Algorithms for the Satisfiability Problem

John Franco
University of Cincinnati
School of Computing Sciences and Informatics
Cincinnati, OH
franco@gauss.ececs.uc.edu

and

Sean Weaver
U.S. Department of Defense
Ft. George G. Meade, Maryland
weaversa@gmail.com

Contents

Abstract

The Satisfiability problem (SAT) has gained considerable attention over the past decade for two reasons. First, the performance of competitive SAT solvers has improved enormously due to the implementation of new algorithmic concepts. Second, so many real-world problems that are not naturally expressed as instances of SAT can be transformed to SAT instances and solved relatively efficiently using one of the ever improving SAT solvers or solvers based on SAT. This chapter attempts to clarify these successes by presenting the important advances in SAT algorithm design as well as the classic implementations of SAT solvers. Some applications to which SAT solvers have been successfully applied are also presented.

The first section of the chapter presents some logic background and notation that will be necessary to understand and express the algorithms presented. The second section presents several representations for SAT instances in preparation for discussing the wide variety of SAT solver implementations that have been tried. The third section presents some applications of SAT. The final two sections presents algorithms for SAT, including search and heuristic algorithms plus systems that use SAT to manage the logic of complex computations.

1 Logic

An instance of Satisfiability is a propositional logic expression in Conjunctive Normal Form (CNF), the meaning of which is described in the next few paragraphs. The most elementary object comprising a CNF expression is the Boolean variable, shortened to *variable* in the context of this chapter. A variable takes one of two values from the set $\{0, 1\}$. A variable v may be negated in which case it is denoted $\neg v$. The value of $\neg v$ is opposite that of v. A *literal* is either a variable or a negated variable. The term *positive literal* is used to refer to a variable and the term *negative literal* is used to refer to a negated variable. The *polarity* of a literal is positive or negative accordingly.

The building blocks of propositional expressions are binary Boolean operators. A *binary Boolean operator* is a function $\mathcal{O}_b : \{0, 1\} \times \{0, 1\} \mapsto \{0, 1\}$. Often, such functions are presented in tabular form, called *truth tables*, as illustrated in Figure 10. There are 16 possible binary operators and the most common (and useful) are $\vee$ (or), $\wedge$ (and), $\rightarrow$ (implies), $\leftrightarrow$ (equivalent), and $\oplus$ (xor, alternatively exclusive-or). Mappings defining the common operators are shown in Figure 1. The only interesting *unary Boolean operator*, denoted $\neg$ (negation), is a mapping from 0 to 1 and 1 to 0. If $\neg l$ is a literal then $\neg\neg l$ is the same as l.

A *formula* is a propositional expression consisting of literals, parentheses, and operators which has some semantic content and whose syntax is described recursively as follows:

1. Any single variable is a formula.
2. If ψ is a formula, then so is $\neg\psi$.

Operator	Symbol	Mapping
Or	$\vee$	$\{00 \mapsto 0;\ 10, 01, 11 \mapsto 1\}$
And	$\wedge$	$\{00, 01, 10 \mapsto 0;\ 11 \mapsto 1\}$
Implies	$\rightarrow$	$\{01 \mapsto 0;\ 00, 10, 11 \mapsto 1\}$
Equivalent	$\leftrightarrow$	$\{01, 10 \mapsto 0;\ 00, 11 \mapsto 1\}$
XOR	$\oplus$	$\{00, 11 \mapsto 0;\ 01, 10 \mapsto 1\}$

Table 1: The most common binary Boolean operators and their mappings

Operator	Symbol	Mapping
Or	$\vee$	$\{00 \mapsto 0;\ \ 10, 01, 11, 1\perp, \perp 1 \mapsto 1;\ \ 0\perp, \perp 0, \perp\perp \mapsto \perp\}$
And	$\wedge$	$\{00, 01, 10 \mapsto 0;\ \ 11 \mapsto 1;\ \ ?\perp, \perp ?, \perp\perp \mapsto \perp\}$
Implies	$\rightarrow$	$\{10 \mapsto 0;\ \ 00, 01, 11, 0\perp \mapsto 1;\ \ 1\perp, \perp ? \mapsto \perp\}$
Equivalent	$\leftrightarrow$	$\{01, 10 \mapsto 0;\ \ 00, 11 \mapsto 1;\ \ ?\perp, \perp ?, \perp\perp \mapsto \perp\}$
XOR	$\oplus$	$\{00, 11 \mapsto 0;\ \ 01, 10 \mapsto 1;\ \ ?\perp, \perp ?, \perp\perp \mapsto \perp\}$
Negate	$\neg$	$\{0 \mapsto 1;\ \ 1 \mapsto 0,\ \ \perp \mapsto \perp\}$

Table 2: Boolean operator mappings with $\perp$. Symbol ? means 1 or 0.

