$ if and only if $(S, s_0) \models \psi$.

Finally, two LTL formulas $\psi_1$ and $\psi_2$ are equivalent if, for all sequences $S$,

$S \models \psi_1$ if and only if $S \models \psi_2$.

Our example concerns a hardware device of two inputs and one output called a set-reset latch. The electrical behavior of such a device is depicted in Figure 1.4 as a set of six waveforms which show the value of the output $q$ in terms of the history of the values of inputs $r$ and $s$. For example, consider waveform (a) in the Figure. This shows the value of the output $q$, as a function of time, if initially $q$, $r$ and $s$ have value 0 and then $s$ is pulsed or raised to value 1 then some time later dropped to value 0. The waveform shows the value of $q$ rises to 1 some time after $s$ does and stays at value 1 after the value of $s$ drops. The waveform (a) also shows that if $q$ has value 1 and $r$ and $s$ have value 0 and then $r$ is pulsed, the value of

---

**Figure 1.4**: A set-reset latch and complete description of signal behavior. Inputs are $s$ and $r$, output is $q$. The horizontal axis represents time. The vertical axis represents signal value for both inputs and output. Values of all signals are assumed to be either 0 (low) or 1 (high) at any particular time. Two rows for $q$ values are used to differentiate between the cases where the initial value of $q$ is 0 or 1.
1.4 FUNCTIONAL VERIFICATION OF HARDWARE DESIGN

Expressions | Comments
---|---
□¬(s ∧ r) | No two inputs have value 1 simultaneously.
□((s ∧ ¬q) → ((s U q) ∨ □s)) | Input \( s \) cannot change if \( s \) is 1 and \( q \) is 0.
□((r ∧ q) → ((r U¬q) ∨ □r)) | Input \( r \) cannot change if \( r \) is 1 and \( q \) is 1.
□(s → □q) | If \( s \) is 1, \( q \) will eventually be 1.
□(r → □¬q) | If \( r \) is 1, \( q \) will eventually be 0.
□((¬q → ((¬q U s) ∨ □¬q))) | Output \( q \) rises to 1 only if \( s \) becomes 1.
□((q → ((q U r) ∨ □q))) | Output \( q \) drops to 0 only if \( r \) becomes 1.

Table 1.4: Temporal logic formula for a set-reset latch

\( q \) drops to 0. Observe the two cases where (i) \( r \) and \( s \) have value 1 at the same moment and (ii) \( q \) changes value after \( s \) or \( r \) pulses are not allowed. The six waveforms are enough to specify the behavior of the latch because the device is simple enough that only “recent” history matters.

The specification of this behavior is given by the LTTL formula of Table 1.4. Observe that the first three expressions of Table 1.4 represent assumptions needed for the latch to work correctly and do not necessarily reflect requirements that can be realized within the circuitry of the latch itself. Care must be taken to insure that the circuitry in which a latch is placed meets those requirements.

Latch states are triples representing values of \( s \), \( r \), and \( q \), respectively. Some examples, corresponding to state sequences depicted by waveforms (a) – (f) in Figure 1.4, that satisfy the formula of Table 1.4 are:

\[ S_a: \quad (\langle 000 \rangle, \langle 100 \rangle, \langle 101 \rangle, \langle 001 \rangle, \langle 011 \rangle, \langle 010 \rangle, \langle 000 \rangle, \ldots ) \]
\[ S_b: \quad (\langle 000 \rangle, \langle 100 \rangle, \langle 101 \rangle, \langle 001 \rangle, \langle 101 \rangle, \langle 001 \rangle, \ldots ) \]
\[ S_c: \quad (\langle 000 \rangle, \langle 010 \rangle, \langle 000 \rangle, \langle 010 \rangle, \langle 000 \rangle, \ldots ) \]
\[ S_d: \quad (\langle 001 \rangle, \langle 101 \rangle, \langle 001 \rangle, \langle 011 \rangle, \langle 010 \rangle, \langle 000 \rangle, \ldots ) \]
\[ S_e: \quad (\langle 001 \rangle, \langle 101 \rangle, \langle 001 \rangle, \langle 101 \rangle, \langle 001 \rangle, \ldots ) \]
\[ S_f: \quad (\langle 001 \rangle, \langle 011 \rangle, \langle 010 \rangle, \langle 000 \rangle, \langle 010 \rangle, \langle 000 \rangle, \ldots ) \]

Clearly, infinitely many sequences satisfy the formula of Table 1.4, so the problem of verifying functionality for sequential circuits appears daunting. However, by means of careful algorithm design, it is sometimes possible to produce such verifications and successes have been reported. In addition, other successful temporal logic systems such as Computation Tree Logic and Interval Temporal Logic have been introduced, along with algorithms for proving theorems in these logics. The reader is referred to [133], Chapter 6 for details and citations. We defer further discussion to Section ??.
1.5 Bounded Model Checking

In Section 1.4 we were interested in solving verification problems of the form
\[ S \models \psi_1 \equiv S \models \psi_2. \] If we wish instead to determine whether there exists an
\( S \) such that \( S \models \psi \) we can trade temporal operators for Boolean variables,
express the sentence as a propositional formula, and apply a SAT solver.
The propositional formula must have the following parts:

1. Components which force the property or properties of the time dependent
   expression to hold.

2. Components which establish the starting state.

3. Components which force legal state transitions to occur.

In order for the Boolean expression to remain of reasonable size it is generally
necessary to bound the number of time steps in which the time-dependent
expression is to be verified. Thus the name \textit{Bounded Model Checking}.

As an example, consider a simple 2-bit counter whose outputs are
represented by variables \( v_1 \) (LSB) and \( v_2 \) (MSB). Introduce variables \( v^i_1 \) and \( v^i_2 \) whose values are intended to be the same as those of variables \( v_1 \) and \( v_2 \), respectively, on the \( i \)th time step. Suppose the starting state is the case
where both \( v^0_1 \) and \( v^0_2 \) have value 0. The transition relation is

\[
\begin{array}{c|c|c}
\text{Current Output} & \text{Next Output} \\
00 & 01 \\
01 & 10 \\
10 & 11 \\
11 & 00 \\
\end{array}
\]

the \( i \)th line of which can be expressed as the following Boolean function:
\[ (v^{i+1}_1 \iff \neg v^i_1) \land (v^{i+1}_2 \iff v^i_1 \oplus v^i_2). \]

Suppose the time-dependent expression to be proved is:
\[ \text{Can the two-bit counter reach a count of 11 in exactly three time steps?} \]

Assemble the propositional formula having value 1 if and only if the above
query holds as the conjunction of the following three parts:

1. **Force the property to hold:**
   \[ (\neg (v^0_1 \land v^0_2) \land \neg (v^1_1 \land v^1_2) \land \neg (v^2_1 \land v^2_2) \land (v^3_1 \land v^3_2)) \]

2. **Express the starting state:**
   \[ (\neg v^0_1 \land \neg v^0_2) \]
3. Force legal transitions (repetitions of the transition relation):

\[
\begin{align*}
(v_1^1 & \iff \neg v_1^0) \land (v_2^1 & \iff v_1^0 \oplus v_2^0) \land \\
(v_1^2 & \iff \neg v_1^1) \land (v_2^2 & \iff v_1^1 \oplus v_2^1) \land \\
(v_1^3 & \iff \neg v_1^2) \land (v_2^3 & \iff v_1^2 \oplus v_2^2)
\end{align*}
\]

Since \((a \iff b)\) is equivalent to \((a \lor \neg b) \land (\neg a \lor b)\), the last expression can be directly turned into a CNF expression. Therefore, the entire formula can be turned into a CNF expression and solved with an off-the-shelf SAT solver.

The reader may check that the following assignment satisfies the above expressions:

\[
v_1^0 = 0, v_2^0 = 0, v_1^1 = 1, v_2^1 = 0, v_1^2 = 0, v_2^2 = 1, v_1^3 = 1, v_2^3 = 1.
\]

It may also be verified that no other assignment of values to \(v_i^1\) and \(v_i^2\), \(0 \leq i \leq 3\), satisfies the above expressions. Information on the use and success of Bounded Model Checking may be found in [16, 34].

### 1.6 Combinational Equivalence Checking

The power of Bounded Model Checking is not needed to solve the Combinational Equivalence Checking (CEC) problem which is to verify that two given combinational circuit implementations are functionally equivalent. CEC problems are easier, in general, because they carry no time dependency and there are no feedback loops in combinational circuits. It has recently been discovered that CEC can be solved very efficiently, in general, \([86, 88, 89]\) by incrementally building a single-output And/Inverter Graph (AIG), representing the mitre \([23]\) of both input circuits, and checking whether the output has value 0 for all combinations of input values. The AIG is built one vertex at a time, working from input to output. Vertices are merged if they are found to be “functionally equivalent.” Vertices can be found functionally equivalent in two ways: 1) candidates are determined by random simulation \([23, 87]\) and then checked by a Satisfiability solver for functional equivalence; 2) candidates are hashed to the same location in the data structure representing vertices of the AIG (for the address of a vertex to represent function, it must depend solely on the opposite endpoints of the vertex’s incident edges and, therefore, an address change typically takes place on a merge). AIG construction continues until all vertices have been placed into the AIG and no merging is possible. The technique exploits the fact that checking the equivalence of two “topologically similar” circuits is relatively easy \([66]\) and avoids testing all possible input-output combinations, which is \(\mathcal{CoNP}\)-hard.
(a): “and” gate (b): “or” gate (c): “xor” gate

(d): “add” circuit of Figure 1.2

Figure 1.5: Examples of AIGs. Edges with white circles are negated, and are not negated otherwise.

An AIG is a directed acyclic graph where all “gate” vertices have indegree 2, all “input” vertices have in-degree 0, all “output” vertices have in-degree 1, and edges are labeled as either “negated” or “not negated.” Any combinational circuit can be equivalently implemented as a circuit involving only 2-input “and” gates and “not” gates. Such a circuit has an AIG representation: “gate” vertices correspond directly to the “and” gates, negated edges correspond directly to the “not” gates, and inputs and outputs represent themselves directly. Negated edges are typically labeled by overlaying a white circle on the edge which distinguishes them from non-negated edges which are unlabeled.

In CEC the AIG is incrementally developed from gate level representations. For example, Figure 1.5(a) shows the AIG for an “and” gate, Figure 1.5(b) shows the AIG for an “or” gate, Figure 1.5(c) shows the AIG for “exclusive-or” and Figure 1.5(d) shows the AIG for the adder circuit of Figure 1.2. Vertices may take 0-1 values. Values are assigned independently to input vertices. The value of a gate vertex (a dependent vertex) is the product $x_1x_2$ where $x_i$ is either the value of the non-arrow side endpoint of incoming edge $i$, $1 \leq i \leq 2$, if the edge is not negated or 1 minus the value of the endpoint if it is negated.

CEC begins with the construction of the AIG for one of the circuits as exemplified in Figure 1.6(a) which shows an AIG and a circuit it is to be compared against. Working from input to output, a node of the circuit is determined to be functionally equivalent to an AIG vertex and is merged with the vertex. The output lines from the merged node become AIG edges.
outgoing from the vertex involved in the merge. Figure 1.6(b) shows the first two nodes of the circuit being merged with the AIG and Figure 1.6(c) shows the completed AIG.

CEC proceeds with the application of many random input vectors to the input vertices of the AIG for the purpose of partitioning the vertices into potential equivalence classes (two vertices are in the same equivalence class if they take the same value under all random input vectors). In the case of Figure 1.6(c), suppose the potential equivalence classes are \{1, 2\}, \{3, 4\}, \{5, 6\}.

The next step is to use a Satisfiability solver to verify that the equivalences actually hold. Those that do are merged, resulting in a smaller AIG. The cycle repeats until merging is no longer possible. If the output vertices hash to the same address, the circuits are equivalent. Alternatively, the circuits are equivalent if the AIG is augmented by adding a vertex corresponding to an “xor” gate with incident edges connecting to the two output vertices of the AIG and the resulting graph represents an unsatisfiable formula, determined by using a SAT solver.

We have shown CEC being used to check the equivalence of two combinatorial circuits but it can also be used for checking a specification against a circuit implementation or for reverse engineering a circuit.

### 1.7 Transformations to Satisfiability

It is routine in the operations research community to transform a given optimization problem into another, solve the new problem, and use the solution to construct a solution or an approximately optimal solution for the given problem. Usual targets of transformations are Linear Programming and Network Flows. In some cases, where the given problem is \(\mathcal{NP}\)-complete, it may be more efficient to obtain a solution or approximate solution by transformation to a Satisfiability problem than by solving directly. However, care must be taken to choose a transformation that keeps running time down and supports low error rates. In this section we consider a successful transformation from Network Steiner Tree Problems to weighted Maximum CNF Satisfiability problems (taken from [82]).

The Network Steiner Tree Problem originates from the following important network cost problem (see, for example, [5]). Suppose a potential provider of communications services wants to offer private intercity service for all of its customers. Each customer specifies a collection of cities it needs to have connected in its own private network. Exploiting the extraordinary bandwidth of fiber-optic cables, the provider intends to save cable costs by “piggy-backing” the traffic of several customers on single cables when possible. Assume there is no practical limit on the number of customers piggy-backed to a single cable. The provider wants an answer to the following question: Through what cities should the cables be laid to meet all customer connectivity requirements and minimize total cabling cost?
(a) The beginning of the equivalency check - one circuit has been transformed to an AIG.

(b) Node $a$ of the circuit (Figure 1.6(a)) has been merged with vertex $b$ of the AIG.

(c) The completed AIG.

Figure 1.6: Creating the AIG.
This problem can be formalized as follows. Let $G(V, E)$ be a graph whose vertex set $V = \{c_1, c_2, c_3, \ldots \}$ represents all cities and whose edge set $E$ represents all possible connections between pairs of cities. Let $w : E \mapsto \mathbb{Z}^+$ be such that $w(\{c_i, c_j\})$ is the cost of laying fiber-optic cable between cities $c_i$ and $c_j$. Let $R$ be a given set of vertex-pairs $\{c_i, c_j\}$ representing pairs of cities that must be able to communicate with each other due to at least one customer’s requirement. The problem is to find a minimum total weight subgraph of $G$ such that there is a path between every vertex-pair of $R$.

Consider the special case of this problem in which the set $T$ of all vertices occurring in at least one vertex-pair of $R$ is a proper subset of all vertices of $G$ (that is, the provider has the freedom to use as connection points cities not containing customers’ offices) and $R$ requires that all vertices of $T$ be connected. This is known as the Network Steiner Tree Problem. An example and its optimal solution are given in Figure 1.7. This problem is one of the first shown to be $\mathcal{NP}$-complete [81]. The problem appears in
many applications and has been extensively studied. Many enumeration algorithms, heuristics and approximation algorithms are known (see [5] for a list of examples).

A feasible solution to an instance \((G, T, w)\) of the Network Steiner Tree Problem is a tree spanning all vertices of a subgraph of \(G\) which includes \(T\). Such a subgraph is called a Steiner Tree. A transformation from \((G, T, w)\) to an instance of Satisfiability is a uniform method of encoding of \(G\), \(T\), and \(w\) by a CNF formula \(\psi\) with non-negative weights on its clauses. A necessary (feasibility) property of any transformation is that a truth assignment to the variables of \(\psi\) which maximizes the total weight of all satisfied clauses specifies a feasible solution for \((G, T, w)\). A desired (optimality) property is, in addition, that feasible solution have minimum total weight. Unfortunately, known transformations that satisfy both properties produce formulas of size that is superlinearly related to the number of edges and vertices in \(G\). Such encodings are useless for large graphs. More practical linear transformations satisfying the feasibility property are possible but these do not satisfy the optimality property. However, it has been demonstrated that, using a carefully defined linear transformation, one can achieve close to optimal results.

As an example, we consider the linear transformation introduced in [82]. This transformation has a positive integer parameter \(k\) which controls the quality of the approximation. We can, without losing any interesting cases, assume that \(G\) is connected. We also assume that no two paths in \(G\) between the same nodes of \(T\) have the same weight; this can be made true, if necessary, by making very small adjustments to the weights of the edges.

**Preprocessing:**

1. Define an auxiliary weighted graph \(((T, E'), w')\), as follows:
   
   Graph \(G' = (T, E')\) is the complete graph on all vertices in \(T\).
   
   For edge \(e' = \{c_i, c_j\}\) of \(G'\), let \(w'(e')\) be the total cost of the minimum cost path between \(c_i\) and \(c_j\) in \(G\). (This can be found, for example, by Dijkstra’s algorithm.)

2. Let \(W\) be a minimum-cost spanning tree of \((G', w')\). We intend to choose, for each edge \(\{c_i, c_j\}\) of \(W\), (all the edges on) one entire path in \((G, w)\) between \(c_i\) and \(c_j\) to be included in the Steiner tree. This will produce a Steiner tree for \(T, G\), as long as no cycles in \(G\) are included, although perhaps not a minimum-cost tree.

   For some pre-specified, fixed \(k\): Find the \(k\) minimum-cost paths in \(G\) between \(c_i\) and \(c_j\) (again, for example, by a variation of Dijkstra’s algorithm); call them \(P_{i,j,1}, \ldots, P_{i,j,k}\). Thus, the algorithm will choose one of these \(k\) paths between each pair of elements of \(W\).

**The Formula:**
### 1.7 Transformations to Satisfiability

<table>
<thead>
<tr>
<th>Variable</th>
<th>Subscript Range</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e_{i,j}$</td>
<td>edge ${c_i, c_j} \in E$</td>
<td>${c_i, c_j}$ ∈ the Steiner Tree</td>
</tr>
<tr>
<td>$p_{i,j,l}$</td>
<td>${c_i, c_j} \in W$, $1 \leq l \leq k$</td>
<td>$P_{i,j,l}$ ⊆ the Steiner Tree</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clause</th>
<th>Subscript Range</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\neg e_{i,j})$</td>
<td>${c_i, c_j} \in E$</td>
<td>$e_{i,j}$ ∉ Steiner Tree</td>
</tr>
<tr>
<td>$(p_{i,j,1} \lor \ldots \lor p_{i,j,k})$</td>
<td>${c_i, c_j} \in W$</td>
<td>Include at least one of $k$ shortest paths between $c_i, c_j$.</td>
</tr>
<tr>
<td>$(\neg p_{i,j,l} \lor e_{m,n})$</td>
<td>${c_m, c_n} \in P_{i,j,l}$</td>
<td>Include all edges of that path.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clause</th>
<th>Weight</th>
<th>Significance</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\neg e_{i,j})$</td>
<td>$w({i, j})$</td>
<td>Maximize weights of edges <strong>not</strong> included</td>
</tr>
<tr>
<td>$(p_{i,j,1} \lor \ldots \lor p_{i,j,k})$</td>
<td>$I$</td>
<td>Force connected subgraph of $G$ containing all vertices of $T$</td>
</tr>
<tr>
<td>$(\neg p_{i,j,l} \lor e_{m,n})$</td>
<td>$I$</td>
<td>Force connected subgraph of $G$ containing all vertices of $T$</td>
</tr>
</tbody>
</table>

Table 1.5: Transformation from an instance of the Network Steiner Tree Problem to a CNF formula. The instance has already been preprocessed to spanning tree $W$ plus lists of $k$ shortest paths $P_{i,j,x}$, $1 \leq x \leq k$, corresponding to edges $\{c_i, c_j\}$ of $W$. The edges of the original graph are given as set $E$. Boolean variables created expressly for the transformation are defined at the top and clauses are constructed according to the middle chart. Weights are assigned to clauses as shown at the bottom. The number $I$ is chosen to be greater than the sum of all weights of edges in $E$. 
The output of the preprocessing step is a tree $W$ spanning all vertices of $T$. Each edge $\{c_i, c_j\} \in W$ represents one of the paths $P_{i,j,1}, \ldots, P_{i,j,k}$ in $G$. The tree $W$, the list of edges in $G$, the lists of edges comprising each of the $k$ shortest paths between pairs of vertices of $W$, and the number $k$ are input to the transformation step. The path weights are not needed by the transformation step and are therefore not provided as an input. The transformation is then carried out as shown in Table 1.5. In the table, $E$ is the set of edges of $G$ and $I$ is any number greater than the sum of the weights of all edges of $G$.

A maximum weight solution to a formula of Table 1.5 must satisfy all non-unit clauses because the weights assigned to those clauses are so high. Satisfying those clauses corresponds to choosing all the edges of at least one path between pairs of vertices in the list. Therefore, since the list represents a spanning tree of $G'$, a maximum weight solution specifies a connected subgraph of $G$ which includes all vertices of $T$.

On the other hand, the subgraph must be a tree. If there is a cycle in the subgraph then there is more than one path from a $T$ vertex $u$ to a non-$T$ vertex $v$, so there is a shorter path that may be substituted for one of the paths going from $u$ through $v$ to some $T$ vertex. Choosing a shorter path is always possible because it will still be one of the $k$ shortest. Doing so removes at least one edge and cycle. This process may be repeated until all cycles are broken and the number of edges remaining is minimal for all $T$ vertices to be connected. Due to the unit clauses, all edge variables whose values are not set to 1 by an assignment add their edge weights to that of the formula. Therefore, the maximum weight solution contains only edges “forced” by $p$ variables and must specify a Steiner Tree.

A maximum weight solution specifies an optimal Steiner Tree only if $k$ is sufficiently large to admit all the shortest paths in an optimal solution. Generally this is not practical since too large a $k$ will cause the transformation to be too large. However, good results can be obtained even with $k$ values up to 30.

Once the transformation to Satisfiability is made, an incomplete algorithm such as Walksat or even a branch-and-bound variant can be applied to obtain a solution. The reader is referred to [82] for empirical results showing speed and approximation quality. We will return to this subject in Section ??.

### 1.8 Boolean Data Mining

The field of *data mining* is concerned with discovering hidden structural information in databases; it is a form of (or variant of) *machine learning*. In particular, it is concerned with finding hidden correlations among apparently weakly-related data. For example, a credit card company might look for
patterns of charging purchases that are frequently correlated with using stolen credit card numbers. Using this data, the company can look more closely at suspicious patterns in an attempt to discover thefts before they are reported by the card owners.

Many of the methods for data mining involve essentially numerical calculations. However, others work on Boolean data. Typically, the relevant part of the database consists of a set \( B \) of \( m \) Boolean \( n \)-tuples (plus keys to distinguish tuples, which presumably are unimportant here). Each of these \( n \) attributes might be the results of some monitoring equipment or experts’ answers to questions.

Some of the tuples in \( B \) are known to have some property; the others are known not to have the property. Thus, what is known is a partially defined Boolean function \( f_B \) of the \( n \) variables. An important problem is to predict whether tuples to be added later will have the same property. This amounts to finding a completely defined Boolean function \( f \) which agrees with \( f_B \) at every value for which it is defined. Frequently, the goal is also to find such an \( f \) with, in some sense, a simple definition: such a definition, it is hoped, will reveal some interesting structural property or explanation of the data points.

For example, binary vectors of dimension two describing the condition of an automobile might have the following interpretation:

<table>
<thead>
<tr>
<th>Vector</th>
<th>Value</th>
<th>Overheating</th>
<th>Coolant Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>No</td>
<td>Low</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>No</td>
<td>Normal</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>Yes</td>
<td>Low</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>Yes</td>
<td>Normal</td>
</tr>
</tbody>
</table>

where the value associated with each vector is 1 if and only if the automobile’s thermostat is defective.

One basic method is to search for such a function \( f \) with a relatively short disjunctive normal form (DNF) definition. An example formula in DNF is

\[
( v_1 \land v_2 \land \neg v_3 ) \lor ( \neg v_1 \land v_4 ) \lor ( v_3 \land v_4 \land v_5 \land v_6 ) \lor ( \neg v_3 ) .
\]

DNF formulas have the same form as that of CNF formulas except that the positions of the \( \land \)’s and the \( \lor \)’s are reversed. In this case the formula is in 4-DNF since each disjunct contains at most 4 conjuncts. Each term, e.g., \(( v_1 \land v_2 \land \neg v_3 )\) is an implicant: whenever it is true, the property holds.

There is extensive research upon the circumstances under which, when some random sample points \( f_B \) from an actual Boolean function \( f \) are supplied, a program can, with high probability, learn a good approximation to \( f \) within reasonable time \(( e.g., [130])\). Of course, if certain additional properties of \( f \) are known, or assumed, more such functions \( f \) can be determined.
Applications are driving interest in the field. The problem has become more interesting due to the use of binarization which allows non-binary data sets to be transformed to binary data sets [22]. Such transformations allow the application of special binary tools for cause-effect analysis that would otherwise be unavailable and may even sharpen “explainability.” For example, for a set of 290 data points containing 9 attributes related to Chinese labor productivity, it was observed in [21] that the short clause

Not in the Northwest Region and Time is later than 1987,

where Time ranges over the years 1985 to 1994, explains all the data, and the clause

SOE is at least 71.44% and Time is earlier than 1988,

where SOE means state owned enterprises, explains 94% of the data. The technique of logical analysis of data has been successfully applied to inherently non-binary problems of oil exploration, psychometric analysis, and economics, among others.
Chapter 2

Math and Logic Foundations

This chapter presents definitions, problems, algorithms, and proof techniques which will be used in following chapters.

2.1 Definitions

In this section we present some definitions that will be needed to talk about formulas, the formulation of problems concerning formulas, graphical representations of formulas, the structure of algorithms for solving the problems, and some complexity concepts to support the analysis of those algorithms.

2.1.1 Logic

The most elementary object we consider is the propositional or Boolean variable, also shortened to variable in the context of this monograph. Other names for variables include letter and atom. A variable takes one of two values from the set \{0, 1\}. A literal is either a variable or a negated variable. A negated variable is written \(\neg v\) where \(v\) is some variable. The value of \(\neg v\) is opposite the value of \(v\). We use the term positive literal to refer to a variable and negative literal to refer to a negated variable. The polarity of a literal is positive or negative accordingly.

The building blocks of propositional statements are binary Boolean operators. A binary Boolean operator is a function \(O_b : \{0, 1\} \times \{0, 1\} \mapsto \{0, 1\}\). Often, such functions are presented in tabular form, called truth tables, as illustrated in Figure 1.3. There are 16 possible mappings and, therefore, operators. The most common (and useful) are \(\lor\) (or), \(\land\) (and), \(\rightarrow\) (implies), \(\leftrightarrow\) (equivalent), and \(\oplus\) (xor, alternatively exclusive-or). Actual mappings for these operators are given in the glossary under Operator, Boolean. The only unary Boolean operator, denoted \(\neg\) (negation), is a mapping from
0 to 1 and 1 to 0. Observe that the expression $\neg\neg v$ usually means the negation of the literal $\neg v$ which is equivalent to the literal $v$. Use of “binary” or “unary” to describe an operator will be dropped when the context is clear.

Operators for time dependent propositions are more complicated and typically depend on the notion of state. The state of a system is a set of parameters and their values. Time dependent systems often are modeled as moving from state to state where the new state is determined only by the current state and input values. Hence, a possible history of a time dependent system may be regarded as a possibly infinite sequence $S = (s_0, s_1, s_2, \ldots)$ of states starting from a distinguished initial one $(s_0)$. Suppose that $\psi_1$ and $\psi_2$ are propositions that may independently have value 0 or 1 for every state of a particular sequence. Some typical temporal operators on $\psi_1$ and $\psi_2$ are given in Table 1.3. Since formulas of temporal logic depend on the particular logic, as was the case with linear temporal logic discussed on page 11, we defer discussion of the description of such formulas until the type of temporal logic is defined. The remainder of this section treats only non-temporal formulas.

Formulas are expressions consisting of literals, parentheses, and operators which have some semantic content. A well-formed formula is defined recursively as follows:

1. Any single variable is a well-formed formula.
2. If $\psi$ is a well-formed formula, then so is $\neg \psi$ (alternatively written $\overline{\psi}$).
3. If $\psi_1$ and $\psi_2$ are both well-formed formulas and $\mathcal{O}$ is a Boolean binary operator, then the following is a well-formed formula: $(\psi_1 \mathcal{O} \psi_2)$. In this case we call $\psi_1$ the left operand of $\mathcal{O}$ and $\psi_2$ the right operand of $\mathcal{O}$.

A well-formed formula is also referred to as a propositional formula or simply a formula when the context is clear. A useful parameter associated with a formula is depth. The depth of a well-formed formula is determined as follows:

1. The depth of a formula consisting of a single variable is 0.
2. The depth of a formula $\neg \psi$ is the depth of $\psi$ plus 1.
3. The depth of a formula $(\psi_1 \mathcal{O} \psi_2)$ is the maximum of the depth of $\psi_1$ and the depth of $\psi_2$ plus 1.

Any assignment of values to the variables of a formula induces a value on the formula. A formula is evaluated from innermost $\neg$ or parentheses out. Many algorithms that will be considered later iteratively build assignments and it will be important to distinguish variables that have been assigned value 0 or 1 from those that have not been assigned a value. Therefore, we
allow a variable to hold a third value, denoted $\perp$, which means the variable is unassigned, that is, the variable is not assigned a value of 0 or 1. This requires that the evaluation of operations be augmented to account for $\perp$. For major operations this is as follows: $(x \land y)$ evaluates to $\perp$ if at least one operand evaluates to $\perp$ and the other operand does not evaluate to 0; $(x \lor y)$ evaluates to $\perp$ if at least one operand evaluates to $\perp$ and the other operand does not evaluate to 1; $(x \oplus y)$ and $(x \leftrightarrow y)$ evaluate to $\perp$ if either operand evaluates to $\perp$; $(x \rightarrow y)$ evaluates to $\perp$ if $x = 0$ and $y = \perp$ or $y = 1$ and $x = \perp$; $\neg x$ evaluates to $\perp$ if $x$ does. Given a set of variables and an assignment $M$ of values to those variables such that at least one has value $\perp$, we say that $M$ is a partial assignment.

Formulas can sometimes be simplified by removing some or all parentheses. Parentheses around nestings involving the same associative operators such as $\lor$, $\land$, and $\leftrightarrow$ may be removed. For example, $(\psi_1 \lor (\psi_2 \lor \psi_3))$, $((\psi_1 \lor \psi_2) \lor \psi_3)$, and $(\psi_1 \lor \psi_2 \lor \psi_3)$ are regarded to be the same formula. In the case of non-associative operators such as $\rightarrow$, parentheses may be removed but right associativity is then assumed.

Formulas we will be dealing with often have many components of the same type which are called clauses. Two common special types of clauses are disjunctive and conjunctive clauses. A disjunctive clause is a well-formed formula consisting only of literals and the operator $\lor$. If all the literals of a clause are negative (positive) then the clause is called a negative clause (alternatively, positive clause). In this monograph we often use the convention of representing disjunctive clauses as sets of literals. When it is understood that an object is a disjunctive clause, we refer to it simply as a clause. A conjunctive clause is a well-formed formula consisting only of literals and the operator $\land$. Again, we often represent a conjunctive clause as a set of literals and call it a clause when it is unambiguous to do so. The number of literals in any clause is referred to as the width of the clause.

Often, formulas are expressed in some normal form. Four of the most frequently arising forms are defined as follows.

A CNF formula is a well-formed formula consisting of a conjunction of two or more disjunctive clauses. We often represent a CNF formula as a set of clauses to facilitate concise algorithmic descriptions. The following two lines show the same CNF formula expressed, above, using Boolean operators and, below, as a set of sets of literals.

\[
(\neg v_0 \lor v_1 \lor \neg v_7) \land (\neg v_2 \lor v_3) \land (v_0 \lor \neg v_6 \lor \neg v_7) \land (\neg v_4 \lor v_5 \lor v_9) \\
\{\{\neg v_0, v_1, \neg v_7\}, \{\neg v_2, v_3\}, \{v_0, \neg v_6, \neg v_7\}, \{\neg v_4, v_5, v_9\}\}
\]

Given CNF formula $\psi$ and $L_\psi$, the set of all literals in $\psi$, a literal $l$ is said to be a pure literal in $\psi$ if $l \in L_\psi$ but $-l \notin L_\psi$. A clause $c \in \psi$ is said to be a unit clause if $c$ has exactly one literal.

A $k$-CNF formula, $k$ fixed, is a CNF formula restricted so that the width of each clause is exactly $k$. 

A **Horn formula** is a CNF formula with the restriction that all clauses contain at most one positive literal. Observe that a clause \((\neg a \lor \neg b \lor \neg c \lor g)\) is functionally the same as \((a \land b \land c \rightarrow g)\) so Horn formulas are closely related to logic programming. In fact, logic programming was originally the study of Horn formulas.

A **DNF formula** is a well-formed formula consisting of a disjunction of two or more conjunctive clauses. We also often represent a DNF formula as a set of clauses.

When discussing a CNF or DNF formula \(\psi\) we use \(V_\psi\) to denote its variable set and \(C_\psi\) to denote its clause set. The subscripts are dropped when the context is clear.

A formula takes a value depending on the values of its variables. A **truth assignment**, or **assignment**, is an assignment of values to the variables of a formula. As mentioned earlier, evaluation of a formula is from innermost parentheses out using the truth tables for each operator encountered and a given truth assignment. If the formula evaluates to 1, then the assignment is called a **satisfying assignment**, **model**, or a **solution**. We treat a truth assignment as a set of variables with the interpretation that these have value 1 and the rest have value 0.

There are \(2^n\) ways to assign values to \(n\) Boolean variables. Any subset of those assignments is a Boolean function on \(n\) variables. Thus, the number of such function is \(2^{2^n}\). Any Boolean function \(f\) on \(n\) variables can be expressed as a CNF formula \(\psi\) where the subset of assignments comprising \(f\) is identical to the set of assignments satisfying \(\psi\) (see Section ??). However, \(k\)-CNF formulas express only a proper subset of Boolean functions: for example, since a width \(k\) clause eliminates the fraction \(2^{-k}\) of potential models, any Boolean function comprising more than \(2^n(1 - 2^{-k})\) assignments cannot be represented by a \(k\)-CNF formula. Similarly, all Boolean functions can be expressed by DNF formulas but not by \(k\)-DNF formulas.

The partial evaluation of a given formula is possible when a subset of its variables are assigned values. Such a partial evaluation usually results in a new formula. We write \(\psi\mid_{v=1}\) to denote the formula resulting from the partial evaluation of formula \(\psi\) due to assigning value 1 to variable \(v\). An obvious similar statement is used to express the partial evaluation of \(\psi\) when \(v\) is assigned value 0 or some subset of variables is assigned values. For example,

\[
(v_1 \lor \neg v_2) \land (\neg v_1 \lor v_3) \land (\neg v_2 \lor \neg v_3) \mid_{v_1=1} = (v_3) \land (\neg v_2 \lor \neg v_3)
\]

since \((v_3) \land (\neg v_2 \lor \neg v_3)\) expresses all solutions to \((v_1 \lor \neg v_2) \land (\neg v_1 \lor v_3) \land (\neg v_2 \lor \neg v_3)\) given \(v_1\) has value 1.

If a truth assignment \(M\) is such that all the literals of a disjunctive (conjunctive) clause have value 0 (respectively, 1) under \(M\), then the clause is said to be **falsified** (respectively, **satisfied**) by \(M\). If \(M\) is such that at least one literal of a disjunctive (conjunctive) clause has value 1 (respectively, 0)
under \( M \), then the clause is said to be \textit{satisfied} (respectively, \textit{falsified}) by \( M \). If a clause evaluates to \( \bot \) then it is neither satisfied nor falsified.

A formula \( \psi \) is \textit{satisfiable} if there exists at least one truth assignment under which \( \psi \) has value 1. In particular, a CNF formula is \textit{satisfiable} if there exists a truth assignment to its variables which satisfies all its clauses. Otherwise, the formula is \textit{unsatisfiable}. Every DNF formula is satisfiable but a DNF formula that is satisfied by \textit{every} truth assignment to its variables is called a \textit{tautology}. The negation of a DNF tautology is an unsatisfiable CNF formula.

Several truth assignments may satisfy a given formula. Any satisfying truth assignment containing the smallest number of variables of value 1 among all satisfying assignments is called a \textit{minimal model} with respect to 1. Thus, consistent with our definition of model as a set of variables of value 1, a minimal model is a set of variables of least cardinality. The usual semantics for Horn formula logic programming is the minimal model semantics: the only model considered is the (unique) minimal one\(^4\).

If a CNF formula is unsatisfiable but removal of any clause makes it satisfiable, then the formula is said to be \textit{minimally unsatisfiable}. Minimally unsatisfiable formulas play an important role in understanding the difference between “easy” and “hard” formulas.

We finish the section with a discussion of equivalence. There are three types of equivalence we will be concerned with. Definitions of these equivalences and symbols used to denote them are as follows.

1. \textit{equality of formulas} (\( \psi_1 = \psi_2 \)): two formulas are \textit{equal} if they are the same string of symbols. We also use “=” for equality of Boolean values, for example \( v = 1 \).

2. \textit{logical equivalence} (\( \psi_1 \Leftrightarrow \psi_2 \)): two formulas \( \psi_1 \) and \( \psi_2 \) are said to be \textit{logically equivalent} if, for every truth assignment \( M \) to the variables of \( \psi_1 \) and \( \psi_2 \), \( M \) satisfies \( \psi_1 \) if and only if \( M \) satisfies \( \psi_2 \). For example, in the following expression, the two leftmost clauses on each side of “\( \Leftrightarrow \)” force \( v_1 \) and \( v_3 \) to have the same value so \( (v_2 \lor v_3) \) may be substituted for \( (v_1 \lor v_2) \). Therefore, the expression on the left of “\( \Leftrightarrow \)” is logically equivalent to the expression on the right.

\[
(\neg v_1 \lor v_3) \land (v_1 \lor \neg v_3) \land (v_1 \land v_2) \Leftrightarrow (\neg v_1 \lor v_3) \land (v_1 \lor \neg v_3) \land (v_2 \land v_3)
\]

Another example is:

\[
(v_1 \lor \neg v_2) \land (\neg v_1 \lor v_3) \land (\neg v_2 \lor \neg v_3) \mid_{v_1 = 1} \Leftrightarrow (v_3) \land (\neg v_2 \lor \neg v_3)
\]

\(^4\)Each satisfiable set of Horn clauses has a unique minimal model with respect to 1, which can be computed in linear time by a well-known algorithm [48, 74] which is discussed in Section 4.2. Implications of the minimal model semantics may be found in Section 4.5, among others. More information may be found in the glossary.
In the second example, assigning \( v_1 = 1 \) has the effect of eliminating the leftmost clause and the literal \( \neg v_1 \). After doing so, equivalence is clearly established.

At this time we feel it is important to point out the difference between \( \psi_1 \Leftrightarrow \psi_2 \) and \( \psi_1 \leftrightarrow \psi_2 \). The former is an assertion that \( \psi_1 \) and \( \psi_2 \) are logically equivalent. The latter is just a formula of formal logic upon which we can ask whether there exists a satisfying truth assignment. The symbol “\( \Leftrightarrow \)” may not be included in a formula, and it makes no sense to ask whether a given assignment satisfies \( \psi_1 \Leftrightarrow \psi_2 \). It is easy to show that \( \psi_1 \Leftrightarrow \psi_2 \) if and only if \( \psi_1 \leftrightarrow \psi_2 \) is a tautology (that is, it is satisfied by every truth assignment to the variables of \( \psi_1 \) and \( \psi_2 \)).

Similarly, we define \( \psi_1 \) logically implies \( \psi_2 \) (\( \psi_1 \Rightarrow \psi_2 \)): for every truth assignment \( M \) to the variables in \( \psi_1 \) and \( \psi_2 \), if \( M \) satisfies \( \psi_1 \) then \( M \) also satisfies \( \psi_2 \). Thus, \( \psi_1 \Leftrightarrow \psi_2 \) if and only if \( \psi_1 \Rightarrow \psi_2 \) and \( \psi_2 \Rightarrow \psi_1 \). Also, \( \psi_1 \Rightarrow \psi_2 \) if and only if \( \psi_1 \rightarrow \psi_2 \) is a tautology.

3. **functional equivalence** (\( \psi_1 \models_V \psi_2 \)): two formulas \( \psi_1 \) and \( \psi_2 \), with variable sets \( V_{\psi_1} \) and \( V_{\psi_2} \), respectively, are said to be functionally equivalent with respect to base set \( V \subseteq V_{\psi_1} \cap V_{\psi_2} \) if, for every truth assignment \( M_V \) to just the variables of \( V \), either

(a) there is a truth assignment \( M_1 \) to \( V_{\psi_1} \setminus V \) and a truth assignment \( M_2 \) to \( V_{\psi_2} \setminus V \) such that \( M_V \cup M_1 \) satisfies \( \psi_1 \) and \( M_V \cup M_2 \) satisfies \( \psi_2 \), or

(b) there is no truth assignment to \( V_{\psi_1} \cup V_{\psi_2} \) which contains \( M_V \) as a subset and satisfies either \( \psi_1 \) or \( \psi_2 \).

The assignments \( M_V \cup M_1 \) and \( M_V \cup M_2 \) are called extensions to the assignment \( M_V \). If formulas \( \psi_1 \) and \( \psi_2 \) are functionally equivalent with respect to base set \( V \) we write \( \psi_1 \equiv_V \psi_2 \). If \( V = V_{\psi_1} \cap V_{\psi_2} \) then we just write \( \psi_1 \equiv \psi_2 \). In this case logical equivalence and functional equivalence are the same.

For example, \( (\neg a) \land (a \lor b) \equiv (a) \land (a \lor c) \) because \( \exists b : (\neg a) \land (a \lor b) \) if and only if \( \exists c : (\neg a) \land (a \lor c) \). But \( (\neg a) \land (a \lor b) \neq (\neg a) \land (a \lor c) \) since \( a = 0, b = 1, c = 0 \) satisfies \( (\neg a) \land (a \lor b) \) but falsifies \( (\neg a) \land (a \lor c) \).

Functional equivalence is useful when transforming a formula to a more useful formula. For example, consider the Tseitin transformation [129] (Section 3.3) which extends resolution: for CNF formula \( \psi \) containing variables from set \( V_{\psi} \), any pair of variables \( x, y \in V_{\psi} \), and variable \( z \notin V_{\psi} \),

\[
\psi \models_{V_{\psi}} \psi \land (z \lor x) \land (z \lor y) \land (\neg z \lor \neg x \lor \neg y).
\]
The term “functional equivalence” is somewhat of a misnomer. For any set \( V \) of variables, \( \equiv_V \) is an equivalence relation but \( \equiv \) is not transitive: for example \( a \lor c \equiv a \lor b \) and \( a \lor b \equiv a \lor \neg c \) but \( a \lor c \not\equiv a \lor \neg c \).

The identity stated in the following lemma is useful in proving Theorem 14.

**Lemma 1.** Let \( O \) be any binary Boolean operator except the two trivial ones that evaluate to 0 or 1 regardless of the value of their operands. Given formulas \( \psi_1 \) and \( \psi_2 \), variables \( v_1 \in V_{\psi_1} \setminus V_{\psi_2} \) and \( v_2 \in V_{\psi_2} \setminus V_{\psi_1} \), and some base set \( V \) such that \( \psi_1 \big|_{v_1=1} \equiv_V \psi_1 \big|_{v_1=0} \) and \( \psi_2 \big|_{v_2=1} \equiv_V \psi_2 \big|_{v_2=0} \),

\[
(v_1 \ O \ v_2) \land \psi_1 \land \psi_2 \equiv_V \neg(v_1 \ O \ v_2) \land \psi_1 \land \psi_2.
\]

**Proof.** We must show that for every truth assignment \( M_V \) to \( V \): 1) if there is an extension that satisfies \( \psi_1 \), \( \psi_2 \), and \( (v_1 \ O \ v_2) \), then there is an extension that satisfies \( \psi_1 \), \( \psi_2 \), and \( \neg(v_1 \ O \ v_2) \); and 2) if there is an extension that satisfies \( \psi_1 \), \( \psi_2 \), and \( \neg(v_1 \ O \ v_2) \), then there is an extension that satisfies \( \psi_1 \), \( \psi_2 \), and \( (v_1 \ O \ v_2) \). We only need to take care of 1) since a similar argument applies for 2).

Assume there is an extension \( M_1 \) that satisfies \( \psi_1 \), \( \psi_2 \), and \( (v_1 \ O \ v_2) \). From \( \psi_1 \big|_{v_1=1} \equiv_V \psi_1 \big|_{v_1=0} \) and \( \psi_2 \big|_{v_2=1} \equiv_V \psi_2 \big|_{v_2=0} \) and the fact that \( M_1 \) satisfies both \( \psi_1 \) and \( \psi_2 \), all four extensions identical to \( M_1 \) except in \( v_1 \) and \( v_2 \) will satisfy \( \psi_1 \) and \( \psi_2 \). One of those will also satisfy \( \neg(v_1 \ O \ v_2) \) since \( O \) is non-trivial. Hence, if there is an extension to \( M_V \) that satisfies \( \psi_1 \), \( \psi_2 \), and \( (v_1 \ O \ v_2) \) there is also an extension that satisfies \( \psi_1 \), \( \psi_2 \), and \( \neg(v_1 \ O \ v_2) \). \( \Box \)

### 2.1.2 Graphs

A mathematical structure that is quite useful for representing and describing manipulations on various kinds of formulas is a graph. An **undirected graph**, or simply **graph**, is a set \( V \) of points called **vertices** and a set \( E \) of vertex pairs called **edges**. An edge \( \{v_i, v_j\} \) is said to **connect** vertices \( v_i \) and \( v_j \). The two vertices connected by an edge are said to be its **endpoints** and the edge is said to be **incident** to each of its endpoints. Edges may be visualized as lines connecting vertex points. Usually we write \( G(V, E) \), or simply \( G \) when the sets \( V \) and \( E \) are understood, to denote a graph of vertex set \( V \) and edge set \( E \). Graph \( G' = (V', E') \) is a **subgraph** of \( G = (V, E) \) if \( E' \subset E \) and \( V' \subset V \) and for every \( \{v_i, v_j\} \in E' \), \( v_i \in V' \) and \( v_j \in V' \).

A **bipartite graph** is a graph \( G = (V_1, V_2, E) \) whose vertex set is partitioned into two sets \( V_1 \) and \( V_2 \) and is such that every edge of \( E \) connects a vertex of \( V_1 \) with a vertex of \( V_2 \). The set of all vertices of \( V_2 \) that is connected by one edge to some vertex of subset \( V_1' \subset V_1 \) is called the **neighborhood** of \( V_1' \). A **matching** is a subset \( E' \subseteq E \) of edges such that no two edges of \( E' \) share the same endpoint. The following is a fundamental and useful result on bipartite graphs well known as Hall’s Theorem. It is stated without proof (see [70]).
Chapter 2. MATH AND LOGIC FOUNDATIONS

Theorem 2. Given a bipartite graph with vertex sets $V_1$ and $V_2$, a matching that includes every vertex of $V_1$ exists if and only if no subset of $V_1$ has a smaller neighborhood than itself.

One use for a bipartite graph is to represent all clauses of a CNF formula as vertices of $V_1$, all variables as vertices of $V_2$, and to interpret an edge as a statement that its clausal endpoint in $V_1$ contains the variable associated with its endpoint in $V_2$. Then, a matching expresses a truth assignment satisfying clauses in $V_1$ to which one of its edges is incident. This and other graphical representations of formulas will be used heavily to develop and prove correctness of algorithms by making use of known results in graph theory.

A somewhat different type of graph is sometimes needed to represent other important structural relationships between clauses and variables in CNF or DNF formulas. A directed graph, also known as a digraph, is a graph in which some orientation is given to each edge. Each edge of a digraph is called a directed edge and is visualized as a line with an arrowhead at one endpoint showing the direction of orientation and written as two vertices enclosed by angle brackets with orientation from left to right (e.g. $\langle v_1, v_2 \rangle$ is oriented from $v_1$ to $v_2$). Usually we write $\vec{G}(V, \vec{E})$, or simply $\vec{G}$ when $V$ and $\vec{E}$ are understood, to denote a digraph of vertex set $V$ and edge set $\vec{E}$. In a digraph, a vertex with at least one outward oriented edge is called an internal vertex. All other digraph vertices are leaf vertices. If the maximum number of outward oriented edges among all vertices is two, we say the digraph is a binary digraph. In a binary digraph an order is implicitly assigned to each pair of edges that is oriented outward from the same vertex. Therefore, we distinguish between the left and right edges of such a pair. One use for a digraph is to represent literals as vertices and use directed edges to represent inferences.

It will be important to explore the structure of various graphs and to determine the existence of graph properties that correspond to formula properties. Doing so will be easier if we define some general terms here. For a given undirected graph $G(V, E)$, there is a path from one chosen vertex $v_s \in V$ to another chosen vertex $v_t \in V$ if and only if there is a sequence $P = (v_s, u_0, u_1, \ldots, u_p, v_t)$ of vertices in $V$, beginning with $v_s$ and ending in $v_t$ such that, for all $0 \leq i < p$, there is an edge $\{u_i, u_{i+1}\} \in E$ and edge $\{v_s, u_0\} \in E$ and edge $\{u_p, v_t\} \in E$. We call such a $P$ a path. For a digraph, we add the restriction that all edges be oriented from left to right between adjacent pairs of $P$. A path $P$ from $v_s$ to $v_t$ may be traversed by “visiting,” in sequence, all edges identified by adjacent pairs of $P$. If, for all pairs of vertices $v_i, v_j \in V$, either there is a path from $v_i$ to $v_j$ or from $v_j$ to $v_i$ or paths in both directions, then the graph is connected. We say $P$ is a cycle if $P$ is a path and some vertex appears twice in $P$. If there is no sequence $P$ that is a cycle, then $G$ (or $\vec{G}$ in the case of a digraph) is said to be acyclic. In a connected acyclic digraph there is at least one vertex $v$
such that the orientation of all edges incident to $v$ is away from $v$. If there is exactly one such vertex $v$ then the graph is a rooted acyclic digraph, or rooted dag for short, and $v$ is distinguished as its root. A rooted dag is a binary rooted acyclic digraph if it is also a binary digraph. A subset $V' \subset V$ of vertices of graph $G(V,E)$ such that for all pairs $v_i, v_j \in V'$, $\{v_i, v_j\} \notin E$ is called an independent set of $G(V,E)$. A maximum independent set of $G$ is an independent set of $G$ of maximum cardinality. The size of a maximum independent set of $G$ is denoted $\alpha(G)$. The problem of finding a maximum independent set is $NP$-complete.

2.1.3 Algorithmic Structures and Operations

Algorithms are presented in figures, such as Figure 3.2, using a format described here. Each algorithm begins with a number for reference. This is followed by a line prototyping the algorithm, including its descriptive name and list of arguments. Next are comments describing what is expected as input, what is delivered as output, the names and types of all local variables used in the algorithm, plus any other comments deemed necessary to understand the operation of the algorithm clearly. The body of the algorithm follows and the end of the algorithm is marked by a box ($\square$).

The body of an algorithm consists of standard control constructs and operations on standard mathematical structures. Control constructs used are variations of the following and are familiar enough that no explanation is given here:

- **Repeat** the following while ⟨condition⟩:  
  ⟨statement(s)⟩

- **Repeat** the following for ⟨iterator range⟩:  
  ⟨statement(s)⟩

- If ⟨condition⟩ then do the following:  
  ⟨statement(s)⟩

- Otherwise  
  ⟨statement(s)⟩

Only indentation is used to indicate the scope of each construct to keep the number of non-descriptive lines to a minimum. The outcome of operations on structures depend on the type of the structure. Table 2.1 defines the operations used here.

To accommodate partial assignments in algorithmic statements we use the notation $M_{i;1}$ to mean a partial assignment to a set of $i$ variables and implicitly maintain a corresponding set $V_M$ of $i$ variables such that if $v \in V_M$ and $v \in M$, then $v$ has value 1, if $v \in V_M$ and $v \notin M$ then $v$ has value 0, and $v$ has value $\perp$ otherwise. The algorithmic statements
<table>
<thead>
<tr>
<th>Type</th>
<th>Operation</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set, CNF Formula</td>
<td>$A \cup B$</td>
<td>Evaluates to a set consisting of all elements of set $A$ and all elements of set $B$ with no duplicates</td>
</tr>
<tr>
<td></td>
<td>Set $\psi \leftarrow A$</td>
<td>Assign set $A$ to $\psi$</td>
</tr>
<tr>
<td>List</td>
<td>Pop $e \leftarrow L$</td>
<td>Remove the first object of list $L$ and assign it to $e$</td>
</tr>
<tr>
<td></td>
<td>Append $L \leftarrow e$</td>
<td>Append object $e$ to list $L$</td>
</tr>
<tr>
<td>Stack</td>
<td>Pop $e \leftarrow S$</td>
<td>Remove the first object of Stack $S$ and assign it to $e$</td>
</tr>
<tr>
<td></td>
<td>Push $S \leftarrow e$</td>
<td>Add object $e$ to the front of Stack $S$</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>Pop $e \leftarrow P$</td>
<td>Remove the highest priority object from $P$ and assign it to $e$</td>
</tr>
<tr>
<td></td>
<td>Insert $P \leftarrow \langle e, w \rangle$</td>
<td>Insert object $e$ into $P$ with priority $w$</td>
</tr>
<tr>
<td>Integer</td>
<td>Set $i \leftarrow j$</td>
<td>Assign the value of integer $j$ to $i$</td>
</tr>
<tr>
<td>Quoted Expression</td>
<td>Set $\psi^q \leftarrow &quot;e&quot;$</td>
<td>Assign to $\psi^q$ the textual string representing expression $e$</td>
</tr>
<tr>
<td></td>
<td>Next $s \leftarrow \psi^q$</td>
<td>Remove the leftmost symbol from $\psi^q$ and call it $s$</td>
</tr>
<tr>
<td>Well-Formed Formula</td>
<td>Replace $\psi^q$ with $e$ in $\psi$</td>
<td>Any occurrences of the quoted expression $\phi^q$ in formula $\phi$ should be replaced with expression $e$</td>
</tr>
</tbody>
</table>

Table 2.1: Algorithmic operations on mathematical structures
2.1.4 Complexity

In this subsection we review enough of the elementary concepts of complexity theory to allow the analysis of the algorithms to proceed. For each algorithm presented, we are interested in a worst case and probabilistic analysis of the number of major operations required for the algorithm to complete successfully in terms of the length of input formulas. The length of a general formula will be the number of variable symbols it contains. In the case of CNF or DNF formulas this is the number of literals in the formula. For example, a 3-CNF formula of \( m \) clauses has length \( 3m \). The “major operations” counted will depend on the algorithm and will be specifically stated.