3. If ψ_1 and ψ_2 are both formulas and $\mathcal{O}$ is a Boolean binary operator, then $(\psi_1\ \mathcal{O}\ \psi_2)$ is a formula, ψ_1 is called the *left operand* of $\mathcal{O}$, and ψ_2 is called the *right operand* of $\mathcal{O}$.

Formulas can be simplified by removing some or all parentheses. Parentheses around nestings involving the same associative operators such as $\vee$, $\wedge$, and $\leftrightarrow$ may be removed. For example, $(\psi_1 \vee (\psi_2 \vee \psi_3))$, $((\psi_1 \vee \psi_2) \vee \psi_3)$, and $(\psi_1 \vee \psi_2 \vee \psi_3)$ are considered 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. The following is an example of a simplified formula:

$$(\neg v_0 \vee v_1 \vee \neg v_7) \wedge (\neg v_2 \vee v_3) \wedge (v_0 \vee \neg v_6 \vee \neg v_7) \wedge (\neg v_4 \vee v_5 \vee v_9)$$

A useful parameter associated with a formula is depth. The *depth* of a 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 ψ plus 1.
3. The depth of a formula $(\psi_1\ \mathcal{O}\ \psi_2)$ is the maximum of the depth of ψ_1 and the depth of ψ_2 plus 1.

A truth assignment or *assignment* is a set of variables all of whom have value 1. If M is an assignment and V is a set of variables then, if $v \in V$ and $v \notin M$, v has value 0. 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 using mappings associated with the Boolean operators that are found in Table 1.

Many algorithms that will be considered later iteratively build assignments and it will be necessary to distinguish variables that have been assigned a value from those that have not been. In such cases, a variable will be allowed to hold a third value, denoted $\perp$, which means the variable is unassigned. This requires that the evaluation of operations be augmented to account for $\perp$ as shown in Table 2. If M is an assignment of values to a set of variables where at least one variable has value $\perp$ then M is said to be a *partial assignment*.

Formulas that are dealt with in this chapter 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 formula consisting only of literals and the operator $\vee$. If all the literals of a clause are negative (positive) then the clause is called a *negative clause* (respectively, *positive clause*). In this chapter disjunctive clauses will usually be represented as sets of literals. When it is understood that an object is a disjunctive clause, it will be referred to simply as a *clause*. The following two lines show the same formula of four clauses expressed, above, in conventional logic notation and, below, as a set of sets of literals.

$$(\neg v_0 \vee v_1 \vee \neg v_7) \wedge (\neg v_2 \vee v_3) \wedge (v_0 \vee \neg v_6 \vee \neg v_7) \wedge (\neg v_4 \vee v_5 \vee v_9)$$
$$\{\{\neg v_0, v_1, \neg v_7\}, \{\neg v_2, v_3\}, \{v_0, \neg v_6, \neg v_7\}, \{\neg v_4, v_5, v_9\}\}$$

A *conjunctive clause* is a formula consisting only of literals and the operator $\wedge$. A conjunctive clause will also usually be represented as a set of literals and called 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 formula consisting of a conjunction of two or more disjunctive clauses.

Given CNF formula ψ and L_ψ, the set of all literals in ψ, a literal l is said to be a *pure literal* in ψ if $l \in L_\psi$ but $\neg l \notin L_\psi$. 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 $(\neg a \vee \neg b \vee \neg c \vee g)$ is functionally the same as $(a \wedge b \wedge c \rightarrow g)$ 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 formula consisting of a disjunction of two or more conjunctive clauses.

When discussing a CNF or DNF formula ψ, V_ψ is used to denote its variable set and C_ψ is used to denote its clause set. The subscripts are dropped when the context is clear.

As mentioned earlier, evaluation of a formula is from innermost paren-

theses 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*.

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 functions is 2^{2^n}. Any Boolean function f on n variables can be expressed as a CNF formula ψ where the set of assignments for which f has value 1 is identical to the set of assignments satisfying ψ [110]. 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 [110].

The partial evaluation of a given formula ψ is possible when a subset of its variables are assigned values. A partial evaluation usually results in the replacement of ψ by a new formula which expresses exactly those assignments satisfying ψ under the given partial assignment. Write $\psi|_{v=1}$ to denote the formula resulting from the partial evaluation of ψ due to assigning value 1 to variable v. An obvious similar statement is used to express the partial evaluation of ψ when v is assigned value 0 or when some subset of variables is assigned values. For example,

$$(v_1 \vee \neg v_2) \wedge (\neg v_1 \vee v_3) \wedge (\neg v_2 \vee \neg v_3)\,|_{v_1=1} \;=\; (v_3) \wedge (\neg v_2 \vee \neg v_3)$$

since $(v_3) \wedge (\neg v_2 \vee \neg v_3)$ expresses all solutions to $(v_1 \vee \neg v_2) \wedge (\neg v_1 \vee v_3) \wedge (\neg v_2 \vee \neg v_3)$ given v_1 has value 1.

If an 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 *satisfied* (respectively, *falsified*) by M. If a clause evaluates to $\perp$ then it is neither satisfied nor falsified.

A formula ψ is *satisfiable* if there exists at least one assignment under which ψ has value 1. In particular, a CNF formula is *satisfiable* if there exists a truth assignment to its variables which satisfies all its clauses. Otherwise, the formula is *unsatisfiable*. Every non-empty DNF formula is satisfiable but a DNF formula that is satisfied by *every* truth assignment to its variables is called a *tautology*. The negation of a DNF tautology is an unsatisfiable CNF formula.

Several assignments may satisfy a given formula. Any satisfying assignment containing the smallest number of variables of value 1 among all satisfying assignments is called a *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[1].

[1]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 [41, 63] which is discussed

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

This section ends with a discussion of formula equivalence. Three types of equivalence are defined along with symbols that are used to represent them as follows:

1. *equality of formulas* ($\psi_1 = \psi_2$): two formulas are *equal* if they are the same string of symbols. Also "=" is used for equality of Boolean values, for example $v = 1$.

2. *logical equivalence* ($\psi_1 \Leftrightarrow \psi_2$): two formulas ψ_1 and ψ_2 are said to be logically equivalent if, for every assignment M to the variables of ψ_1 and ψ_2, M satisfies ψ_1 if and only if M satisfies ψ_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 $(v_2 \vee v_3)$ may be substituted for $(v_1 \vee v_2)$. Therefore, the expression on the left of "$\Leftrightarrow$" is logically equivalent to the expression on the right.

   $$(\neg v_1 \vee v_3) \wedge (v_1 \vee \neg v_3) \wedge (v_1 \wedge v_2) \Leftrightarrow (\neg v_1 \vee v_3) \wedge (v_1 \vee \neg v_3) \wedge (v_2 \wedge v_3)$$

   Another example is:

   $$(v_1 \vee \neg v_2) \wedge (\neg v_1 \vee v_3) \wedge (\neg v_2 \vee \neg v_3) \mid_{v_1=1} \;\Leftrightarrow\; (v_3) \wedge (\neg v_2 \vee \neg v_3)$$

   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.

   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 ψ_1 and ψ_2 are logically equivalent. The latter is just a formula of formal logic upon which one can ask whether there exists a satisfying 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 assignment to the variables of ψ_1 and ψ_2).

   Similarly, define ψ_1 *logically implies* ψ_2 ($\psi_1 \Rightarrow \psi_2$): for every assignment M to the variables in ψ_1 and ψ_2, if M satisfies ψ_1 then M also satisfies ψ_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* ($\psi_1 \leftrightharpoons_V \psi_2$): two formulas ψ_1 and ψ_2, with variable sets V_{ψ_1} and V_{ψ_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 assignment M_V to just the variables of V, either

   (a) there is an assignment M_1 to $V_{\psi_1} \setminus V$ and an assignment M_2 to $V_{\psi_2} \setminus V$ such that $M_V \cup M_1$ satisfies ψ_1 and $M_V \cup M_2$ satisfies ψ_2, or

in Section 5.2. Implications of the minimal model semantics may be found in Section 5.5, among others.

(b) there is no assignment to $V_{\psi_1} \cup V_{\psi_2}$ which contains M_V as a subset and satisfies either ψ_1 or ψ_2.

The assignments $M_V \cup M_1$ and $M_V \cup M_2$ are called *extensions* to the assignment M_V.

The notation $\psi_1 \leftrightharpoons_V \psi_2$ is used to represent the functional equivalence of formulas ψ_1 and ψ_2 with respect to base set V. The notation $\psi_1 \leftrightharpoons \psi_2$ is used if $V = V_{\psi_1} \cap V_{\psi_2}$. In this case logical equivalence and functional equivalence are the same.

For example, $(\neg a) \wedge (a \vee b) \leftrightharpoons_{\{a\}} (\neg a) \wedge (a \vee c)$ because $\exists b : (\neg a) \wedge (a \vee b)$ if and only if $\exists c : (\neg a) \wedge (a \vee c)$. But $(\neg a) \wedge (a \vee b) \neq (\neg a) \wedge (a \vee c)$ since $a = 0$, $b = 1$, $c = 0$ satisfies $(\neg a) \wedge (a \vee b)$ but falsifies $(\neg a) \wedge (a \vee c)$.

Functional equivalence is useful when transforming a formula to a more useful formula. For example, consider the Tseitin transformation [120] (Section 4.3) which extends resolution: for CNF formula ψ containing variables from set V_ψ, any pair of variables $x, y \in V_\psi$, and variable $z \notin V_\psi$,

$$\psi \leftrightharpoons_{V_\psi} \psi \wedge (z \vee x) \wedge (z \vee y) \wedge (\neg z \vee \neg x \vee \neg y).$$

The term "functional equivalence" is somewhat of a misnomer. For any set V of variables, $\leftrightharpoons_V$ is an equivalence relation but $\leftrightharpoons$ is not transitive: for example $a \vee c \leftrightharpoons a \vee b$ and $a \vee b \leftrightharpoons a \vee \neg c$ but $a \vee c \not\leftrightharpoons a \vee \neg c$.