Concerning worst case complexity, we seek a function of input length which bounds the number of major operations in the limit. Thus, worst
case complexity is expressed using “big-oh” notation (see [84]) as in \( O(m^2) \) or \( O(2^m) \). If the function is a polynomial on the input length we say the algorithm has \textit{polynomial worst case complexity}.

Concerning probabilistic analysis, we usually seek one or more of several kinds of results for a given algorithm: 1) the probability that a given algorithm solves a particular problem for a random input formula of length \( m \) in less than some polynomial function \( p(m) \) steps in the limit; 2) the probability that a given algorithm solves a particular problem for a particular random input formula (in this case the algorithm is known to have polynomial worst case complexity).

Many of the problems we consider are \( \mathcal{NP} \)-hard. A proper, formal discussion of the concept known as \( \mathcal{NP} \)-hardness is too lengthy to be repeated here. The reader is referred to [60] for details.

### 2.2 Problems

This monograph addresses the question of how to solve logic problems of the kind defined in this section. All the problems are \( \mathcal{NP} \)-hard but some have restricted versions that are efficiently solved and these are pointed out.

#### 2.2.1 Satisfiability

The most fundamental problem we consider is the Satisfiability problem or SAT. It is stated as follows.

\textbf{Satisfiability (SAT)}:

\begin{itemize}
  \item \textbf{Given:} A Boolean formula \( \psi \).
  \item \textbf{Question:} Determine whether \( \psi \) is satisfied by some truth assignment to the variables of \( \psi \).
\end{itemize}

Frequently, we will assume \( \psi \) is a CNF formula. If \( \psi \) is a 3-CNF formula, the question of satisfiability is still \( \mathcal{NP} \)-hard.

Some well known efficiently solved subclasses are the following. If \( \psi \) is a CNF formula with at most 2 literals per clause, then satisfiability can be determined for \( \psi \) using the linear time Algorithm 16 of Section 4.1. If \( \psi \) is a Horn formula, then satisfiability can be determined in linear time using Algorithm 17 of Section 4.2. If \( \psi \) is a CNF formula such that reversing the polarity of some subset of its variables results in a Horn formula, then \( \psi \) is \textit{renamable Horn}. Satisfiability of a renamable Horn formula can be determined in linear time by Algorithm 18 of Section 4.4.
2.2.2 Finding Minimal Models

The following two problems are of interest because solutions to these can be used to solve quite a number of other problems.

**Minimal Model:**
**Given:** A Boolean formula $\psi$.
**Question:** Determine whether $\psi$ is satisfied by some truth assignment and, if so, find all minimal models of $\psi$.

**Unique Minimum Model:**
**Given:** A Boolean formula $\psi$.
**Question:** Determine whether $\psi$ is satisfied by some truth assignment and, if so, determine whether $\psi$ has a unique minimum model.

All Horn formulas have unique minimum models (see Theorem 29) and these can be obtained in linear time by Algorithm 17 of Section 4.2.

The next three problems are commonly used extensions of the basic Satisfiability problem.

2.2.3 Finding Stable Models

Recall that Horn formulas consist of clauses of the form $(a \land b \land c \rightarrow g)$. A normal logic program has this form except that the use of negative subgoals is allowed, for example in $(a \land \neg b \land c \rightarrow g)$. At this time there is no completely standard semantics for logic programs, but the most commonly accepted is the **stable semantics** of [62] which seems to capture the asymmetry of causality.

Recall that a subset $M$ of variables is a model of a set of clauses if the assignment of value 1 to each variable of $M$ and 0 to all other variables satisfies every clause and that the semantics of Horn formulas is the minimal model semantics mentioned on Page 29. The stable model semantics is more restrictive: in addition to requiring a minimal model we suppose that the variables of value 1 should all be derivable directly from the given facts and the negative literals of value 1. This may be interpreted as reasoning “forward” through the $\rightarrow$’s.

Given a normal logic program $\psi$ and model $M$, evaluate all negative literals of $\psi$ in $M$ and simplify, removing all negative literals of value 1 and removing all clauses with negative literals of value 0. The result is a Horn formula $\psi^M$ called the **Gelfond-Lifschitz transform of $\psi$ with respect to $M$**. Since $\psi^M$ is a Horn formula, it has a unique minimum model. If $M$ is that unique minimum model, then $M$ is said to be a **stable model** of $\psi$. For example, for $\psi = (\neg b \rightarrow a)$, the classical models are $\{a\}$, $\{b\}$, and $\{a, b\}$. The
first two are minimal but only the first is stable. All logic programs have
models and thus also have minimal models, but such a program may have
0, 1, or many stable models.

**Stable Model:**

Given: A normal logic program $\psi$.

Question: Find all the stable models of $\psi$.

**Unique Stable Model:**

Given: A normal logic program $\psi$.

Question: Determine whether there is a unique stable model of $\psi$.

The problem of finding a stable model of a logic program is $\mathcal{NP}$-complete.

### 2.2.4 Well-Founded Models

A well-founded model for a normal logic program allows three values to be
assigned to all variables: 0, 1 and $\perp$ where $\perp$ has the meaning lacks a
classic logic value. The reason for finding a well-founded model is to be able
to infer as much as possible without introducing conflicts. For example, a
logic program which includes clauses $(p \rightarrow \neg p)$ and $(1 \rightarrow p)$ has no model
in classical two valued logic but partial models for the program might exist
with $p$ taking value $\perp$.

A well-founded model can be found with a simple iterative process. The
process begins by assigning $\perp$ to all variables and by adding a clause $(v \rightarrow 0)$
for every variable that appears only on the left side of $\rightarrow$ in clauses. This
has the intuitive interpretation of “negation as failure” - if you cannot prove
something is true then say it is false if doing so causes no conflicts. At each
iteration: 1) for any clause where the left side of $\rightarrow$ has value 1 and the right
side is $\perp$, change the value of the variable on the right side to 1; 2) for any
clause where the right side variable has value 0, force the left side to have
value 0. For example, consider

$$(1 \rightarrow a) \land (\neg b \land a \rightarrow c) \land (\neg c \rightarrow b) \land (\neg d \rightarrow e) \land (e \rightarrow f) \land (\neg a \rightarrow f) \land (d \rightarrow 0)$$

where the rightmost clause is not part of the original program but was added
because $d$ is always on the left side of $\rightarrow$ in the other clauses. After the first
iteration, $a$ gets the value 1 and $d$ gets the value 0. After the third iteration,
variables $e$ and $f$ have taken value 1. Any additional iteration will cause no
change so the well-founded model is found for this example.

From the above, it is not hard to see that a well-founded model can be
found in quadratic time. The semantics of well-founded models has gained
wide acceptance partly because it is amenable to non-monotonic reasoning.
Since finding a stable model is $NP$-complete, it can be difficult to determine a stable model for a normal program. But the variables that cause this difficulty simply get the value $\bot$ in the well-founded model for that program. The problem with well-founded models is that, too frequently, the number of variables taking value $\bot$ is much larger than hoped for.

**Well-Founded Model:**

**Given:** A normal logic program $\psi$.

**Question:** Find the well-founded model of $\psi$.

### 2.2.5 Variable Weighted Satisfiability

This problem is a generalization of the common problem of determining a minimal model for a given formula. The question of minimal models is considered in Sections 4.2 and ??.

**Variable Weighted Satisfiability:**

**Given:** A CNF formula $\psi$ and a function $w : V \mapsto Z^+$ where $w(v)$ is the weight of variable $v$.

**Question:** Determine whether $\psi$ is satisfied by some truth assignment and, if so, determine the assignment $M$ such that

$$\sum_{v \in M} w(v)$$

is minimized.

### 2.2.6 Maximum Satisfiability

This problem is seen in applications where it is desired to find the minimum number of facts to remove from a data base to make it consistent.

**MAX-SAT (Maximum Satisfiability):**

**Given:** A CNF formula $\psi$.

**Question:** Determine the assignment $M$ that satisfies the maximum number of clauses in $\psi$.

This problem remains $NP$-hard even when $\psi$ is restricted to 2-CNF formulas.

### 2.2.7 Weighted Maximum Satisfiability

This is a generalization of MAX-SAT and is useful for formulating many optimization problems such as the Consistency Matrix Problem and Network
Steiner Tree Problem stated in Chapter 1.

**Weighted MAX-SAT:**
*Given:* A CNF formula $\psi$ and a function $w : C_\psi \rightarrow Z^+$ where $w(c)$ is the weight of clause $c \in C_\psi$.

*Question:* Determine the assignment $M$ such that
\[ \sum_{c \in \psi} w(c) \cdot s_M(c) \] is maximized,

where $s_M(c)$ is 1 if clause $c$ is satisfied by $M$ and is 0 otherwise.

2.2.8 Equivalence of Boolean Formulas
The example of VLSI hierarchical design in Chapter 1 shows the importance of the question of functional equivalence.

**Equivalent Formulas:**
*Given:* Two Boolean formulas $\psi_1$ and $\psi_2$.

*Question:* Determine whether $\psi_1$ is functionally equivalent to $\psi_2$.

2.2.9 Binary Decision Diagram
Binary decision diagrams are an important graphical representation of propositional formulas. They are explained in Section 2.3.2.

**Finding a BDD:**
*Given:* A propositional formula $\psi$.

*Question:* Find the Binary Decision Diagram representing $\psi$ with the fewest vertices.

2.3 Representations, Structures, and Measures
Algorithmic concepts are usually easier to specify and understand if a formula or a particular part of a formula is suitably represented. Many representations are possible for Boolean formulas, particularly when expressed as CNF or DNF formulas. As mentioned earlier, we often use the convention that CNF and DNF formulas be represented as sets of clauses and clauses as
2.3.1 (0, ±1) Matrix

A CNF formula of \( m \) clauses and \( n \) variables may be represented as an \( m \times n \) \((0, \pm 1)\)-matrix \( \mathcal{M} \) where the rows are indexed on the clauses, the columns are indexed on the variables, and a cell \( \mathcal{M}(i, j) \) has the value +1 if clause \( i \) contains variable \( j \) as a positive literal, the value -1 if clause \( i \) contains variable \( j \) as a negative literal, and the value 0 if clause \( i \) does not contain variable \( j \) as a positive or negative literal. Figure 2.1 shows an example of a CNF formula and its \((0, \pm 1)\) matrix representation.

It is well known that the question of satisfiability of a given CNF formula \( \psi \) can be cast as an Integer Program as follows:

\[
\mathcal{M}_\psi \alpha + b \geq Z, \quad \alpha_i \in \{0, 1\}, \quad \text{for all } 0 \leq i < n,
\]

where \( \mathcal{M}_\psi \) is the \((0, \pm 1)\) matrix representation of \( \psi \), \( b \) is an integer vector.
with $b_i$ equal to the number of -1 entries in row $i$ of $M_\psi$, and $Z$ is a vector of 1s. A solution to this system of inequalities certifies $\psi$ is satisfiable. In this case a model can be obtained directly from $\alpha$ as $\alpha_i$ is the value of variable $v_i$. If there is no solution to the system, then $\psi$ is unsatisfiable.

In addition to the well-known matrix operations we will be interested in two operations including one that is specific to $(0, \pm 1)$ matrices. A column may be multiplied or scaled by -1. This has the effect of reversing the polarity of a single variable. Every solution before scaling corresponds to a solution after scaling; the difference is that the values of variables associated with scaled columns are reversed. Similarly, rows and columns may be permuted with the effect on a solution being only a possible relabeling of variables taking value 1.

### 2.3.2 Binary Decision Diagram

Binary Decision Diagrams (BDD) [7, 100] are a general, graphical representation for arbitrary Boolean functions. Various forms have been put into use, especially for solving VLSI design and verification problems. A canonical form ([25, 26]) has been shown to be quite useful for representing some particular, commonly occurring, Boolean functions. An important advantage of BDDs is that the complexity of binary and unary operations such as existential quantification, oring, anding, among others, is efficient with respect to the size of the BDD operands. Typically, a formula is given as
a large collection of BDDs and operations such as those stated above are applied repeatedly to create a single BDD which expresses the models, if any, of the given formula. Intermediate BDDs are created in the process. The problem is that the size of intermediate BDDs may become extraordinarily and impractically large even if the final BDD is small. So, in some applications BDDs are useful and in some they are not.

A Binary Decision Diagram (BDD) is a rooted, directed acyclic graph. A BDD is used to compactly represent the truth table, and therefore complete functional description, of a Boolean function. Vertices of a BDD are called terminal if they have no outgoing edges and are called internal otherwise. There is one internal vertex, called the root, which has no incoming edge. There is at least one terminal vertex, labeled 1, and at most two terminal vertices, labeled 0 and 1. Internal vertices are labeled to represent the variables of the corresponding Boolean function. An internal vertex has exactly two outgoing edges, labeled 1 and 0. The vertices incident to edges outgoing from vertex \( v \) are called then\((v)\) and else\((v)\), respectively. Associated with any internal vertex \( v \) is an attribute called index\((v)\) which satisfies the properties

\[
\text{index}(v) < \min\{\text{index}(\text{then}(v)), \text{index}(\text{else}(v))\}
\]

\[^{5}\text{In Section ?? BDDs will be used to represent a branching temporal logic as well.}\]
index(v) = index(w) if and only if vertices v and w have the same labeling (that is, correspond to the same variable). Thus, the index attribute imposes a linear ordering on the variables of a BDD. An example of a formula and one of its BDD representations is given in Figure 2.2.

Clearly, there is no unique BDD for a given formula. In fact, for the same formula, one BDD might be extraordinarily large and another might be rather compact. It is usually advantageous to use the smallest BDD possible. At least one canonical form of BDD, called reduced ordered BDD, does this [25, 26]. The idea is to order the variables of a formula and construct a BDD such that: 1) variables contained in a path from the root to any leaf respect that ordering; and 2) each vertex represents a unique Boolean function. Two Boolean functions are equivalent if their reduced ordered BDDs are isomorphic. A more detailed explanation is given in Section 3.9.

### 2.3.3 Implication Graph

An implication graph of a CNF formula \( \psi \) is a directed graph \( G_{\psi}(V, E) \) where \( V \) consists of one special vertex \( T \) which corresponds to the value 1, other vertices which correspond to the literals of \( \psi \), and the edge set \( E \) such that there is an edge \( \langle v_i, v_j \rangle \in E \) if and only if there is a clause \( \neg v_i \lor v_j \) in \( \psi \) and an edge \( \langle T, v_i \rangle \) or \( \langle T, \neg v_i \rangle \) if and only if there is a unit clause \( v_i \) (respectively, \( \neg v_i \)) in \( \psi \). Figure 2.3 shows an example of an implication graph for a particular 2-CNF formula.

Implication graphs are most useful for, but not restricted to, 2-CNF formulas. In this role the meaning of an edge \( \langle v_i, v_j \rangle \) is: if variable \( v_i \) is assigned the value 1 then variable \( v_j \) is inferred to have value 1 or else the clause \( \neg v_i \lor v_j \) will be falsified.
2.3.4 Propositional Connection Graph

A propositional connection graph for a CNF formula \( \psi \) is an undirected graph \( G_\psi(V,E) \) whose vertex set corresponds to the clauses of \( \psi \), and whose edge set is such that there is an edge \( \{c_i, c_j\} \in E \) if and only if the clause in \( \psi \) represented by vertex \( c_i \) has a literal that appears negated in the clause represented by vertex \( c_j \), and there is no other literal in \( c_i \)'s clause that appears negated in \( c_j \)'s clause. An example of a connection graph for a particular CNF formula is given in Figure 2.4. Propositional connection graphs are a specialization of the first-order connection graphs developed by Kowalski [97].

2.3.5 Variable-Clause Matching Graph

A variable-clause matching graph for a CNF formula is an undirected bipartite graph \( G = (V_1, V_2, E) \) where \( V_1 \) vertices correspond to clauses and \( V_2 \) vertices correspond to variables, and whose edge set contains an edge \( \{v_i, v_j\} \) if and only if \( v_j \in V_1 \) corresponds to a variable that exists, either as positive or negative literal, in clause \( v_j \in V_2 \). An example of a variable-clause matching graph is shown in Figure 2.5.

2.3.6 WFF Digraph

Well-founded formulas are defined recursively on page 26. Any Boolean formula can be adapted to fit this definition with the suitable addition of parentheses and can be represented by a binary rooted acyclic digraph. In such a digraph, each internal vertex corresponds to a binary operator or the unary operator \( \neg \) occurring in the formula. A vertex corresponding to a binary operator has two outward oriented edges: the left edge corresponds to the left subformula and the right edge to the right subformula operated
on. A vertex corresponding to $\neg$ has one outward directed edge. The root represents the operator applied at top level. The leaves are variables. We call such a representation a \textit{wff digraph}. An example is given in Figure 2.6. An efficient algorithm for constructing a wff digraph is given in Section 3.1.

### 2.3.7 Satisfiability Index

Let $\psi$ be a CNF formula and let $\mathcal{M}_\psi$ be its $(0, \pm 1)$ matrix representation. Let $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ be an $n$ dimensional vector of real variables. Let $z$ be a real variable and let $Z = (z, z, \ldots, z)$ be an $m$ dimensional vector where every component is the variable $z$. Finally, let $b = (b_0, b_1, \ldots, b_{m-1})$ be an $m$ dimensional vector such that for all $0 \leq i < m$, $b_i$ is the number of negative literals in clause $c_i$. Form the system of inequalities

$$
\mathcal{M}_\psi \alpha + b \leq Z,
$$

$$
0 \leq \alpha_i \leq 1 \quad \text{for all } 0 \leq i < n.
$$

The satisfiability index of $\psi$ is the minimum $z$ for which no constraints of the system are violated. For example, the CNF formula

$$
((x_1 \lor \neg x_2) \land (x_2 \lor \neg x_3 \lor x_5) \land (x_3 \lor \neg x_4 \lor \neg x_5) \land (x_4 \lor \neg x_1))
$$

has satisfiability index of $5/4$. 

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{wff_digraph.png}
\caption{A wff digraph and associated formula.}
\end{figure}
2.4 Probability

Probabilistic and average-case analysis can give useful insight into the question of what SAT algorithms might be effective and why. Under certain circumstances, one or more structural properties shared by each of a collection of formulas may be exploited to solve such formulas efficiently; or structural properties might force a class of algorithms to require superpolynomial time. Such properties may be identified and then, using probabilistic analysis, one may argue that these properties are so common that the performance of an algorithm or class of algorithms can be predicted for most of a family of formulas.

The main drawbacks of this approach are: 1) some distribution of input formulas must be assumed and a chosen distribution may not represent reality very well; 2) results are usually sensitive to the choice of distribution; 3) the state of analytical tools is such that distributions yielding to analysis are typically symmetric with independent components; 4) few algorithms have yielded to analysis. Despite these drawbacks, probabilistic results can be a useful supplement to worst-case results, which can be overly pessimistic for \( \mathcal{NP} \)-complete problems, in understanding algorithmic behavior.

In this section we present probabilistic tools used in analyses presented later in the monograph. We start with some foundational results.

2.4.1 Identities and Inequalities

The following inequalities and identities will be used to prove probabilistic results. If \( X \) is a random variable then \( \Pr(X \geq 1) \) is the probability that \( X \) has value at least as great as 1, \( \mu_X \) is the expected value of \( X \), and \( \sigma_X^2 \) is the variance of \( X \).

\[
(1 - x) \leq e^{-x} : 0 \leq x \leq 1 \\
(1 - x) \geq e^{-x/(1-x)} : 0 \leq x < 1
\] (2.3) (2.4)

2.4.2 Distributions and the Central Limit Theorem

Probabilistic analysis is greatly assisted if the underlying random variable \( X \) has its probability heavily concentrated around its mean. In that case, the uncertainty of the values of \( X \) gives way to a high level of predictability, effectively squeezing the probability out of the analysis. A well-known distribution with this property is the binomial distribution. A binomial distribution has parameters \( n \) and \( p \) and

\[
\Pr(X = k) = \begin{cases} 
{n \choose k} p^k (1-p)^{n-k} & 0 \leq k \leq n \\
0 & \text{Otherwise}
\end{cases}
\]
If $X$ is binomially distributed with parameters $n$ and $p$, its mean is $np$, its variance is $np(1-p)$, and Chernoff bounds for $X$ are

$$Pr(X < (1 - \beta)np) \leq e^{-\beta^2np/3}$$  \hspace{1cm} (2.5)
$$Pr(X > (1 + \beta)np) \leq e^{-\beta^2np/4}.$$  \hspace{1cm} (2.6)

The Chernoff bounds demonstrate the concentration of probability around the mean.

A normal distribution of mean $\mu$ and variance $\sigma^2$ has the density function

$$f(x; \mu, \sigma) = \frac{1}{\sigma \sqrt{2\pi}} \exp \left( -\frac{(x - \mu)^2}{2\sigma^2} \right).$$

The normal distribution is important to us for two reasons. The first is that the binomial distribution, with parameters $n$ and $p$, tends to a normal distribution, in the limit, with mean $np$ and variance $np(1-p)$. The second is that the sum of values of a collection of random variables is normally distributed provided no subset dominates the others, in some rather weak sense. The following is a statement of this fact involving iid variables.

**Theorem 3.** (Classic Central Limit Theorem) Let $X_1, X_2, X_3, ..., X_n$ be identically distributed random variables with mean $\mu$ and variance $\sigma^2$. Suppose values for all random variables are chosen independently. Let $S = X_1 + X_2 + X_3 + ... + X_n$. Then $S$ tends to a normal distribution with mean $n\mu$ and variance $n\sigma^2$ as $n$ gets large.

The theorem holds even if $X_i$ are integer random variables, even in some cases where the values of $X_i$ are not independently chosen, and even in some cases where $X_i$ are not identically distributed. If $\sigma$ is not too great, $S$ has most of its probability concentrated around its mean (see Section 2.4.4).

### 2.4.3 First Moment Method

The first moment method is a tool for providing an upper bound on the probability that a specified property holds on a class of random structures. Let $X$ be a positive real-valued random variable. Then

$$Pr(X \geq 1) = \int_{t=1}^{\infty} p_X(t)dt < \int_{t=1}^{\infty} tp_X(t)dt < \mu_X$$  \hspace{1cm} (2.7)

where we have used $p_X$ to denote the distribution density function of $X$. The above is known as Markov’s inequality. It says the mean of $X$ is an upper bound on the probability that $X$ takes value at least 1. Suppose $X$ takes value at least 1 if and only if a random structure has property $P$ and takes value 0 otherwise. Then the mean is an upper bound on the probability that a random structure has property $P$. The bound will be useful if the mean is small.
2.4.4 Second Moment Method

The second moment method is a tool for proving that a specified property holds on a class of random structures with high probability as the size parameter of the class tends to \( \infty \). It is applied here in two major ways: to determine bounds, in probability, on the running time of a proposed algorithm for SAT; and to determine a bound on the probability that a random formula is a member of a particular class of formulas.

For a given random structure (in our case, a random CNF formula \( \psi \) - a random formula will be defined in Section 2.4.5), a witness \( w \) is a substructure (in our case, a set of clauses, \( w \subseteq \psi \)) whose presence implies that the structure has some specified property \( P \). The second moment method entails proving that a randomly chosen \( \psi \) fails to contain any witness for \( P \) with probability tending to zero.

Let \( w \) denote a witness, and also represent the event that \( w \subseteq \psi \); which meaning is intended will be clear from context. Let \( W \) denote the set of all possible witnesses. Usually \( W \) is chosen so its elements are symmetric; that is, for any pair \( w, z \in W \), there is an automorphism of the probability space that maps \( w \) to \( z \). We shall assume this is the case in this section. Then \( p = Pr(w) \) is independent of \( w \). Let \( I_w \) be the indicator random variable that is 1 if event \( w \) occurs and 0 otherwise.

\[
\mu_{I_w} = p \quad \text{and} \quad \sigma_{I_w}^2 = \mu(I_w - p)^2 = p(1 - p).
\]

Define the random variable \( I = \sum_{w \in W} I_w \). Then \( \mu_I = |W|p \). A special case of the Chebyshev inequality states that

\[
Pr(I = 0) \leq \frac{\sigma_I^2}{\mu_I^2} \tag{2.8}
\]

Thus it suffices to show that this ratio approaches zero as the size parameter goes to infinity (in our case, the number of variables, \( n \), tends to \( \infty \)).

Notice that, if the events \( w \) were independent, then \( \sigma_I^2 = |W|p(1 - p) = O(\mu_I) \), and it would be sufficient to show that \( \mu_I \rightarrow \infty \) as \( n \rightarrow \infty \). The crux of the second moment method is to show that, although the events \( w \) are not independent, the dependencies are weak enough that \( \sigma_I^2 = o(\mu_I^2) \).

To analyze the expression for \( \sigma_I^2 \), we introduce the notation:

\( A(w) \) is the set of witnesses for \( P \) having some clause in common with \( w \), other than \( w \) itself;

\( D(w) \) is the set of witnesses for \( P \) having no clause in common with \( w \).

We can now state a basic lemma of the second moment method.

**Lemma 4.** (Alon and Spencer [9, Ch. 4.3, Cor. 5]) *With the above notation, if:*

---

**2.4.4 Second Moment Method**

The *second moment method* is a tool for proving that a specified property holds on a class of random structures with high probability as the size parameter of the class tends to \( \infty \). It is applied here in two major ways: to determine bounds, in probability, on the running time of a proposed algorithm for SAT; and to determine a bound on the probability that a random formula is a member of a particular class of formulas.

For a given random structure (in our case, a random CNF formula \( \psi \) - a random formula will be defined in Section 2.4.5), a witness \( w \) is a substructure (in our case, a set of clauses, \( w \subseteq \psi \)) whose presence implies that the structure has some specified property \( P \). The second moment method entails proving that a randomly chosen \( \psi \) fails to contain any witness for \( P \) with probability tending to zero.

Let \( w \) denote a witness, and also represent the event that \( w \subseteq \psi \); which meaning is intended will be clear from context. Let \( W \) denote the set of all possible witnesses. Usually \( W \) is chosen so its elements are symmetric; that is, for any pair \( w, z \in W \), there is an automorphism of the probability space that maps \( w \) to \( z \). We shall assume this is the case in this section. Then \( p = Pr(w) \) is independent of \( w \). Let \( I_w \) be the indicator random variable that is 1 if event \( w \) occurs and 0 otherwise.

\[
\mu_{I_w} = p \quad \text{and} \quad \sigma_{I_w}^2 = \mu(I_w - p)^2 = p(1 - p).
\]

Define the random variable \( I = \sum_{w \in W} I_w \). Then \( \mu_I = |W|p \). A special case of the Chebyshev inequality states that

\[
Pr(I = 0) \leq \frac{\sigma_I^2}{\mu_I^2} \tag{2.8}
\]

Thus it suffices to show that this ratio approaches zero as the size parameter goes to infinity (in our case, the number of variables, \( n \), tends to \( \infty \)).

Notice that, if the events \( w \) were independent, then \( \sigma_I^2 = |W|p(1 - p) = O(\mu_I) \), and it would be sufficient to show that \( \mu_I \rightarrow \infty \) as \( n \rightarrow \infty \). The crux of the second moment method is to show that, although the events \( w \) are not independent, the dependencies are weak enough that \( \sigma_I^2 = o(\mu_I^2) \).

To analyze the expression for \( \sigma_I^2 \), we introduce the notation:

\( A(w) \) is the set of witnesses for \( P \) having some clause in common with \( w \), other than \( w \) itself;

\( D(w) \) is the set of witnesses for \( P \) having no clause in common with \( w \).

We can now state a basic lemma of the second moment method.

**Lemma 4.** (Alon and Spencer [9, Ch. 4.3, Cor. 5]) *With the above notation, if:*

---
1. elements of $W$ are symmetric,

2. $\mu_1 \to \infty$ as $n \to \infty$,

3. $\sum_{z \in A(w)} Pr(z|w) = o(\mu_1)$ for an arbitrary $w \in W$,

then $Pr(P) \to 1$ as $n \to \infty$. 

2.4.5 Distributions Over Input Formulas

Of course, probabilistic results depend on the notion of random formula. This is formalized by specifying an input distribution: that is, an assignment of probabilities to specific formulas reflecting the frequency of occurrence of each formula. Most probabilistic results concern CNF formulas. For these, although several input distributions have been proposed, most probabilistic results are based on the two presented below.

The Constant-Width Distribution

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of $n$ variables. The parameters of the constant-width distribution are the cardinality $n$ of $V$, the number $k$ of literals that comprise a clause, and the number $m$ of clauses that make up a random CNF formula. Each clause is constructed as follows: uniformly choose a size $k$ subset of $V$ and negate each variable in the subset independently with probability $1/2$. A random $(m, n, k)$-SAT formula is a collection of $m$ such clauses, independently generated.

The constant-width distribution does not generate unrealistic clauses as does the variable-width distribution (the distribution that is described next). That is, null clauses, and tautological clauses (complementary literals in the same clause) are not allowed in a constant-width random formula.

The probabilistic analysis of SAT algorithms and structural properties under the constant-width distribution is often difficult. This is partly due to the change in structure of subformulas when assigning a value to a variable on an iteration of a particular algorithm. If subformulas are distributed as constant-width random formulas, the analysis can usually proceed easily. But, statistical dependence between clauses and literals often crops up and prevents this. Notable exceptions, however, include the analysis of the pure literal rule and a non-myopic greedy method. These are described in Section ??.

The Variable-Width Distribution

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of $n$ variables and $\bar{V} = \{\neg v_1, \neg v_2, \ldots, \neg v_n\}$ be their complements. The parameters of the variable-width distribution are
the cardinality \( n \) of \( V \), the number \( m \) of clauses, and the probability \( p(n, m) \) that a particular literal from \( V \cup \hat{V} \) is in a clause. Construct each clause \( c \) in a random formula as follows: for each literal \( l \in V \cup \hat{V} \), add \( l \) to \( c \) with probability \( p(n, m) \), independently of all other literals and clauses. A random variable-width formula is a collection of \( m \) such clauses, independently constructed. We have used \( p(n, m) \) to emphasize that the probability of variable occurrence can be a function of the number of clauses and the number of variables. From here on we use \( p \) in place of \( p(n, m) \) for simplicity.

Since literals are placed independently in clauses, it is possible for null clauses, or clauses with complementary literals (tautological clauses) to exist in a random formula. This does not mimic reality very well. However, the mathematics associated with average-time analyses under the variable-width distribution is often tractable. It would be straightforward but tedious to modify average-case analyses to account for no null or tautological clauses, but such results are unknown to us. Moreover, it appears that doing so will not change the results much if the average clause size is \( (1 + \epsilon) / \sqrt{m} < pn < (1 - \alpha) \ln(m) / 2 \) for any \( 0 < \epsilon \leq 1 \) and \( 0 < \epsilon \). The following results show.

**Theorem 5.** Under the variable-width distribution, the probability that a random formula contains

1. a tautological clause tends to 0 if \( nmp^2 \rightarrow 0 \);
2. a null clause tends to 1 if \( p \rightarrow 0 \) and \( pn < (1 - \epsilon) \ln(m) / 2 \), for any \( 0 < \epsilon < 1 \);
3. a null clause tends to 0 if \( pn > (1 + \epsilon) \ln(m) / 2 \), for any \( 0 < \epsilon < 1 \);
4. two complementary unit clauses capable of resolving to a null clause in one step tends to 0 if \( p\sqrt{n} \rightarrow 0 \), and \( pn > (1 + \epsilon) \ln(m) / 2 \), for any \( 0 < \epsilon < 1 \).

**Proof.**

1. The probability that a random clause is not tautological is \( (1 - p^2)^n \). The probability that \( m \) clauses are not tautological is \( (1 - p^2)^{nm} \). But \( (1 - p^2)^{nm} > e^{-nmp^2/(1-p^2)} \) from inequality (2.3). Hypothesis 1 follows.

2. The probability that a clause is null is \( (1 - p)^2n \). The probability that a formula has no null clause is \( (1 - (1 - p)^2n)^m \leq e^{-m(1-p)^2n} \) by inequality (2.4). By inequality (2.3) \( e^{-m(1-p)^2n} \leq e^{-me^{-2pn/(1-p)}} \). By hypothesis, this is bounded from above by

\[
e^{-me^{-(1-\epsilon)\ln(m)/(1-p)}} = e^{-m^{1+\epsilon/(1-p)}} = e^{-m^{(\epsilon-1)/(1-p)}} = e^{-m(\epsilon-p)/(1-p)}.
\]

This tends to 0 if \( p < \epsilon \). Hypothesis 2 follows.
3. The probability that all \( m \) clauses are not null is \((1 - (1 - p)^{2n})^m \geq e^{-m(1-p)^{2n}/(1-(1-p)^{2n})}\) by inequality (2.3). By inequality (2.4),

\[
e^{-m(1-p)^{2n}/(1-(1-p)^{2n})} > e^{-m^2np/(1-(1-p)^{2n})} > e^{-m^{1-(1+\epsilon)/(1-1/m)}} = e^{-m^{-\epsilon/(1-1/m)}}.
\]

Hypothesis 3 follows.

4. The probability that a pair of clauses is unit and complementary is \(\binom{n}{1} p^2 (1 - p)^2 (n-1)\). Hence the average number of pairs yielding a null clause on one resolution step is no greater than \(mnp^2 e^{-2pn}\). By inequality (2.7) this bounds the probability that at least one pair directly resolves to a unit clause. If \(pn > (1 + \epsilon) \ln(m)/2\), \(mnp^2 e^{-2pm} < mnp^2 m^{-(1+\epsilon)} = np^2 m^{-\epsilon}\). Hypothesis 4 follows.

To add some perspective to this result we show that random formulas with \(pm < (1 + \epsilon)/\sqrt{m}\) are trivially satisfiable when excluding null clauses and intuit that formulas with \(pn > \ln(m) * \sqrt{n}\) are trivially satisfiable. Consider the first case. The probability that a literal occurs at least twice in a random formula is

\[
1 - (1 - p)^m - mp(1 - p)^{m-1} \approx (mp)^2.
\]

Then, the average number of literals occurring more than once is about \(2n(mp)^2\) which, by Markov’s inequality, bounds the probability of the existence of any such literals. If \(m/n\) is constant and \(pn = o(1)/\sqrt{m}\) this bound tends to 0. Now consider the second case. A clause of \(k\) literals can eliminate \(2^{-k}\) fraction of \(2^n\) possible models. If \(k = \log_2(m)\) then \(2^n / m\) models can be eliminated by a clause. But then at most \(m(2^n/m) = 2^n\) models can be eliminated by \(m\) clauses. Thus, CNF formulas of minimum clause width of \(\log_2(m)\) (and at least one clause exceeding that) must be satisfiable. But \(pn\) is the average clause width of a random formula so, if \(pn > \ln(m) * \sqrt{n}\), the average clause width exceeds that minimum. Moreover, since the width of a random clause is binomially distributed, we know from 2.5 that the probability of a random clause having width \(\ln(m)\) is exponentially small. Hence, in this case, a random formula can be easily shown to be satisfiable by checking clause widths.

\[2.4.6 \text{ Formula density}\]

We will use the notion of formula density extensively in this manuscript. The performance of an algorithm will be determined asymptotically, as \(n\) increases. For \((m, n, k)\)-random formulas that increase will be accompanied by a corresponding increase in the number of clauses according to \(m = \lambda_k(n) \cdot n\). The density of CNF formulas, defined as the ratio of clauses
to variables, is then asymptotically $m/n = \lambda_k(n)$. The reason for doing this is that formula properties change dramatically as density changes. For example,

**Theorem 6.** ([55]) Let $p_k(m, n)$ denote the probability that a random $(n, m, k)$-CNF formula has a model. For every $k \geq 2$ there exists a sequence $\lambda_k(n)$ such that for any $0 < \epsilon$,

$$\lim_{n \to \infty} p_k((\lambda_k(n) - \epsilon)n, n) = 1 \quad \text{and} \quad \lim_{n \to \infty} p_k((\lambda_k(n) + \epsilon)n, n) = 0.$$

Theorem 6 says for each $k$ there is a sharp change in the satisfiability of a random $(m, n, k)$-formula at some density $\lambda_k(n)$ for every $n$. That density is approximated by the following:

**Theorem 7.** ([3]) Let $\lambda_k(n)$ be the sequence of Theorem 6.

$$2^k \ln(2) > \lambda_k(n) \geq 2^k \ln(2) - (k + 1) \ln 2 - 1 - \delta_k$$

where $\delta_k \to 0$ for increasing $k$.

### 2.4.7 Markovian approximation via differential equations

Probabilistic analysis was originally applied to search algorithms acting on CNF formulas, most notably variants of DPLL (shown in Figure 3.4) to gain intuition leading to better search heuristics. Unfortunately, a good heuristic heavily conditions the probability space of residual formulas as search progresses making analysis quite difficult or impossible with known techniques. It was realized early that some sort of analysis is possible if computation is restricted to progress in one direction, through a succession of states: specifically, unassigned literals are selected one at a time, all occurrences of that literal are set to value 1 and all occurrences of its negation are set to value 0, and those values are not changed thereafter (in other words, no backtracking). A state, in this case, is a partial assignment $\mathcal{T}$ of values to variables and associated with each state is a residual formula which is the original formula with literals falsified by $\mathcal{T}$ and clauses satisfied by $\mathcal{T}$ removed. We will refer to algorithms of this type as straight line algorithms.

Under certain conditions, distributions of residual formulas, when partitioned on clause width, are the same as the original distribution except for a change in parameters. This fact facilitates a Markovian analysis. Unfortunately, it was shown that those conditions also force an upper limit on the usefulness of the algorithms under analysis. But later it was shown that under weaker conditions that a Markovian analysis is still possible, leading the way to greatly improved results. This section presents the tools used in
a Markovian analysis and states the kind of results possible.

**Myopic algorithms**

Most performance results on random \((m, n, k)\)-CNF formulas have been obtained for myopic algorithms. As stated above, when moving from state to state, satisfied clauses and falsified literals are dropped from the formula. This means a given state corresponds to a formula with clauses of various widths. Partition the clauses of a state’s formula so that all clauses of equal width are in the same class. We say a state’s *spectrum* is the set of cardinalities of clauses for each class of the partition plus the number of unassigned variables. A straight line algorithm is called *myopic* if, under the spectral coalescence of states, the distribution of formulas corresponding to a particular coalesced state can be expressed by its spectral components alone: that is, by the number of clauses of width \(i\), for all \(1 \leq i \leq k\), and the number of unassigned variables. Thus, given random \((m, n, k)\)-CNF formulas, the distribution of formulas for the coalesced “start” state is determined by the distribution of such formulas; and the distribution for the coalesced state corresponding to \(j\) assigned variables and \(m_1, m_2, ..., m_k\) clauses of width 1, 2, ..., \(k\) is that of random \((m_1, n-j, 1)\)-CNF, \((m_2, n-j, 2)\)-CNF, ..., \((m_k, n-j, k)\)-CNF formulas, respectively for each clause width.

To determine whether an algorithm is myopic it is sufficient to show that no information about residual clauses and literals, other than number, is revealed after a literal is selected and assigned a value. This is the case if literal assignments are made randomly. In Section 7.1 more illuminating examples are given. We say that a literal selection strategy is myopic if the straight line program using the strategy is myopic.

Not all, in fact very few, literal selection strategies are myopic. For example, always choosing to satisfy a pure literal when one exists among residual clauses cannot be myopic because the distribution of number of occurrences in clauses depends on whether a pure literal is chosen or not. To see this, consider the full set of random \((4, 6, 3)\)-CNF formulas which we will denote by \(\psi_{4,6}^{4,6}\). Label the variables \(v_1, v_2, ..., v_6\). For each formula \(\psi \in \psi_{3}^{4,6}\) let \(p_{\psi}(e_1, e_2, e_3)\) be the probability that \(v_4, v_5\), or \(v_6\) is chosen first and the residual formula resulting from that choice is \((v_1 \lor v_2 \lor v_3) \land (\langle e_1, v_1 \rangle \lor \langle e_2, v_2 \rangle \lor \langle e_3, v_3 \rangle) \land *\) where \(\langle e, v \rangle\) means variable \(v\) occurs as an unnegated literal if \(e = 1\) and as a negated literal if \(e = 0\), and * means arbitrary (note that no literals are falsified since the chosen variable is a pure literal). Let \(p(e_1, e_2, e_3) = \sum_{\psi \in \psi_{3}^{4,6}} p_{\psi}(e_1, e_2, e_3)\). Obviously, \(p(e_1, e_2, e_3)\) should be independent of \(e_1, e_2, e_3\) if choosing pure literals is myopic. Moreover, \(p(1, 1, 1)\) and \(p(0, 0, 0)\) should be equal. But, since \(v_1, v_2, v_3\) can be pure in the latter case but cannot be pure in the former case, this requires that the pure literal selection heuristic would have to ignore \(v_1, v_2,\) and \(v_3\) and pick only from \(v_4, v_5\), or \(v_6\). Reversing the roles of \(v_1, v_2, v_3\) and \(v_4, v_5, v_6\), the
pure literal selection heuristic would have to ignore \( v_4, v_5, \) and \( v_6 \). Then no literals can be chosen! Thus, any form of the pure literal heuristic\(^6\) cannot be myopic.

In another much-studied, non-myopic literal selection strategy known as the Johnson heuristic, clauses are weighted by their width and literals are weighted by the sum of the weights of the clauses containing it. The strategy is to select the literal of highest weight. The weights are designed to maximize the probability that the partial assignment of the current state plus the variable assignment implied by the selected literal is a subassignment of a model for the formula. Experimental evidence suggests this is quite effective relative to all myopic strategies (see Section ??) but there is as of yet no probabilistic analysis supporting this. Such an analysis probably would reveal much about the distinction between hard and easy formulas and would be more than welcome.

In Section 7.1 some examples of myopic algorithms are discussed. What they have in common is that variable elimination choices can be based on the number of occurrences of literals in remaining clauses but ties must be broken randomly.

**Differential equations to approximate discrete processes**

The idea of using differential equations to approximate discrete random processes goes back at least to [99] and its application to the analysis of algorithms goes back to [96]. Given an initial random \((m, n, k)\)-CNF formula, the process we will approximate is the flow of clauses out of the formula as they become satisfied and the flow of \( i \) literal clauses down to \( i - 1 \) literal clauses as literals are falsified during iterations of a straight line algorithm. To this end let \( m_i(j) \) be the number of clauses containing \( i \) literals at the start of the \( j \)th iteration of a straight line algorithm. Initially, \( m_k(1) = m \) and \( m_i(1) = 0 \) for \( 0 < i < k \). Represent a coalesced state as a vector \( \langle m_1(j), m_2(j), ..., m_k(j) \rangle \).

The following theorem from [1] (based on Theorem 2 of [132]) is used to approximate clause flows by differential equations. In the statement of the theorem \( X = o(f(n)) \) *always* means \( \max\{x : Pr(X = x) \neq 0\} = o(f(n)) \), “uniformly” refers to the convergence implicit in the \( o(.) \) terms, and \( f(u_1, ..., u_k) \) satisfies a Lipschitz condition on \( D \subseteq \mathbb{R}^k \) which means there exists a constant \( L > 0 \) such that \( |f(u_1, ..., u_j) - f(v_1, ..., v_j)| \leq L \sum_{i=1}^{j} |u_i - v_i| \) for all \( (u_1, ..., u_j) \) and \( (v_1, ..., v_j) \) in \( D \).

**Theorem 8.** Let \( m_i(j) \) be a sequence of real-valued random variables, \( 0 < i \leq k \) for some fixed \( k \), such that for all \( i, j, \) and all \( n, |m_i(j)| \leq Bn \) for some constant \( B \). Let \( H(j) = \langle m_1(1), m_k(1), ..., m_1(j), ..., m_k(j) \rangle \) be

\(^6\)For example, if some weighting scheme is applied to the set of pure literals existing in a formula. For more details on this see [114].
the state history of sequences.

Let \( I = \{ \langle c_1, ..., c_k \rangle : Pr(\langle m_1(1), ..., m_k(1) \rangle = \langle c_1n, ..., c_kn \rangle) \neq 0 \text{ for some } n \} \).

Let \( D \) be some bounded connected open set containing the intersection of \( \{ s, c_1, ..., c_k : s \geq 0 \} \) with a neighborhood of \( \{ (0, c_1, ..., c_k) : \langle c_1, ..., c_k \rangle \in I \} \).

Let \( f_i : \mathbb{R}^{k+1} \rightarrow \mathbb{R}, \ 0 < i \leq k, \) and suppose that for some function \( m = m(n) \)

(i) for all \( i \) and uniformly over all \( j < m \)

\[ Pr(|m_i(j + 1) - m_i(j)| > n^{1/5}|H(j)|) = o(n^{-3}) \text{ always;} \]

(ii) for all \( i \) and uniformly over all \( j < m \)

\[ E\{m_i(j+1) - m_i(j)|H(j)\} = f_i(j/n, m_1(j)/n, ..., m_k(j)/n) + o(1) \text{ always;} \]

(iii) for each \( i \), the function \( f_i \) is continuous and satisfies a Lipschitz condition on \( D \).

Then

(a) for \( \langle 0, \hat{z}(1), ..., \hat{z}(k) \rangle \in D \) the system of differential equations

\[ \frac{dz_i}{ds} = f_i(s, z_1, ..., z_k), \ 0 < i \leq k \]

has a unique solution in \( D \) for \( z_i : \mathbb{R} \rightarrow \mathbb{R} \) passing through \( z_i(0) = \hat{z}(i), \ 0 < i \leq k \), and which extends to points arbitrarily close to the boundary of \( D \);

(b) almost surely,

\[ m_i(j) = z_i(j/n) \cdot n + o(n), \]

uniformly for \( 0 \leq j \leq \min\{\sigma n, m\} \) and for each \( i \), where \( z_i(j) \) is the solution in (a) with \( \hat{z}(i) = m_i(j)/n \), and \( \sigma = \sigma(n) \) is the supremum of those \( s \) to which the solution can be extended.

\( \square \)

Hypothesis (i) ensures that \( m_i(j) \) does not change too quickly from iteration to iteration of an algorithm; hypothesis (ii) tells us what we expect the rate of change of \( m_i(j) \) (or expected flow) to be and involves functions which are calculated from a knowledge of what the algorithm is doing; and hypothesis (iii) ensures that this expected flow does not change too quickly.

How to use the solution to the differential equations

Theorem 8 is useful particularly because the differential equations are developed using expectations of clause counts and flows which in many cases
are relatively easy to compute. Moreover, these expectations in the discrete world are translated to actual flow and count values in the solution to corresponding differential equations. Thus, the solution found in Theorem 8(b) for $m_2(j)$, say, does not deviate significantly from $E\{m_2(j)\}$ asymptotically. This is significant because in that case the expected flows are often sufficient to predict conditions whereby a myopic algorithm will be successful probabilistically, in some sense. Consider the family of myopic algorithms that always selects a literal that is a unit clause in the residual formula.

**Theorem 9.** ([1]) Let $A$ be a myopic algorithm that always chooses to satisfy a unit clause, when one exists among non-satisfied clauses. Let $U_j$ be the event that on iteration $j$ of $A$ there are no empty or unit clauses existing among remaining non-satisfied clauses. Suppose

$$m_2(j) < (1 - \delta)(n - j)$$

for all $1 \leq j < (1 - \epsilon)n$, $0 < \epsilon$ and $0 < \delta$ fixed, almost always. Then, there exists $\rho = \rho(\delta, \epsilon)$, $\rho > 0$, such that $Pr(U_{(1-\epsilon)n}) > \rho$.

Theorem 9 can be applied directly to the result of Theorem 8(b) but there are two mysteries that need some clarification first. For one thing, Theorem 9 only applies to $j < (1 - \epsilon)n$. This problem is disposed of on an ad-hoc basis. For example, in Section 7.1, page 157 the following is obtained for a particular myopic algorithm:

$$m_i(j) = \frac{1}{2^{k-i}} {k \choose i} \left(1 - \frac{j}{n}\right)^i \left(\frac{j}{n}\right)^{k-i} m.$$  

Then, using the binomial theorem,

$$\sum_{i=2}^{k} m_i(j) = \left((1 - \frac{j}{2n})^k - \frac{j}{2n} - k \left(1 - \frac{j}{n}\right) \left(\frac{j}{2n}\right)^{k-1}\right) m. \quad (2.9)$$

Let $m/n = 1/\lambda$ and suppose $\lambda < 1$, $1/2^k < \lambda^7$, and of course $k \geq 3$. Set $\epsilon = \lambda / {k \choose 2}$. After substituting $1 - \epsilon$ for $j/n$ on the right side of (2.9), expanding terms, and collecting powers of $\epsilon$, it can be seen that (2.9) is bounded from above by $8 {k \choose 2} (\lambda / {k \choose 2})^2 m/2^k = 8\lambda n/2^k = (8/2^k) \epsilon n \leq \epsilon n$. In other words, the number of width 2 and greater clauses remaining when $j = (1 - \epsilon)n$ is no greater than the number of unset variables, with high

---

7The case $\lambda > 1$ is not interesting in this context since the simple minded strategy of randomly removing all but two literals from every clause and applying a 2-CNF algorithm to the result succeeds with high probability in that case.

8Straight line algorithms for finding models will do poorly if $1/2^k \geq \lambda$ since almost all formulas have no models in that case.
probability. By Theorem 9 there are no unit clauses or empty clauses remaining with bounded probability. But, assuming the algorithm is myopic and inputs are random \((m,n,k)\)-CNF formulas, one may randomly remove all but two literals from each clause resulting in a random \((m,n,2)\)-CNF formula. Such a formula is satisfiable with high probability and can be taken care of trivially.

The second mystery concerns the bound \(\rho\) which is only guaranteed to be a constant and may not be close to 1. Following the notation of Theorem 6, let \(p^A_k(m,n)\) be the probability that myopic algorithm \(A\) finds a model for a random \((m,n,k)\)-CNF formula. For many myopic algorithms \(A\), particularly those that select literals in unit clauses, there exist constants \(\lambda_{k,1}^A\) and \(\lambda_{k,2}^A\) such that \(\lim_{n \to \infty} p^A_k(m,n) \to 1\) if \(m/n < \lambda_{k,1}^A\), \(\lim_{n \to \infty} p^A_k(m,n) \to 0\) if \(m/n > \lambda_{k,2}^A\), and otherwise \(\lim_{n \to \infty} p^A_k(m,n)\) tends to a constant with a smooth and gradual the transition from 1 to 0. Typically, for \(k = 3\), \(\lambda_{3,2}^A > 2\lambda_{3,1}^A\). We note that, in the case where the analysis is used to determine a bound on the probability of the existence of a model, finding a constant bound is all that is needed to prove the probability tends to 1. Discussion of this is beyond the scope of this monograph.

2.4.8 Deferred Decisions

A clause flow analysis may still be possible for non-myopic straight line algorithms under some restrictions. For example, flow analysis is still possible if algorithmic operations include [95]:

1. Select uniformly at random a pure literal, assign it the value 1 and remove all satisfied clauses.

2. Select uniformly at random a literal occurring exactly once in the expression and its occurrence is in a 3-literal clause, assign it the value 0, remove it from the clause it appears, and remove all clauses containing its complementary literal (these are satisfied).

3. Select uniformly at random a literal occurring exactly once in the expression and its occurrence is in a 2-literal clause, assign it the value 0, remove it from the clause it appears, and remove all clauses containing its complementary literal (these are satisfied). Then apply the unit clause unit to exhaustion (until no unit clauses remain).