The foundational problem considered in this chapter is the Satisfiability problem, commonly called SAT. It is stated as follows.

Satisfiability (SAT):

GIVEN: A Boolean formula ψ.

QUESTION: Determine whether ψ is satisfied by some truth assignment to the variables of ψ.

For the class of CNF, even 3-CNF, formulas, SAT is $\mathcal{NP} - hard$. However, for the class of CNF formulas of maximum width 2, SAT can be solved in linear time using Algorithm 19 of Section 5.1. For the class of Horn formulas, SAT can be solved in linear time using Algorithm 20 of Section 5.2. If ψ is a CNF formula such that reversing the polarity of some subset of its variables results in a Horn formula, then ψ is *renamable Horn*. Satisfiability of a renamable Horn formula can be determined in linear time by Algorithm 21 of Section 5.4.

A second problem of interest is a generalization of the common problem of determining a minimal model for a given formula. The question of minimal models is considered in Section 5.2.

Variable Weighted Satisfiability:

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

QUESTION: Determine whether ψ is satisfied by some truth assignment and, if so, determine the assignment M such that

$$\sum_{v \in M} w(v) \quad \text{is minimized.}$$

A third problem of interest is the fundamental optimization version of SAT. This problem is probably more important in terms of practical use than even SAT.

Maximum Satisfiability (MAX-SAT):

GIVEN: A CNF formula ψ.

QUESTION: Determine the assignment M that satisfies the maximum number of clauses in ψ.

The following is an important variant of MAX-SAT:

Weighted Maximum Satisfiability (Weighted MAX-SAT):

GIVEN: A CNF formula ψ and a function $w : C_\psi \mapsto Z^+$ where $w(c)$ is the weight of clause $c \in C_\psi$.

QUESTION: Determine the assignment M such that

$$\sum_{c \in \psi} w(c) \cdot s_M(c) \quad \text{is maximized,}$$

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

2 Representations and Structures

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, CNF and DNF formulas will often be represented as sets of clauses and clauses as sets of literals. In this section additional representations are presented; these will later be used to express algorithms and explain algorithmic behavior. Some of these representations involve only a part of a given formula.

$$
\begin{array}{c}
 \\
c_0 \\
c_1 \\
c_2 \\
c_3 \\
c_4 \\
c_5 \\
c_6 \\
c_7
\end{array}
\begin{array}{c}
\begin{array}{cccccccccc}
v_0 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9
\end{array} \\
\left(\begin{array}{cccccccccc}
\text{-1} & 1 & 0 & 0 & 0 & 0 & 0 & \text{-1} & 0 & 0 \\
0 & \text{-1} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \text{-1} & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{-1} & 0 & 0 & \text{-1} & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & \text{-1} & \text{-1} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \text{-1} & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \text{-1} & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right)
\end{array}
$$

$$
\begin{aligned}
&((\neg v_0 \vee v_1 \vee \neg v_7) \wedge (\neg v_1 \vee v_2) \wedge (\neg v_2 \vee v_3) \wedge (\neg v_0 \vee \neg v_3 \vee v_8) \wedge \\
&(v_0 \vee \neg v_6 \vee \neg v_7) \wedge (\neg v_5 \vee v_6) \wedge (\neg v_4 \vee v_5 \vee v_9) \wedge (v_0 \vee v_4))
\end{aligned}
$$

Figure 1: *A* $(0, \pm 1)$ *matrix representation and associated CNF formula where clauses are labeled* c_0, c_1,...,c_7, *from left to right, top to bottom*

2.1 $(0, \pm 1)$ 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 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 ψ can be cast as an Integer Program as follows:

$$
\begin{aligned}
\mathcal{M}_\psi \boldsymbol{\alpha} \ + \ \boldsymbol{b} \ &\geq \ \boldsymbol{Z}\,, \\
\alpha_i \in \{0, 1\}, \qquad &\text{for all } 0 \leq i < n,
\end{aligned}
\tag{1}
$$

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

In addition to the well-known matrix operations, two matrix operations that are relevant to SAT algorithms on $(0, \pm 1)$ matrix representations are applied in this chapter. The first is column scaling: a column may be multiplied or *scaled* by -1 which 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. The second is row and column reordering: rows and columns may be *permuted* with the effect on a solution being only a possible relabeling of variables taking value 1.