2.4.9 Martingale Processes

A process in which a sequence \(X_1, X_2, X_3...\) of random numbers is generated such that

\[
\mu_{X_n | X_1, ..., X_{n-1}} = X_{n-1}
\]  

(2.10)
is a Martingale. In other words, the expected value of the next number in the sequence is the value of the current number, regardless of prior history. Alternatively, equation (2.10) may be rewritten

$$\mu_{X_n - X_{n-1}\mid X_1,\ldots,X_{n-1}} = 0.$$ 

The probabilities of a Martingale sequence may be shown to be highly concentrated around their means, effectively squeezing the probability out of the variables. Thus, regardless of statistical dependences, the outcomes of a Martingale are predictable. Specifically,

**Theorem 10.** (Martingale Central Limit Theorem) Let $X_1, X_2, X_3, \ldots$ be a Martingale. Define

$$\sigma_n^2 = \mu((X_n - X_{n-1})^2\mid X_1,\ldots,X_{n-1})$$

and let

$$\tau_v = \min \left\{ n : \sum_{i=1}^{n} \sigma_i^2 \geq v \right\}.$$ 

Then $X_{\tau_v}/\sqrt{v}$ converges in distribution to the normal distribution with mean 0 and variance 1.

and

**Theorem 11.** (Azuma’s Inequality, Alon and Spencer [9, Ch. 7.2, Cor. 2]) Let $X_1, X_2, X_3, \ldots$ be a Martingale process with $|X_{i+1} - X_i| \leq 1$ for all $1 \leq i$. Then

$$\Pr(|X_n - \mu_{X_n}| > \lambda\sqrt{n} \mid X_1,\ldots,X_{n-1}) < 2 \cdot e^{-\lambda^2/2}.$$ 

2.4.10 Jensen’s Inequality

Following the intuitive notion of convexity, a **convex function** is a continuous function of one argument whose value at the midpoint of every interval in its domain does not exceed the average of its values at the ends of the interval. That is, $f(x)$ is convex on interval $[a, b]$ if for any two points $x_1, x_2 \in [a, b]$ and any $0 < \rho < 1$, $f(\rho x_1 + (1-\rho)x_2) \leq \rho f(x_1) + (1-\rho)f(x_2)$ [115]. If $X$ is a random variable and $f(x)$ a convex function defined on at least a segment containing the range of $X$, then

$$\mu_{f(X)} \geq f(\mu_X).$$

This is known as Jensen’s inequality.
2.5 Eigenvalues

Eigenvalues and eigenvectors are perhaps one of the most underutilized tools in the analysis of logic algorithms. Let $A$ be an $n \times n$ matrix and $v$ an $n$-dimensional vector. Then $Av$ is regarded to be a transformation of $v$ in $n$ space. A vector whose direction is unchanged by $A$ is called an eigenvector. The factor by which $v$ is scaled due to the transformation is called its eigenvalue. Thus, an eigenvector is a vector satisfying

$$Av = \lambda v$$

where $\lambda$, its eigenvalue, is a scalar constant.

The problem of determining the eigenvalues of $A$ can be handled, for example, by the following

**Theorem 12.** ([108]) The eigenvalues of $A$ can be computed with relative error less than $2^{-b}$ in time $O((n')^3 + (n' \log^2(n')) \log(b))$. \hfill $\square$

The following theorem shows a connection between independent sets of a graph and eigenvalues of a matrix representation of that graph.

**Theorem 13.** ([98]) Let $G(V, E)$ be an undirected graph with vertex set $V = \{1, ..., n\}$. Define an $n \times n$ matrix $A_G$ with columns and rows indexed on vertices of $G$ and such that $A_G(i, j) = 1$ if edge $\langle v_i, v_j \rangle \notin E$ and $A_G(i, j) = -\frac{1-p}{p}$ if edge $\langle v_i, v_j \rangle \in E$ for some $0 < p < 1$ where $p$ may be a function of $n$. Let $\lambda_1(A_G)$ be the greatest eigenvalue of $A_G$ and $\alpha(G)$ be the size of the greatest independent set of $G$. Then $\alpha(G) < \lambda_1(A_G)$.

**Proof.** It is easy to verify that an eigenvector of a $k \times k$ all 1s matrix is the all 1s vector and its eigenvalue is $k$. If the maximum independent set in $G$ is of size $\alpha(G) = k$ then, after re-indexing, $A_G$ has a $k \times k$ submatrix of all 1s. By the interlacing theorem $\lambda_1(A_G) > k = \alpha(G)$. \hfill $\square$

There are a number of reasons we might be interested in independent sets of a graph. One example leads to an alternative algorithm for certifying unsatisfiability that is probabilistically superior to resolution on random CNF formulas. It is based on the simple observation that, given $(m, n, k)$-CNF formula $\psi$, if the minimum number of variables that must be set to 1 to satisfy all positive clauses is $n_+$, the minimum number of variables that must be set to 0 to satisfy all negative clauses is $n_-$, and $n_+ + n_- > n$, then no model exists for $\psi$ (otherwise, some variable would have to be set to both 0 and 1 which is not possible).

Alternatively, if $\psi$ has a model then either there is a subset $V_+$ of $n_+ \leq n/2$ positive literals such that $\psi$ has no positive clause taken strictly from $V_+$ or there is a subset $V_-$ of $n_- \leq n/2$ negative literals such that $\psi$ has no negative clause taken strictly from $V_-$. This may be expressed formally by constructing graphs $G_+$ and $G_-$, where each vertex of $G_+$ (respectively,
2.5 EIGENVALUES

$G_-$) is uniquely labeled by a set of positive (negative) literals and an edge between vertices of $G_+$ ($G_-$) exists if and only if a positive (negative) clause of $\psi$ consists of all literals labeling both its endpoints. Then, with a proper labeling, $|V_+|$ and $|V_-|$ are bounded by expressions directly related to $\alpha(G_+)$ and $\alpha(G_-)$.

Finding the size of a maximum independent set in a graph is $NP$-complete. Theorem 13 will be used to obtain a bound. Since finding eigenvalues or at least useful bounds on eigenvalues is usually tractable, there is hope of finding a certificate of unsatisfiability if the bound of the theorem and the bounds relating $|V_+|$ to $\alpha(G_+)$ and $|V_-|$ to $\alpha(G_-)$ are reasonably tight for the problem at hand. In some cases it is, as is shown in Section 7.5. It is beyond the scope of this monograph to discuss algorithms for finding eigenvalues and eigenvectors. The reader is instead referred to [108].
Chapter 3

General Algorithms

In this section we look at some algorithms, at least in part analyzable, for solving the problems of Section 2.2. Most of these algorithms operate on CNF formulas. Therefore, the following efficient transformation to CNF formulas is needed to demonstrate the general applicability of these algorithms.

3.1 Efficient Transformation to CNF Formulas

We discuss an algorithm, due to Tseitin [129], which efficiently transforms an arbitrary Boolean formula $\phi$ to a CNF formula $\psi$ such that $\psi$ has a model if and only if $\phi$ has a model and if a model for $\psi$ exists, it is a model for $\phi$. The transformation is important because it supports the general use of many of the algorithms which are described in following sections and require CNF formulas as input.

The transformation can best be visualized graphically. As discussed in Section 2.3.6, any Boolean formula $\phi$ can be represented as a binary rooted acyclic digraph $W_\phi$ where each internal vertex represents some operation on one or two operands (an example is given in Figure 2.6). Associate with each internal vertex $x$ of $W_\phi$ a new variable $v_x$ not occurring in $\phi$. If $x$ represents a binary operator $O_x$, let $v_l$ and $v_r$ be the variables associated with the left and right endpoints, respectively, of the two outward oriented edges of $x$ ($v_l$ and $v_r$ may be variables labeling internal or leaf vertices). If $x$ represents the operator $\neg$ then call the endpoint of the outward oriented edge $v_g$. For each internal vertex $x$ of $W_\phi$ write the logical equivalence

\[
v_x \iff (v_l \ O_x \ v_r) \quad \text{if } O_x \text{ is binary, or} \\
v_x \iff \neg v_g \quad \text{if vertex } x \text{ represents } \neg.
\]

For each equivalence there is a short, functionally equivalent CNF expression. The table of Figure 3.1 shows the equivalent CNF expression for all 16 possible binary operators where each bit pattern in the left column
expresses the functionality of an operator for each of four assignments to \( v_l \) and \( v_r \), in increasing order, from 00 to 11\(^9\). The equivalent CNF expression for \( \neg \) is \((v_9 \lor v_{10}) \land (\neg v_9 \lor \neg v_{10})\).

The target of the transformation is a CNF formula consisting of all clauses in every CNF expression from Figure 3.1 that corresponds to an equivalence expressed at a non-leaf vertex of \( W_\phi \) plus a unit clause that forces the root expression to evaluate to 1. For example, the expression of Figure 2.6, namely

\[ v_0 \Leftrightarrow ((\neg v_0 \Leftrightarrow (v_1 \lor \neg v_2)) \land (v_1 \lor \neg v_2) \land (\neg v_2 \rightarrow \neg v_3 \rightarrow v_4)), \]

transforms to

\[
\begin{align*}
&(v_0 \lor v_{x_1}) \land (\neg v_0 \lor \neg v_{x_1}) \land \\
&(v_{x_1} \lor v_{x_2}) \land (\neg v_{x_1} \lor \neg v_{x_2}) \land \\
&(v_{x_2} \lor v_{x_3}) \land (\neg v_{x_2} \lor \neg v_{x_3}) \land \\
&(\neg v_{x_1} \lor v_{x_2} \lor v_{x_3}) \land (\neg v_{x_1} \lor v_{x_2} \lor \neg v_{x_3}) \land \\
&(v_{x_2} \lor v_{x_3} \lor v_{x_4}) \land (\neg v_{x_2} \lor v_{x_3} \lor \neg v_{x_4}) \land \\
&(v_{x_3} \lor v_{x_4} \lor v_{x_5}) \land (\neg v_{x_3} \lor v_{x_4} \lor \neg v_{x_5}) \land \\
&(v_{x_4} \lor v_{x_5} \lor v_{x_6}) \land (\neg v_{x_4} \lor v_{x_5} \lor \neg v_{x_6}) \land (\neg v_{x_2} \lor v_{x_4} \lor v_{x_6}) \land (v_{x_3} \lor v_{x_5} \lor v_{x_6}) \land \\
\end{align*}
\]

\(^9\)See **Operator, Boolean** in the glossary for more details
where each line except the last corresponds to an internal vertex $x_i$ of $W_\phi$ of Figure 2.6, $i$ increasing from 1 in left-to-right and bottom-to-top order, and the new variables labeling those vertices are $v_{x1}, \ldots, v_{x10}$ correspondingly. The last line forces the root expression to evaluate to 1. The algorithm of Figure 3.2 expresses these ideas formally. For simplicity, it is assumed that the input is a well-formed formula, as described on Page 26.

The next two results show the algorithm does what is needed.

**Theorem 14.** Let $\phi$ be a well-formed formula and $V_\phi = \{v_0, v_1, \ldots, v_{n-1}\}$ the set of $n$ variables contained in it. The output of Algorithm 1 on input $\phi$ represents a CNF formula $\psi$, written $(v_0)$ if $\phi$ has no operators and otherwise written $(v_1) \land \psi_\phi$, $n \leq i$, where clause $(v_i)$ is due to the line “Set $\psi := \{\text{Pop } S\}$”, and $\psi_\phi$ is such that $\psi_\phi |_{v_i=1} \models_{V_\phi} \psi_\phi |_{v_i=0}$ \footnote{The meaning of $\models$ is given on Page 30}. In addition, $\psi \models_{V_\phi}$. That is, any truth assignment $M_{V_\phi} \subseteq V_\phi$ is a model for $\phi$ if and only if there is a truth assignment $M_1 \subseteq \{v_n, v_{n+1}, \ldots, v_i\}$ such that $M_{V_\phi} \cup M_1$ is a model for $\psi$.

**Proof.** The output is a set of sets of variables and therefore represents a CNF formula. The line “Set $\psi := \{\text{Pop } S\}$” is reached after all input symbols are read. This happens only after either the “Evaluate ‘$s$’” block or the “Push $S := s$” line. In the former case, $v_i$ is left at the top of the stack. In the latter case, $s$ must be $v_0$, in the case of no operators, or $v_i$ from execution of the “Evaluate ‘$v\text{\&}w$’” block immediately preceding execution of the “Push $S := s$” line (otherwise the input is not a well-formed formula). Therefore, either $v_0$, or $v_i$ is the top symbol of $S$ when “Set $\psi := \{\text{Pop } S\}$” is executed so $\{v_0\}$ or $\{v_i\}$ is a clause of $\psi$.

We show $\psi_\phi$ and $\psi$ have the stated properties by induction on the depth of the input formula $\phi$. For improved clarity, we use $\models$ instead of $\models_{V_\psi_\phi}$ below.

The base case is depth 0. In this case $\phi$ is the single variable $v_0$ and $n = 1$. The line “Push $S := s$” is executed in the first Repeat block, then the line “Set $\psi := \{\text{Pop } S\}$” is executed so $\psi = \{v_0\}$. Since $L = \emptyset$, execution terminates. Obviously, $\psi = \phi$ so both hypotheses are satisfied.

For the induction step, suppose $\phi$ is a formula of positive depth $k+1$ and the hypotheses hold for all formulas of depth $k$ or less. We show they hold for $\phi$. Consider first the case $\phi = \neg \phi'$ ($\phi'$ has depth $k$). Algorithm 1 stacks $\neg$ in the line “Push $S := s$” and then, due to the very next line, proceeds...
Algorithm 1. WFF Transformation to CNF ($\phi$)

/* Input $\phi$ is a well-formed formula as defined on Page 26 */
/* Output $\psi$ is a CNF expression functionally equivalent to $\phi$ */
/* Assume variables of $\phi$ are labeled $v_0, v_1, \ldots, v_{n-1}$ */
/* Additional variables $v_n, v_{n+1}, v_{n+2}, \ldots$ are created as needed */
/* Locals: stack $S$, integer $i$, quoted expression $\phi^q$, set of clauses $\phi^c$ */

Set $\phi^q \leftarrow "\phi"$.
Next $s \leftarrow \phi^q$.
Repeat the following while $\phi^q \neq \emptyset$:
   If $s$ is a variable and the symbol at the top of $S$ is $\neg$,
      Set $i \leftarrow i + 1$. // Evaluate ‘$\neg s$’
      Replace the top of $S$ with $v_i$.
      Append $L \leftarrow "v_i \leftrightarrow \neg s"$.
      Next $s \leftarrow \phi^q$.
   Otherwise, if $s$ is ‘)’,
      Pop $w \leftarrow S$. // Evaluate ‘($v_0 w$)’
      Pop $O \leftarrow S$.
      Pop $v \leftarrow S$.
      Pop $S$.
      Set $i \leftarrow i + 1$.
      Append $L \leftarrow "v_i \leftrightarrow (v \ O \ w)"$.
      Set $s \leftarrow v_i$.
   Otherwise, // Push an operator, positive variable, or ‘(’ onto $S$
      Push $S \leftarrow s$.
      Next $s \leftarrow \phi^q$.
Set $\psi \leftarrow \{\{\text{Pop } S\}\}$
Repeat the following while $L \neq \emptyset$:
   Pop $\psi^q \leftarrow L$.
   If $\psi^q$ equals “$v_j \leftrightarrow \neg v_g$” for some $v_j$ and $v_g$,
      Set $\psi \leftarrow \psi \cup \{\{v_j, v_g\}, \{-v_j, -v_g\}\}$.
   Otherwise, $\psi^q$ equals “$v_j \leftrightarrow (v_i \ O \ v_r)$” so do the following:
      Build $\phi^c$ based on $O, v_j, v_l, v_r$ using the table of Figure 3.1.
      Set $\psi \leftarrow \psi \cup \phi^c$.
Output $\psi$.

Figure 3.2: Algorithm for transforming a well-formed formula to an equivalent CNF formula
as though it were operating on $\phi'$. The algorithm reaches the same point it would have if $\phi'$ were input. However, in this case, $\neg$ and a variable are on the stack. This requires one pass through the “Evaluate ‘$\neg$ ‘” block which adds “$v_i \iff \neg s$” to $L$ and leaves $v_i$ as the only symbol in $S$. Thus, upon termination,

$$
\psi = (v_i) \land \psi_{\phi} = (v_i) \land (v_{i-1} \lor v_i) \land (\neg v_{i-1} \lor \neg v_i) \land \psi_{\phi'}
$$

and $v_i$ is not in $\psi_{\phi'}$. Hence,

$$
\psi_{\phi} \mid_{v_i=1} = ((v_{i-1} \lor v_i) \land (\neg v_{i-1} \lor \neg v_i)) \mid_{v_i=1} \land \psi_{\phi'}
$$

$$
= (\neg v_{i-1}) \land \psi_{\phi'}
$$

$$
\equiv (v_{i-1}) \land \psi_{\phi'} \quad \text{(by the induction hypothesis and Lemma 1, Page 31)}
$$

$$
= ((v_{i-1} \lor v_i) \land (\neg v_{i-1} \lor \neg v_i)) \mid_{v_i=0} \land \psi_{\phi'}
$$

$$
= \psi_{\phi} \mid_{v_i=0}.
$$

Next, we show $\psi \models \phi$ for this case. The expression $(v_i) \land \psi_{\phi}$ can evaluate to 1 only when the value of $v_i$ is 1. Therefore, we can write

$$
\psi = (v_i) \land \psi_{\phi} \models \psi_{\phi} \mid_{v_i=1}
$$

$$
= (\neg v_{i-1}) \land \psi_{\phi'}
$$

$$
\equiv (v_{i-1}) \land \psi_{\phi'} \quad \text{(from above)}
$$

$$
\equiv \phi' \quad \text{(by $\psi \models \phi$ induction hypothesis)}
$$

$$
\equiv \phi.
$$

Finally, consider the case that $\phi = (\phi_l \ O \ \phi_r)$ ($\phi_l$ and $\phi_r$ have depth at most $k$). The Algorithm stacks a ‘(’ then, by the inductive hypothesis and the recursive definition of a well-formed formula, completes operations on $\phi_a$ which results in $\psi_{\phi_l}$ in $L$. The line “Set $\psi \leftarrow \{ \{ \text{Pop } S \} \}”$ is avoided because there are still unread input symbols. Thus, there are two symbols on the stack at this point: a ‘(’ which was put there initially and a variable. The symbol $O$ is read next and pushed on $S$ by the line “Push $S \leftarrow s$”. Then $\psi_{\phi_r}$ is put in $L$ but again the line “Set $\psi \leftarrow \{ \{ \text{Pop } S \} \}”$ is avoided because there is still a ‘)’ to be read. Thus, the stack now contains ‘(’, a variable, say $v_1$, an operator symbol $O$, and another variable, say $v_r$. The final ‘)’ is read and the “Evaluate ‘($vOv$)’” section causes the stack to be popped, $v \iff (v_lOv_r)$ to be added to $L$, and variable $v_i$ to be put on $S$ (in the final iteration of the first loop). Therefore, upon termination,

$$
\psi = (v_i) \land \psi_{\phi} = (v_i) \land (v_i \iff (v_l \ O \ v_r)) \land \psi_{\phi_l} \land \psi_{\phi_r}
$$

where $(v_l) \land \psi_{\phi_l}$ is what the algorithm output represents on input $\phi_l$ and $(v_r) \land \psi_{\phi_r}$ is represented by the output on input $\phi_r$ (some clauses may be
duplicated to exist both in $\psi_{\phi_l}$ and $\psi_{\phi_r}$). Then,

$$
\psi_{\phi} \mid v_i = 1 = (v_i \Leftrightarrow (v_l \lor v_r)) \mid v_i = 1 \land \psi_{\phi_l} \land \psi_{\phi_r},
$$

$$
= (v_l \lor v_r) \land \psi_{\phi_l} \land \psi_{\phi_r},
$$

$$
\Rightarrow - (v_l \lor v_r) \land \psi_{\phi_l} \land \psi_{\phi_r} \quad \text{(induction hypothesis and Lemma 1)}
$$

$$
= (v_i \Leftrightarrow (v_l \lor v_r)) \mid v_i = 0 \land \psi_{\phi_l} \land \psi_{\phi_r}.
$$

It remains to show $\psi = \phi$ for this case. By the induction hypothesis, $\psi_{\phi_l} \mid v_i = 1 \Rightarrow \psi_{\phi_l} \mid v_i = 0$. Therefore, for a given truth assignment $M \phi$, if there is an extension such that $v_l$ has value 1 (0) and $\psi_{\phi_l}$ has value 0, then there is always an extension such that $v_l$ has value 0 (1), respectively, and $\psi_{\phi_l}$ has value 1. Since, by the induction hypothesis, $\phi \equiv (v_l \land \phi_{\phi_l})$, there is always an extension such that $\psi_{\phi_l}$ has value 1 and $v_l$ has the same value as $\phi_l$. The same holds for $v_r$ and $\phi_r$. Therefore,

$$
\psi = (v_i) \land \psi = \psi_{\phi} \mid v_i = 1
$$

$$
= ((v_i) \land (v_i \Leftrightarrow (v_l \lor v_r))) \mid v_i = 1 \land \psi_{\phi_l} \land \psi_{\phi_r}
$$

$$
\Rightarrow (v_l \lor v_r) \land \psi_{\phi_l} \land \psi_{\phi_r}
$$

$$
\Rightarrow (\phi_l \lor \phi_r) = \phi
$$

\[\square\]

**Theorem 15.** Algorithm 1 produces a representation using a number of symbols that is no greater than a constant times the number of symbols the input formula has if there are no double negations in the input formula.

**Proof.** Each binary operator in the input string accounts for a pair of parentheses in the input string plus itself plus a literal. Hence, if there are $m$ binary operators, the string has at least $4m$ symbols.

If there are $m$ binary operators in the input string, $m$ new variables associated with those operators will exist in $\psi$ due to the Append of the ‘Evaluate ‘$(v_l \lor w_r)$’ block. For each, at most 12 symbols involving literals, 12 involving operators (both $\land$ and $\lor$ represented as “,”), and 8 involving parentheses (represented as “{” “}”) are added to $\psi$ (see Figure 3.1), for a total of at most 32 symbols. At most $m + n$ new variables associated with negations are added to $\psi$ (both original variables and new variables can be negated) due to the Append of the ‘Evaluate ‘$\neg s$’ ” block. For each, at most 4 literals, 4 operators, and 4 parentheses are added to $\psi$ (from $\{v_i, v_j\}, \{-v_i, -v_j\}$), for a total of $12(m + n)$ symbols. Therefore, the formula output by Algorithm 1 has at most $44m + 12n$ symbols.

Since $n - 1 \leq m$, $44m + 12n \leq 56m + 12$. Therefore, the ratio of symbols of $\psi$ to $\phi$ is not greater than $(56m + 12)/4m = 14 + 3/m < 17$.

\[\square\]

The next result shows that the transformation is computed efficiently.
Theorem 16. Algorithm 1 has $O(m)$ worst case complexity if input formulas do not have double negations, where $m$ is the number of operators, and one symbol is used per parenthesis, operator, and variable.

Proof. Consider the first Repeat block. Every right parenthesis is read once from the input, is never stacked, and results in an extra loop of the Repeat block due to the line “Set $s \leftarrow v_i$” in the first Otherwise block. Since all other symbols are read once during an iteration of the Repeat block, the number of iterations of this block is the number of input symbols plus the number of right parentheses. Since the number of operators ($m$) must be at least one fifth of the number of input symbols, the number of iterations of the Repeat block is no greater than $10m$. Since all lines of the Repeat block require unit time, the complexity of the block is $O(m)$.

The second Repeat block checks items in $L$, in order, and for each makes one of a fixed number of substitutions in which the number of variables is bounded by a constant. An item is appended to $L$ every time a $\neg$ or right parenthesis is encountered and there is one right parenthesis for every binary operator. Thus there are $m$ items in $L$ and the complexity of the second Repeat block is $O(m)$. Therefore, the complexity of the algorithm is $O(m)$. \qed

If $\phi$ contains duplicate subformulas, the output will consist of more variables and CNF blocks than are needed to represent a CNF formula equivalent to $\phi$. This is easily remedied by implementing $L$ as a balanced binary search tree keyed on $v_l$, $v_r$, and $O$ and changing the Append to a tree insertion operation. If, before insertion in the “Evaluate ‘$(vOw)$’” block, it is discovered that ‘$v_j \Leftrightarrow (vOw)$’ already exists in $L$ for some $v_j$, then $s$ can be set to $v_j$, $i$ need not be incremented, and the insertion can be avoided. A similar change can be made for the “Evaluate ‘$\neg s$’” block. With $L$ implemented as a balanced binary search tree, each query and each insertion would take at most $\log(m)$ time since $L$ will contain no more than $m$ items. Hence, the complexity of the algorithm with the improved data structure would be $O(m \log(m))$.

We remark that there is no comparable, efficient transformation from a well-formed formula to a DNF formula.

3.2 Resolution

Resolution is a general procedure that is primarily used to certify that a given CNF formula is unsatisfiable, but can also be used to find a model, if one exists. The idea originated as consensus in [17] and [122] (ca. 1937) and was applied to DNF formulas in a form exemplified by the following:

$$(x \land y) \lor (\neg x \land z) \Leftrightarrow (x \land y) \lor (\neg x \land z) \lor (y \land z)$$
Consensus in DNF was rediscovered in [116] and [111] (ca. 1954) where it was given its name. A few years later its dual, as resolution in propositional logic, was applied to CNF [45] (1958), for example:

\[(x \lor y) \land (\neg x \lor z) \iff (x \lor y) \land (\neg x \lor z) \land (y \lor z)\]

The famous contribution of Robinson [113] in 1965 was to lift resolution to first-order logic.

Let \(c_1\) and \(c_2\) be disjunctive clauses such that there is exactly one variable \(v\) that occurs negated in one clause and unnegated in the other. Then, the resolvent of \(c_1\) and \(c_2\), denoted by \(R_{c_1}^{c_2}\), is a disjunctive clause which contains all the literals of \(c_1\) and all the literals of \(c_2\) except for \(v\) or its complement. The variable \(v\) is called a pivot variable. If \(c_1\) and \(c_2\) are treated as sets, their resolvent is \(\{l : l \in c_1 \cup c_2 \setminus \{v, \neg v\}\}\).

The usefulness of resolvents derives from the following Lemma.

**Lemma 17.** Let \(\psi\) be a CNF formula represented as a set of sets. Suppose there exists a pair \(c_1, c_2 \in \psi\) of clauses such that \(R_{c_1}^{c_2} \notin \psi\) exists. Then \(\psi \iff \psi \cup \{R_{c_1}^{c_2}\}\).

**Proof.** Clearly, any assignment that does not satisfy CNF formula \(\psi\) cannot satisfy a CNF formula which includes a subset of clauses equal to \(\psi\). Therefore, no satisfying assignment for \(\psi\) implies none for \(\psi \cup \{R_{c_1}^{c_2}\}\).

Now suppose \(M\) is a model for \(\psi\). Let \(v\) be the pivot variable for \(c_1\) and \(c_2\). Suppose \(v \in M\). One of \(c_1\) or \(c_2\) contains \(\neg v\). That clause must also contain a literal that has value 1 under \(M\) or else it is falsified by \(M\). But, that literal exists in the resolvent \(R_{c_1}^{c_2}\), by definition. Therefore, \(R_{c_1}^{c_2}\) is satisfied by \(M\) and so is \(\psi \cup \{R_{c_1}^{c_2}\}\). A similar argument applies if \(v \notin M\).

The resolution method makes use of Lemma 17 and the fact that a clause containing no literals cannot be satisfied by any truth assignment. A resolution algorithm for CNF formulas is presented in Figure 3.3. It uses the notion of pure literal (Page 27) to help find a model, if one exists. Recall, a literal is pure in formula \(\psi\) if it occurs in \(\psi\) but its complement does not occur in \(\psi\). If the algorithm outputs “unsatisfiable” then the set of all resolvents generated by the algorithm is a resolution refutation for the input formula.

**Theorem 18.** Let \(\psi\) be a CNF formula represented as a set of sets. The output of Algorithm 2 on input \(\psi\) is “unsatisfiable” if and only if \(\psi\) is unsatisfiable. If the output is a set of variables, then it is a model for \(\psi\).

**Proof.** If the algorithm returns “unsatisfiable” one resolvent is the empty clause. From Lemma 17 and the fact that an empty clause cannot be satisfied, \(\psi\) is unsatisfiable.

If the algorithm does not return “unsatisfiable” the **Otherwise** block is entered because new resolvents cannot be generated from \(\psi\). Next, a Repeat
block to remove clauses containing pure literals is executed, followed by a final Repeat block which adds some variables to $M$.

Consider the result of the Repeat block on pure literals. All the clauses removed in this block are satisfied by $M$ because all variables associated with negative pure literals are absent from $M$ and all variables associated with positive pure literals are in $M$. Call the set of clauses remaining after this Repeat block $\psi'$. Now, consider the final Repeat block. If $\psi' = \emptyset$ then all clauses are satisfied by $M$ from the previous Repeat block, and $M_{1,1}$ will never falsify a clause for any $i$. In that case $M$ is returned and is a model for $\psi$. So, suppose $\psi' \neq \emptyset$. The only way a clause can be falsified by $M_{1,1}$ is if it is $\{v_1\}$. In that case, the line “Set $M \leftarrow M \cup \{v_1\}$” changes $M$ to satisfy that clause. The clause $\{\neg v_1\} \notin \psi'$ because otherwise it would have resolved with $\{v_1\}$ to give $\emptyset \in \psi$ which violates the hypothesis that the pure literal Repeat block was entered. Therefore, no clauses are falsified by $M_{1,1}$ at the beginning of the second iteration of the Repeat block when $M_{1,2}$ is considered.

Assume the general case that no clause is falsified for $M_{1,i-1}$ at the beginning of the $i^{th}$ iteration of the final Repeat block. A clause $c_1$ will be falsified by $M_{1,1}$ but not by $M_{1,i-1}$ if it contains literal $v_i$, which cannot be a pure literal that was processed in the previous Repeat block. Then, the line “Set $M \leftarrow M \cup \{v_i\}$” changes $M$ to satisfy $c_1$ without affecting any previously satisfied clauses. However, we must consider the possibility that a clause that was not previously satisfied becomes falsified by the change. If there were such a clause $c_2$ it must contain literal $\neg v_i$ and no other literal in $c_2$ would have a complement in $c_1$ otherwise it would already have been satisfied by $M_{1,i-1}$. That means $\psi'$, before the pure literal Repeat block, would contain $R^{c_1}c_2$. Moreover, $R^{c_1}_2$ could not be satisfied by $M_{1,i-1}$ because such an assignment would have satisfied $c_1$ and $c_2$. But then $R^{c_1}_2$ must be falsified by $M_{1,i-1}$ because it contains literals associated only with variables $v_1, \ldots, v_{i-1}$. But this is impossible by the inductive hypothesis. It follows that the line “Set $M \leftarrow M \cup \{v_i\}$” does not cause any clauses to be falsified and that no clause is falsified for $M_{1,i}$. Since no clause is falsified by $M_{1,n}$, all clauses must be satisfied by $M$ which is then a model. The statement of the Theorem follows.

Resolution is a powerful tool for mechanizing the solution to variants of the Satisfiability problem. However, in the case of an unsatisfiable input formula, since the resolution algorithm offers complete freedom to choose which pair of clauses to resolve next, it can be difficult to control the total number of clauses resolved and therefore the running time of the algorithm. The situation in the case of satisfiable input formulas is far worse since the total number of resolvents generated can be quite large even when a certificate of satisfiability can be generated quite quickly using other methods.

It is easy to find examples of formulas on which the resolution algorithm may perform poorly or well, depending on the order of resolvents, and we present one infinite family here. We use $n$ “normal” variables
Algorithm 2.

Resolution ($\psi$)
/* Input $\psi$ is a set of sets of literals representing a CNF formula */
/* Output is either “unsatisfiable” or a model for $\psi$ */
/* Locals: set of variables $M$ */

Set $M \leftarrow \emptyset$.
Repeat the following until some statement below outputs a value:
If $\emptyset \in \psi$, Output “unsatisfiable.”
If there are two clauses $c_1, c_2 \in \psi$ such that $R_{c_1}^{c_2} \notin \psi$ exists,
Set $\psi \leftarrow \psi \cup \{R_{c_1}^{c_2}\}$.
Otherwise,
Repeat the following while there is a pure literal $l$ in $\psi$:
If $l$ is a positive literal, Set $M \leftarrow M \cup \{l\}$.
Set $\psi \leftarrow \{c : c \in \psi, l \notin c\}$.
Arbitrarily index all variables in $V_\psi$ as $v_1, v_2, \ldots, v_n$.
Repeat the following for $i$ from 1 to $n$:
If $M_{1;i}$ falsifies some clause in $\psi$, Set $M \leftarrow M \cup \{v_i\}$.
Output $M$.

Figure 3.3: Resolution algorithm for CNF formulas.

{$v_0, v_1, \ldots, v_{n-1}$} and $n - 1$ “selector” variables {$z_1, z_2, \ldots, z_{n-1}$}. Each clause contains one normal variable and one or two selector variables. For each $0 \leq i \leq n - 1$ there is one clause with $v_i$ and one with $\neg v_i$. The selector variables allow the clauses to be “chained” together by resolution to form any $n$-literal clause consisting of normal variables. Therefore, the number of resolvents generated could be as high as $2^{O(n)}$ although the input length is $O(n)$. The family of input formulas is specified as follows:

{$\{v_0, \neg z_1\}, \{z_1, v_1, \neg z_2\}, \{z_2, v_2, \neg z_3\}, \ldots, \{z_{n-2}, v_{n-2}, \neg z_{n-1}\},$
{$z_{n-1}, v_{n-1}\}, \{\neg v_0, \neg z_1\}, \{z_1, \neg v_1, \neg z_2\}, \{z_2, \neg v_2, \neg z_3\}, \ldots,$
{$z_{n-2}, \neg v_{n-2}, \neg z_{n-1}\}, \{z_{n-1}, \neg v_{n-1}\}$}.

The number of resolvents generated by the resolution algorithm greatly depends on the order in which clauses are chosen to be resolved. Resolving vertically, the 0th column adds resolvent {$\neg z_1$}. This resolves with the clauses of column 1 to add {$v_1, \neg z_2$} and {$\neg v_1, \neg z_2$} and these two add resolvent {$\neg z_2$}. Continuing, resolvents {$\neg z_3$}, \ldots, {$\neg z_{n-1}$} are added. Finally, {$\neg z_{n-1}$} resolves with the two clauses of column $n - 1$ to generate resolvent $\emptyset$ showing that the formula is unsatisfiable. The total number of resolutions executed is $O(n)$ in this case. On the other hand, resolving horizontally
(resolve one clause of column 0 with a clause of column 1, then resolve the resolvent with a clause from column 2, and so on), all \(2^n\) \(n\)-literal clauses of "normal" variables can be generated before \(\emptyset\) is.

The number of resolution steps executed on a satisfiable formula can be outrageous. Remove column \(n-1\) from the above formulas. They are then satisfiable. But that is not known until \(2^{O(n)}\) resolutions fail to generate \(\emptyset\).

This example may suggest that something is wrong with resolution and that it should be abandoned in favor of faster algorithms. While this is reasonable for satisfiable formulas, it may not be for unsatisfiable formulas. As illustrated by Theorem 20, in the case of providing a certificate of unsatisfiability, the resolution algorithm can always "simulate" the operation of many other algorithms in time bounded by a polynomial on input length. So, the crucial problem for the resolution algorithm, when applied to unsatisfiable formulas, is determining the order in which clauses should be resolved to keep the number of resolvents generated at a minimum. A significant portion of this monograph deals with this question. However, we first consider an extension to resolution that has shown improved ability to admit short refutations.

### 3.3 Extended Resolution

In the previous section we showed that the size of a resolution refutation can vary enormously depending on the order resolvents are formed. That is, for at least one family of formulas there are both exponentially long and linearly long refutations. But for some families of formulas nothing shorter than exponential size resolution refutations is possible. Examples, such as the pigeon hole formulas, will be presented in Section 6.2. However, the simple idea of extended resolution can yield short refutations in such cases.

Extended resolution is based on a result of Tseitin [129] who showed that, for any pair of variables \(v, w\) in a given CNF formula \(\psi\), the following expression may be appended to \(\psi\):

\[
(z \lor v) \land (z \lor w) \land (\neg z \lor \neg v \lor \neg w)
\]  

(3.1)

where \(z\) is a variable that is not in \(\psi\). From Figure 3.1 this is equivalent to:

\[
z \leftrightarrow (\neg v \lor \neg w)
\]

which means either \(v\) and \(w\) both have value 1 (then \(z = 0\)) or at least one of \(v\) or \(w\) has value 0 (then \(z = 1\)). Observe that as long as \(z\) is never used again, any of the expressions of Figure 3.1 can be used in place of or in addition to (3.1). More generally,

\[
z \leftrightarrow f(v_1, v_2, \ldots, v_k)
\]
can be used as well where $f$ is any Boolean function of arbitrary arity. Judicious use of such extensions can result in polynomial size refutations for problems that have no polynomial size refutations without extension, a notable example being the pigeon hole formulas as will be shown in Section 6.3.

By adding variables not in $\psi$ one obtains, in linear time, a satisfiability-preserving translation from any propositional expression to CNF with at most a constant factor blowup in expression size as shown in Section 3.1.

### 3.4 Davis-Putnam Resolution

The Davis-Putnam procedure (DPP) [46] is presented here mainly for historical reasons. In DPP, a variable $v$ is chosen and then all resolvents with $v$ as pivot are generated. When no more such resolvents can be generated, all clauses containing $v$ or $\neg v$ are removed and the cycle of choosing a variable, generating resolvents, and eliminating clauses is repeated to exhaustion. The fact that it works is due to the following result.

**Lemma 19.** Let $\psi$ be a CNF formula. Perform the following operations:

$$
\psi_1 \leftarrow \psi.
$$

Choose any variable $v$ from $\psi_1$.

Repeat the following until no new resolvents with $v$ as pivot can be added to $\psi_1$:

- If there is a pair of clauses $c_1, c_2 \in \psi_1$ such that $R_{c_1}^c$ with $v$ as pivot exists and $R_{c_2}^c \notin \psi_1$,

$$
\psi_1 \leftarrow \psi_1 \cup \{R_{c_2}^c\}.
$$

$$
\psi_2 \leftarrow \psi_1.
$$

Repeat the following while there is a clause $c \in \psi_2$ such that $v \in c$ or $\neg v \in c$:

$$
\psi_2 \leftarrow \psi_2 \setminus \{c\}.
$$

Then $\psi$ is satisfiable if and only if $\psi_2$ is satisfiable.

**Proof.** By Lemma 17 $\psi_1$ is functionally equivalent to $\psi$. Since removing clauses cannot make a satisfiable formula unsatisfiable, if $\psi$ and therefore $\psi_1$ is satisfiable, then so is $\psi_2$.

Now, suppose $\psi$ is unsatisfiable. Consider any pair of assignments $M$ (without $v$) and $M \cup \{v\}$. Either both assignments falsify some clause not containing $v$ or $\neg v$ or else all clauses not containing $v$ or $\neg v$ are satisfied by both $M$ and $M \cup \{v\}$. In the former case, some clause common to $\psi$ and $\psi_2$ is falsified by $M$ so both formulas are falsified by $M$. In the latter case, $M$ must falsify a clause $c_1 \in \psi$ containing $v$ and $M \cup \{v\}$ must falsify a clause containing $\neg v$. Then the resolvent $R_{c_2}^c \in \psi_2$ is falsified by $M$. Therefore, since every assignment falsifies $\psi$, $\psi_2$ is unsatisfiable. \qed
3.5 Davis-Putnam Loveland Logemann Resolution

Despite the apparent improvement in the management of clauses over the resolution algorithm, DPP is not considered practical. However, it has spawned some other commonly used variants, especially the next algorithm to be discussed.

Here we present a reasonable implementation of the key idea in DPP which is to repeat the following: choose a variable, take care of all resolutions due to that variable, and then erase clauses containing it. The algorithm, called DPLL [47], is shown in Figure 3.4. It was developed when Loveland and Logemann attempted to implement DPP but found that it used too much RAM. So they changed the way variables are eliminated by employing the splitting rule: assignments are recursively extended by one variable in both possible ways looking for a satisfying assignment. Thus, DPLL is of the divide-and-conquer family.

The DPLL algorithm of Figure 3.4 is written iteratively instead of recursively for better comparison with other algorithms to be discussed later. We omit a formal proof of the fact that the output of DPLL on input $\psi$ is “unsatisfiable” if and only if $\psi$ is unsatisfiable, and if the output is a set of variables, then it is a model for $\psi$. The reader may consult [47] for details.

A common visualization of the flow of control of DPLL and similar algorithms involves a graph structure known as a search tree. Since we will use search trees several times in this monograph to assist in making some difficult points clear, we take the trouble of explaining the concept here. A search tree is a rooted acyclic digraph where each vertex has out degree at most two, and in degree one except for the root. Each internal vertex represents some Boolean variable and each leaf may represent a clause or may have no affiliation. If an internal vertex represents variable $v$ and it has two outward oriented edges then one is labeled $v$ and the other is labeled $\neg v$; if it has one outward oriented edge, that edge is labeled either $v$ or $\neg v$. All vertices encountered on a path from root to a leaf represent distinct variables. The labels of edges on such a path represent a truth assignment to those variables: $\neg v$ means set the value of $v$ to 0 and $v$ means set the value of $v$ to 1.

The remaining details are specific to a particular CNF formula $\psi$ which is input to the algorithm modeled by the search tree. A leaf is such that the partial truth assignment represented by the path from the root either minimally satisfies all clauses of $\psi$, or minimally falsifies at least one clause of $\psi$. In the latter case, one of the clauses falsified becomes the label for the leaf. If $\psi$ is unsatisfiable, all leaves are labeled and the search tree is a refutation tree. A fully labeled refutation tree modeling one possible run of DPLL on a particular unsatisfiable CNF formula is shown in Figure 3.5. With reference to Figure 3.4, a path from the root to any vertex represents
Algorithm 3.

**DPLL ($\psi$)**

- **Input** $\psi$ is a set of sets of literals representing a CNF formula
- **Output** is either “unsatisfiable” or a model for $\psi$
- **Locals:** integer $d$, variable stack $V_P$, list of formulas $\psi_0, \psi_1, \ldots$
  - partial assignments $M_{1:d}$.

Set $d \leftarrow 0$; Set $M_{1:0} \leftarrow \emptyset$; Set $\psi_0 \leftarrow \psi$; Set $V_P \leftarrow \emptyset$.

Repeat the following until some statement outputs a value:

1. If $\psi_d = \emptyset$, Output $M_{1:d}$.
2. If $\emptyset \in \psi_d$, Repeat the following until $l$ is “tagged.”
   - If $V_P = \emptyset$, Output “unsatisfiable.”
   - Pop $l \leftarrow V_P$.
   - Set $d \leftarrow d - 1$.
   - Push $V_P \leftarrow \neg l$. /* $l$ and $\neg l$ are not tagged */
3. If $l$ is a negative literal then $M_{1:d+1} \leftarrow M_{1:d} \cup \{-l\}$.
   Otherwise, $M_{1:d+1} \leftarrow M_{1:d}$.
   Set $\psi_{d+1} \leftarrow \{c - \{-l\} : c \in \psi_d, l \notin c\}$.
   Set $d \leftarrow d + 1$.

Otherwise,

1. If there exists a pure literal in $\psi_d$,
   Choose a pure literal $l$.
2. Otherwise, if there is a unit clause in $\psi_d$,
   Choose a literal $l$ in a unit clause and “tag” $l$.
3. Otherwise,
   Choose a literal $l$ in $\psi_d$ and “tag” $l$.

Push $V_P \leftarrow l$.

If $l$ is a positive literal, $M_{1:d+1} \leftarrow M_{1:d} \cup \{l\}$.
Otherwise, $M_{1:d+1} \leftarrow M_{1:d}$.
Set $\psi_{d+1} \leftarrow \{c - \{-l\} : c \in \psi_d, l \notin c\}$.
Set $d \leftarrow d + 1$.

\[ \square \]

Figure 3.4: DPLL algorithm for CNF formulas.
the state of $V_P$ at some point in the run. For a vertex with out degree two, the left edge label is a "tagged" literal. For vertices with out degree one, the single edge label is a "pure" literal.

The DPLL algorithm is a performance compromise. Its strong point is sets of resolvents are constructed incrementally, allowing some sets of clauses and resolvents to be entirely removed from consideration long before the algorithm terminates. But this is offset by the weakness that some freedom in choosing the order of forming resolvents is lost, so more resolvents may have to be generated. Applied to a satisfiable formula, DPLL can be a huge winner over the resolution algorithm: if the right choices for variable elimination are made, models may be found in $O(n)$ time. However, for an unsatisfiable formula it is not completely clear which algorithm performs best generally. Theorem 20 below shows that if there is a short DPLL refutation for a given unsatisfiable formula, then there must be a short resolution refutation for the same formula. But, Theorem ?? below shows that it is possible for a shortest resolution proof to be exponentially smaller than the smallest DPLL refutation on a given formula. These facts seem to give the edge to resolution. On the other hand, it may actually be relatively easy to find a reasonably short DPLL refutation but very hard to produce an equally short or shorter resolution refutation. An important determining factor is the order in which variables or resolutions are chosen and the best

\[ (v_0 \lor v_4) \land (v_0 \land \neg v_4) \land (v_0 \lor v_1) \land (v_1 \lor \neg v_2) \land (\neg v_2 \lor v_3) \land \]
\[ (\neg v_0 \lor \neg v_1 \lor \neg v_2) \land (\neg v_0 \land \neg v_2 \land \neg v_3) \land (v_2 \lor \neg v_3) \land (v_2 \lor v_3) \]

Figure 3.5: A DPLL refutation tree for the above CNF formula.

---

\[ ^{11}\text{See} \ [38] \text{for other results along these lines.} \]
order is often sensitive to structural properties of the given formula (more on structural properties of formulas is found in Section ??). For this reason, quite a number of choice heuristics have been studied and we present some results on these later in this monograph.

We finish this section with the following classic result.

**Theorem 20.** Suppose DPLL is applied to a given unsatisfiable CNF formula \( \psi \) and suppose the two lines “Set \( \psi_{d+1} = \{c - \{\neg l\} : c \in \psi_d, l \notin c\} \)” are together executed \( p \) times. Then there is a resolution refutation in which no more than \( p/2 \) resolvents are generated.

**Proof.** We will run DPLL on \( \psi \) and, in parallel, generate resolvents from clauses of \( \psi \) and other resolvents. At most one resolvent will be generated every time the test “If \( V_P = \emptyset \)” succeeds or the line “Pop \( l \leftarrow V_P \)” is executed and \( l \) is neither tagged nor a pure literal. But this is at most half the number of times \( \psi_{d+1} \) is set. Hence, when DPLL terminates on an unsatisfiable formula, at most \( p/2 \) resolvents will be generated. Next, we show how to generate resolvents and show that \( \emptyset \) is a resolvent of two of them.

The idea can be visualized as a series of destructive operations on a DPLL refutation tree. Assume clauses in \( \psi \) label leaves as discussed on Page 73. Repeat the following: if there are two leaf siblings, replace the subtree of the two siblings and their parent either with the resolvent of the leaf clauses, if they resolve, or with the leaf clause that is falsified by the assignment represented by the path from root to the parent (there must be one if \( \psi \) is unsatisfiable); otherwise there is a pair of vertices consisting of one leaf and a parent so replace this pair with the leaf. Replacement entails setting the edge originally pointing to the parent to point to the replacement vertex. Eventually, the tree consists of a root only and its label is the empty clause.

This visualization is implemented as follows. Define a stack \( S \) and set \( S \leftarrow \emptyset \). The stack will hold clauses of \( \psi \) and resolvents. Run DPLL on \( \psi \). In parallel, manipulate \( S \) as follows:

1. The test “If \( \emptyset \in \psi_d \)”:
   Whenever this test succeeds, there is a clause \( c \in \psi \) whose literals can only be complements of those in \( V_P \) so push \( S \leftarrow c \).

2. The line “Pop \( l \leftarrow V_P \)”:
   Immediately after execution of this line, if \( l \) is not “tagged,” and is not a pure literal then do the following. Pop \( c_1 \leftarrow S \) and pop \( c_2 \leftarrow S \). If \( c_1 \) and \( c_2 \) can resolve, push \( S \leftarrow R_{c_1c_2} \); otherwise, at least one of \( c_1 \) or \( c_2 \), say \( c_1 \), does not contain \( l \) or \( \neg l \), so push \( S \leftarrow c_1 \).
3. The line “If $V_P = \emptyset$, Output “unsatisfiable.””:

The algorithm terminates so pop $c_1 \leftarrow S$, pop $c_2 \leftarrow S$, and form the resolvent $c_t = R_{c_1}^{c_2}$.

We claim that, when the algorithm terminates, $c_t = \emptyset$.

To prove this claim we show that when $c_1$ and $c_2$ are popped in Step 2, $c_2$ contains literals that are only complementary to those of a subset of $V_P \cup \{l\}$ and $c_1$ contains literals that are only complementary to those of a subset of $V_P \cup \{-l\}$ and if $l$ is a pure literal in the same step then $c_1$ contains literals complementary to those of a subset of $V_P$. Since $V_P = \emptyset$ when $c_1$ and $c_2$ are popped to form the final resolvent, $c_t$, it follows that $c_t = \emptyset$.

By induction on the maximum depth of stacking of $c_2$ and $c_1$. The base case has $c_2$ and $c_1$ as clauses of $\psi$. This can happen only if $c_2$ had been stacked in Step 1 then Step 2 was executed and then $c_1$ was stacked soon after in Step 1. It is straightforward to check that the hypothesis is satisfied in this case.

Suppose the hypothesis holds for maximum depth up to $k$ and consider a situation where the maximum depth of stacking is $k + 1$. There are two cases. First, suppose $l$ is a pure literal. Then neither $l$ nor $\neg l$ can be in $c_1$ so, by the induction hypothesis, all of the literals of $c_1$ must be complementary to literals of $V_P$. Second, suppose $l$ is not “tagged” and not a pure literal. Then $c_1$ and $c_2$ are popped. By the induction hypothesis, $c_1$ contains only literals complementary to some in $V_P \cup \{-l\}$ and $c_2$ contains only literals complementary to some in $V_P \cup \{l\}$. Therefore, if they resolve, the pivot must be $l$ and the resolvent contains only literals complementary to some in $V_P$. If they don’t resolve, by the induction hypothesis, one must not contain $l$ or $\neg l$. That one has literals only complementary to those of $V_P$, so the hypothesis holds in this case.

\[\square\]

3.6 Decompositions

The variable elimination methods of Sections 3.5 and 3.7 recursively decompose a given formula $\psi$ into overlapping subformulas $\psi_1$ and $\psi_2$ such that the solution to $\psi$ can be inferred from the solutions to $\psi_1$ and $\psi_2$. The decompositions are based on occurrences of a selected variable $v$ in clauses of $\psi$ and each subformula has at least as many clauses as those of $\psi$ which contain neither literal $v$ nor literal $\neg v$. Intuitively, the speed of the methods usually depends on the magnitude of the size reduction from $\psi$ to $\psi_1$ and $\psi_2$. However, it is often the case that the number of occurrences of most variables in a formula or subformula is small which usually means small size reductions for most variables. For example, the average number of occurrences of a variable in a random $k$-SAT formula of $m$ clauses developed from $n$ variables is $km/n$ (see Section ??): so, if $k = 3$ and $m/n$ is, say 4 (see Section ??), then the average number of occurrences of a randomly chosen
variable is only 12. Hence, the methods often suffer computational inade-
quacies which can make them unusable in some cases. On the positive side, 
such methods can be applied to any CNF formula.

But there are other decomposition methods that sacrifice some general-
ity for the sake of producing subformulas of relatively small size. Trumpe r's 
book [128] presents quite a few of these, all of which are capable of computationally efficient solutions to some problems that would be considered
difficult for the more general variable elimination methods. This section 
presents one of these, called monotone decomposition, for illustration and 
because it is related to material that is elsewhere in this monograph.

3.6.1 Monotone Decomposition

Let CNF formula $\psi$ of $m$ clauses and $n$ variables be represented as a $m \times n$
$(0, \pm 1)$-matrix $M_\psi$. A monotone decomposition of $M_\psi$, is a permutation of
rows and columns of $M_\psi$ and the multiplication by $-1$ of some or all of its
columns, referred to below as a column scaling, resulting in a partition into
four submatrices as follows:

$$
\begin{pmatrix}
A_1 & E \\
D & A_2
\end{pmatrix}
$$

(3.2)

where the submatrix $A_1$ has at most one +1 entry per row, the submatrix
$D$ contains only $-1$ or 0 entries, the submatrix $A_2$ has no restrictions other
than the three values of $-1$, +1, and 0 for each entry, and the submatrix $E$
has only 0 entries.

The submatrix $A_1$ represents a Horn Formula. In Section 4.2 Horn formu-
las are shown to have the following two important properties: they are
solved efficiently, for example by Algorithm 17 of Figure 4.3, and, by The-
orem 29, there is always a unique minimum model for a satisfiable Horn
formula. The second property means there is always a satisfying assignment
$M$ such that, for any other satisfying assignment $M'$, the variables that have
value 1 according to $M'$ are a superset of the variables set to 1 according
to $M$ (more succinctly, $M \subseteq M'$). This property, plus the nature of sub-
matrices $D$ and $E$, effectively allow a split of the problem of determining
the satisfiability of $\psi$ into two independent problems: namely, determine sasifiability for the Horn formula represented by $A_1$ and determine sasifiability for the subformula represented by $A_2$. The algorithm of Figure 3.6
shows this in more detail. The following theorem proves correctness of this
algorithm.

**Theorem 21.** Let CNF formula $\psi$ be represented as a monotone decomposi-
tion $(0, \pm 1)$-matrix. On input $\psi$, Algorithm 4 outputs “unsatisfiable” if and
only if $\psi$ is unsatisfiable and if $\psi$ is satisfiable, then the output set $M_1 \cup M_2$
is a model for $\psi$. 
Algorithm 4.

Monotone Decomposition Solver (ψ)
/* Input: CNF formula ψ as (0, ±1) Mψ monotone decomposition */
/* Output: “unsatisfiable” or a model for ψ */
/* Locals: set of variables M1, M2 */

Let Mψ be partitioned according to (3.2).
If Horn formula A1 is unsatisfiable, Output “unsatisfiable.”
Let M1 be a unique minimum model for the Horn formula A1.
Remove from A2 all rows common to D’s that are satisfied by M1.
If A2 is unsatisfiable, Output “unsatisfiable.”
Let M2 be a model for A2.
Output M1 ∪ M2.

Figure 3.6: Algorithm for determining satisfiability of a Monotone Decomposition.

Proof. Clearly, if Horn formula A1 is unsatisfiable then so is ψ. So, suppose there is a model M1 for A1 and consider the rows of A2 remaining after rows common to those of D which are satisfied by M1 are removed. Since M1 is a unique minimum model for A1, no entries of D are +1, and variables of A1 are distinct from variables of A2, no remaining row of A2 can be satisfied by any model for A1. Therefore, if these rows of A2 are unsatisfiable, then so is ψ. On the other hand, if these rows are satisfied by model M2, then clearly, M1 ∪ M2 is a model for ψ.

A (0, ±1) matrix Mψ representing CNF formula ψ may have more than one monotone decomposition. We are interested in the maximum monotone decomposition of Mψ. That is, we wish to find the monotone decomposition of ψ such that A1 has the greatest number of rows and columns. A monotone decomposition is said to be maximal with respect to the dimensions of A1.

The following theorem says a unique maximal monotone decomposition is always possible.

Theorem 22. Any (0, ±1) matrix M has a unique maximal monotone decomposition.

Proof. Suppose M has two distinct maximal monotone decompositions, say M1 and M2. Let A1i, A2i, and D, i ∈ {1, 2}, be the partition of M, after column scaling, corresponding to M1 (see the partition (3.2) on page 78).

Construct a new partition M′ of M into A′1, A′2 and D′ such that A′1 includes all rows and columns of A1i and A2i. For those columns of M′ that are also columns of A1i use a column scaling that is exactly the same as the one used in M1. For all other columns use the same scaling as in M2. The
submatrix of $A^1$ that includes rows and columns of $A^1_1$ is the same as $A^1_1$ because the scaling of those columns is the same as for $M_1$. The submatrix of $A^1$ including rows of $A^1_1$ and columns not in $A^1_1$ must be a 0 submatrix by the monotone decomposition $M_1$. The submatrix of $A^1$ including columns of $A^1_1$ and no rows of $A^1_1$ must contain only 0 or -1 entries due to the $M_1$ scaling and the submatrix including neither columns or rows of $A^1_1$ must be Horn due to $M_2$ column scaling. It follows that submatrix $A^1$ is Horn (at most one +1 in each row). It is similarly easy to check that the submatrix of $M^\prime$ consisting of rows of $A^1$ and columns other than those of $A^1$ is 0 and that the submatrix of $M^\prime$ consisting of columns of $A^1$ and rows other than those of $A^1$ contains no +1 entries. It follows that $M^\prime$ is a monotone decomposition. Since $A^1_1 \supset A^1$ and $A^1_2 \supset A^1$ neither $M_1$ nor $M_2$ is a maximal monotone decomposition in violation of the hypothesis. The theorem follows.

From Theorem 22 there is always a maximum monotone decomposition for $M_\psi$.

A maximum monotone decomposition is useful because: 1) $A^1$, representing a Horn formula, is as large as possible so $A^2$ is as small as possible; 2) Horn formulas may be efficiently solved by Algorithm 17; and 3) a maximum monotone decomposition can be found efficiently, as will now be shown.

A maximum monotone decomposition can be found using Algorithm 5 of Figure 3.7. The algorithm completes one or more stages where each stage produces a proper monotone decomposition of some matrix. All submatrices change dimensions during the algorithm so we use primes as in $E^\prime$ to refer to the current incantation of corresponding submatrices. Initially, that matrix is $M^\prime$. At the end of a stage, if the algorithm needs another stage to produce a bigger decomposition, $A^2$ of the current stage becomes the entire input of the next stage and the next stage proceeds independently of previous stages. This can be done since the operation to be mentioned next does not multiply by -1 any of the rows and columns of the $A^1$ and $D^\prime$ matrices of previous stages. The important operation is to move a non-positive column that intersects $A^2$ to just right of the border of the current stage’s $A^1$ matrix, move the border of $A^1$ and $D^\prime$ to the right by one column, tag and move the rows containing 1 on the right boundary of the changed $D^\prime$ up to just below the border of $A^1$, and finally lower the border of $A^1$ and $E^\prime$ down to include the tagged rows. Doing so keeps $A^1$ Horn and $D^\prime$ non-positive and enlarges $A^1$. If no non-positive column exists through $A^2$, no column can be made non-positive through $A^2$ by a -1 multiplication, and the initial moved column is not multiplied by -1, then the initial moved column of the stage is multiplied by -1 and the stage is restarted.

Because of the following theorem, backtracking is limited to just one per stage and is used only to try to decompose with the initial moved column of the stage multiplied by -1.
Theorem 23. Refer to Algorithm 5 for specific variable names and terms.

1. If \( z \) is not a non-positive column in \( \mathcal{E}' \), and \( z \) multiplied by -1 is not non-positive in \( \mathcal{E}' \), then there is no monotone decomposition at the current stage with the initial moved column \( v \) of the stage left as is.

2. If multiplying \( v \) by -1 also fails because \( z \) cannot be made non-positive in \( \mathcal{E}' \), then not only does \( z \) block a monotone decomposition but multiplying any of the other columns in \( \mathcal{A}' \) except \( v \) by -1 blocks a monotone decomposition as well.

Proof.
1. There is no way to extend the right boundary of \( \mathcal{A}' \) and stay Horn while making \( \mathcal{E}' \) 0 because column \( z \) prevents it.

2. Consider columns in \( \mathcal{A}' \) first. The proof is by induction on the number of columns processed in \( \mathcal{A}' \). The base case has no such column: that is, \( \mathcal{A}' \) only contains the column \( v \), and is trivially satisfied. For the inductive step, change the column scaling to 1 for all columns and run the algorithm in the same column order it had been when it could not continue. Assume the hypothesis holds to \( k \) columns and consider processing at the \( k + 1 \)st column, call it column \( x \). At this point \( \mathcal{A}' \) has one 1 in each row, \( \mathcal{D}' \) is non-positive, and since \( x \) is multiplied by 1, it is non-zero and non-positive through \( \mathcal{E}' \). If there is a monotone decomposition where \( x \) is multiplied by -1, then \( x \) goes through \( \mathcal{A}' \) of that decomposition. The multiplication by -1 changes the non-zero non-positive elements of \( x \) through \( \mathcal{E}' \) to non-zero non-negative elements. Therefore, at least one of these elements, say in row \( r \), is +1. But \( \mathcal{A}' \) of the decomposition must have a +1 in each row so it must be that row \( r \) has this +1 in say column \( c \), a column of \( \mathcal{A}' \) that is multiplied by -1. But \( c \) cannot be the same as \( v \) since \( v \) multiplied by -1 blocks a monotone decomposition by hypothesis. On the other hand, if \( c \) is not \( v \), then by the inductive hypothesis \( c \) cannot be multiplied by -1 in a monotone decomposition. Therefore, by contradiction, there is no monotone decomposition and the hypothesis holds to \( k + 1 \) columns.

Now consider column \( z \). No scaling of column \( z \) can make \( z \) non-positive in \( \mathcal{E}' \). Then that part of \( z \) that goes through \( \mathcal{E}' \) has -1 and 1 entries. The hypothesis follows from the same induction argument as above.

Any column blocking a monotone decomposition need never be checked again.

The algorithm keeps track of blocking columns with set \( N \), the long term record of blocking columns, and set \( L \), the temporary per stage record. If column indicator \( w \) is placed in \( N \) it means the unmultiplied column \( w \) blocks, and if \( \neg w \) is placed in \( N \) it means column \( w \) multiplied by -1 blocks.

The algorithm has quadratic complexity. Complexity can be made linear by running the two possible starting points of each stage, namely using column \( v \) as is and multiplying column \( v \) by -1, concurrently and breaking off computation when one of the two succeeds.
Algorithm 5.

Find Maximum Monotone Decomposition ($\psi$)
/* Input: CNF formula $\psi$ as (0,±1) matrix $\mathcal{M}_\psi$ */
/* Output: A maximum monotone decomposition of $\mathcal{M}_\psi$ */
/* Locals: set of variables $M_1, M_2$, set of unusable literals $N, L$ */
Set $N \leftarrow \emptyset$.
Set $A^2 \leftarrow \mathcal{M}_\psi$.
Repeat while there is a column $v$ of $A^2$ such that $v \notin N$ or $-v \notin N$:
    Remove 0 rows from $A^2$.
    Choose any $v$ such that either $v \notin N$ or $-v \notin N$.
    Set $L \leftarrow \emptyset$ and $\alpha \leftarrow 1$.
    If $v \in N$:
        Multiply all entries in column $v$ of $A^2$ by $-1$.
        Set $\alpha \leftarrow -1$.
        Set $N \leftarrow N \setminus \{v\} \cup \{-v\}$.
    Set $p \leftarrow v$.
    Define $A'^1 = D' = E' = 0$, $A'^2 = A^2$, the initial partition of $A^2$.
Repeat the following:
    Move column $p$ of $A'^2$ to the right border of $A'^1$.
    Move the right border of $A'^1$ to the right by 1 column.
    Move and tag rows of $D'$ with 1 in its right column to the top.
    Move the bottom border of $A'^1$ down to include tagged rows.
    If $E' = 0$, Set $A^2 \leftarrow A'^2$ and Break.
    Choose column $z$ through $E'$ with a non-zero entry in $E'$.
    If ($z \in N$ and $-z \in N$) or
        ($z \in N$ and column $z$ has $-1$ entry) or
        ($-z \in N$ and column $z$ has $+1$ entry) or
        (column $z$ has $+1$ and $-1$ entries):
            If $-v \in N$ or $\alpha = -1$: Set $N \leftarrow N \cup \{v, -v, z, -z\} \cup L$.
            Break.
    Otherwise,
        If $-z \notin N$ and column $z$ has no $-1$ entries:
            Multiply all entries in column $z$ of $A^2$ by $-1$.
            If $z \in N$: Set $N \leftarrow N \setminus \{z\} \cup \{-z\}$.
            If $-z \in L$: Set $L \leftarrow L \setminus \{-z\} \cup \{z\}$.
            Set $L \leftarrow L \cup \{-z\}$.
            Set $p \leftarrow z$.
    Remove 0 rows from $A^2$.
    Let $\mathcal{M}$ be the permuted and scaled $\mathcal{M}_\psi$ with lower right matrix $A^2$.
    Output $\mathcal{M}$.
\[\square\]

Figure 3.7: Algorithm for finding the maximum monotone decomposition of a (0,±1) matrix.
3.6.2 AUTARKIES

A formula $\psi$ which has a maximum monotone decomposition where $A^2$ is a member of an efficiently solved subclass of Satisfiability obviously may be solved efficiently by Algorithm 4 if $A^2$ can efficiently be recognised as belonging to such a subclass. Chapter 4 discusses several efficiently solved subclasses of Satisfiability problems which may be suitable for testing. If $A^2$ represents a 2-SAT formula (See Section 4.1) then $\psi$ is said to be q-Horn. The class of q-Horn formulas was discovered and efficiently solved in [18, 19] and it was the results of that that work led to the development of maximum monotone decompositions [127].

3.6.2 Autarkies

Definition 24. An assignment to a set of variables is said to be autark if all clauses that contain at least one of those variables are satisfied by the assignment.

3.7 Branch-and-bound

The DPLL algorithm of Figure 3.4 is sequential in nature. At any point in the search one node, representing a partial assignment $M_1$, of the search tree is active; that is, open to exploration. This means exploration of a promising branch of the search space may be delayed for a considerable period until search finally reaches that branch. Branch-and-bound aims to correct this to some extent. In branch-and-bound, quite a few nodes of the search space may be active at any point in the search. Each of the active nodes has a number $l(M_1)$ which is an aggregate estimate of how close the assignment represented at a node is to a solution or confirms that assignment cannot be extended to a “best” solution. Details concerning how $l(M_1)$ is computed will follow. When it is time to choose a variable $v$ for assignment, the variable is chosen from the subformula of the active node of lowest $l$ value and the node is expanded. That expansion eliminates one active node and may create up to two others, one for each value to $v$. To help control the growth of active nodes branch-and-bound maintains a monotonically decreasing number $u$ for preventing nodes known to be unfruitful from becoming active. If the $l$ value of any potential active node is greater than $u$, it is thrown out and not made active. Eventually, there are no active nodes left and the algorithm completes.

Branch-and-bound is intended to solve more problems than Satisfiability. It requires a function $g_\psi(M_1)$ which maps a given formula $\psi$ and partial or total truth assignment $M_1$ to a non-negative number. The objective of branch-and-bound is to return an assignment $M$ such that $g_\psi(M)$ is minimum over all truth assignments, partial or complete. That is,

$$M : \forall x, g_\psi(X) \geq g_\psi(M).$$
For example, if \( g_\psi(M_1) \) is just the number of clauses in \( \psi_{M_1} \) then branch-and-bound seeks to find \( M \) which satisfies the greatest number of clauses, maybe all.

Branch-and-bound discovers and discards search paths that are known to be fruitless, before they are explored, by means of a heuristic function \( h(\psi_{M_1}) \) where \( \psi_{M_1} \) is obtained from \( \psi \) by removing clauses satisfied by and literals falsified by \( M_1 \). The heuristic function returns a non-negative number which, when added to \( g_\psi(M_1) \), is a lower bound on \( g(\psi_X) \) over all possible extensions \( X \) to \( M_1 \). That sum is the \( l(M_1) \) that was referred to above. That is,

\[
l(M_1) = h(\psi_{M_1}) + g_\psi(M_1) \leq \min\{g_\psi(X) : X \text{ is an extension of } M_1\}.
\]

The algorithm maintains a number \( u \) that records the lowest \( g_\psi \) value that has been seen so far during search. If partial assignment \( M_{1:i} \) is extended by one variable to \( M_{1:i+1} \) and \( g_\psi(M_{1:i+1}) < u \) then \( u \) is updated to that value and \( M_{1:i+1} \), the assignment that produced that value, is saved as \( M \). In that case, \( l(M_{1:i+1}) \) must also be less than \( u \) because it is less than \( g_\psi(M_{1:i+1}) \). But, if \( l(M_{1:i+1}) > u \) then there is no chance, by definition of \( l(M_1) \), that any extension to \( M_{1:i+1} \) will yield the minimum \( g_\psi \). Hence, if that test succeeds, the node that would correspond to \( M_{1:i+1} \) is thrown out, eliminating exploration of that branch.

The algorithm in its general form for Satisfiability is shown in Figure 3.8 as Algorithm 6. A priority queue \( P \) is used to hold all active nodes as pairs where each pair contains a reduced formula and its corresponding partial assignment. Pairs are stored in \( P \) in increasing order of \( l(M_1) \). It is easy to see, by definition of \( l(M_1) \), that no optimal assignment gets thrown out. It is also not difficult to see that, given two heuristic functions \( h_1 \) and \( h_2 \), if \( h_1(\psi_{M_1}) \leq h_2(\psi_{M_1}) \) for all \( M \), then the search explored using \( h_1 \) will be no larger than the search space explored using \( h_2 \). Thus, to keep the size of the search space down, as tight a heuristic function as possible is desired. However, since overall performance is most important and since tighter heuristic functions typically mean more overhead, it is sometimes more desirable to use a weaker heuristic function which generates a larger search space in less time. Section 3.10.2 shows how Linear Programming relaxations of Integer Programming representations of search nodes can be used as heuristic functions. We finish this section with an alternative to illustrate what else is possible.

Recall the problem of Variable Weighted Satisfiability which was defined in Section 2.2.5: given CNF formula \( \psi \) and positive weights on variables, find a satisfying assignment for \( \psi \), if one exists, such that the sum of weights of variables of value 1 is minimized. Let \( \psi_{M_1} \) be defined as above and let \( Q_{\psi_{M_1}} \) be a subset of positive clauses of \( \psi_{M_1} \) such that no variable appears twice in \( Q_{\psi_{M_1}} \). For example, \( Q_{\psi_{M_1}} \) might look like this:

\[
(v_1 \lor v_3 \lor v_7) \land (v_2 \lor v_6) \land (v_4 \lor v_5 \lor v_8).
\]
Algorithm 6.

**Branch-and-bound** \((\psi, h, g_\psi)\)

/* Input: \(n\) variable CNF formula \(\psi\), heuristic function \(h\), * /
/* objective function \(g_\psi\) mapping partial assignments to \(Z\) * /
/* Output: Assignment \(M\) such that \(g_\psi(M)\) is minimized * /
/* Locals: Integer \(u\), priority queue \(P\) * /

Set \(M \leftarrow \emptyset\); Set \(M_{1:0} \leftarrow \emptyset\); Set \(u \leftarrow \infty\).
Insert \(P \leftarrow (\langle \psi, M_{1:0} \rangle, 0)\).

Repeat the following while \(P \neq \emptyset\):

Pop \((\psi', M_{1:i}) \leftarrow P\).
If \(\psi' = \emptyset\) then Output \(M\).
Choose variable \(v\) from \(\psi'\).
Set \(\psi'_1 \leftarrow \{c \setminus \{v\} : c \in \psi' \text{ and } \neg v \notin c\}\).
Set \(\psi'_2 \leftarrow \{c \setminus \{-v\} : c \in \psi' \text{ and } v \notin c\}\).
Repeat the following for \(j = 1\) and \(j = 2\):

If \(j = 1\) then do the following:

Set \(M_{1:i+1} \leftarrow M_{1:i}\).
Otherwise

Set \(M_{1:i+1} \leftarrow M_{1:i} \cup \{v\}\).
If \(h(\psi'_j) + g_\psi(M_{1:i+1}) < u\) then do the following:

Insert \(P \leftarrow (\langle \psi'_j, M_{1:i+1} \rangle, h(\psi'_j) + g_\psi(M_{1:i+1}))\).
If \(g_\psi(M_{1:i+1}) < u\) then do the following:

Set \(u \leftarrow g_\psi(M_{1:i+1})\).
Set \(M \leftarrow M_{1:i+1}\).

Output \(M\).

Figure 3.8: Classic branch-and-bound procedure adapted to Satisfiability.
A strictly lower bound on the minimum weight solution over all extensions to is \(M_1\) is \(g_0(M_1)\) plus \(h(\psi_{M_1})\), the sum of the weights of the minimum weight variable in each of the clauses of \(Q_{\psi_{M_1}}\). Clearly, this is not a very tight bound. But, it is computationally fast to acquire this bound, and the trade off of accuracy for speed often favors this approach [57], particularly when weight calculations are made incrementally. Additionally, there are some tricks that help to find a “good” \(Q_{\psi_{M_1}}\). For example, a greedy approach may be used as follows: choose a positive clause \(c\) with variables independent of clauses already in \(Q_{\psi_{M_1}}\) and such that the ratio of the weight of the minimum weight variable in \(c\) to the number of variables in \(c\) is maximum [104]. The interested reader can consult [104] and [43] for additional ideas.

### 3.8 Local Search

#### 3.8.1 Walksat

#### 3.8.2 Novelty

### 3.9 Binary Decision Diagram

Binary Decision Diagrams (BDDs) as graphs were discussed in Section 2.3.2. In this section the associated BDD data structure and efficient operations on that data structure are discussed. Attention is restricted to Reduced Ordered Binary Decision Diagrams (ROBDDs) due to its compact, efficient, canonical properties.

The following is a short review of Section 2.3.2. A ROBDD is a BDD such that: 1) There is no vertex \(v\) such that \(\text{then}(v) = \text{else}(v)\); 2) The subgraphs of two distinct vertices \(v\) and \(w\) are not isomorphic. A ROBDD represents a Boolean function uniquely in the following way (symbol \(v\) will represent both a vertex of a ROBDD and a variable labeling a vertex). Define \(f(v)\) recursively as follows:

1. If \(v\) is the terminal vertex labeled 0, then \(f(v) = 0\);
2. If \(v\) is the terminal vertex labeled 1, then \(f(v) = 1\);
3. Otherwise, \(f(v) = (v \land f(\text{then}(v))) \lor (\neg v \land f(\text{else}(v)))\).

Then \(f(\text{root}(v))\) is the function represented by the ROBDD. A Boolean function has different ROBDD representations, depending on the variable order imposed by index, but there is only one ROBDD for each ordering. Thus, ROBDDs are known as a canonical representation of Boolean functions. From now on we will use BDD to refer to a ROBDD.

A data structure representing a BDD consists of an array of Node objects (or nodes), each corresponding to a BDD vertex and each containing three
elements: a variable label \( v \), \( \text{then}(v) \), and \( \text{else}(v) \), where the latter two elements are BDD array indices. The first two nodes in the BDD array correspond to the 0 and 1 terminal vertices of a BDD. For both, \( \text{then}(..) \) and \( \text{else}(..) \) are empty. We will use \( \text{terminal}(1) \) and \( \text{terminal}(0) \) to denote the BDD array locations of these nodes. All other nodes fill up the remainder of the BDD array in the order they are created. A node that is not in the BDD array can be created, added to the BDD array, and its array index returned in constant time. We denote this operation by \( \text{node} = \text{createNode}(v, t, e) \), where \( \text{node} \) is the array index of the newly created node, \( t \) is \( \text{then}(v) \) and \( e \) is \( \text{else}(v) \), both BDD array indices. A hashtable is maintained for the purpose of finding a node’s array location given \( v, t, e \). If no such node exists, \( \text{createNode} \) is called to create and insert it. Otherwise, its array location is returned from the hashtable. We use \( \text{lookup}((v, t, e)) \) to represent this operation. If \( \text{lookup} \) returns \text{null} then the node does not exist, otherwise \( \text{lookup} \) returns its BDD array location. We use \( \text{insert}((v, t, e), \text{node}) \) to represent the act of inserting the node BDD array index into the hashtable at key \( (v, t, e) \). The procedure \( \text{findOrCreateNode} \) for returning the BDD array index of a node is shown in Figure 3.9.

The main BDD construction operation is to find and attach two descendant nodes (\( \text{then}(v) \) and \( \text{else}(v) \)) to a parent node \( (v) \). The procedure \( \text{findOrCreateNode} \) is used to ensure that no two nodes in the final BDD data structure represent the same function. The procedure for building a BDD data structure is \( \text{buildBDD} \), shown in Figure 3.9. It is assumed that variable indices match the value of \( \text{index} \) applied to that variable (thus, \( i = \text{index}(v_i) \)). The complexity of \( \text{buildBDD} \) is proportional to the number of nodes that must be created. In all interesting applications, many BDDs are constructed. But they may all share the BDD data structure above. Thus, a node may belong to many BDDs.

The operations \( \text{reduce}_1 \) and \( \text{reduce}_0 \), shown in Figure 3.11 will be used to describe several important BDD operations in subsequent sections. Assuming \( v \) is the root of the BDD representing \( f \), the operation \( \text{reduce}_1(v, f) \) returns \( f \) constrained by the assignment of 1 to variable \( v \) and \( \text{reduce}_0(v, f) \) returns \( f \) constrained by the assignment of 0 to the variable \( v \).

Details on performing the common binary operations of \( \land \) and \( \lor \) on BDDs will be ignored here. The reader may refer to [11] for detailed descriptions. We only mention that, using a dynamic programming algorithm, the complexity of these operations is proportional to the product of the sizes of the operands and the size of the result of the operation can be that great as well. Therefore, using \( \land \) alone, for example (as so many problems would require), could lead to intermediate structures that are too large to be of value. This problem is mitigated somewhat by operations of the kind discussed in the next four subsections, particular existential quantification.

The operations considered next are included not only because they assist BDD oriented solutions but mainly because they can assist search-oriented solutions when used properly. For example, if inputs are expressed as a
Algorithm 7.

\textbf{findOrCreateNode} (v, t, e)

\begin{verbatim}
/* Input: variable label v, Node object indices t and e */
/* Output: an array index of a Node object \langle v, t, e \rangle */

If \( t == e \) then Return \( t \).
Set node \leftarrow \text{lookup}(\langle v, t, e \rangle).
If node \neq \text{null} then Return node.
Set node \leftarrow \text{createNode}(\langle v, t, e \rangle).
\text{insert}(\langle v, t, e \rangle, node).
Return node.
\end{verbatim}

Figure 3.9: Procedure for finding a node or creating and inserting it.

Algorithm 8.

\textbf{buildBDD} (f, i)

\begin{verbatim}
/* Input: Boolean function f, index i */
/* Output: root Node of BDD representing f */

If \( f \equiv \text{1} \) return \text{terminal}(1).
If \( f \equiv \text{0} \) return \text{terminal}(0).
Set t \leftarrow \text{buildBDD}(f|_{v_i=1}, i+1).
Set e \leftarrow \text{buildBDD}(f|_{v_i=0}, i+1).
Return \text{findOrCreateNode}(v_i, t, e).
\end{verbatim}

Figure 3.10: Algorithm for building a BDD: invoked using \textbf{buildBDD}(f,1).

Algorithm 9.

\textbf{reduce}_1 (v, f) \hspace{2cm} \textbf{reduce}_0 (v, f)

\begin{verbatim}
/* Input: variable v, BDD f */
/* Output: reduced BDD */
If root(f) \equiv v then
    Return then(root(f)).
Return f.

/* Input: variable v, BDD f */
/* Output: reduced BDD */
If root(f) \equiv v then
    Return else(root(f)).
Return f.
\end{verbatim}

Figure 3.11: Operations \textbf{reduce}_1 and \textbf{reduce}_0.
Algorithm 10.

\textbf{exQuant}(f,v)
/* Input: BDD f, variable v */
/* Output: BDD f with v existentially quantified away */
\begin{itemize}
  \item If root(f) == v then Return then(root(f))\lor else(root(f)).
  \item If index(v) > index(root(f)) then Return 0. // If v is not in f do nothing
  \item Set $h_f_1 \leftarrow \text{exQuant}(\text{then}(root(f)), v)$.
  \item Let $h_f_0 \leftarrow \text{exQuant}(\text{else}(root(f)), v)$.
  \item If $h_f_0 == h_f_1$ then Return $h_f_1$.
  \item Return \text{FindOrCreateNode}(root(f), h_f_1, h_f_0).
\end{itemize}

\hfill \Box

Figure 3.12: Algorithm for existentially quantifying variable v away from BDD f. The $\lor$ denotes the “or” of BDDs.

A Boolean function which can be written
\[ f(v, \bar{x}) = (v \land h_1(\bar{x})) \lor (\neg v \land h_2(\bar{x})) \]
can be replaced by
\[ f(\bar{x}) = h_1(\bar{x}) \lor h_2(\bar{x}) \]
where $\bar{x}$ is a list of one or more variables. There is a solution to $f(\bar{x})$ if and only if there is a solution to $f(v, \bar{x})$ so it is sufficient to solve $f(\bar{x})$ to get a solution to $f(v, \bar{x})$. Obtaining $f(\bar{x})$ from $f(v, \bar{x})$ is known as existentially quantifying v away from $f(v, \bar{x})$. This operation is efficiently handled if $f(v, \bar{x})$ is represented by a BDD. However, since most interesting BDD problems are formulated as a conjunction of functions, and therefore as conjunctions of BDDs, existentially quantifying away a variable v succeeds easily only when just one of the input BDDs contains v. Thus, this operation is typically used together with other BDD operations for maximum effectiveness. The algorithm for existential quantification is shown in Figure 3.12.

If inferences can be revealed in preprocessing they can be applied immediately to reduce input size and therefore reduce search complexity. Although existential quantification can, by itself, uncover inferences (see, for
example, Figure 3.13), those same inferences are revealed during BDD construction if inference lists for each node are built and maintained (see Section ??). Therefore, a more effective use of existential quantification is in support of other operations, such as strengthening (see Section 3.9.5), to uncover those inferences that cannot be found during BDD construction or in tandem with $\land$ to retard the growth of intermediate BDDs.

Existential quantification, if applied as a preprocessing step prior to search, can increase the number of choicepoints expanded per second but can increase the size of the search space. The increase in choicepoint speed is because existentially quantifying a variable away from the function has the same effect as branching from a choicepoint in both directions. Then overhead is reduced by avoiding heuristic computations. However, search space size may increase since the elimination of a variable can cause subfunctions that had been linked only by that variable to become merged with the result that the distinction between the subfunctions becomes blurred. This is illustrated in Figure 3.14. The speedup can overcome the lost intelligence but it is sometimes better to turn it off.

### 3.9.2 Reductions and Inferences

Consider the truth tables corresponding to two BDDs $f$ and $c$ over the union of variable sets of both $f$ and $c$. Build a new BDD $g$ with variable set no larger than the union of the variable sets of $f$ and $c$ and with a truth table such that on rows which $c$ maps to 1 $g$ maps to the same value that $f$ maps to, and on other rows $g$ maps to any value, independent of $f$. It should be clear that $f \land c$ and $g \land c$ are identical so $g$ can replace $f$ in a collection of BDDs without changing its solution space.

The are at least three reasons why this might be done. The superficial reason is that $g$ can be made smaller than $f$. A more important reason is that inferences can be discovered. The third reason is that BDDs can be removed from the collection without loss. Consider, for example, BDDs representing functions

$$f = (v_1 \lor \neg v_2 \lor v_3) \land (\neg v_1 \lor v_2) \quad \text{and}$$
$$c = (v_1 \lor \neg v_2).$$

Let a truth table row be represented by a 0-1 vector which reflects assignments of variables indexed in increasing order from left to right. Let $g$ have the same truth table as $f$ except for row $\langle 011 \rangle$ which $c$ maps to 0 and $g$ maps to 1. Then $g = (v_1 \leftrightarrow v_2)$ and $f \land c$ is the same as $g \land c$ but $g$ is smaller than $f$. As an example of discovering inferences consider

$$f = (v_1 \rightarrow v_2) \land (\neg v_1 \rightarrow (\neg v_3 \land v_4)) \quad \text{and}$$
$$c = (v_1 \lor v_3).$$
Let \( g \) have the same truth table as \( f \) except \( g \) maps rows \( \langle 0001 \rangle \) and \( \langle 0101 \rangle \) to 0, as does \( c \). Then \( g = (v_1) \land (v_2) \) which reveals two inferences. The BDDs for \( f, c, \) and \( g \) of this example are shown in Figure 3.16. The example showing BDD elimination is deferred to Theorem 25, Section 3.9.4.

Clearly, there are numerous strategies for creating \( g \) from \( f \) and \( c \) and replacing \( f \) with \( g \). An obvious one is to have \( g \) and \( c \) map the same rows to 0. This strategy, which we will call zero-restrict, turns out to have weaknesses. Its obvious dual, which has \( g \) map to 1 all rows that \( c \) maps to 0, is no better. For example, applying zero-restrict to \( f \) and \( c \) of Figure 3.18 produces \( g = \neg v_3 \land (v_1 \lor (\neg v_1 \land \neg v_2)) \) instead of the inference \( g = \neg v_3 \) which is obtained from a more intelligent replacement strategy. The alternative approach, one of many possible ones, judiciously chooses some rows of \( g \) to map to 1 and others to map to 0 so that \( g \)'s truth table reflects a logic pattern that generates inferences. The truth table of \( c \) has many 0 rows and this is exploited. Specifically, \( c \) maps rows \( \langle 010 \rangle, \langle 011 \rangle, \langle 101 \rangle, \) and \( \langle 111 \rangle \) to 0. The more intelligent strategy lets \( g \) map rows \( \langle 011 \rangle \) and \( \langle 111 \rangle \) to 0 and rows \( \langle 010 \rangle \) and \( \langle 101 \rangle \) to 1. Then \( g = \neg v_3 \).

Improved replacement strategies might target particular truth table patterns, for example equivalences, or they might aim for inference discovery. Since there is more freedom to manipulate \( g \) if the truth table of \( c \) has many zeros, it is important to choose \( c \) as carefully as the replacement strategy. This is illustrated by the examples of Figure 3.17 and Figure 3.18 where, in the first case, no inference is generated but after \( f \) and \( c \) are swapped an inference is generated. The next two sections show two replacement strategies that are among the more commonly used.

### 3.9.3 Restrict

The original version of restrict is what we call zero-restrict above. That is, the original version of restrict is intended to remove paths to terminal\((1)\) from \( f \) that are made irrelevant by \( c \). The idea was introduced in [41]. In this monograph we present our own version which is implemented as Algorithm 11 of Figure 3.15. We use symbol \( \downarrow \) to denote the restrict operator. Then \( g = f \downarrow c \) is the result of zero-restrict after all variables in \( c \) that are not in \( f \) are existentially quantified away from \( c \). Figures 3.16 to 3.18 show examples that were referenced in the previous section.

Procedure restrict is similar to a procedure called generalized co-factor (gcf) or constrain (see Section 3.9.4 for a description). Both restrict\((f, c)\) and gcf\((f, c)\) agree with \( f \) on interpretations where \( c \) is satisfied, but are generally somehow simpler than \( f \). Procedure restrict can be useful in preprocessing because the BDDs produced from it can never be larger than the BDDs they replace.

On the negative side, it can, in odd cases, cause a garbling of local information. Although restrict may reveal some of the inferences that strengthening would (see below), it can still cause the number of search
choice points to increase. Both these issues are related: restrict can spread an inference that is evident in one BDD over multiple BDDs (see Figure 3.19 for an example).

3.9.4 Generalized Co-factor

The generalized co-factor operation, also known as constrain, is denoted here by $|$ and implemented as $\text{gcf}$ in Figure 3.20 as Algorithm 12. It takes BDDs $f$ and $c$ as input and produces $g = f/c$ by sibling substitution. BDD $g$ may be larger or smaller than $f$ but, more importantly, systematic use of this operation can result in the elimination of BDDs from a collection. Unfortunately, by definition, the result of this operation depends on the underlying BDD variable ordering so it cannot be regarded as a logical operation. It was introduced in [42].

BDD $g$ is a generalized co-factor of $f$ and $c$ if for any truth assignment $t$, $g(t)$ has the same value as $f(t')$ where $t'$ is the “nearest” truth assignment to $t$ that maps $c$ to 1. The notion of “nearest” truth assignment depends on a permutation $\pi$ of the numbers 1, 2, ..., $n$ which states the variable ordering of the input BDDs. Represent a truth assignment to $n$ variables as a vector in $\{0, 1\}^n$ and, for truth assignment $t$, let $t_i$ denote the $i^{th}$ bit of the vector representing $t$. Then the distance between two truth assignments $t'$ and $t''$ is defined as $\sum_{i=1}^{n} 2^{n-i}(t'_n \oplus t''_n)$. One pair of assignments is nearer to each other than another pair if the distance between that pair is less. It should be evident that distances between pairs are unique for each pair.

For example, Figure 3.21 shows BDDs $f$ and $c$ under the variable ordering given by $\pi = \langle 1, 2, 3, 4 \rangle$. For assignment vectors $\langle * * 01 \rangle$, $\langle * * 10 \rangle$, $\langle * * 11 \rangle$ (where * is a wildcard meaning 0 or 1), $\text{gcf}(f, c)$, shown as the BDD at the bottom of Figure 3.21, agrees with $f$ since those assignments cause $c$ to evaluate to 1. The closest assignment to $\langle 0000 \rangle$, $\langle 0100 \rangle$, $\langle 1000 \rangle$, and $\langle 1100 \rangle$ causing $c$ to evaluate to 1 is $\langle 0001 \rangle$, $\langle 0101 \rangle$, $\langle 1010 \rangle$, and $\langle 1101 \rangle$, respectively. On all these inputs $\text{gcf}(f, c)$ has value 1, which the reader can check in Figure 3.21.

The following expresses the main property of $|$ that makes it useful.

**Theorem 25.** Given BDDs $f_1, ..., f_k$, for any $1 \leq i \leq k$, $f_1 \land f_2 \land ... \land f_k$ is satisfiable if and only if $(f_1|f_{i-1}|f_i) \land (f_{i+1}|f_{i+2}) \land \ldots \land (f_k|f_{k+1})$ is satisfiable. Moreover, any assignment satisfying the latter can be mapped to an assignment that satisfies $f_1 \land \ldots \land f_k$.

**Proof.** If we show

\[(f_1|f_{i-1}|f_i) \land (f_{i+1}|f_{i+2}) \land \ldots \land (f_k|f_{k+1}) \land f_i\]  \hspace{1cm} (3.3)

is satisfiable if and only if

\[(f_1|f_{i-1}|f_i) \land (f_{i+1}|f_{i+2}) \land \ldots \land (f_k|f_{k+1}) \land f_i\]  \hspace{1cm} (3.4)
is satisfiable then, since (3.4) is equivalent to \( f_1 \land \ldots \land f_k \), the first part of the theorem will be proved. Suppose (3.4) is satisfied by truth assignment \( t \). That \( t \) represents a truth table row that \( f_i \) maps to 1. Clearly that assignment also satisfies (3.3). Suppose no assignment satisfies (3.4). Then all assignments for which \( f_i \) maps to 1 do not satisfy (3.3) since otherwise (3.4) would be satisfied by any that do. We only need to consider truth assignments \( t \) which \( f_i \) maps to 0. Each \((f_j|f_i)\) in (3.3) and (3.4) maps to the same value that \( f_j \) maps the “nearest” truth assignment, say \( r \), to \( t \) that satisfies \( f_i \). But \( r \) cannot satisfy (3.4) because it cannot satisfy (3.3) by the argument above. Hence, there is no truth assignment falsifying \( f_i \) but satisfying (3.3) so the first part is proved.

For the second part, observe that any truth assignment that satisfies (3.3) and (3.4) also satisfies \( f_1 \land \ldots \land f_k \) so we only need to consider assignments \( t \) that satisfy (3.3) but not (3.4). In that case, by construction of \((f_j|f_i)\), the assignment that is “nearest” to \( t \) and satisfies \( f_i \) also satisfies \((f_j|f_i)\). That assignment satisfies \( f_1 \land \ldots \land f_k \).

This means that, for the purposes of a solver, generalized co-factoring can be used to eliminate one of the BDDs among a given conjoined set of BDDs: the solver finds an assignment satisfying \( \text{gcf}(f_1, f_i) \land \ldots \land \text{gcf}(f_k, f_i) \) and then extends the assignment to satisfy \( f_i \), otherwise the solver reports that the instance has no solution. However, unlike \textit{restrict}, generalized co-factoring cannot by itself reduce the number of variables in a given collection of BDDs. Other properties of the \textit{gcf} operation, all of which are easy to show, are:

1. \( f = c \land \text{gcf}(f, c) \lor \neg c \land \text{gcf}(f, \neg c) \).
2. \( \text{gcf}(\text{gcf}(f, g), c) = \text{gcf}(f, g \land c) \).
3. \( \text{gcf}(f \land g, c) = \text{gcf}(f, c) \land \text{gcf}(g, c) \).
4. \( \text{gcf}(f \land c, c) = \text{gcf}(f, c) \).
5. \( \text{gcf}(f \land g, c) = \text{gcf}(f, c) \land \text{gcf}(g, c) \).
6. \( \text{gcf}(f \lor g, c) = \text{gcf}(f, c) \lor \text{gcf}(g, c) \).
7. \( \text{gcf}(f \lor \neg c, c) = \text{gcf}(f, c) \).
8. \( \text{gcf}(\neg f, c) = \neg \text{gcf}(f, c) \).
9. If \( c \) and \( f \) have no variables in common and \( c \) is satisfiable then \( \text{gcf}(f, c) = f \).

Care must be taken when co-factoring in “both” directions (exchanging \( f \) for \( c \)). For example, \( f \land g \land h \) cannot be replaced by \((g|f) \land (f|g) \land h \) since the former may be unsatisfiable when the latter is satisfiable.

Examples of the application of \textit{gcf} are shown in Figures 3.21 and 3.22. Figure 3.21 illustrates the possibility of increasing BDD size. Figure 3.22
presents the same example after swapping $v_1$ and $v_3$ under the same variable ordering and shows that the result produced by \texttt{gcf} is sensitive to variable ordering. Observe that the functions produced by \texttt{gcf} in both figures have different values under the assignment $v_1 = 1$, $v_2 = 1$, and $v_3 = 0$. Thus, the function returned by \texttt{gcf} depends on the variable ordering as well.

### 3.9.5 Strengthen

This binary operation on BDDs helps reveal inferences that are missed by \texttt{restrict} due to its sensitivity to variable ordering. Given two BDDs, $b_1$ and $b_2$, strengthening conjoins $b_1$ with the projection of $b_2$ onto the variables of $b_1$: that is, $b_1 \land \exists \vec{v}b_2$, where $\vec{v}$ is the set of variables appearing in $b_2$ but not in $b_1$. Strengthening each $b_i$ against all other $b_j$s sometimes reveals additional inferences or equivalences. Algorithm \texttt{strengthen} is shown in Figure 3.23. Figure 3.24 shows an example.

Strengthening provides a way to pass important information from one BDD to another without causing a size explosion. No size explosion can occur because, before $b_1$ is conjoined with $b_2$, all variables in $b_2$ that don’t occur in $b_1$ are existentially quantified away. If an inference (of the form $v = 1$, $v = 0$, $v = w$, or $v = \neg w$) exists due to just two BDDs, then strengthening those BDDs against each other (pairwise) can move those inferences, even if originally spread across both BDDs, to one of the BDDs. Because \texttt{strengthen} shares information between BDDs, it can be thought of as sharing intelligence and \texttt{strengthening} the relationships between functions; the added intelligence in these strengthened functions can be exploited by a smart search heuristic. We have found that \texttt{strengthen} usually decreases the number of choicepoints when a particular search heuristic is employed, but sometimes it causes more choicepoints. We believe this is due to the delicate nature of some problems where duplicating information in the BDDs leads the heuristic astray.

Procedure \texttt{strengthen} may be applied to CNF formulas and in this case it is the same as applying Davis-Putnam resolution selectively on some of the clauses. When used on more complex functions it is clearer how to use it effectively as the clauses being resolved are grouped with some meaning. Evidence for this comes from Bounded Model Checking examples (see Section ??).

We close this section by mentioning that for some classes of problems resolution has polynomial complexity while strictly BDD manipulations require exponential time and for other classes of problems resolution has exponential complexity while BDD manipulations require polynomial time. This topic is treated in detail in Section ??.
3.10 Algebraic Methods

In an algebraic system Boolean constraints are expressed as a system of algebraic equations or inequalities which has a solution if and only if the constraints are satisfiable. Equations or inequalities may represent a larger class of Boolean functions than just disjunctions. The attraction of these methods is that, in some cases, a single algebraic operation can simulate a large number of resolution operations. The problem is that it is not always obvious how to choose the optimal sequence of operations to take advantage of this and often performance is disappointing due to a non-optimal choice.

3.10.1 Gröbner Bases Applied to SAT

It is interesting that at the same time resolution was being developed and understood as a search tool in the 1960's, an algebraic tool for computing a basis for highly “non-linear” systems of equations was introduced: the basis it found was given the name Gröbner basis and the tool was called the Gröbner basis algorithm [27]. But it wasn’t until the mid 1990’s that Gröbner bases crossed paths with Satisfiability when it was shown that Boolean expressions can be written as systems of multi-linear equations which a simplified Gröbner basis algorithm can solve using a number of derivations that is guaranteed to be within a polynomial of the minimum number possible [35]. Also in that paper it is shown that the minimum number of derivations cannot be much greater than, and may sometimes be far less than, the minimum number needed by resolution. Such powerful results led the authors to say “these results suggest the Gröbner basis algorithm might replace resolution as a basis for heuristics for NP-complete problems.”

This has not happened, perhaps partly because of the advances in the development of CNF SAT solvers in the 1990’s and partly because, as is the case for resolution, it is generally difficult to find a minimum sequence of derivations leading to the desired conclusion. However, the complementary nature of algebraic and logic methods makes them an important alternative to resolution. Generally, in the algebraic world problems that can be solved essentially using Gaussian elimination with a small or modest increase in the degree of polynomials are easy. A classic example where this is true is systems of equations involving only the exclusive-or operator. By contrast, just expressing the exclusive-or of $n$ variables requires $2^n - 1$ clauses.

In the algebraic proof system outlined here, facts are represented as multi-linear equations and new facts are derived from a database of existing facts using rules described below. Let $\langle c_0, c_1, \ldots, c_{2^n-1} \rangle$ be a 0-1 vector of $2^n$ coefficients. For $0 \leq j < n$, let $b_{i,j}$ be the $j^{th}$ bit in the binary representation of the number $i$. An input to the proof system is a set of equations of the
following form:
\[
\sum_{i=0}^{2^n-1} c_i v_1^{b_{i,0}} v_2^{b_{i,1}} \cdots v_n^{b_{i,n-1}} = 0
\] (3.5)

where all variables \( v_i \) can take values 0 or 1, and addition is taken modulo 2. An equation of the form (3.5) is said to be multi-linear. A product \( t_i = v_1^{b_{i,0}} v_2^{b_{i,1}} \cdots v_n^{b_{i,n-1}} \), for any \( 0 \leq i \leq 2^n - 1 \), will be referred to as a multi-linear term or simply a term. The degree of \( t_i \), denoted \( \deg(t_i) \), is \( \sum_{0 \leq j < n} b_{i,j} \). A term that has a coefficient of value 1 in an equation is said to be a non-zero term of that equation.

New facts may be derived from known facts using the following rules:

1. Any even sum of like non-zero terms in an equation may be replaced by 0. Thus, \( v_1 v_2 + v_1 v_2 \) reduces to 0 and \( 1 + 1 \) reduces to 0. This reduction rule is needed to eliminate terms when adding two equations (see below).

2. A factor \( v^2 \) in a term may be replaced by \( v \). This reduction rule is needed to ensure terms remain multi-linear after multiplication (see below).

3. An equation of the form (3.5) may be multiplied by a term and the resulting equation may be reduced to the form (3.5) by rule 2. above. Thus, \( v_1 v_3 (v_1 + v_3 = 0) \) becomes \( v_1 v_3 v_4 + v_3 v_4 = 0 \).

4. Two equations may be added to produce an equation that may be reduced by rule 1. above to the form (3.5). Examples will be given below.

An equation that is created by rule 3. or 4. is said to be derived. All derived equations are reduced by rules 1. and 2. before being added to the proof.

Observe that the solution spaces of two equations are complementary if they differ only in that \( c_0 = 0 \) for one and \( c_0 = 1 \) for the other. For example, the sets of solutions for the two equations

\[
\begin{align*}
v_1 v_2 v_3 + v_1 v_2 + v_2 v_3 + v_1 + 1 &= 0 \\
v_1 v_2 v_3 + v_1 v_2 + v_2 v_3 + v_1 &= 0
\end{align*}
\]

are complementary.

We leave as evident that performing operations 3. and 4. with reductions 1. and 2. as needed results in a set of derived equations whose solution space is a superset of the original set. The set of all possible derived equations has a solution space which is identical to that of the original set of equations.
Theorem 26. The equation 1=0 is always derivable using rules 3. and 4. (and implicitly rules 1. and 2.) from an inconsistent input set of multi-linear equations and never derived from a consistent set.

Proof. Assume 1=0 is not one of the input equations. Suppose 1=0 is derived from a consistent input set. Then two equations were added to derive 1=0. But the solution space of both must be complementary and therefore the solution space of the entire system must be empty. That is not possible since application of rules 3. and 4. do not reduce the solution space below that of the original set of equations and there is at least one solution because the input set is consistent.

Suppose 1=0 is not derivable from an inconsistent input set. We re-index terms, with the term of degree 0 taking the lowest index, and construct a sequence of derivations such that no two derived equations have the same non-zero term. If the derivation cannot continue to the lowest (0th) term then the resulting system of equations is linearly independent and therefore must have a solution. But that is impossible by assumption.

Re-indexing is as follows: terms of degree $i$ all have higher index than terms of degree $j$ if $i > j$; among terms of the same degree, the order of index is decided lexicographically. Call the equations $\psi$ and create set $B$, initially empty. Repeat the following until $\psi$ is empty. Pick an equation $e$ of $\psi$ that has the highest index, non-zero term. As long as there is an equation $g$ in $B$ whose highest non-zero term has the same index as the highest index non-zero term of $e$, replace $e$ with $e + g$. If $0=0$ is not produced, add $e$ to $B$. This ensures $B$ remains linearly independent. Create as many as $n$ new equations by multiplying $e$ by every variable and add those equations to $\psi$ that have never been in $\psi$. This sets up the addition of $e$ with all other equations in $B$. When $\psi$ is empty, all original equations have been replaced by equations with the same solution space and are such that no two of them have the same highest index non-zero term.

Next we show some examples of inputs and then two short derivations.

The CNF clause

\[(v_1 \lor v_2 \lor v_3)\]

is represented by the equation

\[v_1(1 + v_2)(1 + v_3) + v_2(1 + v_3) + v_3 + 1 = 0\]

which may be rewritten

\[v_1v_2v_3 + v_1v_2 + v_1v_3 + v_2v_3 + v_1 + v_2 + v_3 + 1 = 0.\]

The reader can verify this from the truth table for the clause. Negative literals in a clause are handled by replacing variable symbol $v$ with $(1 + v)$. For example, the clause

\[\neg(v_1 \lor v_2 \lor v_3)\]
is represented by
\[(1 + v_1)(1 + v_2)(1 + v_3) + v_2(1 + v_3) + v_3 + 1 = 0\]
which reduces to
\[v_1v_2v_3 + v_1v_2 + v_1v_3 + v_1 = 0. \quad (3.6)\]
As can be seen, just the expression of a clause introduces non-linearities. However, this is not the case for some Boolean functions. For example, the exclusive-or formula
\[v_1 \oplus v_2 \oplus v_3 \oplus v_4\]
is represented by
\[v_1 + v_2 + v_3 + v_4 + 1 = 0.\]
An equation representing a BDD (Section 3.9) can be written directly from the BDD as a sum of algebraic expressions constructed from paths to 1 because each path represents one or more rows of a truth table and the intersection of rows represented by any two paths is empty. Each expression is constructed incrementally while tracing a path as follows: when a 1 branch is encountered for variable \(v\), multiply by \(v\), and when a 0 branch is encountered for variable \(v\), multiply by \((1 + v)\). Observe that for any truth assignment, at most one of the expressions has value 1. The equation corresponding to the BDD at the upper left in Figure 3.13 is
\[(1 + v_1)(1 + v_2)(1 + v_3) + (1 + v_1)v_2v_3 + v_1(1 + v_2)v_3 + v_1v_2 + 1 = 0\]
which reduces to
\[v_1 + v_2 + v_3 + v_1v_2v_3 = 0.\]
Since there is a BDD for every Boolean function, this example illustrates the fact that a single equation can represent any complex function. It should be equally clear that a single equation addition may have the same effect as many resolution steps.

Addition of equations and the Gaussian-elimination nature of algebraic proofs is illustrated by showing steps that solve the following simple formula:
\[(v_1 \lor \neg v_2) \land (v_2 \lor \neg v_3) \land (v_3 \lor \neg v_1) \quad (3.7)\]
The equations corresponding to the above are (1), (2), and (3). All equations below the line are derived as stated on the right.
\[v_1v_2 + v_2 = 0 \quad (1)\]
\[v_2v_3 + v_3 = 0 \quad (2)\]
\[v_1v_3 + v_1 = 0 \quad (3)\]
The solution is given by the bottom two equations which state that $v_1 = v_2 = v_3$. If, say, the following two clauses are added to (3.7)

$$(\neg v_1 \lor \neg v_2) \land (v_3 \lor v_1)$$

the equation $v_3 + v_2 + 1 = 0$ could be derived. Adding this to (10) would give $1 = 0$ which proves that no solution exists.

Ensuring a derivation of reasonable length is difficult. One possibility is to limit derivations to equations of bounded degree where the degree of a term $t$, $\deg(t)$, is defined in the proof of Theorem 26 and the degree of an equation is $\deg(e) = \max\{\deg(t) : t \text{ is a non-zero term in } e\}$. An example is Algorithm 14 of Figure 3.25 which is adapted from [35]. In the algorithm terms are re-indexed as in Theorem 26. Then $\text{first\_non\_zero}(e_i)$ is used to mean the highest index of a non-zero term of $e_i$. The function $\text{reduce}(e)$ is an explicit statement that says reduction rules 1. and 2. are applied as needed to produce a multi-linear equation.