2.2 Binary Decision Diagrams

Binary Decision Diagrams [3, 85] 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 ([21, 22]) 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, logical or, and logical and, among others, is efficient with respect to the size of the BDD operands. Typically, a given formula ψ is represented 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 ψ. Intermediate BDDs are created in the process. Unfortunately, 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

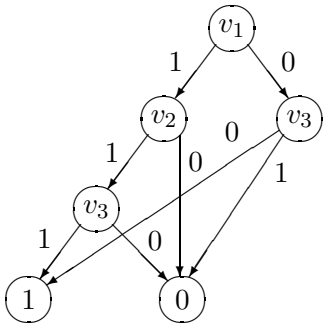


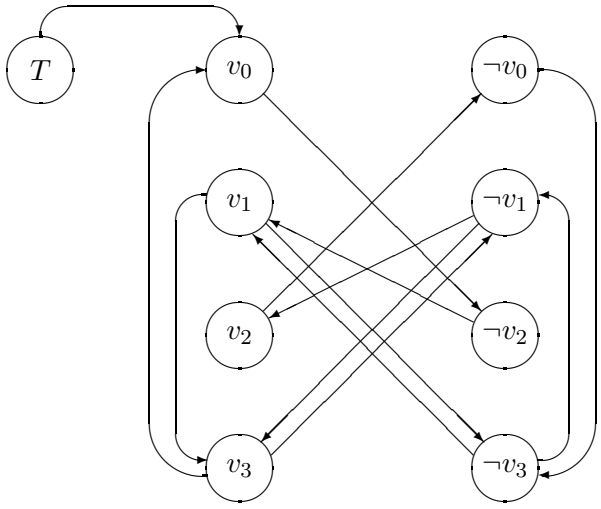
Figure 2: The formula $(v_1 \vee \neg v_3) \wedge (\neg v_1 \vee v_2) \wedge (\neg v_1 \vee \neg v_2 \vee v_3)$ represented as a BDD. The topmost vertex is the root. The two bottom vertices are terminal vertices. Edges are directed from upper vertices to lower vertices. Vertex labels (variable names) are shown inside the vertices. The 0 branch out of a vertex labeled v means v takes the value 0. The 1 branch out of a vertex labeled v means v takes the value 1. The *index* of a vertex is, in this case, the subscript of the variable labeling that 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 $index(v) < \min\{index(then(v)), index(else(v))\}$ and $index(v) = index(w)$ 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.

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 [21, 22]. 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 is the root of a BDD that represents a Boolean function that is unique with respect to all other vertices. Two Boolean functions are equivalent if their reduced ordered BDDs are isomorphic. A more detailed explanation is given in Section 4.9.

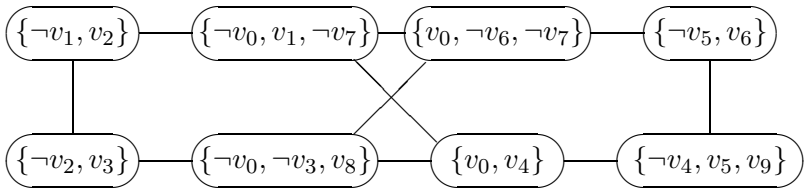
2.3 Implication Graph

An *implication graph* of a CNF formula ψ is a directed graph $\vec{G}_\psi(V, \vec{E})$ where V consists of one special vertex T which corresponds to the value 1, other vertices which correspond to the literals of ψ, and the edge set $\vec{E}$ such that



$$((v_0) \wedge (\neg v_0 \vee \neg v_2) \wedge (v_1 \vee v_2) \wedge (\neg v_1 \vee \neg v_3) \wedge (\neg v_1 \vee v_3) \wedge (v_1 \vee v_3) \wedge (v_0 \vee \neg v_3))$$

Figure 3: *An implication graph and associated 2-CNF formula.*



$$((\neg v_0 \vee v_1 \vee \neg v_7) \wedge (\neg v_1 \vee v_2) \wedge (\neg v_2 \vee v_3) \wedge (\neg v_0 \vee \neg v_3 \vee v_8) \wedge (v_0 \vee \neg v_6 \vee \neg v_7) \wedge (\neg v_5 \vee v_6) \wedge (\neg v_4 \vee v_5 \vee v_9) \wedge (v_0 \vee v_4))$$

Figure 4: *A propositional connection graph and associated CNF formula.*

there is an edge $\langle v_i, v_j \rangle \in \vec{E}$ if and only if there is a clause $(\neg v_i \vee v_j)$ in ψ, and an edge $\langle T, v_i \rangle \in \vec{E}$ ($\langle T, \neg v_i \rangle \in \vec{E}$) if and only if there is a unit clause (v_i) (respectively, $(\neg v_i)$) in ψ. Figure 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 $(\neg v_i \vee v_j)$ will be falsified.*

2.4 Propositional Connection Graph

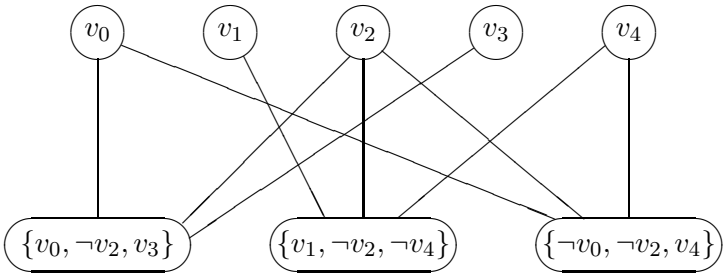
A *propositional connection graph* for a CNF formula ψ is an undirected graph $G_\psi(V, E)$ whose vertex set corresponds to the clauses of ψ, and whose edge set is such that there is an edge $\{c_i, c_j\} \in E$ if and only if the clause in ψ represented by vertex c_i has a literal that appears negated 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 4. Propositional connection graphs are a specialization of the first-order connection graphs developed by Kowalski [84].