We close by comparing equations and the algebraic method with BDDs and BDD operations. Consider an example taken from Section 3.9. Equations corresponding to $f$ and $c$ in Figure 3.17 are

$$f : \quad v_1 v_3 + v_2 + v_1 v_2 = 0$$

$$c : \quad v_2 v_3 + v_3 = 0$$

Multiply $f$ by $v_2 v_3$ to get $v_2 v_3 = 0$ which adds with $c$ to get $v_3 = 0$, the inference that is missed by $\text{restrict}(f, c)$ in Figure 3.17. The inference can be derived from BDDs by reversing the role of $f$ and $c$ as shown in Figure 3.18. Consider what multiplying $f$ by $v_2 v_3$ and adding to $c$ means in the BDD world. The BDD representing $v_2 v_3$, call it $d$, consists of two internal nodes $v_2$ and $v_3$, a path to 0 following only 1 branches, and all other paths terminating at 1. Every path that terminates at 1 in $f$ also terminates at 1 in $d$. Therefore, $d \land c$ can safely be added as a BDD as long as $f$ remains. But it is easy to check that $d \land c$ is simply $v_3 = 0$.

The process used in the above example can be applied more generally. All that is needed is some way to create a “best” factor $d$ from $f$ and $c$. This
is something a generalized co-factor, which is discussed in Section 3.9.4, can sometimes do. However, the result of finding a generalized co-factor depends on BDD variable ordering. For the ordering \( v_1 < v_2 < v_3 \) the generalized co-factor \( g = \text{gcf}(f, c) \) turns out to be \((v_1 \lor \neg v_2 \lor v_3) \land (\neg v_2 \lor \neg v_3)\) which is different from \( d \) in the leading clause but is sufficient to derive the inference when conjoined with \( c \). By the definition of \( \text{gcf} \), since \( f \land c = g \land c \), \( g \) may replace \( f \) - this is not the case for \( d \) above.

Existentially quantifying \( v \) away from a BDD has a simple counterpart in algebra: just multiply two polynomials, one with restriction \( v = 1 \) and the other with restriction \( v = 0 \). For example, the BDD of Figure 3.13 may be expressed as

\[
v_1v_2v_3 + v_1v_3 + v_1 + 1 = 0.
\]

The equations under restrictions \( v_2 = 1 \) and \( v_2 = 0 \), respectively, are

\[
v_1 + 1 = 0 \quad \text{and} \quad v_1v_3 + v_1 + 1 = 0.
\]

The result of existential quantification is

\[
(v_1 + 1)(v_1v_3 + v_1 + 1) = v_1 + 1 = 0
\]

which reveals the same inference. As with BDDs, this can be done only if the quantified variable is in no other equation.

The counterpart to strengthening is just as straightforward. The BDDs of Figure 3.24 have equation representations

\[
b_2 : \quad v_1v_3 + v_2 + v_1v_2 = 0 \]
\[
b_1 : \quad v_3 + v_2v_3 = 0.
\]

Existentially quantify \( v_1 \) away from \( b_2 \) to get \( v_2v_3 = 0 \) and add this to \( b_1 \) to get \( v_3 = 0 \).

### 3.10.2 Integer Programming

An Integer Program models the problem of maximizing or minimizing a linear function subject to a system of linear constraints, where all \( n \) variables are integral:

\[
\begin{align*}
\text{maximize or minimize} \quad & \mathbf{c} \quad \mathbf{\alpha} \\
\text{subject to} \quad & \mathbf{M}\mathbf{\alpha} \leq \mathbf{b} \\
& l \leq \mathbf{\alpha} \leq u \\
& \alpha_i \text{ integral, } 1 \leq i \leq n
\end{align*}
\]

where \( \mathbf{M} \) is a constraint matrix, \( \mathbf{c} \) is a linear objection function, \( \mathbf{b} \) is a constant vector, and \( \mathbf{\alpha} \) is a variable vector.
The Integer Programming problem and its relaxation to Linear Programming are very well studied and a large body of techniques have been developed to assist in establishing an efficient solution to (3.8). Since it was shown in Section 2.3.1 that an instance of Satisfiability can be modeled as an Integer Program, these techniques can be applied to Satisfiability. They are divided into the categories of preprocessing and solving. However, an important third aspect concerns the matrix $M$ that is used to model a given instance.

Modeling is important because the effective solution of Integer Programs often entails the use of Linear Programming relaxations. A solution to such a relaxation generally provides a bound on the actual solution and the relaxation of one formulation of the input may provide a tighter bound than another. Generally, the tighter the bound, the better.

For example, consider two formulations of the pigeon-hole problem which is discussed further in Section 6.2 in regards to resolution. The pigeon-hole problem is: can $n+1$ pigeons be placed in $n$ holes so that no two pigeons are in the same hole. Define Boolean variables $v_{i,j}, 1 \leq i \leq n, 1 \leq j \leq n+1$ with the interpretation that $v_{i,j}$ will take the value 1 if and only if pigeon $j$ is in hole $i$ and otherwise will take value 0. The following equations, one per pigeon, express the requirement that every pigeon is to be assigned to a single hole:

$$\sum_{i=1}^{n} v_{i,j} = 1, \quad 1 \leq j \leq n+1, \quad (3.9)$$

and the following inequalities express the requirement that two pigeons cannot occupy the same hole:

$$v_{i,j} + v_{i,k} \leq 1, \quad 1 \leq i \leq n, 1 \leq j < k \leq n+1. \quad (3.10)$$

There is no solution to this system of equations and inequalities. Relaxing integrality constraints there is a solution at $v_{i,j} = 1/n$ for all $i,j$. Running the algorithm to be shown later, practically complete enumeration is necessary before finally it is determined that no solution exists [77]. However, the requirement that at most one pigeon is in a hole may alternatively be represented by

$$\sum_{j=1}^{n+1} v_{i,j} \leq 1, \quad 1 \leq i \leq n. \quad (3.11)$$

which may be used instead of (3.10). The new constraints are much tighter than (3.10) and the system (3.9) and (3.11) is easily solved [77]. The reader may compare this formulation with that given in Section 6.2.

The purpose of preprocessing is to reformulate a given Integer Program and tighten its Linear Programming relaxation. In the process it may eliminate redundant constraints and may even be able to discover unsatisfiability.
3.11 Cutting Planes
3.12 Elliptical Cuts
3.13 Satisfiability Modulo Theories
3.14 Simulated Annealing
3.15 Genetic Algorithms
3.16 Constraint Programming
3.17 Lagrangian Duality
3.18 TABU Search
3.19 Stable Models
3.20 Well-Founded Models
3.21 Answer Set Programming
Figure 3.13: Two examples of existentially quantifying a variable away from a function. Functions are represented as BDDs on the left. Variable \( v_3 \) is existentially quantified away from the top BDD leaving 1, meaning that regardless of assignments given to variables \( v_1 \) and \( v_2 \) there is always an assignment to \( v_3 \) which satisfies the function. Variable \( v_2 \) is existentially quantified away from the bottom BDD leaving the inference \( v_1 = 1 \).
Figure 3.14: Existential quantification can cause blurring of functional relationships. The top function is seen to separate variables $v_6$, $v_7$, and $v_8$ from $v_3$, $v_4$, and $v_5$ if $v_2$ is chosen during search first. Existentially quantifying $v_2$ away from the top function before search results in the bottom function in which no such separation is immediately evident. Without existential quantification the assignment $v_1 = 0$, $v_2 = 1$, $v_3 = 1$ reveals the inference $v_4 = 1$. With existential quantification the assignment must be augmented with $v_7 = 0$ and $v_8 = 0$ (but $v_2$ is no longer necessary) to get the same inference.
Algorithm 11.

```
restrict (f, c) /* Input: BDD f, BDD c */
    /* Output: BDD f restricted by c */
    If c or f is terminal(1) or if f is terminal(0) return f.
    If c == ¬f return terminal(0).
    If c == f return terminal(1).
    // f and c have a non-trivial relationship
    Set v_f ← root(f). // v_f is a variable
    Set v_c ← root(c). // v_c is a variable
    If index(v_f) > index(v_c) return restrict(f, exQuant(c, v_c)).
    If reduce_0(v_f, c) is terminal(0) then
        Return restrict(reduce_1(v_f, f), reduce_1(v_f, c)).
    If reduce_1(v_f, c) is terminal(0) then
        Return restrict(reduce_0(v_f, f), reduce_0(v_f, c)).
    Set h_f1 ← restrict(reduce_1(v_f, f), reduce_1(v_f, c)).
    Set h_f0 ← restrict(reduce_0(v_f, f), reduce_0(v_f, c)).
    If h_f1 == h_f0 then Return h_f1.
    Return findOrCreateNode(v_f, h_f1, h_f0).
```

Figure 3.15: Algorithm for restricting a BDD f by a BDD c.
Chapter 3. GENERAL ALGORITHMS

\[ f = (v_1 \rightarrow v_2) \land (\neg v_1 \rightarrow (\neg v_3 \land v_4)) \]

Figure 3.16: A call to \texttt{restrict}(f, c) returns the BDD \( g \) shown on the right. In this case inferences \( v_1 = 1 \) and \( v_2 = 1 \) are revealed. The symbol \( \Rightarrow \) denotes the operation.

\[ f = (v_1 \lor \neg v_2) \land (\neg v_1 \lor \neg v_3) \quad c = (v_2 \lor \neg v_3) \]

Figure 3.17: A call to \texttt{restrict}(f, c) results in no change.

\[ f = (v_2 \lor \neg v_3) \quad c = (v_1 \lor \neg v_2) \land (\neg v_1 \lor \neg v_3) \quad g = (\neg v_3) \]

Figure 3.18: Reversing the roles of \( f \) and \( c \) in Figure 3.17, a call to \texttt{restrict}(f, c) results in the inference \( g = \neg v_3 \) as shown on the right. In this case, the large number of 0 truth table rows for \( c \) was exploited to advantage.
\[ f = (v_1 \lor \neg v_2) \land (\neg v_1 \lor v_3) \quad c = (\neg v_1 \lor v_3) \quad g = (v_1 \lor \neg v_2) \]

Figure 3.19: A call to \texttt{restrict}(f, c) spreads an inference that is evident in one BDD over multiple BDDs. If \( v_3 \) is assigned 0 in \( f \) then \( v_1 = 0 \) and \( v_2 = 0 \) are inferred. After replacing \( f \) with \( g = \texttt{restrict}(f, c) \), to get the inference \( v_2 = 0 \) from the choice \( v_3 = 0 \), visit \( c \) to get \( v_1 = 0 \) and then \( g \) to get \( v_2 = 0 \). Thus, \texttt{restrict} can increase work if not used properly. In this case, restricting in the reverse direction leads to a better result.

Algorithm 12.

\[
\texttt{gcf}(f, c) \\
\text{/* Input: BDD } f, \text{ BDD } c \text{ */} \\
\text{/* Output: greatest co-factor of } f \text{ by } c \text{ */} \\
\]

If \( f = \text{terminal}(0) \) or \( c = \text{terminal}(0) \) return \( \text{terminal}(0) \).

If \( c = \text{terminal}(1) \) or \( f = \text{terminal}(1) \) return \( f \).

Set \( v_m \leftarrow \text{index}^{-1} (\min\{\text{index}(\text{root}(c)), \text{index}(\text{root}(f))\}) \).

// \( v_m \) is the top variable of \( f \) and \( c \)

If \( \text{reduce}_0(v_m, c) = \text{terminal}(0) \) then

Return \( \text{gcf}(\text{reduce}_1(v_m, f), \text{reduce}_1(v_m, c)) \).

If \( \text{reduce}_1(v_m, c) = \text{terminal}(0) \) then

Return \( \text{gcf}(\text{reduce}_0(v_m, f), \text{reduce}_0(v_m, c)) \).

Set \( h_1 \leftarrow \text{gcf}(\text{reduce}_1(v_m, f), \text{reduce}_1(v_m, c)) \).

Set \( h_0 \leftarrow \text{gcf}(\text{reduce}_0(v_m, f), \text{reduce}_0(v_m, c)) \).

If \( h_1 = h_0 \) then Return \( h_1 \).

Return \( \text{FindOrCreateNode}(v_m, h_1, h_0) \).

Figure 3.20: Algorithm for finding a greatest common co-factor of a BDD.
\[ f = (v_1 \rightarrow \neg v_2) \lor (\neg v_1 \rightarrow v_3) \quad c = (v_2 \lor v_3) \]

\[ gcf(f, c) = (v_1 \rightarrow \neg v_2) \lor (\neg v_1 \rightarrow (v_2 \rightarrow v_3)) \]

Figure 3.21: Generalized co-factor operation on \( f \) and \( c \) as shown. In this case the result is more complicated than \( f \). The variable ordering is \( v_1 < v_2 < v_3 \).
\[ f = (v_3 \rightarrow \neg v_2) \vee (\neg v_3 \rightarrow v_1) \]

3.21 ANSWER SET PROGRAMMING

\[ c = (v_1 \vee v_2) \]

Figure 3.22: Generalized co-factor operation on the same \( f \) and \( c \) of Figure 3.21 and with the same variable ordering but with \( v_1 \) and \( v_3 \) swapped. In this case the result is less complicated than \( f \) and the assignment \( \{v_1, v_2\} \) causes the output of \( \text{gcf} \) in this figure to have value 1 whereas the output of \( \text{gcf} \) in Figure 3.21 has value 0 under the same assignment.

Algorithm 13.

\[ \text{Strengthen} \ (b_1, b_2) \]

/* Input: BDD \( b_1 \), BDD \( b_2 \) */
/* Output: BDD \( b_1 \) strengthened by \( b_2 \) */

Set \( \bar{x} \leftarrow \{x : x \in b_2, x \notin b_1\} \).

Repeat the following for all \( x \in \bar{x} \):

Set \( b_2 \leftarrow \text{exQuant}(b_2, x) \).

Return \( b_1 \land b_2 \).

Figure 3.23: Algorithm for strengthening a BDD by another.
Strengthening example: Existentially quantify $v_1$ away from $b_2$...

$\exists v_1 b_2 : \quad b_1 :$

Figure 3.24: ...then conjoin the two BDDs. Inference $v_3 = 0$ is revealed.
Algorithm 14.

**An Algebraic Solver** $(\psi, d)$

/* Input: List of equations $\psi = \langle e_1, \ldots, e_m \rangle$, integer $d$ */
/* Output: “satisfiable” or “unsatisfiable” */
/* Locals: Set $B$ of equations */

Set $B \leftarrow \emptyset$.

Repeat while $\psi \neq \emptyset$:

Pop $e \leftarrow \psi$.

Repeat while $\exists e' \in B : \text{first\_non\_zero}(e) = \text{first\_non\_zero}(e')$:

Set $e \leftarrow \text{reduce}(e + e')$. /* Rule 4. */

If $e$ is $1 = 0$: Output “unsatisfiable”

If $e$ is not $0 = 0$:

Set $B \leftarrow B \cup \{e\}$.

If degree$(e) < d$:

Repeat for all variables $v$:

If $\text{reduce}(ve)$ has not been in $\psi$:

Append $\psi \leftarrow \text{reduce}(ve)$. /* Rule 3. */

Output “satisfiable”.

Figure 3.25: Simple algebraic algorithm for SAT.
Chapter 4

Algorithms for Easy Classes of CNF Formulas

For certain classes of CNF formulas, the Satisfiability problem is known to be solved efficiently by specially designed algorithms. Some classes, such as the 2-SAT and Horn classes are quite important because they show up in real applications and others are quite interesting because results on these add to our understanding of the structural properties that make formulas hard or easy. Such an understanding can help develop a search heuristic that will obtain a solution more efficiently. In particular, knowledge that a large subset of clauses of a given formula belongs to some easy class of formulas can help reduce the size of the search space needed to determine whether some partial truth assignment can be extended to a solution. Because CNF formulas are so rich in structure we have devoted much space in this chapter to a few special cases to help fully appreciate the possibilities. In particular, we have greatly detailed results on minimally unsatisfiable and nested formulas.

The reader may have the impression that the number of polynomial time solvable classes is quite small due to the famous dichotomy theorem of Schaefer [117]. But this is not the case. Schaefer proposed a scheme for defining classes of propositional formulas with a generalized notion of “clause.” He proved that every class definable within his scheme was either \( \mathcal{NP} \)-complete or polynomial-time solvable, and he gave criteria to determine which. But not all classes can be defined within his scheme. The class of Horn formulas can be but we will describe several others including q-Horn, extended Horn, CC-balanced and SLUR that cannot be so defined. The reason is that Schaefer’s scheme is limited to classes that can be recognized in log space.

Below, we identify some of the more notable easy classes and present algorithms for their solution. A crucial component of many of these algorithms
Algorithm 15.

Unit Resolution ($\psi$)
/* Input: set of sets CNF formula $\psi$  */
/* Output: pair $\langle$CNF formula $\phi$, partial assignment $P$$\rangle$ */
/* $\psi$ is satisfiable if and only if $\phi$ is satisfiable */
/* $\phi$ has no unit clauses */
/* Locals: set of variables $P$, set of sets CNF formula $\phi$ */
Set $\phi \leftarrow \psi$; Set $P \leftarrow \emptyset$.
Repeat the following while $\emptyset \notin \phi$ and there is a unit clause in $\phi$:
Let $\{l\} \in \phi$ be a unit clause.
If $l$ is a positive literal, Set $P \leftarrow P \cup \{l\}$.
Set $\phi \leftarrow \{c - \{\neg l\} : c \in \phi, l \notin c\}$.
Output $\langle \phi, P \rangle$.

Figure 4.1: Unit resolution for CNF formulas.

is unit resolution. An implementation is given in Figure 4.1.

4.1 2-SAT

All clauses of a 2-SAT formula contain at most two literals. A given 2-SAT formula $\psi$ may be solved efficiently by constructing the implication graph $\vec{G}_\psi$ of $\psi$ (see Section 2.3.3) and traversing its vertices, ending either at a cycle containing complementary literals or with no additional vertices to explore. The algorithm of Figure 4.2 implicitly does this.

Unit resolution drives the exploration of strongly connected components of $\vec{G}_\psi$. The initial application of unit resolution, if necessary, is analogous to traversing all vertices reachable from the special vertex $F$. Choosing a literal $l$ arbitrarily and temporarily assigning it the value 1 after unit resolution completes is analogous to starting a traversal of some strongly connected component with another round of unit resolution. If that round completes with an empty clause, a contradiction exists so $l$ is set to 0, $\phi$ and $M'$ are reset to what they were just before $l$ was set to 1, and exploration resumes. If an empty clause is encountered before the entire component is visited (that is, while there still exist unit clauses) then the formula is unsatisfiable. Otherwise, the value of $l$ is made permanent and so are values that were given to other variables during traversal of the component. This process repeats until the formula is found to be unsatisfiable or all components have been explored. The variable named $s$ keeps track of whether variable $l$ has been given one value or two, the variable $M'$ holds the temporary assignments to
Algorithm 16.

**2-SAT Solver** ($\psi$)

/* Input: set of sets 2-CNF formula $\psi$ */
/* Output: “unsatisfiable” or a model for $\psi$ */
/* Locals: variable $s$, set of variables $M, M'$, set of sets formula $\phi$ */

Set $\phi \leftarrow \psi$; Set $s \leftarrow 1$; Set $M \leftarrow \emptyset$; Set $M' \leftarrow \emptyset$; Set $\phi' \leftarrow \emptyset$.

Repeat the following until some statement outputs a value:

Set $(\phi, M') \leftarrow \text{Unit Resolution} (\phi)$.

If $\emptyset \in \phi$ then do the following:

- If $s$ has value 1 or $\phi' = \emptyset$ then Output “unsatisfiable”.
- Set $s \leftarrow 1$
- Set $M' \leftarrow \emptyset$; $\phi \leftarrow \phi'$; $l \leftarrow l'$.
- If $l$ is a negative literal, Set $M' \leftarrow \neg l$.
- Set $\phi \leftarrow \{ c - \{ l \} : c \in \phi, \neg l \notin c \}$.

Otherwise, if $\phi \neq \emptyset$ then do the following:

- Set $s \leftarrow 0$.
- Choose a literal $l$ arbitrarily from a clause of $\phi$.
- Set $M \leftarrow M \cup M'$.
- Set $\phi' \leftarrow \emptyset$.
- Set $l' \leftarrow l$.
- If $l$ is a positive literal, Set $M' \leftarrow \{ l \}$.
- Set $\phi \leftarrow \{ c - \{ \neg l \} : c \in \phi, l \notin c \}$.

Otherwise, Output $M \cup M'$.

Figure 4.2: Algorithm for determining satisfiability of 2-CNF formulas.

variables during traversal of a component, and the variables $\phi'$ and $l'$ save the point to return to if a contradiction is found. This algorithm is adapted from [51].

The next two theorems are stated without proof.

**Theorem 27.** On input CNF formula $\psi$, Algorithm 2-SAT Solver outputs “unsatisfiable” if and only if $\psi$ is unsatisfiable and if it outputs a set $M$, then $M$ is a model for $\psi$.

**Theorem 28.** On input CNF formula $\psi$ containing $m$ clauses and $n$ variables, Algorithm 2-SAT Solver has $O(m + n)$ worst case complexity.
Algorithm 17.

**Horn Solver** \((\psi)\)

/* Input: set of sets Horn formula \(\psi\) */

/* Output: “unsatisfiable” or a model for \(\psi\) */

/* Locals: set of variables \(M\) */

Set \(M \leftarrow \emptyset\).

Repeat the following until no positive literal unit clauses are in \(\psi\):

Choose \(v\) from a positive literal unit clause \(\{v\} \in \psi\).

Set \(M \leftarrow M \cup \{v\}\).

Set \(\psi \leftarrow \{c - \{\neg v\} : c \in \psi, v \notin c\}\).

If \(\emptyset \in \psi\), Output “unsatisfiable.”

Output \(M\).

\(\square\)

Figure 4.3: Algorithm for determining satisfiability of Horn formulas.

### 4.2 Horn Formulas

A CNF formula is Horn if every clause in it has at most one positive literal. This class is widely studied, in part because of its close association with Logic Programming. To illustrate, a Horn clause \((\neg v_1 \lor \neg v_2 \lor \ldots \lor \neg v_i \lor v)\) is equivalent to the rule \(v_1 \land v_2 \land \ldots \land v_i \rightarrow v\) or the implication \(v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i \rightarrow v\). However, the notion of causality is generally lost when translating from rules to Horn formulas.

The following states an important property of Horn formulas.

**Theorem 29.** Every Horn formula has a unique minimum model.

*Proof.* Let \(\psi\) be a Horn formula. Let \(M\) be a minimal model for \(\psi\), that is, a smallest subset of variables of value 1 that satisfies \(\psi\). Choose any \(v \in M\). Since \(M \setminus \{v\}\) is not a model for \(\psi\), there must be a clause \(c \in \psi\) such that positive literal \(v \in c\). Since all other literals of \(c\) are negative and \(M\) is minimal, all assignments not containing \(v\) cannot satisfy \(c\) and therefore \(\psi\). It follows that all models other than \(M\) must have cardinality greater than \(|M|\). Hence \(M\) is a unique minimum model for \(\psi\). \(\square\)

The satisfiability of Horn formulas can be determined in linear time using unit resolution [48, 74, 121]. One of several possible variants is shown in Figure 4.3.

**Theorem 30.** Given Horn formula \(\psi\) as input, Algorithm **Horn Solver** outputs “unsatisfiable” if and only if \(\psi\) is unsatisfiable and if it outputs a set of variables \(M\), then \(M\) is a unique minimum model for \(\psi\).
4.3 RENAMEABLE HORN FORMULAS

Proof. When Algorithm **Horn Solver** completes with output set $M$, all remaining clauses have at least one negative literal. Since none of the remaining clauses are null and since $v$ added to $M$ serves to falsify negative literals, at least one of the remaining negative literals in a remaining clause has not been added to $M$ and is therefore satisfied by $M$. Therefore, all remaining clauses are satisfied by $M$. A clause is removed when adding $v$ to $M$ only because it contains literal $v$. Therefore all removed clauses are satisfied by $M$. Hence, $M$ satisfies $\psi$.

Suppose $M$ is not the unique minimum model for $\psi$. Then, for some variable $v$, the assignment $M \setminus \{v\}$ satisfies $\psi$. Variable $v$ was added to $M$ because some clause $c \in \psi$ containing positive literal $v$ became a unit clause after all variables associated with the negative literals of $c$ were placed in $M$. But then $M \setminus \{v\}$ cannot satisfy $c$. Therefore $M$ is the unique minimum model.

Now suppose the algorithm outputs “unsatisfiable” but there is a model for $\psi$. Let $M'$ be the unique minimum model. Run the algorithm until reaching the point at which an empty clause is generated. Let this happens on the $i^{th}$ iteration of the Repeat block and let $\psi'$ be the set of all clauses removed up to the test for an empty clause on the $i - 1$st iteration. Let $v$ be the last variable added to $M$ and $c$ be the unit clause from which it was obtained. Clearly, $\psi'$ is Horn and $M \setminus \{v\}$ is its unique minimum model. $M'$ must contain all variables of $M \setminus \{v\}$ since it is the unique minimum model for $\psi$. Therefore, it cannot contain $\{v\}$. But then $c$ is not satisfied by $M'$, a contradiction.

Algorithm **Horn Solver** is only useful if the input formula is known to be Horn. It is easy to see that this can be checked in linear time.

### 4.3 Renameable Horn Formulas

Given CNF formula $\psi$ and variable subset $V' \subset V$, define $\text{switch}(\psi, V')$ to be the formula obtained from $\psi$ by reversing the polarity of all occurrences of $v$ and $\neg v$ in $\psi$ for all $v \in V'$. If there exists a $V' \subset V$ such that $\text{switch}(\psi, V')$ is Horn, then $\psi$ is said to be **renameable Horn** or **hidden Horn**.

Renameable Horn formulas can be recognized and solved in $O(|\psi|)$ time [13, 101]. Algorithm **SLUR** on Page 120 solves renameable Horn formulas in linear time.

### 4.4 Linear Programming Relaxations

Let $M_\psi$ be a $(0, \pm 1)$ matrix representing a CNF formula $\psi$. If $M_\psi$ has a particular structure it is possible to solve the inequalities 2.1 with non-integer constraints $0 \leq \alpha_i \leq 1$ to obtain a solution to $\psi$ either directly or
by rounding. Notable classes based on particular matrix structures are the extended Horn formulas and what we call the CC-balanced formulas.

The class of **extended Horn formulas** was introduced by Chandru and Hooker [29] who were looking for conditions under which a Linear Programming relaxation could be used to find solutions to propositional formulas. It is based on a theorem of Chandrasekaran [28] which characterizes sets of linear inequalities for which 0-1 solutions can always be found (if one exists) by rounding a real solution obtained using an LP relaxation. Extended Horn formulas can be expressed as linear inequalities that belong to this family of 0-1 problems.

We choose to present an equivalent graph theoretic definition. A formula $\psi$ is in the class of extended Horn formulas if one can construct a rooted directed tree $T$, called an extended Horn tree, indexed on the variables of $\psi$ such that, for every clause $c$ in $\psi$:

1. All the positive literals of $c$ are consecutive on a single path of $T$.
2. There is a partition of the negative literals of $c$ into sets $N_1, N_2, ..., N_{n_c}$, where $n_c$ is at least 1, but no greater than the number of negative literals of $c$, such that for all $1 \leq i \leq n_c$, all the variables of $N_i$ are consecutive on a single path of $T$.
3. For at most one $i$, the path in $T$ associated with $N_i$ begins at the vertex in $T$ from which the path associated with positive literals begins.
4. For all remaining $i$, the path in $T$ associated with $N_i$ begins at the root of $T$.

Disallowing negative paths that do not originate at the root (point 3. above) gives a subclass of extended Horn called **simple extended Horn** [123]. Extended Horn formulas can be solved in polynomial time by Algorithm **SLUR** on Page 120 [119].

Chandru and Hooker showed that unit resolution alone can determine whether or not a given extended Horn formula is satisfiable. This is due to the following two properties of an extended Horn formula:

1. If $\psi$ is extended Horn and has no unit clauses then $\psi$ is satisfiable.
2. If $\psi$ is extended Horn and $v \in V_\psi$ then $\psi_1 = \{c - \{v\} : c \in \psi, \neg v \notin c\}$ and $\psi_2 = \{c - \{-v\} : c \in \psi, v \notin c\}$ are both extended Horn.

Chandru and Hooker proposed an algorithm that finds a model for a satisfiable formula. First, apply unit resolution, setting values of unassigned variables to 1/2 when no unit clauses remain. Then round the result by a matrix multiplication. Their algorithm cannot, however, be reliably applied unless it is known that a given formula is extended Horn. Unfortunately, the
The problem of recognizing extended Horn formulas is not known to be solved in polynomial time. As will be shown later in this section, this problem has become moot since Algorithm \textbf{SLUR} solves extended Horn formulas in linear time without the need for recognition.

The class of CC-balanced formulas has been studied by several researchers (see [36] for a detailed account of balanced matrices and a description of CC-balanced formulas). The motivation for this class is the question, for SAT, when do Linear Programming relaxations have integer solutions? A formula $\psi$ with $(0, \pm 1)$ matrix representation $M_{\psi}$ is \textit{CC-balanced} if in every submatrix of $M_{\psi}$, with exactly two nonzero entries per row and per column, the sum of the entries is a multiple of four (this definition is taken from [128]). Recognizing that a formula is CC-balanced takes linear time. However, the recognition problem is moot because Algorithm \textbf{SLUR} solves CC-balanced formulas in linear time without the need for recognition.

As alluded to above, both extended Horn and CC-balanced formulas are subsets of a larger efficiently solved class of formulas solved by \textit{Single Lookahead Unit Resolution} [119] (SLUR). The SLUR class is peculiar in that it is defined based on an algorithm rather than on properties of formulas. Algorithm \textbf{SLUR} of Figure 4.4, selects variables sequentially and arbitrarily and considers a one-level lookahead, under unit resolution, of both possible values that the selected variable can take. If unit resolution does not result in an empty clause in one direction, the assignment corresponding to that value choice is made permanent and variable selection continues. If all clauses are satisfied after a value is assigned to a variable (and unit resolution is applied), the algorithm returns a satisfying assignment. If unit resolution, applied to the given formula or to both sub-formulas created from assigning values to the selected variable on the first iteration, results in a clause that is falsified, the algorithm reports that the formula is unsatisfiable. If unit resolution results in falsified clauses as a consequence of both assignments of values to the selected variable on any iteration except the first, the algorithm reports that it has given up.

A formula is in the class SLUR if, for all possible sequences of selected variables, Algorithm \textbf{SLUR} does not give up on that formula. Observe that due to the definition of this class, the question of class recognition is avoided.

The worst case complexity of Algorithm \textbf{SLUR}, as written, is quadratic in the length of the input formula. The complexity is dominated by the execution of \{ $c - \{v\} : c \in \psi, \neg v \notin c \}$, \{ $c - \{\neg v\} : c \in \psi, v \notin c \}$, and the number of unit clauses eliminated by unit resolution. The total number of times a clause is checked and a literal removed due to the first two expressions is at most the number of literals existing in the given formula if the clauses are maintained in a linked list indexed on the literals. However, the same unit clause may be removed by unit resolution on successive iterations of the Repeat block of Algorithm \textbf{SLUR} since once branch of execution is always cut. This causes quadratic worst case complexity.

A simple modification to Algorithm \textbf{SLUR} brings the complexity down
Algorithm 18.

\begin{verbatim}
SLUR ($\psi$)
/* Input: a set of sets CNF formula $\psi$ */
/* Output: "unsatisfiable" or a model for $\psi$ */
/* Locals: set of variables $M$, flag $d$, formula $\psi'$ */
Set $\langle \psi', M \rangle \leftarrow$ \textbf{Unit Resolution} ($\psi$).
If $\emptyset \in \psi'$ then Output "unsatisfiable."
Set $d \leftarrow 0$ /* $d = 0$ iff execution is at the top level. */
Repeat the following while $\psi' \neq \emptyset$:
    Choose arbitrarily a variable $v \in V_{\psi'}$.
    Set $\langle \psi_1, M_1 \rangle \leftarrow$ \textbf{Unit Resolution} ($\{c - \{v\} : c \in \psi', \neg v \notin c\}$).
    Set $\langle \psi_2, M_2 \rangle \leftarrow$ \textbf{Unit Resolution} ($\{c - \{\neg v\} : c \in \psi', v \notin c\}$).
    If $\emptyset \in \psi_1$ and $\emptyset \in \psi_2$ then do the following:
        If $d = 0$ then Output "unsatisfiable."
        Otherwise, Output "give up."
    Otherwise, do the following:
        Arbitrarily choose $i$ so that $\emptyset \notin \psi_i$
        Set $\psi' \leftarrow \psi_i$.
        If $i = 1$, $M \leftarrow M \cup M_1$.
        Otherwise, $M \leftarrow M \cup M_2 \cup \{v\}$.
    Set $d \leftarrow 1$
Output $M$
\end{verbatim}

Figure 4.4: Algorithm for determining satisfiability of SLUR formulas.

to linear time: run both calls of unit resolution simultaneously, alternating execution of their Repeat blocks. When one terminates without an empty clause in its output formula, abandon the other call.

\textbf{Theorem 31.} Algorithm SLUR has $O(|\psi|)$ worst case complexity if both calls to unit resolution are applied simultaneously and one call is immediately abandoned if the other finishes first without falsifying a clause.

\textit{Proof.} For reasons mentioned above we only consider the number of steps used by unit resolution. The number of times a literal from a unit clause is chosen and satisfied clauses and falsified literals removed in non-abandoned calls of unit resolution is $O(|\psi|)$ since no literal is chosen twice. Since the Repeat blocks of abandoned and non-abandoned calls alternate, the time used by abandoned calls is no greater than that used by non-abandoned ones. Thus, the worst case complexity of Algorithm SLUR, with the interleave
modification, is $O(|\psi|)$. \hfill \square

All Horn, renameable Horn, extended Horn, and CC-balanced formulas are in the class SLUR. Thus, an important outcome of the results on SLUR is the observation that no special preprocessing or testing is needed for some of the special polynomial time solvable classes of SAT when using a reasonable variant of the DPLL algorithm.

A limitation of all the classes of this section is they do not represent many interesting unsatisfiable formulas. There are several possible extensions to the SLUR class which improve the situation. One is to add a 2-SAT solver to the unit resolution steps of Algorithm SLUR. This extension is at least able to handle all 2-SAT formulas which is something Algorithm SLUR cannot do. It can be elegantly incorporated due to the following observation: whenever Algorithm SLUR completes a sequence of unit resolutions, and if at that time the remaining clauses are nothing but a subset of the original clauses (which they would have to be if all clauses have at most two literals), then effectively the algorithm can start all over. That is, if fixing of a variable to both values leads to an empty clause, then the formula has been proved to be unsatisfiable. Thus, one need not augment Algorithm SLUR by the 2-SAT algorithm, because the 2-SAT algorithm (at least one version of it) does exactly what the extended algorithm does. Another extension of Algorithm SLUR is to allow a polynomial number of backtracks, giving up if at least one branch of the search tree does not terminate at a leaf where a clause is falsified. This enables unsatisfiable formulas with short search trees to be solved efficiently by Algorithm SLUR.

### 4.5 q-Horn Formulas

This class of propositional formulas was developed in [19] and [20]. We choose to characterize the class of q-Horn formulas as a special case of maximum monotone decomposition of matrices [127, 128]. Express a CNF formula of $m$ clauses and $n$ variables as an $m \times n \ (0, \pm 1)$-matrix $M$. In the monotone decomposition of $M$, columns are scaled by $-1$ and the rows and columns are partitioned into submatrices as follows:

$$
\begin{pmatrix}
\mathcal{A}^1 & \mathcal{E} \\
\mathcal{D} & \mathcal{A}^2
\end{pmatrix}
$$

where the submatrix $\mathcal{A}^1$ has at most one +1 entry per row, the submatrix $\mathcal{D}$ contains only $-1$ or 0 entries, the submatrix $\mathcal{A}^2$ has no restrictions other than the three values of $-1$, +1, and 0 for each entry, and the submatrix $\mathcal{E}$ has only 0 entries. If $\mathcal{A}^1$ is the largest possible over columns then the decomposition is a maximum monotone decomposition. If the maximum
Algorithm 19.

**q-Horn Solver** ($\mathcal{M}_\psi$)
/* Input: a $(0, \pm 1)$ matrix representation for q-Horn formula $\psi$ */
/* Output: “unsatisfiable” or a model for $\psi$ */
/* Locals: set of variables $M_1, M_2$ */

Find the maximum monotone decomposition of $\mathcal{M}_\psi$ (Page 82).
If Horn formula $\mathcal{A}^1$ is unsatisfiable then Output “unsatisfiable”.
Let $M_1$ be a unique minimum model for the Horn formula $\mathcal{A}^1$.
Remove from $\mathcal{A}^2$ all rows whose columns in $\mathcal{D}$ are satisfied by $M_1$.
If $\mathcal{A}^2$ is unsatisfiable then Output “unsatisfiable”.
Let $M_2$ be a model for the 2-SAT formula $\mathcal{A}^2$.
Output $M_1 \cup M_2$.

Figure 4.5: Algorithm for determining satisfiability of q-Horn formulas.

monotone decomposition of $\mathcal{M}$ is such that $\mathcal{A}^2$ has no more than two nonzero entries per row, then the formula represented by $\mathcal{M}$ is q-Horn.

Truemper [128] shows that a maximum monotone decomposition for a matrix associated with a q-Horn formula can be found in linear time (this is discussed in Section 3.6.1). Once a q-Horn formula is in its decomposed form it can be solved in linear time by Algorithm *q-Horn Solver* of Figure 4.5.

**Theorem 32.** Given q-Horn formula $\psi$, Algorithm *q-Horn Solver* outputs “unsatisfiable” if and only if $\psi$ is unsatisfiable and if $\psi$ is satisfiable, then the output set $M_1 \cup M_2$ is a model for $\psi$.

*Proof.* Clearly, if Horn formula $\mathcal{A}^1$ is unsatisfiable then so is $\psi$. Suppose $\mathcal{A}^2$ is unsatisfiable after rows whose columns in $\mathcal{D}$ are removed because they are satisfied by $M_1$. Since $M_1$ is a unique minimum model for $\mathcal{A}^1$ and no entries of $\mathcal{D}$ are +1, no remaining row of $\mathcal{A}^2$ can be satisfied by any model for $\mathcal{A}^1$. Therefore, $\psi$ is unsatisfiable in this case. The set $M_1 \cup M_2$ is a model for $\psi$ if it is output since $M_1$ satisfies rows in $\mathcal{A}^1$ and $M_2$ satisfies rows in $\mathcal{A}^2$. □

An equivalent definition of q-Horn formulas comes from the following.

**Theorem 33.** A CNF formula is q-Horn if and only if its satisfiability index is no greater than 1.

*Proof.* Let $\psi$ be a q-Horn formula with variable set $V_\psi$ and suppose $|V_\psi| = n$.
Let $\mathcal{M}_\psi$ be a monotone decomposition for $\psi$. Let $n_a$ be such that for $0 \leq i < n_a$, column $i$ of $\mathcal{M}_\psi$ coincides with submatrices $\mathcal{A}^1$ and $\mathcal{D}$, and for $n_a \leq i < n$, column $i$ coincides with submatrix $\mathcal{A}^2$. Form the inequalities 2.2 from $\mathcal{M}_\psi$. Assign value 1 to all $\alpha_i$, $0 \leq i < n_a$, and value 1/2 to all $\alpha_i$.,
\[ n_a \leq i < n. \] This satisfies \( 0 \leq \alpha_i \leq 1 \) of system 2.2. Since rows of \( A^1 \) have non-zero entries in columns 0 to \( n_a - 1 \) only and at most one of those is +1, the maximum sum of the elements of the corresponding row of inequality 2.2 is 1. Since all rows of \( D \) and \( A^2 \) have at most two non-zero entries in columns \( n_a \) to \( n - 1 \) and no +1 entries in columns 0 to \( n_a - 1 \), the sum of elements of a corresponding row of inequality 2.2 has maximum value 1 too. Thus, inequality 2.2 is satisfied with \( z = 1 \).

Now, suppose \( \psi \) has satisfiability index \( I_\psi \) which is no greater than 1. Choose values for all \( \alpha_i \) terms such that inequality 2.2 is satisfied for \( z = I_\psi \). Let \( \pi \) be a permutation of the columns of \( \mathcal{M}_\psi \) so that \( \alpha_{\pi_i} \leq \alpha_{\pi_j} \) if and only if \( i < j \). Form \( \mathcal{M}'_\psi \) from \( \mathcal{M}_\psi \) by permuting columns according to \( \pi \). Let \( n_b \) be such that \( \alpha_{\pi_i} < 1/2 \) for \( 0 \leq i < n_b \) and \( 1/2 \leq \alpha_{\pi_i} \) for \( n_b \leq i < n \). Scale columns 0 through \( n_b - 1 \) of \( \mathcal{M}'_\psi \) by -1. Then inequality 2.2, using \( \mathcal{M}'_\psi \) for \( \mathcal{M}_\psi \), is satisfied with \( z = I_\psi \) and \( 1/2 \leq \alpha_i \) for all \( 0 \leq i < n \). It follows that each row of \( \mathcal{M}'_\psi \) can have at most two +1 entries or else the elements of the corresponding row of inequality 2.2 sums to greater than 1. Consider all rows of \( \mathcal{M}'_\psi \) with two +1 entries and mark all columns that contain at least one of those entries. Let \( \rho \) be a permutation of the columns of \( \mathcal{M}'_\psi \) so that all marked columns have higher index than all unmarked columns. Form \( \mathcal{M}''_\psi \) from \( \mathcal{M}'_\psi \) by permuting columns according to \( \rho \) and permuting rows so that all rows with only 0 entries in marked columns are indexed lower than rows with at least one non-zero entry in a marked column. Let \( n_c \) be such that columns \( n_c \) to \( n - 1 \) in \( \mathcal{M}''_\psi \) are exactly the marked columns. The value of \( \alpha_{\rho_{\pi_i}} \), \( n_c \leq i < n \), must be exactly 1/2 or else the elements of some row of inequality 2.2 must sum to greater than 1. It follows that the number of non-zero entries in columns \( n_c \) to \( n - 1 \) in any row of \( \mathcal{M}''_\psi \) must be at most two or else the elements of some row of the inequality sums to greater than 1. By construction of \( \mathcal{M}''_\psi \), every row can have at most one +1 entry in columns 0 to \( n_c - 1 \). Hence \( \mathcal{M}''_\psi \) is a monotone decomposition for \( \psi \). 

The following result from [20] is also interesting in light of Theorem 33. It is stated without proof.

**Theorem 34.** The class of all formulas with a satisfiability index greater than \( 1 + 1/n^\epsilon \), for any fixed \( \epsilon < 1 \), is NP-complete.}


There is an almost obvious polynomial time solvable class larger than that of the q-Horn formulas: namely, the class of formulas which have a satisfiability index no greater than \( 1 + \alpha \ln(n)/n \), where \( \alpha \) is any positive constant. The \( \mathcal{M} \) matrix for any formula in this class can be scaled by -1 and partitioned as follows:

\[
\begin{pmatrix}
\mathcal{A}^1 & \mathcal{E} & \mathcal{B}^1 \\
\mathcal{D} & \mathcal{A}^2 & \mathcal{B}^2
\end{pmatrix}
\]
where submatrices $A^1, A^2, E,$ and $D$ have the properties required for q-Horn formulas and the number of columns in $B^1$ and $B^2$ is no greater than $O(\ln(n))$. Satisfiability for such formulas can be determined in polynomial time by solving the q-Horn system obtained after substitution of each of $2^{O(\ln(n))}$ partial truth assignments to the variables of $B^1$ and $B^2$.

4.6 Matched Formulas

The matched formulas have been considered in the literature (see [126]) but not extensively studied, probably because this seems to be a rather useless and small class of formulas. Our interest in matched formulas is to provide a basis of comparison with other, well known, well studied classes. Let $G_\psi(V_1, V_2, E)$ be the variable-clause matching graph (see Section 2.3.5) for CNF formula $\psi$, where $V_1$ is the set of clause vertices and $V_2$ is the set of variable vertices. A total matching with respect to $V_1$ is a subset of edges $E' \subseteq E$ such that no two edges in $E'$ share an endpoint but every vertex $v \in V_1$ is an endpoint for some edge in $E'$. Formula $\psi$ is a matched formula if its variable-clause matching graph has a total matching with respect to $V_1$. A matched formula is trivially satisfied: for each edge $e \in E'$ assign the variable represented by the variable vertex in $e$ a value that satisfies the clause represented by the clause vertex in $e$. The comparison with other classes is discussed in Sections 4.11 and 7.6.

4.7 Generalized Matched Formulas

Due to Szeider [125]. These formulas are always satisfiable but recognizing them is $NP$-complete.

4.8 Nested and Extended Nested Satisfiability

The complexity of nested satisfiability, inspired by Lichtenstein’s theorem of planar satisfiability [102], has been studied in [85]. Index all variables in a CNF formula consecutively from 1 to $n$ and let positive and negative literals take the index of their respective variables. A clause $c_i$ is said to straddle another clause $c_j$ if the index of a literal of $c_j$ is strictly between two indices of literals of $c_i$. Two clauses are said to overlap if they straddle each other. A formula is said to be nested if no two clauses overlap. For example, the following formula is nested

$$(v_6 \lor \neg v_7 \lor v_8) \land (v_2 \lor v_4) \land (\neg v_6 \lor \neg v_9) \land (v_1 \lor \neg v_5 \lor v_{10}).$$

The class of nested formulas is quite limited in size. A variable cannot show up in more than one clause of a nested formula where its index is
strictly between the greatest and least of the clause: otherwise, two clauses overlap. Therefore, a nested formula of \( m \) clauses and \( n \) variables has at most \( 2m + n \) literals. Thus, no CNF formula consisting of \( k \)-literal clauses is a nested formula unless \( m/n < 1/(k-2) \). This particular restriction will be understood better from the probabilistic perspective taken in Section 7.6. Despite this, the class of nested formulas is not contained in either of the SLUR or q-Horn classes as shown in Section 4.11. This adds credibility to the potential usefulness of the algorithm presented here. However, our main interest in nested formulas is due to an enlightening analysis and efficient dynamic programming solution which appears in [85] and is presented here.

Strong dependencies between variables of nested formulas may be exploited for fast solutions. In a nested formula, if clause \( c_i \) straddles clause \( c_j \) then \( c_j \) does not straddle \( c_i \), and if clause \( c_i \) straddles clause \( c_j \) and clause \( c_j \) straddles \( c_k \) then \( c_i \) straddles \( c_k \). Thus, the straddling relation induces a partial order on the clauses of a nested formula. It follows that the clauses can be placed in a total ordering using a standard linear time algorithm for topologically sorting a partial order: in the total ordering a given clause does not straddle any clauses following it. In the example above, if clauses are numbered \( c_0 \) to \( c_3 \) from left to right, \( c_2 \) straddles \( c_0 \), \( c_3 \) straddles \( c_1 \) and \( c_2 \), and no other clause straddles any other, so these clauses are in the desired order already.

Once clauses are topologically sorted into a total order, the satisfiability question may be solved in linear time by a dynamic programming approach where clauses are processed one at a time, in order. The idea is to maintain a partition of variables as a list of intervals such that all variables in any clause seen so far are in one interval. Associated with each interval are four “\( D \)” values that express the satisfiability of corresponding processed clauses under all possible assignments of values to the endpoints of the interval. As more clauses are considered, intervals join and \( D \) values are assigned. By introducing two extra variables, \( v_0 \) and \( v_{n+1} \) and an extra clause \( (v_0 \lor v_{n+1}) \) which straddles all others and is the last one processed, there is one interval \([v_0, v_{n+1}]\) remaining at the end. A \( D \) value associated with that interval determines satisfiability for the given formula. What remains is to determine how to develop the interval list and associated \( D \) values incrementally. This task is made easy by the fact that a variable which appears in a clause \( c \) with index strictly between the highest and lowest indices of variables in \( c \) never appears in a following clause.

An efficient algorithm for determining the satisfiability of nested formulas is shown in Figure 4.6. The following lemma is needed to prove correctness. We refrain from showing the dependence of \( h \) values on \( i \) to prevent the notation from going out of control. However, from the context, this dependence should be clear.

**Lemma 35.** Assume, at the start of any iteration \( 0 \leq i \leq m \) of the outer
Algorithm 20.

**Nested Solver** ($\psi$)

/* Input: set of sets CNF formula $\psi$, with variables indexed 1 to $n$ */
/* and $m$ clauses indexed from 0 to $m - 1$ */
/* Output: “unsatisfiable” or “satisfiable” */
/* Locals: Boolean variables $D_{i,j}(s,t), E_{i,j}(s,t), G_{i,j}(s,t)$, */
/* set $D$ of Boolean variables. */

Apply unit resolution to eliminate all unit clauses in $\psi$.
Topologically sort the clauses of $\psi$ as explained in the text.
Let $c_0, \ldots, c_{m-1}$ denote the clauses, in order.
Add the clause $c_m = \{v_0, v_{n+1}\}$.
Set $D \leftarrow \{D_{j,j+1}(s,t) \leftarrow 1 : \ 0 \leq j \leq n, \ s, t \in \{0, 1\}\}$.
Repeat the following for $0 \leq i \leq m$:
   Let $0 < h_1 < h_2 < \ldots < h_k < n + 1$ be the intervals for $D \in D$.
   // That is, $D = \{D_{0,h_1}(*,*), D_{h_1,h_2}(*,*), \ldots, D_{h_k,n+1}(*,*)\}$.
   Let $c_i = \{l_{p}, \ldots, l_{q}\}$, where $l_p \in \{v_{r}, \neg v_{r}\}$, $p \leq r \leq q$.
   Set $E_{h_p,h_p}(s,t) \leftarrow 0$ for all $s \in \{0, 1\}, \ t \in \{0, 1\}$.
   Set $G_{h_p,h_p}(s,t) \leftarrow 0$ for all $s \in \{0, 1\}, \ t \in \{0, 1\}$.
   If $h_p = v_{h_p}$ then do the following:
      Set $E_{h_p,h_p}(1,1) \leftarrow G_{h_p,h_p}(0,0) \leftarrow 1$.
      Set $E_{h_p,h_p}(0,0) \leftarrow G_{h_p,h_p}(1,1) \leftarrow 0$.
   Otherwise, if $h_p = \neg v_{h_p}$ then do the following:
      Set $E_{h_p,h_p}(1,1) \leftarrow G_{h_p,h_p}(0,0) \leftarrow 0$.
      Set $E_{h_p,h_p}(0,0) \leftarrow G_{h_p,h_p}(1,1) \leftarrow 1$.
Repeat the following for $0 \leq j < q - p, \ s \in \{0, 1\}, \ t \in \{0, 1\}$:
   If $t = 1$ then Set $l \leftarrow v_{h_{p+j+1}}$, Otherwise Set $l \leftarrow \neg v_{h_{p+j+1}}$.
   Set $E_{h_p,h_{p+j+1}}(s,t) \leftarrow (E_{h_p,h_{p+j}}(s,1) \land D_{h_{p+j},h_{p+j+1}}(1,t)) \lor$
      $(E_{h_p,h_{p+j}}(s,0) \land D_{h_{p+j},h_{p+j+1}}(0,t)) \lor$
      $(G_{h_p,h_{p+j}}(s,1) \land D_{h_{p+j},h_{p+j+1}}(1,t) \land l \in c_i) \lor$
      $(G_{h_p,h_{p+j}}(s,0) \land D_{h_{p+j},h_{p+j+1}}(0,t) \land l \in c_i)$.
   Set $G_{h_p,h_{p+j+1}}(s,t) \leftarrow (G_{h_p,h_{p+j+1}}(s,1) \land D_{h_{p+j},h_{p+j+1}}(1,t) \land \neg (l \in c_i)) \lor$
      $(G_{h_p,h_{p+j}}(s,0) \land D_{h_{p+j},h_{p+j+1}}(0,t) \land \neg (l \in c_i))$.
Repeat for $s \in \{0, 1\}, \ t \in \{0, 1\}$: Set $D_{h_p,h_q}(s,t) \leftarrow E_{h_p,h_q}(s,t)$.
Set $D \leftarrow D \setminus \{D_{h_p,h_{p+j+1}}(*,*) \ldots D_{h_q,h_q}(*,*)\} \cup \{D_{h_p,h_q}(*,*)\}$.
If $D_{0,n+1}(1,1) = 1$, Output “satisfiable,”
Otherwise Output “unsatisfiable.”

Figure 4.6: Algorithm for determining satisfiability of nested formulas.
Repeat loop of Algorithm \textit{Nested Solver}, that

\[ \mathcal{D} = \{ D_{h_0,h_1}(\ast, \ast), D_{h_1,h_2}(\ast, \ast), \ldots, D_{h_k,h_{k+1}}(\ast, \ast) \}, \]

where \( h_0 = 0 \) and \( h_{k+1} = n + 1 \). Let

\[ \psi^i_{p,p+j} = \{ c : c \in \{ c_0, \ldots, c_{i-1} \}, h_p \leq \minIndex(c) < \maxIndex(c) \leq h_{p+j} \}, \]

for any \( 0 \leq j \leq k - p + 1 \). Then the following hold:

1. At the start of iteration \( i \) of the outer Repeat loop, each variable in \( c_i \) is the same as one of \( \{ v_{h_0}, v_{h_1}, \ldots, v_{h_{k+1}} \} \) and for every clause \( c \in \{ c_0, \ldots, c_{i-1} \} \) there exists an \( 0 \leq r \leq k \) such that all the variables of \( c \) have index between \( h_r \) and \( h_{r+1} \). Moreover, for every \( h_r \) and \( h_{r+1} \) there is at least one clause whose minimum indexed variable has index \( h_r \) and whose maximum indexed variable has index \( h_{r+1} \).

2. At the start of iteration \( i \) of the outer Repeat loop, \( D_{h_j,h_{j+1}}(s,t) \), \( 0 \leq j \leq k \), has value 1 if and only if \( \psi^j_{p,p+j+1} \) is satisfiable with variable \( v_{h_j} \) set to value \( s \) and variable \( v_{h_{j+1}} \) set to value \( t \).

3. At the start of iteration \( 0 \leq j < q - p \) of the main inner Repeat loop, \( E_{h_p,h_{p+j}}(s,t) \) has value 1 if and only if \( \psi^j_{p,p+j} \cup \{ \{ l_x : l_x \in c_i, x \leq h_{p+j} \} \} \) is satisfiable with variable \( v_{h_p} \) set to value \( s \) and variable \( v_{h_{p+j}} \) set to value \( t \).

4. At the start of iteration \( 0 \leq j < q - p \) of the main inner Repeat loop, \( G_{h_p,h_{p+j}}(s,t) \) has value 1 if and only if \( \psi^j_{p,p+j} \cup \{ \{ l_x : l_x \in c_i, x \leq h_{p+j} \} \} \) is not satisfiable but \( \psi^j_{p,p+j} \) is satisfiable with variable \( h_p \) set to value \( s \) and variable \( v_{h_{p+j}} \) set to value \( t \).

\textbf{Proof.} Consider Point 1. At the start of iteration 0 of the outer Repeat loop Point 1 holds because all variables are in the set \( \{ h_0, \ldots, h_{n+1} \} \). Suppose Point 1 holds at the beginning of iteration \( i \). As a result of the topological sort, \( c_i \) cannot be straddled by any clause in \( \{ c_0, \ldots, c_{i-1} \} \). If one variable of \( c_i \) is indexed strictly between \( h_r \) and \( h_{r+1} \), by Point 1, there is a clause in \( \{ c_0, \ldots, c_{i-1} \} \) whose variable indices are as high as \( h_{r+1} \) and as low as \( h_r \). But, such a clause would straddle \( c_i \). Hence, for every variable \( v \in c_i \), \( v \in \{ v_{h_p}, v_{h_{p+1}}, \ldots, v_{h_q} \} \). Since the last line of the outer Repeat loop replaces all \( D_{h_p,h_{p+1}} \ldots D_{h_{q-1},h_q} \) with \( D_{h_p,h_q} \), Point 1 holds at the beginning of iteration \( i + 1 \) of the outer loop.

Consider Point 2. At the start of iteration \( i = 0 \) all defined variables are \( D_{j,j+1}(\ast, \ast), 0 \leq j \leq n \), and these have value 1. From the definition of \( \psi^i_{p,j} \) we have \( \psi^0_{p,j} = 0 \). Thus, Point 2 holds before iteration \( i = 0 \). Suppose Point 2 holds at the start of iteration \( i \). Since \( c_i \) contains no variables indexed less than \( h_p \) or greater than \( h_q \) and all \( D_{h_r,h_{r+1}}(\ast, \ast) \) are unchanged.
by the algorithm for \( r < p \) and \( r \geq q \), then these \( D \) values are correct for iteration \( i + 1 \) (but the subscripts on \( h \) values change because there are fewer \( D \) variables on the next iteration). So, due to the short inner Repeat loop following the main inner Repeat loop, Point 2 holds at the start of iteration \( i + 1 \) if \( E_{h_p,q}(s,t) \) has value 1 if and only if \( \psi^j_{p,q} \cup \{c_i\} \) is satisfiable with variable \( v_{h_p} \) set to value \( s \) and variable \( v_{h_q} \) set to value \( t \). This is shown below, thus taking care of Point 2.

Assume, during iteration \( i \) of the outer loop and before the start of the main inner loop, that Point 1 and Point 2 hold. Consider the iteration \( j = 0 \) of the main inner loop. As above, \( \psi_{s,s}^0 = \emptyset \). Moreover, by Point 1, 
\[
\{ l : l \in c_0, x \leq h_p \} = \{ h_p \} \quad \text{so} \quad \psi^0_{p,p} \cup \{ \{ h_p \} \} = \{ \{ h_p \} \}.
\]
There is no satisfying assignment for this set if the value of the highest and lowest indexed variables of \( c_0 \) are opposite each other since the highest and lowest indexed variables are the same. Accordingly, \( E_{p,p}(s,t) \) and \( G_{p,p}(s,t) \) are set to 0 in the algorithm when \( s \) and \( t \) are of opposite value. However, if \( s = t = 1 \) and \( l_{h_p} = v_{h_p} \), then \( \psi^0_{p,p} \cup \{ \{ h_p \} \} \) is satisfiable. Accordingly, \( E_{p,p}(1,1) \) is set to 1 and \( G_{p,p}(1,1) \) is set to 0 in the algorithm. Otherwise, if \( s = t = 1 \) and \( l_{h_p} = \neg v_{h_p} \), then \( \psi^0_{p,p} \cup \{ \{ \neg h_p \} \} \) is unsatisfiable. Accordingly, \( E_{p,p}(1,1) \) is set to 0 and \( G_{p,p}(1,1) \) is set to 1 in the algorithm. Similar reasoning applies to the case \( s = t = 0 \). Thus, Points 3 and 4 hold for the case \( j = 0 \).

Now consider iteration \( i \) of the outer loop and iteration \( j > 0 \) of the main inner loop. Assume, at the start of iteration \( j - 1 \), that Points 3 and 4 hold. Points 1 and 2 hold from before since no changes to these occur in the main inner loop. Let \( c_i^{j-1} = \{ l : l \in c_i, x \leq p + j - 1 \} \) be the subset of literals of \( c_i \) that have index no greater than \( h_{p+j-1} \). From Point 1, for every clause \( c \in \{ c_0, \ldots, c_{i-1} \} \), there exists a positive integer \( 0 \leq r \leq k \) such that all variables of \( c \) have index between \( h_r \) and \( h_{r+1} \). It follows that
\[
\psi^i_{p,p+j} \cup \{ c_i^{j-1} \} = (\psi^i_{p,p+j-1} \cup \{ c_i^{j-1} \}) \cup \psi^i_{p,p+j-1,p+j}.
\]
and \((\psi^i_{p,p+j-1} \cup \{ c_i^{j-1} \}) \cap \psi^i_{p,p+j-1,p+j} = \emptyset \). For the sake of visualization, define \( \psi_1 = \psi_{p,p+j-1} \) and \( \psi_2 = \psi_{p,p+j-1,p+j} \). At most one variable, namely \( v_{h_{p+j-1}} \), is common to both \( \psi_1 \) and \( \psi_2 \). Hence, when \( v_{h_{p+j-1}} \) is set to some value, the following holds: both \( \psi_1 \cup \{ c_i^{j-1} \} \) and \( \psi_2 \) are satisfiable if and only if \( \psi^i_{p,p+j} \cup \{ c_i^{j-1} \} \) is satisfiable. Now consider \( \psi^i_{p,p+j} \cup \{ c_i^{j-1} \} \). Since every variable of \( c_i \) has an index matching one of \( \{ h_p, h_{p+1}, \ldots, h_q \} \), \( c_i \setminus c_i^{j-1} \) either is the empty set or \( \{ v_{h_{p+j}} \} \) or \( \{ \neg v_{h_{p+j}} \} \). Therefore, given \( v_{h_{p+j-1}} \), \( \psi^i_{p,p+j-1} \cup \{ c_i^{j-1} \} \) is satisfiable if and only if either both \( \psi_1 \cup \{ c_i^{j-1} \} \) and \( \psi_2 \) are satisfiable or \( \psi_1 \) and \( \psi_2 \) are satisfiable, \( \psi_1 \cup \{ c_i^{j-1} \} \) in unsatisfiable, but \( \psi_1 \cup \{ c_i^{j-1} \} \cup \{ l \} \) is satisfiable where \( l \in \{ v_{h_{p+j}}, \neg v_{h_{p+j}} \} \) or \( \{ l \} = \emptyset \). But, since \( l \) does not occur in \( \psi_1 \), \( \psi_1 \cup \{ c_i^{j-1} \} \) can be unsatisfiable with \( \psi_1 \) and \( \psi_1 \cup \{ c_i^{j-1} \} \cup \{ l \} \) satisfiable only for assignments which satisfy \( l \). Then, by hypothesis, the
association of the $E$ and $G$ variables with $\psi_{p,p+j-1}^i$ and $\psi_{p+j-1,p+j}^i$, and the assignments to $E$ and $G$ in the main inner loop of Algorithm 20, the values of $E$ and $G$ match Points 3 and 4 for the $j$th iteration of the main inner loop.

From Point 3, upon completion of the main inner loop $E_{h_p,h_q}(s,t)$ has value 1 if and only if $\psi_{h_p,h_q}^i \cup \{c_i\}$ is satisfiable. This matches the hypothesis of Point 2 thereby completing the proof that Point 2 holds.

**Corollary 36.** Algorithm Nested Solver correctly determines satisfiability for a given nested formula.

**Proof.** Algorithm Nested Solver determines $\psi$ is satisfiable if and only if $D_{0,n+1}(1,1)$ has value 1. But, from Lemma 35, $D_{0,n+1}(1,1)$ has value 1 if and only if $\psi \cup \{v_0, v_{n+1}\}$ is satisfiable and $v_0$ and $v_{n+1}$ are set to value 1. The corollary follows.

The operation of Nested Solver is demonstrated by means of an example taken from [85]. Suppose the algorithm is applied to a nested formula containing the following clauses which are shown in proper order after the topological sort:

$$(v_1 \lor v_2) \land (v_2 \lor v_3) \land (\neg v_2 \lor \neg v_3) \land (v_3 \lor v_4) \land (\neg v_3 \lor \neg v_4) \land (\neg v_1 \lor v_2 \lor v_4)$$

The following table shows the $D$ values that have been computed prior to the start of iteration 6 of the outer loop of the algorithm. The columns show $s, t$ values and the rows show variable intervals.

<table>
<thead>
<tr>
<th>$D_{s,s}(s, t)$</th>
<th>0, 0</th>
<th>0, 1</th>
<th>1, 0</th>
<th>1, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2, 3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3, 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Such a table could result from the following set of processed clauses at the head of the topological order (and before $c$): processing clause $c$, using the initial values and recurrence relations for $E$ and $G$ variables given above, produces values as shown in the following tables.

<table>
<thead>
<tr>
<th>$E_{p,p+j}(s, t)$</th>
<th>0, 0</th>
<th>0, 1</th>
<th>1, 0</th>
<th>1, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1, 2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1, 3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1, 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
The last line of the $E$ table holds the new $D$ values for the interval $[v_1, v_4]$ as shown by the following table.

<table>
<thead>
<tr>
<th>$G_{p,p+j}(s,t)$</th>
<th>0, 0</th>
<th>0, 1</th>
<th>1, 0</th>
<th>1, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1, 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1, 3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Linear time is achieved by careful data structure design and from the fact that no more than $2m + n$ literals exist in a nested formula.

**Theorem 37.** Algorithm Nested Solver has worst-case time complexity that is linear in the size of $\psi$.

The question of whether the variable indices of a given formula can, in linear time, be permuted to make the formula nested appears to be open.

An extension to nested satisfiability has been proposed in [73]. We prefer to skip the details and just mention that this extension can be recognized and solved in linear time. For details, the reader is referred to [73].

### 4.9 Linear Autark Formulas

This class is based on the notion of an autark assignment which was introduced in [107]. Repeating the definition from Page 83, an assignment to a set of variables is an autark assignment if all clauses that contain at least one of those variables are satisfied by the assignment. An autark assignment provides a means to partition the clauses into two groups: one that is satisfied by the autark assignment and one that is completely untouched by that assignment. Therefore, autark assignments provide a way to reduce a formula to one that is equivalent in satisfiability.

We now show how to find an autark assignment in polynomial time. Let CNF formula $\psi$ of $m$ clauses and $n$ variables be represented as a $(0, \pm 1)$ matrix $M_\psi$. Let $\alpha$ be an $n$ dimensional real vector with components $\alpha_1, \alpha_2, \ldots, \alpha_n$. Consider the following system of inequalities:

\[
M_\psi \alpha \geq 0, \\
\alpha \neq 0.
\]  

**Theorem 38.** ([103]) A solution to (4.1) implies an autark assignment for $\psi$.  

<table>
<thead>
<tr>
<th>$D_{<em>,</em>}(s,t)$</th>
<th>0, 0</th>
<th>0, 1</th>
<th>1, 0</th>
<th>1, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Proof. Create the autark assignment as follows: if $\alpha_i < 0$ then assign $v_i = 0$, if $\alpha_i > 0$ then assign $v_i = 1$, if $\alpha_i = 0$ then keep $v_i$ unassigned. We show that either a clause is satisfied by this assignment or it contains only unassigned variables.

For every clause, by (2.1), we have to satisfy the following:

$$a_1v_1 + a_2v_2 + \ldots + a_nv_n \geq 1 - b$$

(4.2)

where $a_i$ factors are 0, -1, or +1, and the number of negative $a_i$ terms is $b$. Suppose $\alpha$ is a solution to (4.1). Write an inequality of (4.1) as follows:

$$a_1\alpha_1 + a_2\alpha_2 + \ldots + a_n\alpha_n \geq 0$$

(4.3)

Suppose at least one term, say $a_i\alpha_i$, is positive. If $\alpha_i$ is positive then $a_i = 1$ so, with $v_i = 1$ and since there are at most $b$ terms of value -1, the left side of (4.2) must be at least $1 - b$. On the other hand, if $\alpha_i$ is negative, then $a_i = -1$, $v_i = 0$ and the product $a_i v_i$ in (4.2) is 0. Since there are $b - 1$ negative factors left in (4.2), the left side must be at least $1 - b$. Therefore, in either case, inequalities of the form (4.3) with at least one positive term represent clauses that are satisfied by the assignment corresponding to $\alpha$.

Now consider the case where no terms in (4.3) are positive. Since the left side of (4.3) is at least 0, there can be no negative terms either. Therefore, all the variables for the represented clause are unassigned.

A formula $\psi$ for which the only solution to (4.1) is $\alpha = 0$ is said to be linear autarky-free.

System (4.1) is an instance of Linear Programming and can therefore be solved in polynomial time. Hence, an autark assignment, if one exists, can be found in polynomial time.

Algorithm LinAut of Figure 4.7 is the central topic of study in this section and it will be used to define a polynomial time solvable class of formulas called linear autarky formulas. The algorithm repeatedly applies an autark assignment as long as one can be found by solving (4.1). In a departure from [103] the algorithm begins with a call to Unit Resolution to eliminate all unit clauses. This is done to remove some awkwardness in the description of linear autarky formulas that will become clear shortly.

The following states an important property of Algorithm LinAut.

**Lemma 39.** Let $\psi$ be a CNF formula that is input to Algorithm LinAut and let $\psi'$ be the formula that is output. If $\psi' = \emptyset$ then $\psi$ is satisfiable and $\alpha^S$ transforms to a satisfying assignment for $\psi$.

**Proof.** An autark assignment $t$ to $\psi'$ induces a partition of clauses of $\psi'$ into those that are satisfied by $t$ and those that are untouched by $t$. Due to the independence of the latter group, a satisfying assignment for that group can be combined with the autark assignment for the former group.
to satisfy $\psi'$. If there is no satisfying assignment for either group then $\psi'$ cannot be satisfiable. Therefore, since each iteration finds and applies an autark assignment (via $\alpha$), if $\psi' = \emptyset$ then the composition of the autark assignments of each iteration is a satisfying assignment for $\psi$.

The algorithm has polynomial time complexity. In some cases it solves its input formula.

**Theorem 40.** Let $\psi$ be Horn or renameable Horn. If $\psi$ is satisfiable then $\text{LinAut}$ returns an $\alpha^S$ that transforms to a solution of $\psi$ as described in the proof of Theorem 38 and the formula returned by $\text{LinAut}$ is $\emptyset$. If $\psi$ is unsatisfiable $\text{LinAut}$ returns “unsatisfiable.”

**Proof.** Horn or renameable Horn formula $\psi$ is unsatisfiable only if there is at least one positive unit clause in $\psi$. In this case $\text{Unit Resolution}(\psi)$ will output a formula containing $\emptyset$ and $\text{LinAut}$ will output “unsatisfiable.” Otherwise $\text{Unit Resolution}(\psi)$ outputs a Horn or renameable Horn formula $\psi'$ with no unit clauses and a partial assignment $P$ which is recorded in $\alpha^S$. In the next paragraph we will show that any such Horn or renameable Horn formula is not linear autarky-free so there exists an autark assignment for it. By definition, the clauses that remain after the autark assignment is applied must be Horn or renameable Horn. Therefore, the Repeat loop of $\text{LinAut}$ must continue until there are no clauses left. Then, by Lemma 39, $\alpha^S$ transforms to a satisfying assignment for $\psi$.

Now we show there is always an $\alpha \neq 0$ that solves (4.1). Observe that any Horn formula without unit clauses has at least one negative literal in every clause. Therefore some all negative vector $\alpha$ of equal components solves (4.1). In the case of renameable Horn, the polarity of $\alpha$ components in the switch set is reversed to get the same result. Therefore, there is always at least one autark assignment for a Horn or renameable Horn formula. Note that Algorithm $\text{LinAut}$ may find an $\alpha$ that does not zero out all rows so the Repeat loop may have more than 1 iteration but, since the reduced formula $\psi$ is Horn or renameable Horn, there is always an $\alpha \neq 0$ that solves (4.1) for the clauses that are left.

The following Horn formula would violate Theorem 40 if the call to $\text{Unit Resolution}$ had not been added to $\text{LinAut}$:

$$(v_1) \land (v_2) \land (v_3) \land (v_1 \lor -v_2 \lor -v_3) \land (-v_1 \lor v_2 \lor -v_3) \land (-v_1 \lor -v_2 \lor v_3).$$

This formula is satisfied with variables set to 1 but is linear autarky-free.

**Theorem 41.** If $\psi$ is a 2-SAT formula that is satisfiable, then $\text{LinAut}$ outputs an $\alpha^S$ that represents a solution of $\psi$ and the formula output by $\text{LinAut}$ is $\emptyset$. 

Algorithm 21.

\texttt{LinAut} (ψ)  
/* Input: a set of sets CNF formula ψ */  
/* Output: pair \langle real vector, subformula of ψ \rangle or “unsatisfiable” */  
Set \langle ψ', P \rangle ← \texttt{Unit Resolution}(ψ).
If \emptyset ∈ ψ' then Output “unsatisfiable”.
Set \(α^S ← 0\).
For each \(v_i \in P\) Set \(α^S_i ← 1\).
For each \(v_i \) such that \(v_i \notin P\) and \(v_i \in V_P\) Set \(α^S_i ← −1\).
If \(ψ' = \emptyset\) then Output \(⟨α^S, ψ'⟩\).
Repeat the following until some statement generates output:
Solve for \(α : M_{ψ'}α ≥ 0\) and \(α^S_i = 0\) if column \(i\) in \(M_{ψ'}\) is 0.
If \(α=0\) then Output \(⟨α^S, ψ'⟩\).
Otherwise, do the following:
Set \(α^S ← α^S+α\).
For every \(α^S_i ≠ 0\) do the following:
Zero out rows of \(M_{ψ'}\) with a non-zero entry in column \(i\).
Set \(ψ' ← \{c : c ∈ ψ', v_i ∉ c, ¬v_i ∉ c\}\).

\(\square\)

Figure 4.7: Repeated decomposition of a formula using autark assignments. 
Recall \(V_P\) is the set of variables whose values are set in partial assignment \(P\). 
Output real vector \(α^S\) can be transformed to a truth assignment as described in the proof of Theorem 38.
Proof. In this case Unit Resolution($\psi$) outputs $\psi$ and causes no change to $\alpha^S$. The only non-zero autark solutions to $M_\psi \alpha \geq 0$ have the following clausal interpretation: choose a literal in a clause and assign its variable a value that satisfies the clause, for all non-satisfied clauses that contain the negation of that literal assign a value to the variable of the clause’s other literal that satisfies that clause, continue the process until some clause is falsified or no variable is forced to be assigned a value to satisfy a clause. This is one iteration of Algorithm 2-SAT Solver so, since $\psi$ is satisfiable, the process never ends in a falsified clause. Since what is left is a satisfiable 2-SAT formula this step is repeated until $\emptyset$ is output. As in the proof of Theorem 40, $\alpha^S$ transforms to a satisfying assignment for $\psi$. ◻

If $\psi$ is an unsatisfiable 2-SAT formula then LinAut($\psi$) will output an unsatisfiable 2-SAT formula.

The class of linear autarky formulas is the set of CNF formulas on which the application of LinAut either outputs “unsatisfiable” or a formula that is $\emptyset$ or a linear autarky-free 2-SAT formula. In the middle case the input formula is satisfiable, in the other two cases it is unsatisfiable. Observe that, in the last case, it is unnecessary to solve the remaining 2-SAT formula.

Theorem 42. All $q$-Horn formulas are linear autark formulas

Proof. Apply LinAut to $q$-Horn formula $\psi$. Then a linear autarky-free $q$-Horn formula is output. We can assume that all clauses in the output formula have at least 2 literals because all clauses of $\psi'$ entering the Repeat loop for the first time have at least 2 literals and any subsequent autark assignment would not introduce a clause that is not already in $\psi'$. The rows and columns of $M_\psi$ representing the output $q$-Horn formula $\psi'$ may be permuted and columns may be scaled by -1 to get a form as shown on Page 78 where $A^1$ represents a Horn formula, $D$ is non-positive, and $A^2$ represents a 2-SAT formula. Construct vector $\alpha$ as follows. Assign equal negative values to all $\alpha_i$ where $i$ is the index of a column through $A^1$ (reverse the value if the column had been scaled by -1) and 0 to all other $\alpha_i$. Then, since there are at least as many positive as negative entries in every row through $A^1$, the product of any of those rows, unscaled, and $\alpha$ is greater than 0. Since there are only negative entries in rows through $D$ and all the entries through columns of $A^2$ multiply by $\alpha_i$ values that are 0, the product of any of the rows through $D$, unscaled, and $\alpha$ are also positive. Therefore, $\alpha$ shows that $\psi'$ is not linear autark-free, a contradiction. It follows that $A^1 = D = \emptyset$ and the output formula is either 2-SAT or $\emptyset$. If it’s 2-SAT, it must be unsatisfiable as argued in Theorem 41. ◻

The relationship between linear autarky-free formulas and the satisfiability index provides a polynomial time test for the satisfiability of a given CNF formula.
Theorem 43. If the shortest clause of a CNF formula $\psi$ has width $k$ and if the satisfiability index of $\psi$ is less than $k/2$, then $\psi$ is satisfiable.

Proof. This is seen more clearly by transforming variables as follows. Define real $n$ dimensional vector $\beta$ with components

$$\beta_i = 2\alpha_i - 1 \quad \text{for all} \quad 0 \leq i < n.$$ 

The satisfiability index (2.2) for the transformed variables may be expressed, by simple substitution, as follows:

$$M_{\psi}\beta \leq 2Z - l$$

(4.4)

where $l$ is an $n$ dimensional vector expressing the number of literals in all clauses and $Z = \langle z, z, \ldots, z \rangle$. The minimum $z$ that satisfies (4.4) is the satisfiability index of $\psi$. Apply Algorithm LinAut to $\psi$ which, by hypothesis, has shortest clause of width $k$. The algorithm removes rows so $z$ can only decrease and $k$ can only increase. Therefore, if $z < k/2$ for $\psi$, it also holds for the $\psi'$ that is output by LinAut. Suppose $\psi' \neq \emptyset$. Formula $\psi'$ is linear autarky-free so the only solution to $0 \leq M_{\psi'}\beta$ is $\beta = 0$ which implies $0 \leq 2Z - l$. Since the shortest clause of $\psi'$ is at least $k$, it follows that $k/2 \leq z$. This contradicts the hypothesis that $z < k/2$. Therefore, $\psi' = \emptyset$ and, by Lemma 39, the input formula is satisfiable.

The effectiveness of this test on random $k$-CNF formulas will be discussed in Section 7.6.

4.10 Classes of Unsatisfiable Formulas

The following classes specify collections of unsatisfiable formulas only.

4.10.1 Minimally Unsatisfiable Formulas

An unsatisfiable CNF formula is minimally unsatisfiable if removing any clause results in a satisfiable formula. The class of minimally unsatisfiable formulas is easily solved if the number of clauses exceeds the number of variables by a fixed positive constant $k$. This difference is called the formula’s deficiency. In this section we begin with a discussion of the special case where deficiency is 1, then consider the case of any fixed deficiency greater than 1. The discussion includes some useful properties which lead to an efficient solver for this case and helps explain the difficulty of resolution methods for many CNF formulas (see Section ??).

The following result was proved in one form or another by several people (for example, [6, 94]). We present a simple argument due to Truemper.
Theorem 44. If \( \psi \) is a minimally unsatisfiable CNF formula with \(|V_\psi| = n\) variables, then the number of clauses in \( \psi \) must be at least \( n + 1 \).

Proof. Let \( \psi \) be a minimally unsatisfiable CNF formula with \( n + 1 \) clauses. Suppose \(|V_\psi| \geq n + 1\). Let \( G_\psi(V_1, V_2, E) \) be the variable-clause matching graph for \( \psi \) (see Section 2.3.5) where \( V_1 \) is the set of clause vertices and \( V_2 \) is the set of variable vertices. There is no total matching with respect to \( V_1 \) since this would imply \( \psi \) is satisfiable, a contradiction. Therefore, by Theorem 2, there is a subset \( V'_\psi \subset V_1 \) with neighborhood smaller than \(|V'_\psi|\). Let \( V'_\psi \) be such a subset of maximum cardinality. Define \( \psi_1 \) to be the CNF formula consisting of the clauses of \( \psi \) corresponding to the neighborhood of \( V'_\psi \). By the minimality property of \( \psi \), \( \psi_1 \) must be satisfiable. Delete from \( \psi \) the clauses of \( \psi_1 \) and from the remaining clauses all variables occurring in \( \psi_1 \). Call the resulting CNF formula \( \psi_2 \). There must be a matching of the clauses of \( \psi_2 \) into the variables of \( \psi_2 \) since otherwise \( V'_\psi \) and therefore \( \psi_1 \) was not maximal in size. Hence \( \psi_2 \) is satisfiable. But if \( \psi_1 \) and \( \psi_2 \) are satisfiable then so is \( \psi \), a contraction. \( \square \)

Most of the remaining ideas of this section have been inspired by Oliver Kullman [90].

A saturated minimally unsatisfiable formula \( \psi \) is a minimally unsatisfiable formula such that adding any literal \( l \), existing in a clause of \( \psi \), to any clause of \( \psi \) not already containing \( l \) or \( \neg l \) results in a satisfiable formula. The importance of saturated minimally unsatisfiable formulas is grounded in the following lemma.

Lemma 45. Let \( \psi \) be a saturated minimally unsatisfiable formula and let \( l \) be a literal from \( \psi \). Then

\[
\psi' = \{ c \setminus \{ \neg l \} : c \in \psi, l \notin c \}
\]

is minimally unsatisfiable. In other words, if satisfied clauses and falsified literals due to \( l \) taking value 1 are removed from \( \psi \), what’s left is minimally unsatisfiable.

Proof. Formula \( \psi' \) is unsatisfiable if \( \psi \) is, otherwise there is an assignment which sets \( l \) to 1 and satisfies \( \psi \). Suppose there is a clause \( c \in \psi' \) such that \( \psi' \setminus \{ c \} \) is unsatisfiable. Clause \( c \) must contain a literal \( l' \) that is neither \( l \) nor \( \neg l \). Construct \( \psi'' \) by adding \( l' \) to all clauses of \( \psi \) containing \( \neg l \). Since \( \psi \) is saturated, there is a truth assignment \( M \) satisfying \( \psi'' \) and therefore \( \psi' \) with literals \( l' \) added as above. Assignment \( M \) must set \( l' \) to 1, otherwise \( \psi' \setminus \{ c \} \) is satisfiable. Then, adjusting \( M \) so the value of \( l \) is 0 gives an assignment which also satisfies \( \psi' \), a contradiction. \( \square \)

The following is a simple observation that is needed to prove Theorem 47 below.
Lemma 46. Every variable of a minimally unsatisfiable formula occurs positively and negatively in the formula.

Proof. Let $\psi$ be any minimally unsatisfiable formula. Suppose there is a positive literal $l$ such that $\exists c \in \psi : l \in c$ and $\forall c \in \psi, -l \notin c$. Let $\psi_1 = \{ c : c \in \psi, l \notin c \}$. Let $\psi_2 = \psi \setminus \psi_1$ denote the clauses of $\psi$ which contain literal $l$. Clearly, $|\psi_2| > 0$. Then, by definition of minimal unsatisfiability, $\psi_1$ is satisfiable by some truth assignment $M$. Hence $M \cup \{l\}$ satisfies $\psi_1 \cup \psi_2 = \psi$. This contradicts the hypothesis that $\psi$ is unsatisfiable. A similar argument applies if literal $l$ is negative, proving the lemma.

Theorem 47. Let $\psi$ be a minimally unsatisfiable formula with $n > 0$ variables, $n + 1$ clauses. Then there exists a variable $v \in V_\psi$ such that the literal $v$ occurs exactly one time in $\psi$ and the literal $\neg v$ occurs exactly one time in $\psi$.

Proof. If $\psi$ is not saturated, there is some set of literals already in $\psi$ that may be added to clauses in $\psi$ to make it saturated. Doing so does not change the number of variables and clauses of $\psi$. So, suppose from now on that $\psi$ is a saturated minimally unsatisfiable formula with $n$ variables and $n + 1$ clauses. Choose a variable $v$ such that the sum of the number of occurrences of literal $v$ and literal $\neg v$ in $\psi$ is minimum. By Lemma 46, the number of occurrences of literal $v$ in $\psi$ is at least one and the number of occurrences of literal $\neg v$ in $\psi$ is at least one. By Lemma 45,

$$\psi_1 = \{ c \setminus \{ \neg v \} : c \in \psi, v \notin c \} \quad \text{and} \quad \psi_2 = \{ c \setminus \{ v \} : c \in \psi, \neg v \notin c \}$$

are minimally unsatisfiable. Clearly, $|V_{\psi_1}| = |V_{\psi_2}| = n - 1$ or else the minimality of variable $v$ is violated. By Theorem 44, the fact that $\psi_1$ and $\psi_2$ are minimally unsatisfiable, and the fact that $|\psi_i| < |\psi|$, $i \in \{1, 2\}$, it follows that $|\psi_1| = |\psi_2| = n$. Therefore, the number of occurrences of literal $v$ in $\psi$ is one and the number of occurrences of literal $\neg v$ in $\psi$ is one.

Lemma 48. Let $\psi$ be a minimally unsatisfiable CNF formula with $n > 0$ variables and $n + 1$ clauses. Let $v$ be the variable of Theorem 47 and define clauses $c_v$ and $c_{\neg v}$ such that literal $v \in c_v$ and literal $\neg v \in c_{\neg v}$. Then there is no variable which appears as a positive literal in one of $c_v$ or $c_{\neg v}$ and as a negative literal in the other.

Proof. Suppose there is a variable $w$ such that literal $w \in c_v$ and literal $\neg w \in c_{\neg v}$. By the minimality of $\psi$ and the fact that variable $v$ appears only in $c_v$ and $c_{\neg v}$, there exists an assignment $M$ which excludes setting a value for variable $v$ and satisfies $\psi \setminus \{c_v, c_{\neg v}\}$. But $M$ must also satisfy either $c_v$, if $w$ has value 1, or $c_{\neg v}$ if $w$ has value 0. In the former case $M \cup \{\neg v\}$ satisfies $\psi$ and in the latter case $M \cup \{v\}$ satisfies $\psi$, a contradiction. The argument
can be generalized to prove the lemma.

The next series of results shows the structure of minimally unsatisfiable formulas and how to exploit that structure for fast solutions.

**Theorem 49.** Let \( \psi \) be a minimally unsatisfiable formula with \( n > 1 \) variables and \( n + 1 \) clauses. Let \( v \) be the variable of Theorem 47, let \( c_v \) be the clause of \( \psi \) containing the literal \( v \), and let \( c_{\neg v} \) be the clause of \( \psi \) containing the literal \( \neg v \). Then \( \psi' = \psi \setminus \{c_v, c_{\neg v}\} \cup \{R_{c_{\neg v}}^{c_v}\} \) is a minimally unsatisfiable formula with \( n - 1 \) variables and \( n \) clauses.

**Proof.** By Lemma 48 the resolvent \( R_{c_{\neg v}}^{c_v} \) of \( c_v \) and \( c_{\neg v} \) exists. Moreover, \( R_{c_{\neg v}}^{c_v} \neq \emptyset \) if \( n > 1 \) or else \( \psi \) is not minimally unsatisfiable. Therefore, \( \psi' \) is unsatisfiable, contains \( n - 1 \) variables, and has \( n \) non-empty clauses. Remove a clause \( c \) from \( \psi \setminus \{c_v, c_{\neg v}\} \). By the minimality of \( \psi \), there is an assignment \( M \) which satisfies \( \psi \setminus \{c\} \). By Lemma 17, \( M \) also satisfies \( \psi \setminus \{c_v, c_{\neg v}\} \cup \{R_{c_{\neg v}}^{c_v}\} \). Hence \( M \) satisfies \( \psi' \setminus \{c\} \). Now remove \( c_v \) and \( c_{\neg v} \) from \( \psi \). Again, by the minimality of \( \psi' \), there is an assignment \( M' \) which satisfies \( \psi' \setminus \{c_v, c_{\neg v}\} \) and hence \( \psi' \setminus \{R_{c_{\neg v}}^{c_v}\} \). Thus, removing any clause from \( \psi' \) results in a satisfiable formula and the theorem is proved.

Theorem 49 implies the correctness of the polynomial time algorithm of Figure 4.8 for determining whether a CNF formula with one more clause than variable is minimally unsatisfiable. If the output is “m unsat(1)” the input formula is minimally unsatisfiable, with clause-variable difference of 1, otherwise it is not.

The following results provide an interesting and useful characterization of minimally unsatisfiable formulas of the type discussed here.

**Theorem 50.** Let \( \psi \) be a minimally unsatisfiable formula with \( n > 1 \) variables and \( n + 1 \) clauses. Then there exists a variable \( v \), occurring once as a positive literal and once as a negative literal, and a partition of the clauses of \( \psi \) into two disjoint sets \( \psi_v \) and \( \psi_{\neg v} \) such that literal \( v \) only occurs in clauses of \( \psi_v \), literal \( \neg v \) only occurs in clauses of \( \psi_{\neg v} \) and no variable other than \( v \) that is in \( \psi_v \) is also in \( \psi_{\neg v} \) and no variable other than \( v \) that is in \( \psi_{\neg v} \) is also in \( \psi_v \).

**Proof.** By induction on the number of variables. The hypothesis clearly holds for minimally unsatisfiable formulas of one variable, all of which have the form \((v) \land (\neg v)\). Suppose the hypothesis is true for all minimally unsatisfiable CNF formulas containing \( k > 1 \) or fewer variables and having deficiency 1. Let \( \psi \) be a minimally unsatisfiable CNF formula with \( k + 1 \) variables and \( k + 2 \) clauses. From Theorem 47 there is a variable \( v \) in \( \psi \) such that literal \( v \) occurs in exactly one clause, say \( c_v \), and literal \( \neg v \) occurs in exactly one clause, say \( c_{\neg v} \). By Lemma 48 the resolvent \( R_{c_{\neg v}}^{c_v} \) of \( c_v \) and \( c_{\neg v} \) exists and \( R_{c_{\neg v}}^{c_v} \neq \emptyset \) or else \( \psi \) is not minimally unsatisfiable. Let \( \psi' = (\psi \setminus \{c_v, c_{\neg v}\}) \cup \{R_{c_{\neg v}}^{c_v}\} \). That is, \( \psi' \) is obtained from \( \psi \) by resolving
Therefore, the hypothesis holds for $k$ and $k+1$ clauses. Then, by the induction hypothesis, there is a partition $\psi'_v$ and $\psi'_{\neg v}$ of clauses of $\psi'$ and a variable $v'$ such that literal $v'$ occurs only in one clause of $\psi'_v$, literal $\neg v'$ occurs only in one clause of $\psi'_{\neg v}$, and, excluding $v'$, there is no variable overlap between $\psi'_v$ and $\psi'_{\neg v}$. Suppose $R_{\psi_{v'}} \in \psi'_{\neg v}$. Then $\psi'_v$ and $R_{\psi_{v'}} \subseteq \psi'_{\neg v}$ (denoted $\psi_v$ and $\psi_{\neg v}$, respectively) form a non variable overlapping partition of clauses of $\psi$ (excluding $v'$ and $\neg v'$), literal $v'$ occurs once in a clause of $\psi_v$, and literal $\neg v'$ occurs once in a clause of $\psi_{\neg v}$. A similar statement holds if $R_{\psi_{v'}} \in \psi'_v$.

Theorem 51. A CNF formula $\psi$ with $n$ variables and $n+1$ clauses is minimally unsatisfiable if and only if there is a refutation tree for $\psi$ in which every clause of $\psi$ labels a leaf exactly one time, every variable of $\psi$ labels exactly two edges (once as a positive literal and once as a negative literal), and every edge label of the tree appears in at least one clause of $\psi$.

Proof. $\iff$) By induction on the size of the refutation tree. Suppose there is such a refutation tree $T_\psi$ for $\psi$. If $T_\psi$ consists of two edges, they must be labeled $v$ and $\neg v$ for some variable $v$. In each clause labeling a leaf of $T_\psi$ must consist of one literal that is opposite to that labeling the edge the leaf is the endpoint of. Hence, $\psi$ must be $\psi$ which is minimally unsatisfiable.

Now suppose the theorem holds for refutation trees of $k > 2$ or fewer edges and suppose $T_\psi$ has $k+1$ edges. Let $v$ be the variable associated with the root of $T_\psi$, and let $\psi_v$ and $\psi_{\neg v}$ be the sets of clauses labeling leaves in the subtree joined to the root edges labeled $v$ and $\neg v$, respectively. Define $\psi_1 = \{c \setminus \{\neg v\} : c \in \psi_v\}$ and $\psi_2 = \{c \setminus \{v\} : c \in \psi_{\neg v}\}$. It is straightforward to see that the relationships between each refutation subtree rooted at a child of the root of $T_\psi$ and $\psi_1$ and $\psi_2$ are as described in the statement of the theorem. Hence, by the induction hypothesis, $\psi_1$ and $\psi_2$ are minimally unsatisfiable. Let $c$ be a clause in $\psi_1$. Let $M_1$ be an assignment to variables of $\psi_1$ satisfying $\psi_1 \setminus \{c\}$. Let $M_2 = M_1 \cup \{v\}$. Clearly, $M_2$ satisfies $\psi_2$ as well as all clauses of $\psi_{\neg v}$ which contain literal $v$. The remaining clauses of $\psi_{\neg v}$ are a proper subset of clauses of $\psi_2$ and are satisfied by some truth assignment $M_3$ to variables of $\psi_2$ since $\psi_2$ is minimally unsatisfiable. Since the variables of $\psi_1$ and $\psi_2$ do not overlap, $M_2 \cup M_3$ is an assignment satisfying $\psi_1 \cup \psi_{\neg v} \setminus \{c\}$ and therefore $\psi \setminus \{c\}$. The same result is obtained if $c$ is removed from $\psi_2$. Thus, $\psi$ is minimally unsatisfiable.

$\iff$) Build the refutation tree in conjunction with running Algorithm Min Unsat Solver as follows. Before running the algorithm construct $n+1$ (leaf) nodes and distinctly label each with a clause of $\psi$. While running the algorithm, add a new (non-leaf) node each time a resolvent is computed. Unlike the case for a normal refutation tree, label the new node with the resolvent. Construct two edges from the new node to the two nodes which
Algorithm 22.

**Min Unsat Solver** ($\psi$)

/* Input: a set of sets CNF formula $\psi$ */
/* Output: either “munsat(1)” or “not munsat(1)” */

Repeat the following until all variables of $\psi$ are eliminated:

If the difference between clauses and variables is not 1,
Output “not munsat(1).”

If no variable of $\psi$ occurs once positively and once negatively,
Output “not munsat(1).”

If, for some $v$, $\psi$ includes $(v) \land (\neg v)$ and $|\psi| > 2$,
Output “not munsat(1).”

Choose variable $v$ such that $v \in c_v$ and $\neg v \in c_{\neg v}$
and neither literals $v$ nor $\neg v$ are in $\psi \setminus \{c_v, c_{\neg v}\}$.
Compute the resolvent $R_{c_v, c_{\neg v}}$ of clauses $c_v$ and $c_{\neg v}$.
Set $\psi \leftarrow \psi \setminus \{c_v, c_{\neg v}\} \cup \{R_{c_v, c_{\neg v}}\}$.
Output “munsat(1).”

Figure 4.8: Algorithm for determining whether a CNF formula with one more clause than variable is minimally unsatisfiable.

are labeled by the two clauses being resolved ($c_v$ and $c_{\neg v}$ in the algorithm.
If the pivot variable is $v$, label the edge incident to the node labeled by the
clause containing $v$ (alternatively $\neg v$) $\neg v$ ($v$, respectively). Continue until
a single tree containing all the original leaf nodes is formed.

The graph constructed has all the properties of the refutation tree as
stated in the theorem. It must include one or more trees since each clause,
original or resolvent, is used one time in computing a resolvent. Since two
edges are added for each new non-leaf node and $n$ new non-leaf nodes are
added, there must be $2n + 1$ nodes and $2n$ edges in the structure. Hence,
it must be a single tree. Any clause labeling a node contains all literals in
clauses labeling leaves beneath that node minus all literals of pivot variables
beneath and including that node. In addition, the label of the root is $\emptyset$.
Therefore, all literals of a clause labeling a leaf are a subset of the comple-
ments of edge labels on a path from the root to that leaf. Obviously, the
complement of each edge label appears at least once in leaf clauses.

An example of such a minimally unsatisfiable formula and corresponding
refutation tree is shown in Figure 4.9.

Now we turn to the case of minimally unsatisfiable formulas with defi-
4.10.1 MINIMALLY UNSATISFIABLE FORMULAS

Figure 4.9: A minimally unsatisfiable CNF formula $\psi$ of six variables and seven clauses and a corresponding refutation tree. Edge labels are variable assignments (e.g. $\neg v_1$ means $v_1$ has value 0). Each leaf is labeled with a clause that is falsified by the assignment indicated by the path from the root to the leaf.

The maximum deficiency of a formula can be determined in polynomial time.
Lemma 53. Let $\psi$ be a minimally unsatisfiable CNF formula which has $\delta$ more clauses than variables. Then the maximum deficiency of $\psi$ is $\delta$.

Proof.

Lemma 54. Let $\psi$ be a minimally unsatisfiable formula. Then for every nonempty set $V'$ of variables of $\psi$ there is at least one clause $c \in \psi$ such that some variable of $V'$ occurs in $c$.

Proof.

Lemma 55. Let $\psi$ be a CNF formula with maximum deficiency $\delta$. A maximum matching of the bipartite variable-clause graph $G_\psi$ of $\psi$ does not cover $\delta$ clause vertices.

Proof.

Lemma 55 is important because it is used in the next lemma and because it shows that the maximum deficiency of a CNF formula can be computed with a $O(n^3)$ matching algorithm.

Lemma 56. Let $\psi$ be a 1-expanding CNF formula with maximum deficiency $\delta$. Let $\psi' \subset \psi$ be a subformula of $\psi$. Then the maximum deficiency of $\psi'$ is less than $\delta$.

Proof. We need to show that for any subset $\psi' \subset \psi$, if the number of variables contained in $\psi'$ is $n'$, then $|\psi'| - n' < \delta$. Let $G_\psi$ be the bipartite variable-clause graph of $\psi$. In what follows symbols will be used to represent clause vertices in $G_\psi$ and clauses in $\psi$ interchangeably. Choose a clause $c \in \psi \setminus \psi'$. Let $M_{G_\psi}$ be a maximum matching on $G_\psi$ that does not cover vertex $c$. There is always one such matching because, by hypothesis, every subset of variable vertices has a neighborhood which is larger than the subset and this allows $c$'s cover to be moved to another clause vertex, if necessary. Let $C$ be the set of all clause vertices of $G_\psi$ that are not covered by $M_{G_\psi}$. By Lemma 55 $|C| = \delta$ so, since $c \notin \psi'$, $|C \cap \psi'| < \delta$. Since $M_{G_\psi}$ matches every clause vertex in $\psi' \setminus C$ to a variable that is in $\psi'$, the number of clauses in $\psi' \setminus C$ must be no bigger than $n'$. Therefore, $|\psi'| - n' \leq |\psi'| - |\psi' \setminus C|$. But $|\psi'| - |\psi' \setminus C|$ is the number of clauses in $C \cap \psi'$ which is less than $\delta$. 

Theorem 57. ([124]) Let $\psi$ be a CNF formula with maximum deficiency $\delta$. The satisfiability of $\psi$ can be determined in time $O(2^\delta n^3)$ where $n$ is the number of variables in $\psi$.

Proof. Let $m$ be the number of clauses in $\psi$. By hypothesis, $m \leq n + \delta$. Let $G = (V_1, V_2, E)$ be the variable-clause matching graph for $\psi$. Find a maximum matching $M_G$ for $G$. Since $nm \leq n(n + \delta) = O(n^2)$, this can be done in $O(n^3)$ time by the well known Hopcroft-Karp maximum cardinality
4.10.2 BOUNDED RESOLVENT LENGTH RESOLUTION

The next step is to build a refutation tree for $\psi$ of depth $\delta$.

**Theorem 58.** ([124]) Let $\psi$ be a minimally unsatisfiable CNF formula with $\delta$ more clauses than variables. Then $\psi$ can be recognized as such in time $O(2^\delta n^4)$ where $n$ is the number of variables in $\psi$.

**Proof.** By Theorem 53 the maximum deficiency of $\psi$ is $\delta$ which, by Lemma 55, can be checked in $O(n^3)$ time. Since, by Lemma 56, the removal of a clause from $\psi$ reduces the maximum deficiency of $\psi$, the algorithm inferred by Theorem 57 may be used to check that, for each $c \in \psi$, $\psi \setminus \{c\}$ is satisfiable. The algorithm may also be used to check that $\psi$ is unsatisfiable. Since the algorithm is applied $m + 1$ times and $m = n + \delta$, the complexity of this check is $O(2^\delta n^4)$. Therefore, the entire check takes $O(2^\delta n^4)$ time.

4.10.2 Bounded Resolvent Length Resolution

We consider this class for two reasons: as a counterpoint to minimally unsatisfiable formulas and to assist our understanding of probabilistic results presented in Chapter 7. A CNF formula $\psi$ is $k$-BRLR if Algorithm BRLR of Figure 4.10 either generates all resolvents of $\psi$ or returns “unsatisfiable.”

Algorithm BRLR implements a simplified form of $k$-closure [131]. It repeatedly applies the resolution rule to a CNF formula with the restriction that all resolvents are of size no greater than some fixed $k$. In other words, the algorithm finds what are sometimes called “$k$-bounded” refutations.

For a given CNF formula with $n$ variables and $m$ clauses, the worst-case number of resolution steps required by the algorithm is

$$
\sum_{i=1}^{k} 2^i \binom{n}{i} = 2^k \binom{n}{k} (1 + O(k/n)).
$$

This essentially reflects the product of the cost of finding a resolvent and the maximum number of times a resolvent is generated. The latter is $2^k \binom{n}{k}$. The cost of finding a resolvent depends on the data structures used in implementing the algorithm.

For every clause, maintain a linked list of literals it contains, in order by index, and a linked list of possible clauses to resolve with such that the resolvent has no greater than $k$ literals. Maintain a list $T$ of matrices of dimension $n \times 2$, $\binom{n}{2} \times 4$, …, $\binom{n}{k} \times 2^k$ such that each cell has value 1 if and only if a corresponding original clause of $\psi$ or resolvent exists at any particular iteration of the algorithm. Also, maintain a list of clauses that may resolve with at least one other clause to generate a new resolvent: the list is threaded through the clauses. Assume a clause can be accessed in constant time and a clause can access its corresponding cell in $T$ in constant
Algorithm 23.

\[ \text{BRL}R(\psi, k) \]

/* Input: a set of sets CNF formula \( \psi \) */

/* Output: “unsatisfiable,” “satisfiable,” or “give up” */

Repeat the following until some statement outputs a value:

If there are clauses \( c_1, c_2 \in \psi \) that resolve to \( R_{c_1}c_2 \not\in \psi, |R_{c_1}c_2| \leq k \)

Set \( \psi \leftarrow \psi \cup R_{c_1}c_2 \).

Otherwise, if \( \emptyset \not\in \psi \) do the following:

If all resolvents have been generated then Output “satisfiable”.

Otherwise, Output “give up”.

If \( \emptyset \in \psi \), Output “unsatisfiable.”

Figure 4.10: Finding a \( k \)-bounded refutation

time by setting a link just one time as the clause is scanned the first time or created as a resolvent.

Initially, set to 1 all the cells of \( T \) which correspond to a clause of \( \psi \) and set all other cells to 0. Over \( \binom{m}{2} \) pairs of clauses, use \( 2L \) literal comparisons to determine whether the pair resolves. If so, and their resolvent has \( k \) or fewer literals, and the cell in \( T \) corresponding to the resolvent has value 0, then set the cell to 1, add a link to the clause lists of the two clauses involved, and add to the potential resolvent list any of the two clauses that is not already threaded through it. This accounts for complexity \( O(Lm^2) \).

During an iteration, select a clause \( c_1 \) from the potential resolvent list. Scan through \( c_1 \)’s resolvent list checking whether the cell of \( T \) corresponding to the other clause, \( c_2 \), has value 1. If so, delete \( c_2 \) from \( c_1 \)’s list. If \( c_1 \)’s list is scanned without finding a cell of value 0, delete \( c_1 \) from the potential resolvent list and try the next clause in the potential resolvent list. When some clause is paired with another having a cell of value 0 in \( T \), a new resolvent is generated. In this case, construct a new clause, create a resolvent list for the clause by checking for resolvents with all existing clauses.

4.11 Comparison of Classes

We briefly observe that the SLUR, q-Horn, nested and matched classes are incomparable, the class of q-Horn formulas without unit clauses is subsumed by the class of linear autark formulas and SLUR is incomparable with the linear autark formulas. All the other classes we considered are contained in one or more of the three. For example, Horn formulas are in the intersection of the q-Horn and SLUR classes and all 2-SAT formulas are q-Horn.
4.11 COMPARISON OF CLASSES

Any Horn formula with more clauses than distinct variables is not Matched, but is both SLUR and q-Horn.

The following is a matched and q-Horn formula but is not a SLUR formula:

\[(v_1 \lor \neg v_2 \lor v_4) \land (v_1 \lor v_2 \lor v_5) \land (\neg v_1 \lor \neg v_3 \lor v_6) \land (\neg v_1 \lor v_3 \lor v_7).\]

In particular, in Algorithm SLUR, initially choosing 0 values for \(v_4, v_5, v_6, v_7\), leaves an unsatisfiable formula with no unit clauses. To verify q-Horn membership, set \(\alpha_1 = \alpha_2 = \alpha_3 = 1/2\) and the remaining \(\alpha\)'s to 0 in 2.2.

The following formula is matched and SLUR but is not q-Horn:

\[(\neg v_2 \lor v_3 \lor \neg v_5) \land (\neg v_1 \lor \neg v_3 \lor v_4) \land (v_1 \lor v_2 \lor \neg v_4).\]

In particular, the satisfiability index of this formula is 4/3. To verify SLUR membership, observe that in Algorithm SLUR no choice sequence leads to an unsatisfiable formula without unit clauses.