2.5 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 $v_i \in V_1$ 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 5.

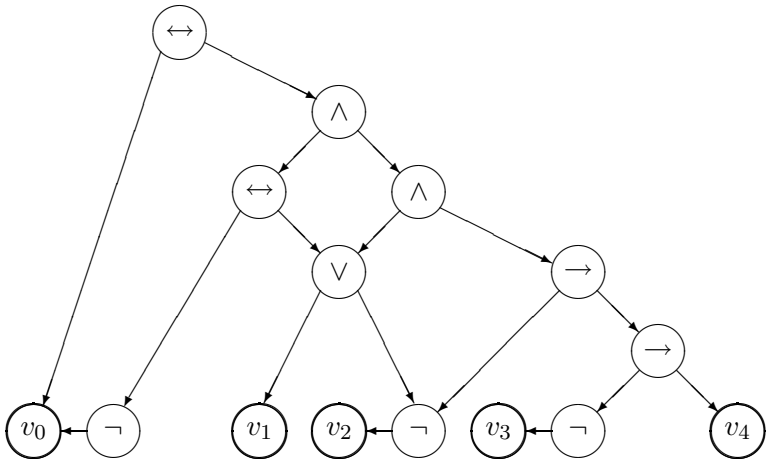
2.6 Formula Digraph

Formulas are defined recursively on page 1. Any Boolean formula can be adapted to fit this definition with the suitable addition of parentheses and its structure (as opposed to its functionality) can be represented by a binary



$$((v_0 \vee \neg v_2 \vee v_3) \wedge (v_1 \vee \neg v_2 \vee \neg v_4) \wedge (\neg v_0 \vee \neg v_2 \vee v_4))$$

Figure 5: *A variable-clause matching graph and associated CNF formula.*



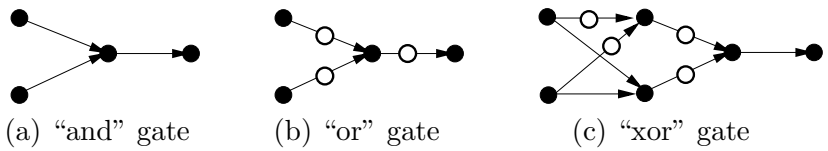
$$v_0 \leftrightarrow ((\neg v_0 \leftrightarrow (v_1 \vee \neg v_2)) \wedge (v_1 \vee \neg v_2) \wedge (\neg v_2 \rightarrow \neg v_3 \rightarrow v_4))$$

Figure 6: *A formula digraph and associated formula.*

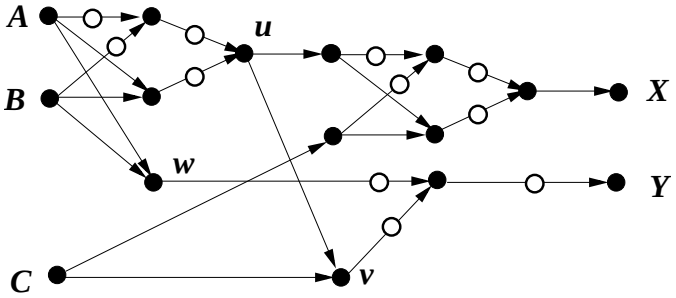
rooted acyclic digraph. In such a digraph, each internal vertex corresponds to a binary operator or a unary operator $\neg$ that occurs in the formula. A vertex corresponding to a binary operator has two outward oriented edges: the left edge corresponds to the left subformula and the right edge to the 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. Call such a representation a *wff digraph*. An example is given in Figure 6. An efficient algorithm for constructing a wff digraph is given in Section 4.1.

2.7 Satisfiability Index

Let ψ be a CNF formula and let $\mathcal{M}_\psi$ be its $(0, \pm 1)$ matrix representation. Let $\boldsymbol{\alpha} = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$ be an n dimensional vector of real variables. Let z be a real variable and let $\boldsymbol{Z} = (z, z, \ldots, z)$ be an m dimensional vector where every component is the variable z. Finally, let $\boldsymbol{b} =$



(a) "and" gate (b) "or" gate (c) "xor" gate



(d) one-bit "add" circuit of Figure 9

Figure 7: Examples of AIGs. Edges with white circles are negated. Edges without a white circle are not negated.