The following formula is nested but not q-Horn (minimum \(Z\) is 5/4):

\[(\neg v_3 \lor \neg v_4) \land (v_3 \lor v_4 \lor \neg v_5) \land (\neg v_1 \lor v_2 \lor \neg v_3) \land (v_1 \lor v_3 \lor v_5).\]

The following formula is nested but not SLUR (choose \(v_3\) first, use the branch where \(v_3 = 0\) to enter a situation where satisfaction is impossible):

\[(v_1 \lor v_2) \land (v_1 \lor \neg v_2 \lor v_3) \land (\neg v_1 \lor v_4) \land (\neg v_1 \lor \neg v_4).\]

The following is a linear autark formula that is not q-Horn (\(\alpha = (0.5, -0.5, 0)\) and the minimum \(Z\) is 3/2):

\[(v_1 \lor v_2 \lor v_3) \land (\neg v_1 \lor \neg v_2 \lor \neg v_3).\]

The following is SLUR but is not a linear autark formula (by symmetry, any variable choice sequence in SLUR leads to the same result which is a satisfying assignment and only \(\alpha = (0, 0, 0)\) satisfies the inequality of (4.1)):

\[(\neg v_1 \lor v_2 \lor v_3) \land (v_1 \lor \neg v_2 \lor v_3) \land (v_1 \lor v_2 \lor \neg v_3) \land (\neg v_1 \lor \neg v_2 \lor \neg v_3).\]

The following formula is minimally unsatisfiable with one more clause than variable, but is not 3-BRLR:

\[(v_9 \lor v_1 \lor v_{18}) \land (v_9 \lor \neg v_1 \lor v_{56}) \land (\neg v_9 \lor v_2 \lor v_{90}) \land (\neg v_9 \lor \neg v_2 \lor v_2) \land \]
\[(v_1 \lor v_4 \lor \neg v_8) \land (v_3 \lor \neg v_5 \lor v_{56}) \land (\neg v_3 \lor v_5 \lor v_{90}) \land (\neg v_3 \lor \neg v_5 \lor v_2) \land \]
\[(v_6 \lor v_7 \lor v_{49}) \land (v_6 \lor \neg v_7 \lor \neg v_{56}) \land (\neg v_6 \lor v_8 \lor v_{90}) \land (\neg v_6 \lor \neg v_8 \lor v_2) \land \]
\[(v_9 \lor v_{19} \lor \neg v_{10}) \land (v_9 \lor \neg v_{10} \lor \neg v_{56}) \land (\neg v_9 \lor v_{11} \lor v_{90}) \land (\neg v_9 \lor \neg v_{11} \lor v_2) \land \]
\[(v_{12} \lor v_{13} \lor \neg v_{20}) \land (v_{12} \lor \neg v_{13} \lor v_{57}) \land (\neg v_{12} \lor v_{14} \lor \neg v_{60}) \land (\neg v_{12} \lor \neg v_{14} \lor v_2) \land \]
\[(v_{15} \lor v_{16} \lor \neg v_{20}) \land (v_{15} \lor \neg v_{16} \lor v_{57}) \land (\neg v_{15} \lor v_{17} \lor \neg v_{60}) \land (\neg v_{15} \lor \neg v_{17} \lor v_2) \land \]
\[(v_{18} \lor v_{19} \lor v_{31}) \land (v_{18} \lor \neg v_{19} \lor \neg v_{57}) \land (\neg v_{18} \lor v_{20} \lor \neg v_{60}) \land (\neg v_{18} \lor \neg v_{20} \lor v_2) \land \]
\[(v_{21} \lor v_{22} \lor \neg v_{31}) \land (v_{21} \lor \neg v_{22} \lor \neg v_{57}) \land (\neg v_{21} \lor v_{23} \lor \neg v_{60}) \land (\neg v_{21} \lor \neg v_{23} \lor v_2) \land \]
This formula was obtained from a complete binary refutation tree, variables \( v_0 \) to \( v_{47} \) labeling edges of the bottom two levels and \( v_{48} \) to \( v_{62} \) labeling edges of the top four levels. Along the path from the root to a leaf, the clause labeling that leaf contains all variables of the bottom two levels and one variable of the top four levels. Thus, resolving all clauses in a subtree rooted at variable \( v_{3i} \), \( 0 \leq i \leq 15 \), leaves a resolvent of four literals, all taking labels from edges in the top four levels.

To any \( k \)-BRLR formula \( \psi \) that is unsatisfiable, add another clause arbitrarily using the variables of \( \psi \). The result is a formula that is not minimally satisfiable.

The above ideas can be extended to show that each of the classes contains a formula that is not a member of any of the others. These examples may be extended to infinite families with the same properties.

The subject of comparison of classes will be revisited in Section 7.6 using probabilistic measures to determine relative sizes of the classes.
Chapter 5

Assisting Search

5.1 Transformations

5.2 Lookahead

5.2.1 Depth-first Lookahead

5.2.2 Breadth-first Lookahead

5.2.3 Function-complete Lookahead

5.3 Learning: Conflict Resolution

5.4 Backjumping

5.5 Adding Uninferred Constraints

5.5.1 Autarkies

5.5.2 Non-monotonic constraints

5.5.3 Unsafe constraints: tunnels
Chapter 6

Lower and Upper Bounds

6.1 Upper Bounds on Complexity

Exploitable properties have been used to find $2^n(1-\epsilon)$ upper bounds on the number of steps needed to solve a given $k$-CNF formula $\psi$. The first non-trivial upper bound is found in the classic paper of Monien and Speckenmeyer [107] and is based on the notion of autark assignments, defined on Page 83.

If an autark assignment is found during search, it is sufficient to fix the autark assignment below that point in the search: that is, backtracking to consider other values for the variables of the autark assignment is unnecessary at the point the autark assignment is discovered. A well-known example of an autark assignment is the assignment of 1 to a pure literal: it is unnecessary to expand the search space with a pure literal assigned value 0 because a formula that is not satisfiable with a pure literal set to 1 is also not satisfiable with that pure literal set to 0. Monien and Speckenmeyer, in [107], analyzed a DPLL variant which branches on all assignments to the variables of a shortest width clause except the one falsifying the clause. Such assignments either produce clauses of shorter width or are autark. This provides an upper bound of $O(\alpha_k^n)$, where $\alpha_k$ is the largest real root of the equation

$$\alpha_k^k - 2\alpha_k^{k-1} + 1 = 0.$$ 

Thus, if $k = 3$, the upper bound is $O(1.618^n)$ steps. The bound increases with increasing $k$ and approaches $O(2^n)$. A number of more complex variants have been studied and the best bounds obtained in this way, so far, appear to be $O(1.505^n)$ [91] and $O(1.497^n)$ [118] for 3-SAT.
### Table 6.1: Formula for proving that some hole must have at least two pigeons given $n + 1$ pigeons are assigned to $n$ holes.

<table>
<thead>
<tr>
<th>Variables</th>
<th>Subscript Range</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| $v_{i,k}$ | $1 \leq i \leq n$  
$1 \leq k \leq n + 1$ | $v_{i,k} = 1$ iff pigeon $k$ is in hole $i$ |

<table>
<thead>
<tr>
<th>Clauses</th>
<th>Subscript Range</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(v_{1,k} \lor \ldots \lor v_{n,k})$</td>
<td>$1 \leq k \leq n + 1$</td>
<td>Every pigeon is in at least one hole</td>
</tr>
</tbody>
</table>
|$(\neg v_{l,k} \lor \neg v_{i,k})$| $1 \leq l < k \leq n + 1$  
$1 \leq i \leq n$| Each hole contains no more than one pigeon |

#### 6.2 Exponential Lower Bounds on Complexity

An important breakthrough in the complexity of resolution was achieved by Armin Haken [69] in 1985 when he proved exponential lower bounds for the class of pigeon-hole formulas. The pigeon-hole problem is to prove that it is impossible to assign $n + 1$ pigeons to $n$ holes without at least one hole containing two pigeons. A pigeon-hole formula for $n$ holes is shown in Table 6.1 along with a description of the meaning of each clause. The formula has $n + 1$ $n$-literal clauses and $n \binom{n + 1}{2}$ 2-literal clauses.

It is straightforward to see that the formulas of Table 6.1 are unsatisfiable. Any truth assignment for which at least two variables of the same hole index have value 1 can be eliminated immediately as a model by one of the 2-literal clauses. For all other assignments at most one variable of each hole index has value 1. Any such variable can satisfy at most one of the $n$-literal clauses. But there are $n + 1$ of those clauses so at least one of them must be falsified. Hence all assignments where at most one variable of the same hole index has value 1 also cannot be models. Thus, no assignments can be models.

It is straightforward to show that the pigeon-hole formulas are unsatisfiable using resolution (Algorithm 2 in Figure 3.3). But despite the simple argument of the previous paragraph, the number of resolution steps required is exponential in $n$ regardless of the order in which resolutions are carried out.

Pigeon-hole formulas are a simple example of a number of formula classes that are difficult for resolution. An important, frequently occurring class is the class of formulas representing minmax functions. A minmax function \(f^m_n(X) : Z \mapsto \{0, 1\}\) takes value 1 if and only if $n \leq X \leq m$. 

6.3 Extended Resolution vs. Resolution

In Section 3.3 extended resolution was introduced as the addition to a CNF formula $\psi$ of expressions of the form

$$z \leftrightarrow f(v_1, v_2, \ldots, v_k),$$

but in clausal form, where $z$ is a variable that is not in a clause of $\psi$, and each of $v_1, \ldots, v_k$ is in one or more clauses of $\psi$. The larger formula is functionally equivalent to $\psi$. However, resolution proofs on the larger formulas may be considerably shorter than the shortest proofs of $\psi$. This is illustrated in this section with the family of pigeon-hole formulas. The discussion is taken from [37] and [39].

The idea is to add to $n$-pigeon-hole formula $\psi_n$ expressions whose clauses will resolve with those of $\psi_n$ to create an $(n-1)$-pigeon-hole formula, then repeat until clauses for a 1-pigeon-hole formula are created. The added expressions introduce relatively few resolvents and the 1-pigeon-hole formula is trivial to prove unsatisfiable.

We start by supposing it is possible to assign $n + 1$ pigeons to $n$ holes. Then there is some function $f : \{1, \ldots, n+1\} \mapsto \{1, \ldots, n\}$ which maps pigeons to holes uniquely. Now consider function $f' : \{1, \ldots, n\} \mapsto \{1, \ldots, n-1\}$ which is defined as follows:

$$f'(j) = \begin{cases} 
    f(j) & \text{if } f(j) \neq n \\
    f(n+1) & \text{otherwise}
\end{cases}.$$ 

Function $f'$ has the same mapping as $f$ except for that $j$ which is mapped to $n$. But if $f$ maps uniquely, then $f(n+1)$ cannot map to $n$, nor to any value equal to some other $f(j)$. So, substituting $f(n+1)$ for $f(j)$ in that case results in $f'$ mapping to $\{1, \ldots, n-1\}$ uniquely. The mapping $f'$ is captured by the following family of expressions:

$$z_{i,j}^{n-1} \leftrightarrow v_{i,j} \lor (v_{n,j} \land v_{i,n+1}), \quad 1 \leq i \leq n-1, 1 \leq j \leq n.$$ 

All the $z_{i,j}^{n-1}$ will act like the $v_{i,j}$ except that the maximum of $i$ and $j$ are reduced by 1. In other words, if a unique mapping is possible and all expressions are satisfied, one of $z_{i,1}^{n-1}, z_{i,2}^{n-1}, \ldots, z_{i,n}^{n-1}$, $1 \leq i \leq n-1$, will have value 1 and all $\neg z_{i,j}^{n-1} \lor \neg z_{i,k}^{n-1}$ will also have value 1. We can add the clausal versions of these expressions to $\psi_n$ and resolve with clauses of $\psi_n$ to produce clauses representing an $n-1$-pigeon-hole problem. The added clauses are, for $1 \leq i \leq n-1$, $1 \leq j \leq n$:

$$(-z_{i,j}^{n-1} \lor v_{i,j} \lor v_{n,j}) \land (-z_{i,j}^{n-1} \lor v_{i,j} \lor v_{i,n+1}) \land (z_{i,j}^{n-1} \lor v_{i,j}) \land (z_{i,j}^{n-1} \lor v_{i,j} \lor v_{i,n+1})$$
Resolving the added clauses and original set produces the set
\[(z_{n,k}^{n-1} \lor \ldots \lor z_{n-1,k}^{n-1}) \quad \text{for} \quad 1 \leq k \leq n, \quad \text{and}\]
\[(-z_{i,l}^{n-1} \lor -z_{i,k}^{n-1}) \quad \text{for} \quad 1 \leq l < k \leq n, 1 \leq i \leq n - 1.\]

New clauses corresponding to
\[z_{i,j}^{n-2} \leftrightarrow z_{i,j}^{n-1} \lor (z_{n,j}^{n-1} \land z_{i,n+1}^{n-1})\]
are added and resolved with those above and the process repeats until finally
the following clauses are added
\[(z_{1,1}^{1} \land z_{1,2}^{1}) \land (\neg z_{1,1}^{1} \lor z_{1,2}^{1})\]
which immediately resolved to a null clause. The total number of resolution
steps for the whole process is \(O(m^{4/3})\) where \(m\) is the number of clauses in
\(\psi_n\).
Chapter 7

Probabilistic Analysis

In this chapter we present examples of analysis using the tools of Section 2.4. We choose examples which represent, without complication, ideas that have led to progressively better results. This means, in some cases, the results obtained here are not the “best” known.

7.1 Myopic algorithms for satisfiable formulas

For densities where random expressions are satisfiable with high probability the behavior of some DPLL variants can be understood adequately by means of performance results on corresponding straight line algorithms. A first wave of many results were obtained for myopic algorithms of a family called Shortest Clause Algorithms because each variable assignment is chosen to satisfy a non-satisfied clause of least width. This family is presented as Algorithm SCA in Figure 7.1. The algorithm takes as input a k-CNF formula $\psi$ and a parameter $s$ which determines how long a short clause can be before randomly selecting a variable for assignment from all of $\psi$ instead of from a shortest clause. Once a variable is selected for assignment, the following line causes some clauses to shrink and some to disappear:

Set $\psi \leftarrow \{c - \{l\} : c \in \psi \text{ and } l \notin c\}$.

Hence, sets $C_i(j)$, $0 \leq i \leq k$, $0 \leq j \leq n$ are defined to contain all clauses of width $i$ that remain after $j$ variables have been assigned values.

The study of SCA is motivated by the observation that if a straight line algorithm is to succeed, then it cannot allow an accumulation of unit clauses (that is, $|C_1(j)|$ must remain small for all $j$) since too many unit clauses makes it likely that a pair of unit clauses with opposite polarity exist in $C_1(j)$ for some $j$ and that would cause the algorithm to terminate without success. Intuitively, then, eliminating shortest clauses first should have high priority.
Algorithm 24.

**SCA** ($\psi, s$):

**Input:** a set of sets CNF formula $\psi$, integer $s$.

**Output:** either “give up” or a model for $\psi$.

Set $M \leftarrow \emptyset$.

Repeat the following until some statement outputs a value:
- If $\psi = \emptyset$ then Output $M$.
- If $\emptyset \in \psi$ then Output “gives up”.
- For $i$ from 0 to $k$ let $C_i = \{c : c \in \psi \text{ and } |c| = i\}$.
  - Set $q \leftarrow \min\{i : C_i \neq \emptyset\}$.
  - if $q \leq s$ then do the following:
    - Set $L_1 \leftarrow \{l : \exists c \in C_q \text{ such that } l \in c\}$.
    - Choose $l$ randomly from $L_1$.
  - Otherwise,
    - Set $L_2 \leftarrow \{l : \exists c \in \psi \text{ such that } l \in c\}$.
    - Choose $l$ randomly from $L_2$.
- If $l$ is a positive literal then Set $M \leftarrow M \cup \{l\}$.
- Set $\psi \leftarrow \{c - \{\neg l\} : c \in \psi \text{ and } l \notin c\}$.

Figure 7.1: Smallest Clause Algorithm for $k$-CNF formulas.

Probabilistic analysis confirms this intuition to some extent. Results for the family of shortest clause first algorithms are quite good compared to some other heuristics, for example the pure literal heuristic alone. In addition, the analysis is based on *clause-flows* and illuminates the algorithmic mechanics that cause successful behavior. The remainder of this section is a high-level presentation of the analysis of several straight line algorithms. The main intention is to focus on motivation and intuition with a minimum of details.

A clause-flow model can be used to analyze the mechanics of many straight line algorithms. In particular, in this section we concentrate on the family identified as Algorithm **SCA** in Figure 7.1. Let $\psi$ be a random $(m, n, k)$-CNF formula. Define $w_i(j)$ to be the number of $i$-literal clauses that are added to $C_i(j)$ as a result of choosing a literal on the $j$th iteration. That is, $w_i(j)$ is the flow of clauses into $C_i(j)$ due to assigning a value to a literal $l$ and removing $\neg l$ from clauses on the $j$th iteration. Define $z_i(j)$ to be the number of clauses eliminated from $C_i(j)$ on the $j$th iteration. That is, $z_i(j)$ is the flow out of $C_i(j)$ due to clauses that contain chosen literal $l$ (these clauses become satisfied and are therefore eliminated). Let $m_i(j) = |C_i(j)|$. 
Figure 7.2 shows clause sets $C_i(j)$ represented by ovals, and clause flows $w_i(j)$ and $z_i(j)$ represented by arcs with arrows indicating flow direction.

The success of a shortest clause algorithm depends critically on what is happening to $w_0(j)$. If $w_0(j) > 0$ for any $j$, any shortest clause algorithm stops and gives up because some clause has just had all its literals falsified by the current partial truth assignment. In turn, $w_0(j)$ can be controlled by keeping complementary pairs of clauses out of $C_1(j)$, for all $j$, since, if such a pair exists in $C_1(j)$ for some $j = j^*$, then eventually $w_0(j) > 0$ for some $j = j^* > j'$. Complementary pairs may be kept out of $C_1(j)$, for all $j$, by preventing a significant accumulation of unit clauses over time since such an accumulation tends to raise the probability that a complementary pair exists. Choosing a unit clause literal first, if one exists, does this by acting like a “pump” that attempts to immediately discharge all clauses which flow into $C_1(j)$. Unfortunately, it is not usually the case that more than one unit clause can be discharged at a time. Therefore, by choosing unit clause literals first, one is unable to prevent an accumulation in $C_1(j)$ when $w_1(j) > 1$ over a significant range of $j$.

An important approach to the analysis of shortest clause algorithms is to model the clause flows and accumulations as a system of differential equations and determine under what conditions $\max_j \{w_1(j)\} = 1$. Those conditions mark the boundary of good probabilistic performance. In what follows we try this idea for Algorithm SCA given in Figure 7.1 with $s = 1$. This algorithm is known in the literature as UC for Unit Clause.

Whether the flows can be modeled according to the approach stated above depends on two things: 1) the clauses in $C_i(j)$, for any $i$ and $j$, should be statistically independent and uniformly distributed; 2) conditions whereby markovian processes may be modeled as differential equations should be satisfied. In the first case, clauses entering $C_i(j)$ are independent of clauses existing in $C_i(j)$ and of each other, and are uniformly distributed since they were so in $C_i(j+1)$ and the chosen literal is selected randomly from the appropriate set of free literals. Also, conditioned on the event that at least one unit clause leaves $C_1(j)$ when $|C_1(j)| > 0$, clauses leave $C_i(j)$ independently as well. This establishes that Algorithm SCA with $s = 1$ is myopic (see Section 2.4.7). In the second case, even as $n$ grows, the flows $w_i(j)$ and $z_i(j)$ are binomially distributed with means that are less than $km/n$ which is bounded by a constant. This is enough to satisfy the conditions of Theorem 8.

We are justified, therefore, in modeling the discrete clause flows and accumulations by a system of differential equations which will now be developed. To describe the sizes of sets $C_i(j+1)$ in terms of set sizes on iteration $j$ write, for $1 \leq i \leq k$, $0 < j < n$,

$$m_i(j+1) = m_i(j) + w_i(j) - z_i(j).$$
Figure 7.2: Clause sets and flows for Algorithm SCA. On the left is a schematic representation. On the right are plots of the expected number of clauses in $C_i(j)$ versus $j/n$ for the case $k = 4$ with $i = 2, 3, 4$.

Taking expectations gives

$$E\{m_i(j + 1)\} = E\{m_i(j)\} + E\{w_i(j)\} - E\{z_i(j)\}.$$  

This can be written

$$E\{m_i(j + 1) - m_i(j)\} = E\{w_i(j)\} - E\{z_i(j)\}. \quad (7.1)$$

Due to the statistical independence of clauses at every $j$, and the fact that $\psi$ is originally constructed from a set of $n$ variables, we have, for all $2 \leq i \leq k$, $0 < j < n$,

$$E\{z_i(j)\} = E\{E\{z_i(j)|m_i(j)\}\}$$

$$= E\left\{ \frac{i \ast m_i(j)}{n-j} \right\} = \frac{i \ast E\{m_i(j)\}}{n-j}.$$
Also, for all \(1 \leq i < k\), \(0 < j < n\),

\[
E \{ w_i(j) \} = E \{ E \{ w_i(j) | m_{i+1}(j) \} \} = E \left\{ \frac{(i+1) * m_{i+1}(j)}{2(n-j)} \right\} = \frac{(i+1) * E \{ m_{i+1}(j) \}}{2(n-j)} \text{ and}
\]

\[
E \{ w_k(j) \} = 0.
\]

After substituting these results into Equation 7.1 and verifying conditions in Theorem 8, we can write the corresponding difference equations as follows:

for \(2 \leq i < k\),

\[
E \{ m_i(j+1) - m_i(j) \} = \frac{(i+1) * E \{ m_{i+1}(j) \}}{2(n-j)} - \frac{i * E \{ m_i(j) \}}{n-j} \text{ and}
\]

\[
E \{ m_k(j) \} = -\frac{k * E \{ m_k(j) \}}{n-j}.
\]

For brevity, we use \(\bar{m}_i(x)\) to mean \(E \{ m_i(x) \}\). Then the corresponding differential equations are

\[
\frac{d\bar{m}_i(x)}{dx} = \frac{(i+1) * \bar{m}_{i+1}(x)}{2(1-x)n} - \frac{i * \bar{m}_i(x)}{(1-x)n} \text{ and}
\]

\[
\frac{d\bar{m}_k(x)}{dx} = -\frac{k * \bar{m}_k(x)}{(1-x)n}
\]

where \(\bar{m}_i(x)\) is \(z_i(x)\) of Theorem 8.

Boundary conditions, assuming \(m\) clauses of \(k\) literals in \(\psi\) initially, are \(\bar{m}_k(0) = m/n\) and \(\bar{m}_i(0) = 0\) for all \(1 \leq i < k\). The solution to the equations with these boundary conditions is, for all \(2 \leq i \leq k\)

\[
\bar{m}_i(x) = \frac{1}{n-1} \binom{k}{i} (1-x)^i (x)^{k-i} m/n.
\]

Thus,

\[
E \{ m_i(j) \} = \frac{1}{2^{k-i}} \binom{k}{i} (1-j/n)^i (j/n)^{k-i} m.
\]

Plots of these functions for \(k = 4\) are given in Figure 7.2.

The important flow is given by

\[
E \{ w_1(j) \} = \frac{E \{ m_2(j) \}}{n-j} = \frac{1}{2^{k-2}} \binom{k}{2} (1-j/n)(j/n)^{k-2}(m/n).
\]

By Theorem 9 and since \(|E \{ m_2(j) \} - m_2(j)| < \beta\), for any \(0 < \beta\), almost always, from Theorem 8(b), Algorithm UC succeeds with probability bounded from below by a constant if

\[
E \{ w_1(j) \} = \frac{E \{ m_2(j) \}}{n-j} < \frac{m_2(j) + \beta}{n-j} = \frac{m_2(j)}{n-j} + o(1) < 1 - \epsilon + o(1)
\]
for any $0 < \epsilon$. That is, the algorithm succeeds as long as the average rate of production of unit clauses is no greater than 1 per iteration at any iteration.

Taking the derivative with respect to $j$ and setting to 0 yields a maximum for $E\{w_1(j)\}$ at $j = j^* = \frac{k-2}{k-1}n$. The value of $E\{w_1(j^*)\}$ is less than 1 if

$$\frac{m}{n} < \frac{2^{k-1}}{k} \left( \frac{k-1}{k-2} \right)^{k-2}.$$

We can now conclude the following.

**Theorem 59.** Algorithm UC determines that a given random $(m, n, k)$-CNF formula has a model with probability bounded from below by a constant if

$$\frac{m}{n} < \frac{2^{k-1}}{k} \left( \frac{k-1}{k-2} \right)^{k-2}.$$

Observe that for $k = 3$, UC succeeds with bounded probability when $m/n < 2.666$. By Theorem 6 almost all random $(m, n, 3)$-CNF formulas have at least one model if $m/n < 2.666$.

A feature of the flow analysis outlined above is that it reveals mechanisms that suggest other, improved heuristics. For example, since increasing $z$ flows decreases $w$ flows, the following adjustment to Algorithm UC is suggested: if there are no unit clauses, choose a literal $l$ randomly from $\psi$, then compare the number of occurrences of $l$ with the number of occurrences of $\neg l$ and choose the literal that occurs most often (this tends to increase $z$ flows at the expense of $w$ flows). This isn’t quite good enough, however, since $C_i(j)$ clauses are approximately twice as influential as $C_{i+1}(j)$ clauses. Roughly, this is because one clause is accounted for in $z_i(j)$ for every two clauses in $z_{i+1}(j)$. Therefore, it is better to compare weights of literals where the weight of a literal is given by:

$$\omega(l) = \sum_{c \in \psi : l \in c} 2^{-|c|}.$$

An analysis of such a heuristic is not known to us but Algorithm UCL in Figure 7.3 is analyzable for 3-CNF formulas and uses a heuristic that is fairly close to that above.

Algorithm UCL is superior to UC in the case of $k = 3$ as the following shows:

**Theorem 60.** ([30]) Algorithm UCL determines that a given random $(m, n, 3)$-CNF formula has a model with probability bounded from below by a constant if $m/n < 2.9$. 

\[\square\]
Algorithm 25.

**UCL** (ψ)

**Input:** a set of sets CNF formula ψ.

**Output:** either “give up” or a model for ψ.

Set $M \leftarrow \emptyset$.
Set $j \leftarrow 0$.
Set $L \leftarrow \{x, \neg x : \exists c \in \psi \text{ such that either } x \in c \text{ or } \neg x \in c\}$.

Repeat the following until some statement returns a value:

If $C_1(j) \neq \emptyset$ then randomly choose literal $x$ from $C_1(j)$.
Otherwise,

Choose literal $y$ randomly from $L$.
If \# occurrences of $\neg y$ in $C_3(j) >$ \# occurrences of $y$ in $C_3(j)$
  Set $x \leftarrow \neg y$.
Otherwise,
  Set $x \leftarrow y$.
Set $L \leftarrow L \setminus \{x, \neg x\}$.
Set $\psi \leftarrow \{c - \{\neg x\} : c \in \psi, x \notin c\}$.
If $x$ is a positive literal then Set $M \leftarrow M \cup \{x\}$.
Set $j \leftarrow j + 1$.
If $\psi = \emptyset$ then Output $M$.
If there are two complementary unit clauses in $\psi$, Output “give up”.

Figure 7.3: A heuristic which chooses variables randomly but assigns values based on the difference in the number of occurrences of complemented and uncomplemented literals.

An improved analysis and improved bound to $m/n < 3.001$ is given in [1].

The reader may be curious about why the number of occurrences of literals in $C_2(j)$ was not taken into account in Algorithm **UCL**. Flow analysis tells us that it is unnecessary. Suppose that, in the case $C_1(j) = \emptyset$, the $j + 1$st literal is chosen on the number of times it occurs in both $C_3(j)$ and $C_2(j)$. Assume the most optimistic case: the literal appears in more clauses of both $C_3(j)$ and $C_2(j)$ than its complement (then the flow into $C_1(j + 1)$ is minimized since the number of two and three literal clauses removed due to the $j + 1$st chosen literal is maximized). Let $E\{w_1^*(j)\}$ denote the new average flow of clauses into $C_1(j)$. Then

$$E\{w_1^*(j)\} = E\{w_1(j)\} - h_1(j)(1 - E\{w_1^*(j)\})$$
where $h_1(j)$ is the extra number of clauses removed from the flow into $C_1(j)$ when the chosen literal is not a unit clause and $1 - E\{w_1^*(j)\}$ is the probability (to within $O(\frac{1}{n})$) that the chosen literal is not a unit clause. Therefore,

$$E\{w_1^*(j)\} = \frac{E\{w_1(j)\} - h_1(j)}{1 - h_1(j)}.$$ 

Thus $E\{w_1^*(j)\} < 1$ is equivalent to $E\{w_1(j)\} < 1$ and no benefit is gained by considering the number of occurrences of the chosen literal in $C_2(j)$.

There is another improved heuristic suggested by flow analysis. If “pumping” clauses at the bottom level by means of the unit clause rule is effective, putting “pumps” at all levels should be more effective. This amounts to adopting the following strategy for literal selection which is a generalization of the unit clause rule called the smallest clause rule: choose a literal from a clause of smallest size. Thus, if there is at least one unit clause, choose from one of them; otherwise, if there is at least one 2-literal clause, choose a literal from a 2-literal clauses; otherwise, if there is at least one 3-literal clause, choose a literal from a 3-literal clause, and so on. This is Algorithm SCA, Figure 7.1 with $s = k$. We call this Algorithm GUC.

The effectiveness of “pumping” at all levels is revealed by a flow analysis applied to Algorithm GUC. The results are

**Theorem 61.** ([31]) Algorithm GUC determines that a random $(m, n, k)$-CNF formula has a model with probability bounded from below by a constant when

$$\frac{m}{n} < 3.09 \frac{2^{k-2}}{k+1} \left(\frac{k-1}{k-2}\right)^{k-2} \text{ and } 4 \leq k \leq 40.$$ 

**Theorem 62.** ([31]) Algorithm GUC determines a random $(m, n, k)$-CNF formula has a model with probability tending to 1 when

$$\frac{m}{n} < 1.845 \frac{2^{k-2}}{k+1} \left(\frac{k-1}{k-2}\right)^{k-2} - 1 \text{ and } 4 \leq k \leq 40.$$ 

But it is not necessary to “pump” at all levels to get this kind of result. Consider Algorithm SCA with $s = 2$ and call it Algorithm SC. The following analysis sketch of Algorithm SC is based on results by Chvátal and Reed [33]. Let $p_j$ denote the probability that a fixed input clause shrinks to two literals after exactly $j$ iterations of Algorithm SC. Then $p_j m$ is the average flow into $C_2(j)$. The following simple expression for $p_j$ can be obtained straightforwardly.

$$p_j = \frac{(\frac{j-1}{k-j})(\frac{n-j}{2})}{\binom{n}{k}} \frac{1}{2^{k-2}}$$
This can be bounded from above by setting \( j \) to the value that maximizes \( p_j \) (the maximum occurs when the ratio \( p_{j+1}/p_j \) crosses the value 1). We are interested in conditions that imply the bound for \( p_j \) is less than \( 1/m \) since that gives an average flow into \( C_2(j) \) that is less than 1. Straightforward calculation reveals, for all \( j \),

\[
p_j < \frac{1 - \epsilon}{m}
\]

for any fixed \( \epsilon > 0 \) such that

\[
\frac{m}{n} < \left( \frac{1 - \epsilon}{1 + \epsilon} \right) \frac{2^k}{8k} \left( \frac{k-1}{k-3} \right)^{k-3} \frac{k-1}{k-2}.
\]

This yields the following

**Theorem 63.** Algorithm SC determines a random \((n, m, k)\)-CNF expression, \( k \geq 3 \), has a model with probability tending to 1 when

\[
\frac{m}{n} < \frac{2^k}{8k} \left( \frac{k-1}{k-3} \right)^{k-3} \frac{k-1}{k-2}.
\]

The difference between the result of Theorem 63 and that of Theorem 62 is due to improved analysis facilitated by working with an easier algorithm.

By adding a limited amount of backtracking to GUC, Frieze and Suen produced an algorithm, called GUCB, for 3-CNF formulas that finds a model, with probability tending to 1, when \( m/n < 3.003 \). Probabilistic analysis of a backtracking algorithm can be difficult due to statistical dependences which arise because information is revealed upon a backtrack. However, Frieze and Suen showed it is possible to carefully manage a limited amount of backtracking so that this does not happen. The following explains how the backtracking in GUCB is accomplished. The initial operation of GUCB is the same as that of GUC. Suppose GUCB has successfully completed \( t \) iterations and has chosen the sequence of literals \( \{x_{\pi_1}, x_{\pi_2}, \ldots, x_{\pi_t}\} \) and assigned them the value 1. Suppose \( |C_1(t')| = 0 \), \( t' < t \) and \( |C_1(j)| > 0 \) for all \( t' < j \leq t \) so the last iteration that saw no unit clauses was iteration \( t' \). Suppose further that choosing and assigning value 1 to literal \( x_{\pi_{t+1}} \) results in the appearance of complementary unit clauses in \( C_1(t+1) \). Then GUCB backtracks by setting \( x_{\pi_{t'}} = x_{\pi_{t'+1}} = \ldots = x_{\pi_t} = x_{\pi_{t+1}} = 0 \). Clauses now satisfied and literals now falsified are removed, removed clauses now not satisfied and removed literals now not falsified and in non-satisfied clauses are reinstated. After backtracking, GUCB continues choosing and assigning literals the value 1 as before. The algorithm succeeds if all clauses are eliminated. The algorithm fails in two ways: 1) the resetting of literals from 1 to 0 results in a null clause; 2) a complementary pair of unit clauses is encountered before \( |C_1| \) has become 0 after a backtrack. The analysis of GUCB is possible because the effect on the distribution of \( C_i(j) \) is slight and because, with probability tending to 1, GUCB backtracks at most \( \ln^5(n) \) times when \( m/n < 3.003 \).
Frieze and Suen also applied limited backtracking to Algorithm SC and call the resulting algorithm SCB. The result is the following:

**Theorem 64.** For \( k \geq 4 \), SCB succeeds, with probability tending to 1, when \( m/n < \eta_k \frac{2^k}{k} \) where \( \eta_4 \approx 1.3836, \eta_5 \approx 1.504, \) and \( \lim_{k \to \infty} \eta_k \approx 1.817 \). \( \square \)

This may be compared to the above result for GUC which, at \( k = 40 \), has a performance bound of \( m/n < (1.2376)^{2^{40}/40} \) (this is with probability tending to 1), and the above result for SC which has a performance bound of \( \lim_{k \to \infty} m/n < (0.9236)2^k/k \) (also with probability tending to 1).

There is a limit to the probabilistic performance of straight line, myopic algorithms. That is, there is an optimal policy for choosing a variable and value on any iteration. For \( k = 3 \), the optimal policy is given by the following:

**Optimal Myopic Literal Selection Policy**

If \( |C_1(j)| = 0 \) choose 2-literal clause \((v \lor w)\) or \((v \lor w)\) or \((v \lor w)\) at random. Select \( v, w, \) or \( v, w \) at random and temporarily assign the value to the chosen variable which satisfies its clause. Then fix the assignment according to the following. Let \( M_3(j) \) be the number of 3-literal clauses satisfied minus the number of literals falsified in 3-literal clauses. Let \( M_2(j) \) be the number of 2-literal clauses satisfied minus the number of literals falsified in 2-literal clauses. Let \( d_3(j) = |C_3(j)|(n-j) \). Let \( d_2(j) = |C_2(j)|(n-j) \). Define \( \Theta(j) = (1.5 \cdot d_3(j) - 2 \cdot d_2(j) + \gamma)/(1 - d_2(j)) \) where \( \gamma \) is some constant. Reverse the assignment if \( \Theta(j) > M_3(j)/M_2(j) \).

**Theorem 65.** ([4]) Algorithm SCA with \( s = 1 \) and using the optimal myopic literal selection policy instead of choosing a literal randomly from \( L_2 \) succeeds in finding a model, given random \((m,n,3)\)-CNF formulas as input, with probability bounded from below by a constant when \( m/n < 3.22 \). This is the best possible literal selection policy for SCA on random \((m,n,3)\)-CNF formulas. \( \square \)

According to the optimal myopic literal selection policy, if there is a literal that will maximize both the number of 3-literal clauses and 2-literal clauses satisfied, one will be selected. Otherwise, a literal is selected as a compromise between reducing the number of 2-literal clauses immediately and increasing the chance of reducing the number of 2-literal clauses at some point in the future.

A better result is due to a literal selection policy which may actually look at 2 literals on an iteration instead of just 1. The result is:

**Theorem 66.** There exists a myopic straight line algorithm which succeeds in finding a model, given random \((m,n,3)\)-CNF formulas as input, with
7.2 NON-MYOPIC ALGORITHMS FOR SATISFIABLE FORMULAS

Algorithm 26.

\textbf{GPL}(\psi)

\textbf{Input}: a set of sets CNF formula $\psi$.
\textbf{Output}: returns either “give up” or a model for $\psi$.

Set $M \leftarrow \emptyset$.
Repeat the following:
- If $\psi = \emptyset$ then Output $M$.
- If $\emptyset \in \psi$ then Output “give up”.
- Let $l$ be a literal occurring at least as frequently as any other in $\psi$.
- If $l$ is a positive literal then Set $M \leftarrow M \cup \{l\}$.
- Set $\psi \leftarrow \{c - \{\neg l\} : c \in \psi, l \notin c\}$
- Repeat the following until no unit clauses exist in $\psi$:
  - Let $\{l\}$ be a unit clause in $\psi$.
  - If $l$ is a positive literal then Set $M \leftarrow M \cup \{l\}$.
  - Set $\psi := \{c - \{\neg l\} : c \in \psi, l \notin c\}$.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{algorithm26.png}
\caption{A non-myopic straight line algorithm.}
\end{figure}

\textit{probability bounded from below by a constant which $m/n < 3.26$. This is the best performance possible by any myopic straight line algorithm.}

7.2 Non-myopic algorithms for satisfiable formulas

In light of experimental evidence suggesting $r_3$ is approximately 4.25 and the results of Theorem 66 there appears to be an unfortunate gap between what is achievable by myopic algorithms and what we would hope is possible from straight line algorithms. However, the tools of Section 2.4.7 may still be applied to the analysis of non-myopic straight line algorithms, under some restrictions. For example, flow analysis is still possible if algorithmic operations include [95]:

1. Select uniformly at random a pure literal, assign it the value 1 and remove all satisfied clauses.
2. Select uniformly at random a literal occurring exactly once in the expression and its occurrence is in a 3-literal clause, assign it the value 0, remove it from the clause it appears, and remove all clauses containing its complementary literal (these are satisfied).
Algorithm 27.

**MPS** \( \psi \)

**Input:** a set of sets CNF formula \( \psi \).

**Output:** either returns “give up” or a model for \( \psi \).

Set \( M \leftarrow \emptyset \).

Repeat the following:

- If \( \emptyset = \emptyset \) then Output \( M \).
- If \( \emptyset \in \psi \) then Output “give up”.

Repeat the following until no unit clauses exist in \( \psi \):

- Let \( \{l\} \) be a unit clause in \( \psi \).
- If \( l \) is a positive literal then \( M \leftarrow M \cup \{l\} \).
- Set \( \psi \leftarrow \{c \setminus \{\neg l\} : c \in \psi, l \notin c \} \).

Repeat the following for all unassigned literals \( l \):

- Set \( \psi_l \leftarrow \{c \setminus \{\neg l\} : c \in \psi, l \notin c \} \).
- Define \( w(l) \), the weight of literal \( l \), to be \( \prod_{c \in \psi_l} (1 - 2^{-|c|}) \).
- Choose \( l \) so that \( w(l) \) is maximum over all unassigned literals.
- If \( l \) is an uncomplemented literal then Set \( M \leftarrow M \cup \{l\} \).
- Set \( \psi \leftarrow \{c \setminus \{\neg l\} : c \in \psi, l \notin c \} \).

\( \Box \)

Figure 7.5: Another non-myopic straight line algorithm.

3. Select uniformly at random a literal occurring exactly once in the expression and its occurrence is in a 2-literal clause, assign it the value 0, remove it from the clause it appears, and remove all clauses containing its complementary literal (these are satisfied). Then apply the unit clause unit to exhaustion (until no unit clauses remain).

An example, which we call **GPL**, from [79] is shown in Figure 7.4.

**Theorem 67.** Algorithm **GPL** succeeds in finding a model given random \((m, n, 3)\)-CNF formulas as input with probability bounded from below by a constant when \( m/n < 3.42 \).

\( \Box \)

Similar algorithms have been reported to have good performance out to \( m/n < 3.52 \) [68, 80].

Other non-myopic straight line algorithms have shown even better performance on random \((m, n, k)\)-CNF inputs. For example, the algorithm of Figure 7.5 seems to do well for \( m/n < 3.6 \). The algorithm is designed to iteratively select a variable and value which maximizes the expected number of models possessed by what is left of the given expression after satisfied clauses and falsified literals are removed, assuming somehow the clauses of
the new expression are statistically independent. Of course, that is not the case. But it is conceivable that the fundamental idea of maximizing expected number of models or maximizing the probability a model exists can be enhanced to provide algorithms of greater performance. An analysis of this and other algorithms like it has not yet been reported.

7.3 Lower bounds on the satisfiability threshold

The results of the previous two sections may be used to obtain lower bounds for the random \((m, n, k)\)-CNF satisfiability threshold. But, unfortunately, the best that can be said for the \(k\)-CNF satisfiability threshold, based on algorithmic analysis, is \(r_k > c \cdot 2^k/k\), \(c\) a constant, and \(r_3 > 3.42\). Threshold results could be obtained without relying on any algorithm by means of the second moment method were it not for the fact that the variance of the number of satisfying assignments is too large for a sufficiently tight bound. However, the second moment method has been applied successfully to a different problem and the results for that problem imply the following result:

**Theorem 68.** [3] For random \((m, n, k)\)-CNF formulas

\[
r_k \geq 2^k \ln(2) - (k + 1) \ln 2 - 1 - \delta_k
\]

where \(\delta_k \to 0\) for increasing \(k\).

These results improve and are built upon those of [2]. We sketch the motivation leading to these results in this section.

The crucial question is which random variable \(X\) can the second moment method be applied to. If \(X\) is the number of models, the bound provided by the second moment method is inadequate because for many random \((m, n, k)\)-CNF formulas the number of models is far from the mean and this causes the variance of \(X\) to be too high. More specifically, \(Pr(z|w)\) (see Lemma 4, Page 49) is too high. This can be seen as follows. Let \(z\) and \(w\) be assignments to variables in random \((m, n, k)\)-CNF formula \(\psi\) which agree in \(\alpha n\) variables. Then

\[
Pr(z \text{ is a model for } \psi | w \text{ is a model for } \psi) = \left(1 - \frac{1 - \alpha^k}{2^k - 1}\right)^m.
\]

Therefore, to apply the second moment method, we will need to control

\[
\sum_{0 \leq \alpha \leq 1} \binom{n}{\alpha n} \left(1 - \frac{1 - \alpha^k}{2^k - 1}\right)^m.
\]

Observe the maximum of this sum does not occur at \(\alpha = 1/2\) and is not \(o(\mu)\) as needed (recall \(\mu = 2^n (1 - 2^{-k})^m\)).
The fix is to use a bound for a closely related problem for which the maximum of the sum occurs at $\alpha = 1/2$. The problem is called Not-All-Equal $k$-CNF (which we denote by NAE$_k$). Given CNF formula $\psi$, NAE$_k$ is the problem of finding a model for $\psi$ such that every clause has at least one literal falsified as well as at least one satisfied. Since a NAE$_k$ model for $\psi$ is also a model for $\psi$

$$Pr(\exists \text{ model for } \psi) > Pr(\exists \text{ NAE}_k \text{ model for } \psi).$$

But

$$Pr(z \text{ is a NAE}_k \text{ model for } \psi | w \text{ is a NAE}_k \text{ model for } \psi) = \left(1 - \frac{1 - \alpha^k - (1 - \alpha)^k}{2^{k-1} - 1}\right)^m$$

and

$$\sum_{0 \leq \alpha \leq 1} \binom{n}{\alpha n} \left(1 - \frac{1 - \alpha^k - (1 - \alpha)^k}{2^{k-1} - 1}\right)^m \approx \frac{2^n(1 - 2^{-k+1})^m}{\sqrt{n}} = o(\mu)$$

because all the significant contributions to the sum occur near the maximum of both terms in the summand which is at $\alpha = 1/2$ and $\mu$ for NAE$_k$ on random $(m,n,k)$-CNF formulas is $2^n(1 - 2^{-k+1})^m$.

### 7.4 Verifying unsatisfiability: resolution

The algorithms discussed above are useless for proving that no satisfying truth assignment exists in the case of a given unsatisfiable formula. Resolution is one popular method capable of doing this and can be extremely effective when certain clause patterns are present, with high probability. Recall from Page 68 that the resolvent of two clauses $c_1$ and $c_2$ on pivot variable $v$ is given by

$$R_{c_1}^{c_2} = \{l : l \in c_1 \cup c_2 \setminus \{v, \neg v\}\}.$$

If resolvents are generated until one is the empty set, then the original formula is unsatisfiable.

There are many algorithms which use the generation of resolvents to determine whether a given formula is satisfiable. They differ in restrictions that are applied to help reduce the total number of resolvents generated. The restricted resolution algorithm of Figure 7.6, called RW for Restricted Width, has polynomial time complexity because resolvents are restricted to be no greater than $k$ in width.

It can be shown that Algorithm RW is effective on random $(m,n,k)$-CNF formulas up to a point. Particular patterns of clauses that RW can
Algorithm 28.

\[ \text{RW}(\psi) \]

**Input:** a set of sets CNF formula \( \psi \), \( \emptyset \notin \psi \).

**Output:** either “unsatisfiable” or “gives up”.

Repeat the following:

If there are clauses \( c_1, c_2 \in \psi \) s.t. \( \mathcal{R}_{c_1}^{c_2} \notin \psi \) and \( |\mathcal{R}_{c_1}^{c_2}| \leq k \) then:

- If \( \emptyset = \mathcal{R}_{c_1}^{c_2} \) then Output “unsatisfiable”.

Set \( \psi \leftarrow \psi \cup \{ \mathcal{R}_{c_2} \} \).

Otherwise, break out of the loop.

Output “cannot determine whether \( \psi \) is unsatisfiable”.

Figure 7.6: A resolution algorithm restricted to resolvents of maximum width \( k \).

exploit to produce a null resolvent are developed as follows. Let \( S_{i,j} \) denote the family of clause sets defined as follows over a set of variables \( X = \{ x_{11}, x_{21}, \ldots, x_{12}, x_{22}, \ldots \} \):

\[
S_{i,j} = \begin{cases} 
\{ \{x_{i1}\}\{\bar{x}_{i1}\}\} & \text{if } j = 1; \\
\{P \cup \{x_{ij}\} : P \in S_{2i-1,j-1}\} \cup \{P \cup \{\bar{x}_{ij}\} : P \in S_{2i,j-1}\} & \text{if } j > 1.
\end{cases}
\]

Thus,

\[
S_{1,1} = \{\{x_{11}\},\{\bar{x}_{11}\}\} \\
S_{1,2} = \{\{x_{12},x_{11}\},\{x_{12},\bar{x}_{11}\},\{\bar{x}_{12},x_{21}\},\{\bar{x}_{12},\bar{x}_{21}\}\} \\
S_{1,3} = \{\{x_{13},x_{12},x_{11}\},\{x_{13},x_{12},\bar{x}_{11}\},\{x_{13},\bar{x}_{12},x_{21}\},\{x_{13},\bar{x}_{12},\bar{x}_{21}\},\{\bar{x}_{13},x_{22},x_{31}\},\{\bar{x}_{13},x_{22},\bar{x}_{31}\},\{\bar{x}_{13},\bar{x}_{22},x_{41}\},\{\bar{x}_{13},\bar{x}_{22},\bar{x}_{41}\}\}
\]

Over variable set \( X \cup \{y_0, y_1, y_2, \ldots \} \) define

\[
\mathcal{R}_{k,r} = \begin{cases} 
\{P \cup \{y_0, \bar{y}_1\} : P \in S_{1,k-2}\} \cup \\
\cup_{i=1}^{r-2} \{P \cup \{y_i, \bar{y}_{i+1}\} : P \in S_{i+1,k-2}\} \cup \\
\{P \cup \{y_{r-1}, y_0\} : P \in S_{r,k-2}\} \cup \\
\{P \cup \{\bar{y}_0, \bar{y}_r\} : P \in S_{r+1,k-2}\} \cup \\
\cup_{i=r+1}^{2r-2} \{P \cup \{y_{i-1}, \bar{y}_i\} : P \in S_{i+1,k-2}\} \cup \\
\{P \cup \{y_{2r-2}, \bar{y}_0\} : P \in S_{2r,k-2}\}
\end{cases}
\]
For example,

\[ \mathcal{R}_{3,3} = \left\{ \{y_0, \bar{y}_1, x_{11}\}, \{y_0, \bar{y}_1, \bar{x}_{11}\}, \{y_1, \bar{y}_2, x_{21}\}, \{y_1, \bar{y}_2, \bar{x}_{21}\}, \{y_2, y_0, x_{31}\}, \{y_2, y_0, \bar{x}_{31}\}, \{\bar{y}_0, \bar{y}_3, x_{41}\}, \{\bar{y}_0, \bar{y}_3, \bar{x}_{41}\}, \{y_3, \bar{y}_4, x_{51}\}, \{y_3, \bar{y}_4, \bar{x}_{51}\}, \{y_4, \bar{y}_0, x_{61}\}, \{y_4, \bar{y}_0, \bar{x}_{61}\} \right\} \]

Algorithm RW, applied to any \(k\)-CNF formula that contains a subset of clauses which can be mapped to \(\mathcal{R}_{k,r}\) (\(r \geq 2\)) after renaming of variables, always results in a certificate of unsatisfiability for that formula. Therefore, if the probability that a random \((m,n,k)\)-CNF formula has such a subset is high, then RW will correctly output “unsatisfiable” with high probability.

It is straightforward to obtain the probability that there is a subset of clauses that maps to \(\mathcal{R}_{k,r}\) for some \(r\). Expression \(\mathcal{R}_{k,r}\) contains \(r^{2k-1}\) clauses and \(r^{2k-1} - 1\) distinct variables, and is minimally unsatisfiable: that is, it is unsatisfiable but removal of any clause produces a satisfiable set. This property is important to the analysis since a minimally unsatisfiable expression can be padded with additional clauses to get other unsatisfiable clause patterns with a much higher ratio of clauses to variables, but the probability that such a padded pattern exists is not greater than the probability that one of its base minimally unsatisfiable sets exist.

The existence probability crosses from tending to 0 to tending to 1 at a particular ratio of \(m/n\). This can be estimated by finding conditions that set the average number of minimally unsatisfiable sets to \(\infty\) in the limit. The probability that a particular subset of \(r^{2k-1}\) clauses matches the pattern \(\mathcal{R}_{k,r}\) for a particular choice of variables is \((r^{2k-1})! / (r^k(m/(2^k \binom{n}{k}))^{r^{2k-1}})\). There are \(\binom{n}{r^{2k-1}}\) ways to choose the variables and each of \(2^{r^{2k-1} - 1}\) ways to complement the chosen variables presents a pattern that RW can handle. Furthermore, any of the \((r^{2k-1} - 1)\)! permutations of those variables also presents a solvable pattern. Finally, there are \(\binom{m}{r^{2k-1}}\) ways to choose clauses. Therefore, the average number of \(\mathcal{R}_{k,r}\) patterns in \(\psi\) is the product of the above terms which is about \((k!m/(2^{k-1}n^{(k-1)+1/r^{2k-1}}))^{r^{2k-1}}\). This tends to \(\infty\) when \(m/n^{(k-1)} > (n\omega(n))^{1/r^{2k-1}}\) where \(\omega(n)\) is any slowly growing function of \(n\). Setting \(r = \lceil \ln^{1+\epsilon}(n) \rceil\), where \(\epsilon\) is a small constant, is sufficient to give the following result (in this case patterns are small enough for a second moment analysis to carry through).

**Theorem 69.** Algorithm RW succeeds, with probability tending to 1, on random \((m,n,k)\)-CNF formulas if \(m/n > n^{k-2}2^{k-1}/k!\). \(\square\)

This analysis shows the importance of the ratio of the number of variables to the number of clauses in a minimally unsatisfiable expression. A higher ratio means a lower cutoff for \(m/n^{k-1}\). We remark that the first results of the above nature appearing in print, as far as we know, are due to Xudong.
7.4 VERIFYING UNSATISFIABILITY: RESOLUTION

Fu [58]. Fu used a minimally unsatisfiable pattern which he called a Flower to achieve the above result. But Flowers have $2^{k-1}(r+k-1)$ clauses and $2^{k-1}r+k-1$ variables. Since the difference between clauses and variables in $\mathcal{R}_{k,r}$ is 1, whereas the difference is $(2^{k-1} - 1)(k-1)$ in the case of Flowers, we chose to present $\mathcal{R}_{k,r}$ instead.

Improving on the above result is hard. An obvious initial attempt consists of finding minimally unsatisfiable clause patterns where the ratio of variables to clauses is higher than 1 and hopefully close to $k$ (recall, a ratio of 1 is roughly what is achieved above). But, the following result, which can be proved by induction on the number of variables in a minimally unsatisfiable set and first appeared in [8] (see also [14] and [83]), is discouraging:

**Theorem 70.** If $\psi$ is a minimally unsatisfiable CNF formula with $m$ clauses, then $\psi$ has less than $m$ variables.

Notice that Theorem 70 does not say anything about the width of a clause: it could be fixed at any $k$ or different for each clause and the result still holds.

Theorems 69 and 70 do not say that resolution fails to prove unsatisfiability in polynomial time with high probability when $m/n^{(k-1)} \to 0$. These theorems suggest only that looking for clause patterns which cause an algorithm such as RW to succeed often for such values of $m/n$ is likely not to bear fruit. However, a deeper and illuminating argument shows that resolution can’t be successful, probabilistically on random $(m,n,k)$-CNF formulas, for a wide range of $m/n$ values.

A number of papers consider the probabilistic performance of resolution for random unsatisfiable $(n,m,k)$-CNF formulas (for example, [14, 15, 32, 58]). The following two results are noteworthy.

**Theorem 71.** For $k > 3$, an unsatisfiable random $(m,n,k)$-CNF formula has only exponential size resolution proofs, with probability tending to 1, if $m/n^{(k+2)/4-\epsilon} < 1$ for any $\epsilon > 0$. A random $(m,n,3)$-CNF formula has only exponential size resolution proofs, with probability tending to 1, if $m/n^{6/5-\epsilon} < 1$, for any $\epsilon > 0$.

**Theorem 72.** There exists a DPLL variant which verifies the unsatisfiability of a random $(m,n,k)$-CNF formula in polynomial time with probability tending to 1 when $m/n > (n/\log(n))^{k-2}$.

The fact that no resolution result better than that of Theorem 72 has been found is surprising. It seems hard to imagine that RW is so close to what is the best possible for resolution. But a result of [15] shows that at least some natural DPLL variants cannot do much better. These pessimistic results motivate the results of the next section.
7.5 Verifying unsatisfiability: a spectral analysis

The pessimistic results obtained for verifying unsatisfiability via resolution algorithms has motivated the search for alternatives. In this section a simple one is presented and an analysis outlined. A decision is made based on counting variables which must be assigned value 1 or 0 if a model exists.

The idea is as follows. Given a CNF formula $\psi$, throw out all except positive and negative clauses. Let $n_+$ be the minimum number of variables that must be assigned value 1 to satisfy the positive clauses and let $n_-$ be the number of variables that must be assigned value 0 to satisfy the negative clauses. If $n_+ + n_- > n$ then some variable must be assigned both 1 and 0 if $\psi$ has a model. But this is impossible, so $\psi$ cannot have a model. Hence, this simple counting argument provides a means to verify unsatisfiability.

The only problem is that finding $n_+$ and $n_-$ is NP-complete. However, it may be sufficient merely to approximate $n_+$ and $n_-$ closely enough. The following explains how to do this based on work reported in [65]. We start with this theorem which is obvious from the above discussion:

**Theorem 73.** If $k$-CNF formula $\psi$ has a model then there is a subset $V'$ of $n/2$ variables such that either $\psi$ has no negative clause or no positive clause taken strictly from $V'$.

For purposes of discussion fix $k = 4$. Construct two graphs $G_+$ and $G_-$, corresponding to the positive and negative clause sets of $\psi$, respectively. Both $G_+$ and $G_-$ have $\binom{n}{2}$ vertices and each vertex of $G_+$ and $G_-$ is uniquely labeled by a pair of variables. An edge between two vertices of $G_+$ exists if and only if there is in $\psi$ a positive clause containing exactly the variables that label the two vertices. Similarly, an edge between two vertices of $G_-$ exists if and only if there is a negative clause containing exactly the variables labeling the vertices. It follows from Theorem 73 that

**Theorem 74.** If $\psi$ has a model, then either $G_+$ or $G_-$ has an independent set of size greater than $\left(\binom{n}{2}\right) \approx n^2/8$.

The following applies to both $G_+$ and $G_-$ but we proceed only for $G_+$ and state that the discussion holds for $G_-$ as well. Let $n'$ be the number of vertices in $G_+$ and define $n' \times n'$ matrix $A_{G_+}$ with columns and rows indexed on vertices of $G_+$ and such that $A_{G_+}(i,j) = 1$ if edge $\langle v_i, v_j \rangle$ exists in $G_+$ and $A_{G_+}(i,j) = -(1-p)/p$ if edge $\langle v_i, v_j \rangle$ does not exist in $G_+$ for some $0 < p < 1$ where $p$ may be a function of $n'$. Let $\lambda_1(A_{G_+})$ be the greatest eigenvalue of $A_{G_+}$ and $\alpha(G_+)$ be the size of the greatest independent set of $G_+$. Then,

**Theorem 75.**

$$\alpha(G_+) < \lambda_1(A_{G_+}) \quad \text{and} \quad \alpha(G_-) < \lambda_1(A_{G_-})$$
7.5 VERIFYING UNSATISFIABILITY: A SPECTRAL ANALYSIS

Proof. Follows directly from Theorem 13.

The above is not a probabilistic result. One could cast any 4-CNF formula $\psi$ in terms of graphs $G_+$ and $G_-$ and matrices $A_{G_+}$ and $A_{G-}$, find the maximum eigenvalues of the matrices, and if both maximum eigenvalues turn out to be less than $\left(\frac{n}{2}\right)$, where $n$ is the number of variables of $\psi$, conclude by Theorem 74 that $\psi$ is not satisfiable. If at least one maximum eigenvalue is greater than $\left(\frac{n}{2}\right)$ then nothing can be concluded. The value $p$ can be chosen freely and affects the tightness of the bound in Theorem 75.

The problem of determining the eigenvalues of $A_{G_+}$ and $A_{G_-}$ can be handled, for example, by Theorem 12, Page 60.

When dealing with random $k$-CNF formulas, $k$ even, an upper bound on the maximum eigenvalues of $A_{G_+}$ and $A_{G_-}$, depending on formula density $m/n$, can be obtained. If $m/n$ is not too small, with very high probability the bound is sufficient to prove that a given random formula is unsatisfiable. The proof of this begins with the following:

**Lemma 76.** ([59]) If upper triangular, off-diagonal elements of symmetric $n' \times n'$ matrix $A$ (the $i,j$ element of $A$ is the same as the $j,i$ element for all $i,j$) are chosen independently from a distribution bounded by constant $K$ with expectation $0$ and variance $\sigma^2$, and the diagonal elements are chosen with expectation $\nu$, then with probability greater than $1 - \frac{1}{10}$
\[
\max_{i \leq n'} \left\{ \left| \lambda_i(A) \right| \right\} = 2\sigma \sqrt{n'} + O(n'^{1/3} \log(n')).
\]

If $p$ is the probability that a particular edge exists in $G_+$ (also $G_-$) then the corresponding element in $A_{G_+}$ is $-(1 - p)/q$ and otherwise it is 1. Therefore, the expectation of each off-diagonal element of $A_{G_+}$ (also $A_{G_-}$) is $1(1 - p) - ((1 - p)/p)p = 0$ and its variance is $1^2(1 - p) + ((1 - p)/p)^2p = (1 - p)/p$. The expectation of the diagonal elements is 1. But element values are not independent and, since $p$ can be arbitrarily small, there is no bound on the distribution. However, some modifications to the proof of the above lemma support the following result:

**Lemma 77.** Assume $p = \ln^7(n')/n'$. With probability greater than $1 - (1/10)n'$, the random matrix $A$ satisfies
\[
\max_{i \leq n'} \left\{ \left| \lambda_i(A) \right| \right\} = 2(n'/\ln^{7/2}(n'))(1 + o(1)).
\]

The restriction on $p$ means the distribution is bounded. The result is not terribly sensitive to the particular choice of $p$ and the stated value is sufficient for the purposes of this exposition. The value of $p$ relates the number of clauses of $\psi$ with the number of variables. Specifically, $p$ is
approximately the number of edges in $G_+$ (alternatively $G_-$) divided by the number possible, or $(m/16)/(n'{}^2)$, where the factor of $1/16$ is due to $G_+$ and $G_-$ representing, with high probability, $1/16$ of the $m$ input clauses in the case that $k = 4$.

**Theorem 78.** (adapted from [65]) Let $n' = \binom{n}{2}$, $m = 16n'(\ln^7(n')/2)$, and $p = m/(16\binom{n'}{2}) = \ln^7(n')/(n' - 1)$. With high probability,

$$\lambda_1(A_{G_+}) \leq 2(n'/\ln^7(n'))(1 + o(1)).$$

The same bound holds for $\lambda_1(A_{G_-})$. $\square$

Theorems 75 and 78 give an adequate bound on the size of the independent sets of $G_+$ and $G_-$ for a wide range of formula densities. These results are used as follows. Suppose $m = 4n^2\ln^7(n^2) = 4(2^7)n^2\ln^7(n)$. By Theorem 78 $\lambda_1(A_{G_+}) \leq n^2/8$ and $\lambda_1(A_{G_-}) \leq n^2/8$ with high probability and by Theorem 75 $\alpha(G_+) < n^2/8$ and $\alpha(G_-) < n^2/8$ with high probability. Hence, by Theorem 74, with high probability, $\psi$ has no model if $m > 4(2^7)n^2\ln^7(n)$ or $m/n > cn/\ln^7(n)$, where $c$ is a constant, when $k = 4$. The general theorem is the following:

**Theorem 79.** Random $(n, m, k)$-CNF formulas can be certified as unsatisfiable in polynomial time with high probability if $m > nk^{k/2 + o(1)}$, $k \geq 4$, and if $m > n^{3/2 + \epsilon}$, $\epsilon$ an arbitrarily small constant, $k = 3$. $\square$

### 7.6 Polynomial time solvable classes.

Formulas that are members of certain polynomial time solvable classes, such as those defined in Chapter 4, are not generated frequently enough to assist in determining whether random $(m, n, k)$-CNF formulas have models. This is unlike the case for random $(m, n, p)$-CNF formulas. We illustrate with a few examples. In what follows we use $\psi$ to denote a random $(m, n, k)$-CNF formula.

We consider the class of Horn formulas (Section 4.2) first. The probability that a randomly generated clause is Horn is $(k + 1)/2^k$ so the probability that $\psi$ is Horn is $(k + 1)/2^m$. This tends to 0 as $m$ tends to $\infty$ for any fixed $k$. For a hidden Horn formula (Section 4.3), regardless of switch set, there are only $k + 1$ out of $2^k$ ways ($k$ ways to place a positive literal and 1 way to place only negative literals) that a random clause can become Horn. Therefore, the expected number of successful switch sets is $2^m((k + 1)/2^k)m$. This tends to 0 for increasing $m$ and $n$ if $m/n > 1/(k - \log_2(k + 1))$. Therefore, by Markov’s inequality, $\psi$ is not hidden Horn, with probability tending to 1, if $m/n > 1/(k - \log_2(k + 1))$. Even when $k = 3$, this is $m/n > 1$. This bound can be improved considerably by finding complex structures
that imply a formula cannot be hidden Horn. Such a structure is presented next.

The following result for q-Horn formulas (Section 4.5) is taken from [54]. For \( p = \lfloor \ln(n) \rfloor \geq 4 \), call a set of \( p \) clauses a \( c \)-cycle if all but two literals can be removed from each of \( p - 2 \) clauses, all but three literals can be removed from two clauses, the variables can be renamed, and the clauses can be reordered in the following sequence

\[
(v_1 \lor \neg v_2) \land (v_2 \lor \neg v_3) \land \ldots \land (v_i \lor \neg v_{i+1} \lor v_{p+1}) \land \\
\ldots \land (v_j \lor \neg v_{j+1} \lor v_{p+1}) \land \ldots \land (v_p \lor \neg v_1) \tag{7.2}
\]

where \( v_i \neq v_j \) if \( i \neq j \). We use the term “cycle” to signify the existence of cyclic paths through clauses which share a variable: that is, by jumping from one clause to another clause only if the two clauses share a variable, one may eventually return to the starting clause. Given a c-cycle \( C \subset \psi \), if no two literals removed from \( C \) are the same or complementary, then \( C \) is called a q-blocked c-cycle.

If \( \psi \) has a q-blocked c-cycle then it is not q-Horn. Let a q-blocked c-cycle in \( \psi \) be represented as above. Develop satisfiability index inequalities (2.2) for \( \psi \). After rearranging terms in each, a subset of these inequalities is as follows

\[
\alpha_1 \leq Z - 1 + \alpha_2 - \ldots \\
\ldots \\
\alpha_i \leq Z - 1 + \alpha_{i+1} - \alpha_{p+1} - \ldots \\
\ldots \\
\alpha_j \leq Z - 1 + \alpha_{j+1} - (1 - \alpha_{p+1}) - \ldots \\
\ldots \\
\alpha_p \leq Z - 1 + \alpha_1 - \ldots . \tag{7.3}
\]

From inequalities (7.3) to (7.4) we deduce

\[
\alpha_1 \leq pZ - p + \alpha_1 - (1 - \alpha_{p+1} + \alpha_{p+1}) - \ldots \\
\text{or} \\
0 \leq pZ - p - 1 + \ldots
\]

where all the terms in \( \ldots \) are non-positive. Thus, all solutions to (7.3) through (7.4) require \( Z > (p+1)/p = 1 + 1/p = 1 + 1/\lfloor \ln^2 n \rfloor > 1 + 1/n^\beta \) for any fixed \( \beta < 1 \). This violates the requirement (Theorem 33) that \( Z \leq 1 \) in order for \( \psi \) to be q-Horn.

The expected number of q-blocked c-cycles can be found and the second moment method applied to give the following result.

**Theorem 80.** A random \((m,n,k)\)-CNF formula is not q-Horn, with probability tending to 1, if \( m/n > 4/(k^2 - k) \). \(\Box\)
A similar analysis yields the same results for hidden Horn, SLUR, CC-balanced, or extended Horn classes. The critical substructure which causes $\psi$ not to be SLUR is called a criss-cross loop. An example is shown in Figure 7.7 as a propositional connection graph. Just one criss-cross loop in $\psi$ prevents it from being SLUR. Comparing Figure 7.7 and expression (7.2) we see that both the SLUR and q-Horn classes are “vulnerable” to certain types of “cyclic” structures. Most other polynomial time solvable classes are similarly vulnerable to cyclic structures of various kinds. But random $k$-CNF formulas are constructed without consideration of such cyclic structures: at some point as $m/n$ is increased cycles begin to appear in $\psi$ in abundance and when this happens cycles that prevent membership in one of the above named polynomial time solvable classes also show up. Cycles appear in abundance when $m/n > 1/O(k^2)$. It follows that a random $k$-CNF formula is not a member of one of the above named polynomial time solvable classes when $m/n > 1/O(k^2)$. By contrast, simple polynomial time procedures will solve a random $k$-CNF formula with high probability when $m/n < 2^k/O(k)$ (Section ??). The disappointing conclusion that many polynomial time solvable classes are relatively rare among random $k$-CNF formulas because they are vulnerable to cyclic structures is given added perspective by considering the class of matched formulas.

A $k$-CNF formula is a matched formula (see Page 124) if there is a total matching in its variable-clause matching bipartite graph: a property that is not affected by cyclic structures as above. A probabilistic analysis tells us that a random $k$-CNF generator produces “many more” matched formulas than SLUR or q-Horn formulas. Let $Q \subset \psi$ be any subset of clauses of $\psi$. Define the neighborhood of $Q$, denoted $V(Q)$, to be the set of variables that occur in $Q$. The variable-clause matching graph (defined in Section 2.3.5) of $\psi$ is a bipartite graph with vertex sets $C_\psi$ and $V_\psi$ and edge set $E_\psi$, such that there is a $c \in C_\psi$ for each clause of $\psi$, there is a $v \in V_\psi$ for each variable in $\psi$, and there is an edge $\langle c, v \rangle \in E$ if and only if variable $v$ appears as a

---

Figure 7.7: A “criss-cross loop” of $t = 3p + 2$ clauses represented as a propositional connection graph. Only “cycle literals” are shown in the nodes; “padding literals,” required for $k \geq 3$ and different from cycle literals, are present but are not shown.
positive or negative literal of clause \( c \) in \( \psi \). Thus \( V(Q) \) is also the set of vertices in \( V_\psi \) adjacent to the vertices corresponding to \( Q \).

**Definition 81.** Define the **deficiency** of \( Q \), denoted \( \delta(Q) \), as \( \delta(Q) = |Q| - |V(Q)| \), that is, the excess of clauses over distinct variables in those clauses. A subset \( Q \subseteq C_\psi \) is said to be deficient if \( \delta(Q) > 0 \).

The following theorem is well known.

**Theorem 82.** (Hall’s Theorem [70])

Given a bipartite graph with vertex sets \( V_\psi \) and \( C_\psi \), a matching that includes every vertex of \( C_\psi \) exists if and only if no subset of \( C_\psi \) is deficient.

**Theorem 83.** Random \((n, m, k)\)-CNF formulas are matched formulas with probability tending to 1 if \( m/n < r(k) \) where \( r(k) \) is given by the following table [54].

<table>
<thead>
<tr>
<th>( k )</th>
<th>( r(k) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>.64</td>
</tr>
<tr>
<td>4</td>
<td>.84</td>
</tr>
<tr>
<td>5</td>
<td>.92</td>
</tr>
<tr>
<td>6</td>
<td>.96</td>
</tr>
<tr>
<td>7</td>
<td>.98</td>
</tr>
<tr>
<td>8</td>
<td>.990</td>
</tr>
<tr>
<td>9</td>
<td>.995</td>
</tr>
<tr>
<td>10</td>
<td>.997</td>
</tr>
</tbody>
</table>

Theorem 83 may be proved by finding a lower bound on the probability that a corresponding random variable-clause matching graph has a total matching. By Theorem 82 it is sufficient to prove an upper bound on the probability that there exists a deficient subset of clause vertices, then show that the bound tends to 0 for \( m/n < r(k) \) as given in the theorem. This bound is obtained by the first moment method, that is, by finding the expected number of deficient subsets.

The results up to now are interesting for at least two reasons. First, Theorem 83 says that random \( k \)-CNF formulas are matched formulas with high probability if \( m/n < r(k) \) which is approximately 1. But, by Theorem 80 and similar results, a random \( k \)-CNF formula is almost never a member of one of the well-studied classes mentioned earlier unless \( m/n < O(k^2) \). As already pointed out, this is somewhat disappointing and surprising because all the other classes were proposed for rather profound reasons, usually reflecting cases when corresponding instances of integer programming present polytopes with some special properties. Despite all the theory that helped establish these classes, the matched class, ignored in the literature because
it is so trivial in nature, turns out to be, in some probabilistic sense, much bigger than all the others.

Second, the results provide insight into the nature of larger polynomial time solvable classes of formulas. Classes vulnerable to cyclic structures appear to be handicapped relative to classes that are not. In fact, the matched class may be generalized considerably to larger polynomial time solvable classes such as Linear Autarkies [92, 103] which are described in Section 4.9.

Consider the test for satisfiability that is implied by Theorem 43: find the satisfiability index $z$ of the given formula and compare against the width, say $k$, of the formula’s shortest clause; if $z < k/2$ then the formula is satisfiable. This test is based on the fact that Algorithm LinAut, Page 133, will output $\psi' = \emptyset$ on any formula for which $z < k/2$.

**Theorem 84.** The above test for satisfiability does not succeed on a random $(m, n, k)$-CNF formula $\psi$ with probability tending to 1 as $m, n \to \infty$ and $m/n > 1$.

**Proof.** We will use the definition of the satisfiability index of $\psi$ on Page 135: the minimum $z$ such that

$$M_\psi \beta \leq 2Z - I$$

(7.5)

where $I$ is an $n$ dimensional vector expressing the width of all clauses, $Z = \langle z, z, \ldots, z \rangle$, and $-1 \leq \beta_i \leq 1$ for all $0 \leq i < n$.

Consider the product $\gamma = M_\psi \beta$. We need to show that, with high probability, there is no assignment of values to the components of $\beta$ so that all elements of $\gamma$ are less than 0.

\[\square\]

### 7.7 Randomized algorithms: upper bounds

Section 6.1 presents upper bounds on complexity which were obtained using deterministic algorithms. Better bounds have been obtained for probabilistic algorithms. A family of such algorithms exploit a relationship between the structure of a $k$-CNF expression and structure of the space of models expressed in terms of isolated models and critical clauses [109].

**Definition 85.** A $j$-isolated model is a model with exactly $n - j$ assignment neighbors differing in one variable that are also models.

If $j$ is close to $n$, the set of such models, called nearly isolated, for a given $k$-CNF expression has a relatively “short” description. Thus, a nearly isolated model, if one exists, is relatively easy to find by searching the space of relatively “short” descriptions. The important observation is that any satisfiable $k$-CNF expression either has a nearly isolated model or has very
many models. In either case, a model can be found relatively quickly. Short descriptions depend on the notion of critical clauses.

**Definition 86.** If \( M \) is a model but reversing variable \( v_i \) in \( M \) is not a model, then there must exist a clause with exactly one literal that has value 1 under \( M \) and that literal is either \( v_i \) if \( v_i \in M \) or \( \bar{v}_i \) if \( v_i \notin M \). This clause is said to be a critical clause for variable \( i \) at model \( M \).

A critical clause cannot be critical for two different variables at the same model. Therefore, a \( j \)-isolated model has \( j \) critical clauses.

A description of a model can be encoded in fewer than \( n \) bits as follows. Let \( \pi \) be a permutation of \( \{1, 2, \ldots, n\} \). Let \( M \) be a model for a \( k \)-CNF expression \( \psi \). Define a string \( x^M \in \{0, 1\}^n \) such that \( x^M_i = 1 \) if and only if \( v_i \in M \) (that is, \( v_i \) is assigned value 1). An encoding of \( x^M \) with respect to \( \pi \), written \( I_\pi(x^M) \), is a permutation of its bits according to \( \pi \) but with the \( i \)th bit deleted for any \( i \) such that 1) there is a critical clause for the variable \( v_{\pi_i} \) at \( M \); and 2) \( v_{\pi_i} \) or \( \bar{v}_{\pi_i} \) is the last literal in the clause if literals are ordered by \( \pi \).

An encoding can be efficiently decoded. That is, a model \( M \) can be recovered efficiently from its encoding as described above. This is done by starting with the original formula \( \psi \), assigning values to variables, one at a time, in order, according to \( \pi \), and reducing \( \psi \) accordingly. Bits of the encoding are used to decide values of assignment \( M \) except in the case that the reduced \( \psi \) has a unit clause consisting of the current variable, in order, or its complement. In that case, the current variable is assigned a value that satisfies that clause. This leads to the following lemma.

**Lemma 87.** - **The Satisfiability Coding Lemma** (from [109]) - If \( M \) is a \( j \)-isolated model of a \( k \)-CNF formula \( \psi \), then its average (over all permutations \( \pi \)) description length under the encoding \( I_\pi(x^M) \) is at most \( n - j/k \).

This has inspired the algorithm shown in Figure 7.8. Notice the algorithm essentially randomly chooses a permutation \( \pi \) and then simultaneously 1) attempts a decoding according to \( \pi \); and 2) checks a potential satisfying assignment according to \( \pi \). Performance of the algorithm is given by the following.

**Theorem 88.** (from [109]) The algorithm of Figure 7.8 runs in \( O(n^2|\psi|2^{n-n/k}) \) time and finds a model for a given satisfiable \( k \)-CNF expression with probability tending to 1.

The complexity is easy to see. The following outlines the reason the probability of finding a model tends to 1. Suppose \( \psi \) has a \( j \)-isolated model \( x^M \). Fix \( j \) critical clauses as above. By Lemma 87, the average number (over all \( \pi \)) of critical variables that appear last among the variables in their critical clauses is at least \( j/k \). Since the maximum number of critical variables is \( n \),
for at least $1/n$ fraction of the permutations the number of such variables is at least $j/k$. Therefore, the probability that there are at least $j/k$ critical clauses where the critical variables occur last for $\pi$ (the variable choices for a single round of the algorithm) is at least $1/n$. Since $x^M$ is $j$-isolated, if the algorithm is to output $x^M$ it makes at most $n - j/k$ random assignments. So the probability that the algorithm outputs $x^M$ on a round is at least $2^{-n+j/k}/n$. A straightforward summation over all models for $\psi$ determines a bound on the probability that the algorithm gives up.

An improved randomized algorithm which uses RandomSatSolver of Figure 7.8 is given in [110]. It finds a model, when one exists, for a 3-CNF expression with probability tending to 1 in $O(1.362^n)$ steps. The algorithm is peculiar because its performance actually deteriorates as the number of models for a given expression increases [75]. This is likely due to the emphasis of the algorithm on finding nearly isolated models and the lack of intelligence in looking for non-isolated models.

A better randomized algorithm for 3-CNF expressions, which is based on local search, is presented in [120] and shown in Figure 7.9. Each round of this algorithm begins with an assignment that is hoped to be fairly close to a model. Then, for $3n$ iterations, single variable assignment flips are made, attempting to eliminate all falsified clauses. After $3n$ iterations, if a model is not found, it is unlikely that the assignment chosen at the beginning of the round is close to a model, so the round terminates and another assignment is picked on the next round and the process repeats.

**Theorem 89.** (adapted from [120]) The algorithm of Figure 7.9 has complexity $O(n|\psi|(2(1 - 1/k))^n)$ and finds a model for a given satisfiable $k$-CNF expression with probability tending to 1.

For 3-CNF expressions, this algorithm has $O(1.333^n)$ complexity. Contrary to the randomized algorithm of Figure 7.8, the performance of this algorithm improves as the number of models of an input increases. This observation is exploited in an algorithm presented in [75] where a round begins with a random assignment $\tau$ and then the $3n$ local search steps of SLS and the $O(n)$ Davis-Putnam steps of RandomSatSolver are run on $\tau$. The resulting algorithm has complexity $O(1.324^n)$ for 3-CNF expressions.

### 7.8 Survey Propagation

New CNF algorithms, startlingly successful, have been developed by observing similarities in CNF structure and models of matter. We describe one here for the purpose of illustrating the impact non-rigorous methods may have on future SAT algorithm development.

We are concerned with the CNF structure that arises from the bipartite graph $G^\phi$ introduced on Page 174. Recall, given CNF expression $\phi$, $G^\phi$ has vertex set $V^\phi$ corresponding to variables, vertex set $C^\phi$ corresponding to
**Procedure** \textsc{RandomSatSolver} ($\psi$)

**Input:** a $k$-CNF formula $\psi$ with $n$ variables.

**Output:** either “give up” or a model for $\psi$.

Repeat the following $n^{2n-\frac{n}{k}}$ times:

Set $M \leftarrow \emptyset$.
Set $V \leftarrow \{v : \exists C \in \psi \text{ such that } v \in C \text{ or } \neg v \in C\}$.
Set $\psi' \leftarrow \psi$.

Repeat the following while $V \neq \emptyset$:

If $\emptyset \in \psi'$ then break. // No extension will be a model
If $\psi' = \emptyset$ then Output $M$.
Randomly choose $v \in V$.
If $\{v\} \in \psi'$ then do the following:
Set $M \leftarrow M \cup \{v\}$.
Set $\psi' \leftarrow \{C - \{\neg v\} : C \in \psi', v \notin C\}$.

Otherwise, if $\{\bar{v}\} \in \psi'$ then do the following:
Set $\psi' \leftarrow \{C - \{v\} : C \in \psi', \neg v \notin C\}$.

Otherwise, with probability $1/2$ do the following:
Set $M \leftarrow M \cup \{v\}$.
Set $\psi' \leftarrow \{C - \{\neg v\} : C \in \psi', v \notin C\}$.

Otherwise, do the following:
Set $\psi' \leftarrow \{C - \{v\} : C \in \psi', \neg v \notin C\}$.
Set $V \leftarrow V \setminus \{v\}$.

Output “give up”.

Figure 7.8: A randomized algorithm for finding a model for a satisfiable $k$-CNF formula.
Procedure SLS ($\psi$)

**Input:** a $k$-CNF formula $\psi$ with $n$ variables.

**Output:** either “give up” or a model for $\psi$.

Set $V \leftarrow \{v : \exists C \in \psi \text{ such that } v \in C \text{ or } \neg v \in C\}$.

Repeat the following $n(2(1 - 1/k))^n$ times:

Set $M \leftarrow \emptyset$.

Repeat the following for each $v \in V$:

Set $M \leftarrow M \cup \{v\}$ with probability $1/2$.

Repeat the following $3n$ times:

Set $\psi' \leftarrow \{C \in \psi : \forall v \in C, v \notin M \text{ and } \forall \neg v \in C, v \in M\}$.

If $\psi' = \emptyset$ then Output $M$.

Randomly choose $C \in \psi'$.

Randomly choose literal $l \in C$.

If $l$ is a positive literal then Set $M \leftarrow M \cup \{l\}$.

Otherwise, if $l$ is a negative literal $\neg v$ then Set $M \leftarrow M \setminus \{v\}$.

Output “give up”.

Figure 7.9: A local-search randomized algorithm for finding a model for a satisfiable $k$-CNF formula. The factor $n$ in the outermost Repeat loop can be improved but is good enough for the purposes of this exposition.
Figure 7.10: Model of a spin glass. Variables \( \sigma_1, \sigma_2, \ldots \), called spins, are binary taking values +1 or -1. Functions \( E_1, E_2, \ldots \) are energy functions, each of which depends on a small number of neighboring spins. The spin glass problem is to find spin values which minimize the total energy of a given collection of energy functions.
among small groups of neighboring spins. This model can be represented graphically, as depicted in Figure 7.10, where circled vertices are spins, boxed vertices are energy functions, and edges show the dependence of each energy function on spins. Such a graph is bipartite: there is no edge between a pair of circle vertices and no edge between a pair of box vertices. It is a general rule in physics that systems tend to migrate toward their minimum energy (that is maximum entropy) state. Thus, the spin glass problem is to determine the minimum energy of a given spin glass model.

The spin glass problem is analogous to the problem of determining a model for a given CNF expression: spins take the role of Boolean variables, energy functions take the role of clauses, and energy corresponds to clauses satisfied. Thus, if the minimum energy of a spin glass is 0, the corresponding CNF expression has a model and if the minimum energy is greater than 0, the expression has no model. Energy functions consistent with the so called Ising model and expressing this behavior for \( k \) literal clauses composed of literals taken from variables \( v_{i1}, v_{i2}, ..., v_{ik} \) corresponding to spins \( \sigma_{i1}, \sigma_{i2}, ..., \sigma_{ik} \) are

\[
E_i = 2 \prod_{r=1}^{k} \frac{1 + J_{ir} \sigma_{ir}}{2}
\]

where \( J_{ir} \) is +1 if the \( r \)-th literal of the \( i \)-th clause is complemented and -1 if it is uncomplemented\(^{12}\). Thus, the energy of clause \( \{v_1, \bar{v}_2, \bar{v}_3\} \) is \((1 - \sigma_1)(1 + \sigma_2)(1 + \sigma_3)/4\). Observe this is 0 if \( \sigma_1 = +1, \sigma_2 = -1, \) or \( \sigma_3 = -1 \) and is 2 otherwise. The energy of an entire system of \( m \) energy functions is

\[
E = \sum_{i=1}^{m} E_i.
\]

All clauses are satisfied if and only if the energy of the analogous spin glass is 0.

Where can physicists help? They can make assumptions analogous to those that apply to the physical world and thereby aim for a level of understanding that computer scientists would not have thought of. It is natural in physics to consider the probability distribution of spins:

\[
Pr(\sigma_1, \sigma_2, ..., \sigma_n) = \frac{1}{Z} e^{-\frac{1}{T}E}
\]

where \( T \) is temperature and \( Z \) is a normalizing constant. This distribution follows from the observation that a system in equilibrium tends to seek its highest entropy, or equivalently, lowest energy state. At \( T = 0 \), the lowest energy states are the only significant terms contributing to this distribution (non-rigorously) and, given a system of energy functions as defined above,

\(^{12}\)The factor of 2 is explained below.
one is interested in the distribution at the 0, or lowest possible, energy state. It can be calculated in a fairly straightforward manner for each spin separately due to the assumption of a thermodynamic limit: that is, \( \lim_{n \to \infty} E/n \) is bounded. By this assumption, what is happening at any particular spin is independent of what is happening at nearly all other spins. Thus, probability distributions are assumed to be decomposed into products of probability distributions of influential variables.

Let’s see how this may apply to \( k \)-CNF analogs. For simplicity of notation, and without loss of generality, let \( E_i \) be a function of spins \( \sigma_1, \ldots, \sigma_k \). Let \( h_{j-i} \cdot \sigma_j \) be the energy contributed to \( E_i \) by the spin \( \sigma_j \) assuming the effect of \( E_i \) on \( \sigma_j \) is disregarded. The \( h \) terms are called magnetic fields in physics. Let \( u_{i-j} \) be the contribution to the magnetic field of spin \( \sigma_j \) from \( E_i \). Consider the marginal distribution for \( \sigma_1 \) of \( E_i \). If the Ising model is assumed, one writes

\[
\sum_{\sigma_2, \ldots, \sigma_k} e^{-\frac{1}{T}(E_i - h_{2-i} \cdot \sigma_2 - \ldots - h_{k-i} \cdot \sigma_k)} = e^{\frac{1}{T}(w_{i-1} + u_{i-1} \cdot \sigma_1)}.
\]

Since one is interested in the case \( T = 0 \), this simplifies to

\[
\min_{\sigma_2, \ldots, \sigma_k} \{E_i - h_{2-i} \cdot \sigma_2 - \ldots - h_{k-i} \cdot \sigma_k\} = -w_{i-1} - u_{i-1} \cdot \sigma_1.
\]

For the \( k \)-CNF problem, \( E_i \) has been given on Page 180 and

\[
w_{i-1} = \sum_{j \in E_i} h_{j-i} \quad u_{i-1} = \sum_{j \in E_i} w_{i-j} \cdot \sigma_j - \theta(J_{j-i}^2) \cdot \sum_{j \in E_i} u_{i-j}
\]

where \( \theta(x) = 1 \) if \( x > 0 \) and \( \theta(x) = 0 \) if \( x < 0 \). Interpreting \( h_{j-i} > 0 \) as evidence that \( \sigma_j \) should be \(+1\) and \( h_{j-i} < 0 \) as evidence that \( \sigma_j \) should be \(-1\) to minimize the \( i \)th energy function (satisfy the \( i \)th clause), \( w_{i-1} + u_{i-1} \cdot \sigma_1 = |h_{2-i}| + \ldots + |h_{k-i}| \) if for any \( \sigma_2, \ldots, \sigma_k \) the evidence supports the current value of the corresponding spin, or if no support by these \( h \) variables is given and \( \sigma_1 \) has a value which minimizes the \( i \)th energy function. But \( w_{i-1} + u_{i-1} \cdot \sigma_1 = |h_{2-i}| + \ldots + |h_{k-i}| - 2 \) if \( \sigma_1 \) is not set to minimize \( E_i \) and no support is given for any of \( \sigma_2, \ldots, \sigma_k \) (the \( i \)th clause is falsified). This explains the factor of 2 used in defining clausal energy on Page 180. The minimum energy of the entire system given spin, say \( \sigma_j \), has a particular value is

\[
E_{\sigma_j} = C - \sum_{i: \sigma_j \in E_i} w_{i-j} \cdot \sigma_j \sum_{i: \sigma_j \in E_i} u_{i-j}
\]

where \( C \) is some constant. Write

\[
h_j = \sum_{i: \sigma_j \in E_i} u_{i-j}.
\]

(7.6)
Let $Q_{i \rightarrow j}(u)$ be the probability distribution of $u_{i \rightarrow j}$. Let $P_{j \rightarrow i}(h)$ be the probability distribution of the contribution of $h_j$ to function $E_i$. Suppose energy functions $E_{\pi_1}, \ldots, E_{\pi_p}$ influence spin $\sigma_j$. By independence of distributions, which follows from the assumption of thermodynamic limit,

$$P_{j \rightarrow i}(h) = C_{j \rightarrow i} \sum_{u_1, \ldots, u_p: u_{j \rightarrow i} = h} Q_{\pi_1 \rightarrow j}(u_1) \cdots Q_{\pi_p \rightarrow j}(u_p) e^{y|h|}.$$ 

For each spin $\sigma_j$ of $E_i$, let $\sigma_{\pi_1}', \ldots, \sigma_{\pi_p}'$ be the remaining spins of $E_i$.

Let $\theta(J_{i \rightarrow j}^\pi \cdot h_1) \cdot \cdots \cdot \theta(J_{i \rightarrow j}^{\pi_p} \cdot h_p')$ be denoted by $w_{i,1 \ldots p'}$. Write

$$Q_{i \rightarrow j}(u) = C_{i \rightarrow j} \sum_{h_1, \ldots, h_p'} P_{\pi_1' \rightarrow i}(h_1) \cdots P_{\pi_p' \rightarrow i}(h_p') e^{-y \cdot w_{i,1 \ldots p'}}.$$ 

The terms $C_{i \rightarrow j}$ and $C_{j \rightarrow i}$ are normalizing constants and $y$ is a parameter which expresses the rate of change of complexity with respect to $E/n$, and complexity is a measure of the number of different energy states possible. Since this information is not known generally, $y$ must be guessed.

The values for $Q_{i \rightarrow j}(u)$ and $P_{j \rightarrow i}(h)$ may be computed by the algorithm of Figure 7.11. It is then a simple matter to determine $P_j(h)$, the distribution for the field acting on spin $\sigma_j$. But the value of $P_j(h)$ suggests a setting for spin $\sigma_j$ which minimizes energy: namely, +1 if $\sum_{h > 0} P_j(h) > \sum_{h < 0} P_j(h)$ and -1 if $\sum_{h > 0} P_j(h) < \sum_{h < 0} P_j(h)$. If $\sum_{h > 0} P_j(h) = \sum_{h < 0} P_j(h) = 1$ no bias is detected. This suggests the algorithm of Figure 7.12 for solving CNF expressions: assign values to “biased” variables first, re-computing $P_j(h)$ after each variable is assigned, then apply a standard SAT solver to complete the assignment.

Considerable success has been reported with this approach [106] on random $(n, m, k)$-CNF expressions and also some benchmarks which have been considered hard for a variety of SAT solvers. What is different about the method discussed here? First, it provides a way to choose initial variables and values, based on apparent probability distributions, that is intelligent enough to know when to stop. Traditional methods for choosing the first so many variables to branch on will choose as many values and variables that the user would like. Mistakes higher up the search process are very serious and, if made, can result in very long searches at the lower end. The method of M´ezard and Zecchina seems to make few mistakes on many expressions. By maximizing entropy, variable/value choices tend to result in reduced expressions such that an additional variable choice will yield essentially the same reduced subexpressions regardless of its value. Second, more interdependence of CNF components is taken into account. We illustrate with the well-known Johnson heuristic [76] which chooses a variable and assigns
a value so as to maximize the probability that a 0 energy state (a model) exists assuming clauses are statistically independent (an unlikely situation). Although successful in several cases, this heuristic cannot really see very far ahead of the current state. But the method of Mézard and Zecchina is designed to, in some sense, explore all possible energy states for a given expression or subexpression, particularly the lowest energy states, and present statistics on those states and their causes.
Procedure $SP(G^\phi)$

**Input:** bipartite graph $G^\phi$ relating energy functions to spins.

**Output:** set of distributions of all spin fields.

**begin**

for all energy function $E_i$ to spin $\sigma_j$ edges in $G^\phi$ define:

$$Q_{i\rightarrow j}(u) = \begin{cases} c_{i\rightarrow j} & \text{if } u = 0 \\ 1 - c_{i\rightarrow j} & \text{if } u = J^i_j \\ 0 & \text{otherwise} \end{cases}$$

and initialize $c_{i\rightarrow j}$ randomly;

repeat the following until convergence {

select an energy function $E_i$;

for each spin $\sigma_j$ input to $E_i$ compute

$$P_{j\rightarrow i}(h) = C_{j\rightarrow i} \sum_{u_1, \ldots, u_p, \sum_{x=1}^p u_x = h} Q_{\pi_1\rightarrow j}(u_1) \cdot \ldots \cdot Q_{\pi_p\rightarrow j}(u_p) e^{y|h|};$$

where $E_{\pi_1}, \ldots, E_{\pi_p}$ are functions depending on spin $\sigma_j$ except $E_i$ and $C_{j\rightarrow i}$ is a normalizing constant;

for each spin $\sigma_j$ input to $E_i$ let $\theta(J^\pi_1_i \cdot h_1) \cdot \ldots \cdot \theta(J^\pi_p_{i'}, h_{i'})$ be denoted by $w_{i, 1, \ldots, p'}$ (that is, from Page 182, $w_{i, 1, \ldots, p'}$ denotes $w_{i\rightarrow j} = |h_1^\pi_1| - \ldots - |h_{i'}|$ where $\sigma_{i'} \ldots \sigma_{p'}$ are spins of $E_i$) and compute $Q_{i\rightarrow j}(u) =$

$$C_{i\rightarrow j} \sum_{h_1, \ldots, h_{i'}, u = -J^i_j \cdot w_{i, 1, \ldots, p'}} P_{\pi'_{1\rightarrow i}}(h_1) \cdot \ldots \cdot P_{\pi'_{p\rightarrow i}}(h_{i'}) e^{-y \cdot w_{i, 1, \ldots, p'}}$$

}

Compute $P_j(h) =$

$$C_j \sum_{u_1, \ldots, u_{p+1}, \sum_{x=1}^{p+1} u_x = h} Q_{\pi_1\rightarrow j}(u_1) \cdot \ldots \cdot Q_{\pi_{p+1\rightarrow j}}(u_{p+1}) e^{y|h|}.$$ 

where $E_{\pi_1}, \ldots, E_{\pi_{p+1}}$ are functions depending on spin $\sigma_j$;

return $P_j(h)$ for all $j$;

end;

Figure 7.11: An algorithm for computing the distributions $P_j(h)$. 
**Procedure SID (ϕ)**

**Input:** a CNF expression ϕ.

**Output:** either “unsatisfiable” or a model for ϕ.

```plaintext
begin
    repeat the following indefinitely {
        M := ∅;
        establish $G^\phi$ and variables $u_{i\rightarrow j}$;
        randomly choose values for $u_{i\rightarrow j}$;
        repeat the following until $\emptyset \in \phi$ {
            run SP on $G^\phi$ to evaluate $P_j(h)$;
            if $P_j(h) = 0$ except at $h = 0$ for all $j$ then {
                apply a SAT solver to $\phi$;
                if it finds a model then return that model;
            }
            for all $\sigma_j$, let $w^-_j = \sum_{h < 0} P_j(h)$ and $w^+_j = \sum_{h > 0} P_j(h)$;
            choose $\sigma_j$ such that $|w^+_j - w^-_j|$ is a maximum;
            if $w^+_j > w^-_j$ then $\sigma_j := +1$; else $\sigma_j := -1$;
            if $\sigma_j = +1$ then {
                $M := M \cup \{v_j\}$;
                $\phi := \{C \setminus \{\bar{v}_j\} : C \in \phi, v_j \notin C\}$;
            } else {
                $\phi := \{C \setminus \{v_j\} : C \in \phi, \bar{v}_j \notin C\}$;
                while there exists a unit clause $\{l\} \in \phi$ do the following {
                    if $l$ is an uncomplemented literal then $M := M \cup \{l\}$;
                    $\phi := \{C \setminus \{l\} : C \in \phi, l \notin C\}$;
                }
            }
        }
    }
end;
```

Figure 7.12: An algorithm for solving a CNF expression based on maximizing entropy.
Bibliography


192 Chapter 7. PROBABILISTIC ANALYSIS


7.8 SURVEY PROPAGATION


7.8 SURVEY PROPAGATION


Appendix A

Glossary

Algorithm: (4,13,20)
A specific set of instructions for carrying out a procedure or solving a problem, usually with the requirement that the algorithm terminate at some point.

Boolean Function: (28)
A mapping \( \{0, 1\} \times \{0, 1\} \times \ldots \times \{0, 1\} \mapsto \{0, 1\} \). If the dimension of the domain is \( n \), the number of possible functions is \( 2^{2^n} \).

Clause: (27 also see Formula, CNF)
In CNF formulas a clause is a disjunction of literals such as the following: \( \bar{a} \lor b \lor c \lor d \). A clause containing only negative (positive) literals is called a negative clause (alternatively, a positive clause). In this monograph a disjunction of literals is also written as a set such as this: \( \{a, b, c, d\} \). Either way, the width of a clause is the number of literals it contains. In logic programming a clause is an implication such as the following: \( a \land b \land c \rightarrow g \). In this monograph, if a formula is a conjunction or disjunction of expressions, we say each expression is a clause.

Clause Width: (see Clause)

Edge: (see Graph)

Endpoint: (31)
One of two vertices spanned by an edge. See Graph.

Formula, DNF: (23)
DNF stands for Disjunctive Normal Form. Let a literal be a variable or a negated variable. Let a *conjunctive clause* be a single literal or a conjunction of two or more literals (see *Clause*). A DNF formula is an expression of the form $C_1 \lor C_2 \lor \ldots C_m$ where each $C_i, 1 \leq i \leq m$, is a conjunctive clause: that is, a DNF formula is a disjunction of some number $m$ of conjunctive clauses. A conjunctive clause evaluates to *true* under an assignment of values to variables if all its literals has value *true* under the assignment.

**Formula, CNF:** (3,5,20,27)

CNF stands for conjunctive normal form. Let a literal be a variable or a negated variable. Let a *disjunctive clause* be a single literal or a disjunction of two or more literals (see *Clause*). A CNF formula is an expression of the form $C_1 \land C_2 \land \ldots C_m$ where each $C_i, 1 \leq i \leq m$, is a disjunctive clause: that is, a CNF formula is a conjunction of some number $m$ of disjunctive clauses. In this monograph we regard a disjunctive clause, sometimes called a clause when the context is clear, to be a set of literals and a CNF formula to be a set of clauses. Thus the following is an example of how a CNF formula is expressed in this monograph.

$$\{\{\bar{a}, b\}, \{a, c, d\}, \{c, \bar{d}, \bar{e}\}\}$$

A CNF formula is said to be *satisfied* by an assignment of values to its variables if the assignment causes all its clauses to evaluate to *true*. A clause evaluates to *true* under an assignment of values to variables if at least one of its literals has value *true* under the assignment.

**Formula, Horn:** (28,5)

A CNF formula in which every clause has at most one positive literal. Satisfiability of Horn formulas can be determined in linear time [48, 74]. Horn formulas have the remarkable property that, if satisfiable, there exists a unique minimum satisfying assignment with respect to the value *true*. In other words, the set of all variables assigned value *true* in any satisfying assignment other than the unique minimum one includes the set of variables assigned value *true* in the minimum one.

**Formula, Minimally Unsatisfiable:** (29)

An unsatisfiable CNF formula such that removal of any clause makes it satisfiable.

**Formula, Propositional or Boolean:** (3,13,20,22,26,27)
A Boolean variable is a formula. If \( \psi \) is a formula, then \( (\psi) \) is a formula. If \( \psi \) is a formula then \( \neg \psi \) is a formula. If \( \psi_1 \) and \( \psi_2 \) are formulas and \( O_b \) is a binary Boolean operator, then \( \psi_1 O_b \psi_2 \) is a formula. In some contexts other than Logic Programming, we use \( \bar{a} \) or \( \bar{\psi} \) instead of \( \neg a \) or \( \neg \psi \) to denote negation of a variable or formula. Formulas evaluate to \textit{true} or \textit{false} depending on the operators involved. Precedence from highest to lowest is typically from parentheses, to \( \neg \), to binary operators. Association, when it matters as in the case of \( \rightarrow \) (implies), is typically from right to left.

**Graph:** (19)
A mathematical object composed of points known as vertices or nodes and lines connecting some (possibly empty) subset of them, known as edges. Each edge is said to span the vertices it connects. If weights are assigned to the edges then the graph is a \textit{weighted graph}. Below is an example of a graph and a weighted graph.

![Graph Example](image)

**Graph, Directed:** (32)
A graph in which some orientation is given to each edge.

**Graph, Directed Acyclic:** (32)
A directed graph such that, for every pair of vertices \( v_a \) and \( v_b \), there is not both a path from \( v_a \) to \( v_b \) and a path from \( v_b \) to \( v_a \).

**Graph, Rooted Directed Acyclic:** (33)
A connected, directed graph with no cycles such that there is exactly one vertex, known as the \textit{root}, whose incident edges are all oriented away from it.

**Literal:** (25, see also \textbf{Formula, CNF})
A Boolean variable or the negation of a Boolean variable. In the context of a formula, a negated variable is called a \textit{negative} literal and an unnegated variable is called a \textit{positive} literal.

**Logic Program, Normal:** (37)
A formula consisting of implicational clauses. That is, clauses have the form \((a \land b \land c \rightarrow g)\) where atoms to the left of \( \rightarrow \) could be positive or negative literals.

**Maximum Satisfiability (MAX SAT):** ()
The problem of determining the maximum number of clauses of a given CNF formula that can be satisfied by some truth assignment.
206 Chapter 7. PROBABILISTIC ANALYSIS

Model, Minimal Model: (29)
For the purposes of this monograph, a model is a truth assignment satisfying a given formula. A minimal model is such that a change in value of any true variable causes the assignment not to satisfy the formula. See also Formula, Horn.

Model, Stable: (37)
In a logic program, a minimal model such that the variables of value 1 are all derivable directly (reasoning “forward” through the \( \rightarrow \)'s) from the given facts and the negative literals of value 1.

Model, Well-founded: (38)
A well-founded model for a normal logic program allows three values to be assigned to all variables: 0, 1 and \( \perp \) where \( \perp \) has the meaning lacks a classic logic value.

\( \mathcal{NP} \)-hard: (3)
A very large class of difficult combinatorial problems. There is no known polynomial time algorithm for solving any \( \mathcal{NP} \)-hard problem and it is considered unlikely that any will be found. For a more formal treatment of this topic see M.R. Garey and D.S. Johnson. Computers and Intractability: A Guide to the Theory of \( \mathcal{NP} \)-completeness, W.H. Freeman, San Francisco, 1979.

Operator, Boolean: (11,25,61)
A mapping from binary Boolean vectors to \{0,1\}. Most frequently used binary operators are

- Or \( \lor \): \{00 \mapsto 0; 10,01,11 \mapsto 1\}
- And \( \land \): \{00,01,10 \mapsto 0; 11 \mapsto 1\}
- Implies \( \rightarrow \): \{01 \mapsto 0; 00,01,11 \mapsto 1\}
- Equivalent \( \leftrightarrow \): \{01,10 \mapsto 0; 00,11 \mapsto 1\}
- XOR \( \oplus \): \{00,11 \mapsto 0; 01,10 \mapsto 1\}

out of the 16 possible mappings. A Boolean operator \( \mathcal{O} \) shows up in a formula like this: \((v_l \mathcal{O} v_r)\) where \( v_l \) is called the left operand of \( \mathcal{O} \) and \( v_r \) is called the right operand of \( \mathcal{O} \). Thus, the domain of \( \mathcal{O} \) is a binary vector whose first component is the value of \( v_l \) and whose second component is the value of \( v_r \). In the text we sometimes use patterns of 4 bits to represent an operator: the first bit is the mapping from 00, the second from 01, the third from 10, and the fourth from 11. Thus, the operator 0001 applied to \( v_l \) and \( v_r \) has the same functionality as \((v_l \land v_r)\), the operator 0111 has the same functionality as \((v_l \lor v_r)\), and the operator 1101 has the same functionality as \((v_l \rightarrow v_r)\). The only meaningful unary operator is \( \neg \): \{1 \mapsto 0; 0 \mapsto 1\}. We also sometimes write \( \bar{a} \) or \( \bar{\psi} \) for \( \neg a \) or \( \neg \psi \) where \( a \) is a variable and \( \psi \) is a formula.

Operator, Temporal: (11,26)
An operator used in temporal logics. Basic operators include henceforth ($\square \psi_1$), eventually ($\lozenge \psi_1$), next ($\circ \psi_1$), and until ($\psi_1 U \psi_2$).

Satisfied Clause: (see Formula, CNF)

Satisfiability (SAT): (3,6, 17)

The problem of deciding whether there is an assignment of values to the variables of a given Boolean formula that makes the formula true. In 1971, Cook showed that the general problem is NP-complete even if restricted to CNF formulas containing clauses of width 3 or greater or if the Boolean operators are restricted to any truth-functionally complete subset. However, many efficiently solved subclasses are known.

State: (26)

A particular assignment of values to the parameters of a system.

Subgraph: (19,31)

A graph whose vertices and edges form subsets of the vertices and edges of a given graph where an edge is contained in the subset only if both its endpoints are.

Unit Clause: (22)

A clause consisting of one literal. See also Clause in the glossary.

Variable, Propositional or Boolean: (7,20,22)

An object taking one of two values. In this monograph we alternate equivalently between the values $\{0,1\}$ and $\{false, true\}$ depending on context: $\{0,1\}$ is typically used to represent logic levels in an electrical circuit and $\{true, false\}$ is applied to literals and formulas.

Vertex: (see Graph)

Vertex Weighted Satisfiability: (4)

The problem of determining an assignment of values to the variables of a given CNF formula, with weights on the variables, that satisfies the formula and maximizes the sum of the weights of true variables. The problem of finding a minimal model for a given satisfiable formula is a special case.

Weighted Maximum Satisfiability: (17, 22)

The problem of determining an assignment of values to the variables of a given CNF formula, with weights on the clauses, which maximizes the sum of the weights of satisfied clauses.