$(b_0, b_1, \ldots, b_{m-1})$ 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

$$\begin{aligned} \mathcal{M}_\psi \boldsymbol{\alpha} \ + \ \boldsymbol{b} \ \leq \ \boldsymbol{Z}\,, & \qquad (2) \\ 0 \leq \alpha_i \leq 1 \qquad \text{for all } 0 \leq i < n. & \end{aligned}$$

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

$$((x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3 \vee x_5) \wedge (x_3 \vee \neg x_4 \vee \neg x_5) \wedge (x_4 \vee \neg x_1))$$

has satisfiability index of 5/4.

2.8 And/Inverter Graphs

An AIG is a directed acyclic graph where all "gate" vertices have in-degree 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: this distinguishes them from non-negated edges which are unlabeled. Examples are shown in Figure 7.

3 Applications

This section presents a sample of real-world problems that may be viewed as, or transformed to, instances of SAT. Of primary interest is to explore the thinking process required for setting up logic problems, and gauging the complexity of the resulting systems. Since it is infeasible to meet this objective *and* thoroughly discuss a large number of known applications, a small but varied and interesting collection of problems are discussed.

3.1 Consistency Analysis in Scenario Projects

This application, taken from the area of scenario management [5, 42, 51], is contributed by Feldmann and Sensen [45] of Burkhard Monien's PC^2 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 [50] 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, consider a simplified example from [45] 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, only the first 10 are shown. 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 8, 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 [45]. But transforming to CNF formulas, in some cases with weights on proposition letters, and solving a variant of SAT, is sometimes a reasonable way to tackle such problems. This context provides 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. It is next shown how to construct representative formulas so that, often, a large subset of clauses is polynomial time solvable and the whole formula is 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

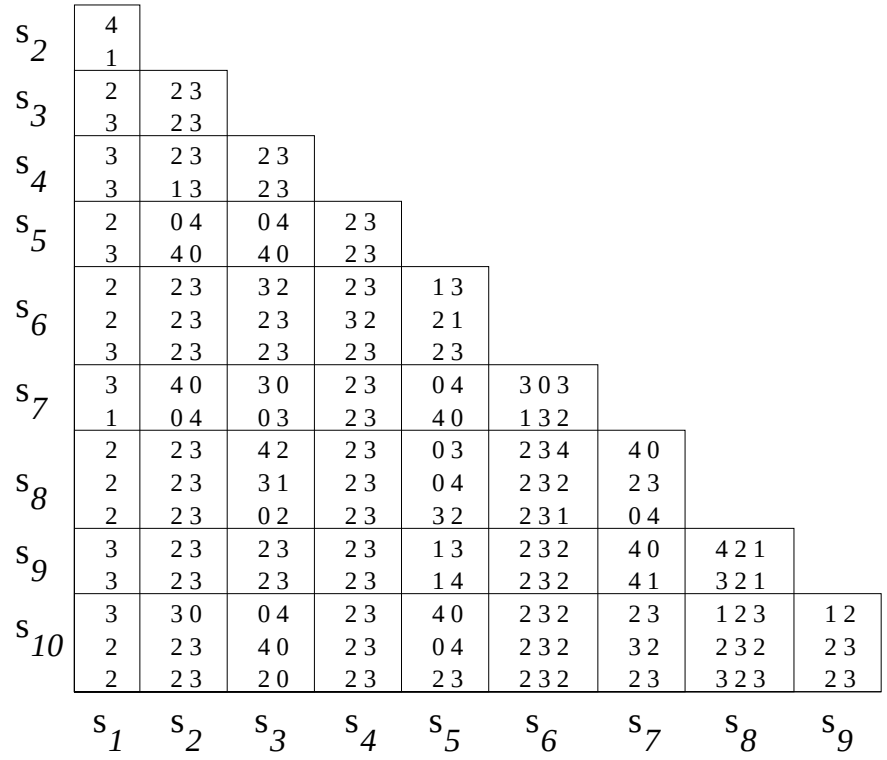


Figure 8: *A consistency matrix for a scenario project.*

Clause of $\psi^{S,\mathcal{D}}$	**Subscript Range**	**Meaning**
$(\neg v_{i,j} \vee \neg v_{i,k})$	$1 \leq i \leq n, 1 \leq j < k \leq m_i$	≤ 1 development/key factor
$(v_{i,1} \vee \ldots \vee v_{i,m_i})$	$1 \leq i \leq n$	≥ 1 development/key factor
$(\neg v_{i,k} \vee \neg v_{j,l})$	$i,j,k,l : C_{i.k,j.l} = 0$	Consistent developments only

Table 3: Formula to determine existence of a consistent bundle.

let $\mathcal{D} = \cup_{i=1}^{n} D_i$. For each future development $d_{i.j}$, 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 $\psi^{S,\mathcal{D}}$ with the clauses described in Table 3 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 to be considered 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 $\psi^{S,\mathcal{D}}$. A simple inclusion-exclusion algorithm exhibits very good performance for some actual problems of this sort [45].

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 (defined on Page 6). The transformed formula consists of $\psi^{S,\mathcal{D}}$ 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 $\psi^{S,\mathcal{D}}$:

$$(\neg v_{i,k} \vee \neg v_{j,l} \vee p_{i,k,j,l}) \wedge (v_{i,k} \vee \neg p_{i,k,j,l}) \wedge (v_{j,l} \vee \neg p_{i,k,j,l}).$$

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.

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. It should be pointed out that, although the number of clauses added to $\psi^{S,\mathcal{D}}$ might be significant compared to the number of clauses originally in $\psi^{S,\mathcal{D}}$, 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 4.11.

3.2 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 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 9. For the sake of discussion, 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 10 where a and b are gate inputs and c is a gate output. Suppose none of the interconnect points enclosed by the dashed line of Figure 9 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 $\psi_1 = (A \wedge B)$. 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 $\psi_2 = (\neg C \vee (A \wedge B) \vee (\neg A \wedge \neg B))$. The conjunction $\psi_1 \wedge \psi_2$ is the

[2]Horn formulas are solved efficiently (see Section 5.2)

[3]Actual circuit voltage levels are abstracted to the values 0 and 1

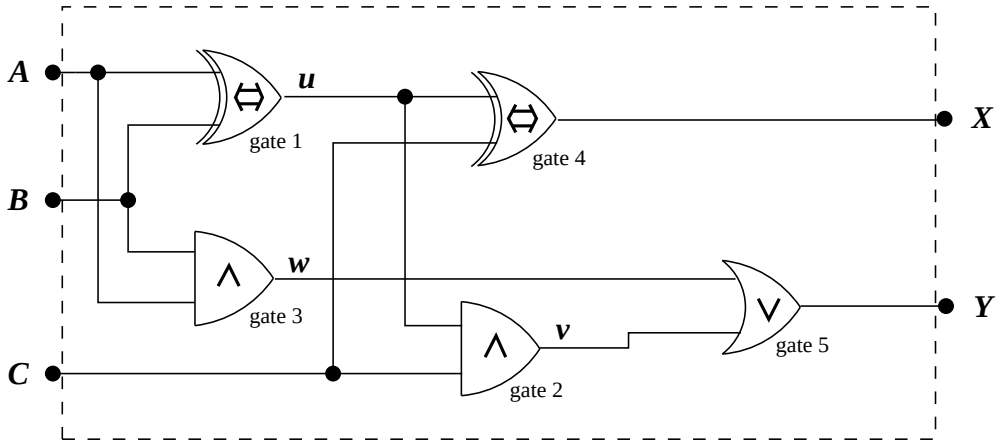
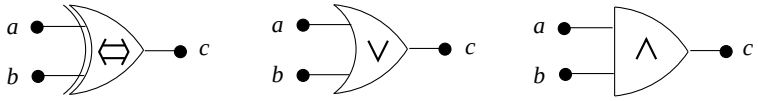


Figure 9: *A 1-bit full adder circuit.*



a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

a	b	c
0	0	0
0	1	1
1	0	1
1	1	1

a	b	c
0	0	0
0	1	0
1	0	0
1	1	1

Figure 10: *Truth tables for logic elements of a 1-bit full adder.*

CNF expression $(A \wedge B)$. This expression is satisfied if and only if A and B are set to 1. Such an input will cause output Y to have 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 continue to increase dramatically.

3.3 Diagnosis of Circuit Faults

A natural extension of the test design discussed in Section 3.2 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 malfunction? 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? The first of these questions is discussed in this section.

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 9. The expression is assembled in stages, one for each gate, starting with gate 3 of Figure 9, which is an *and*-gate. The behavior of a correctly functioning *and*-gate is specified in the right-hand truth table in Figure 10. The following formula expresses the truth table:

$$(a \wedge b \wedge c) \ \vee (\neg a \wedge \neg b \wedge c) \vee (\neg a \wedge b \wedge \neg c) \vee (\neg a \wedge \neg b \wedge \neg c)$$

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,b, and c, to get the following abnormality expression:

$$(A \wedge B \wedge w \wedge \neg Ab_3) \ \vee (\neg A \wedge \neg B \wedge w \wedge \neg Ab_3) \vee$$
$$(\neg A \wedge B \wedge \neg w \wedge \neg Ab_3) \vee (\neg A \wedge \neg B \wedge \neg w \wedge \neg Ab_3) \vee (\neg w \wedge Ab_3)$$

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 Δ 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 the observed output values are $Y = 0$ and $X = 1$ whereas $Y = 1$ and $X = 0$ are the correct outputs. One 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. Suppose gate 4 is working correctly. Since the only gate that gate 4 depends on is gate 1, that gate must be stuck. One cannot tell whether gates 2,3,5 are functioning; normally it would be assumed that they are functioning correctly until evidence to the contrary is obtained. Thus the natural diagnosis is gate 1 is defective (only Ab_1 has value 1).

Alternatively, suppose under the same inputs it is observed that $X, Y = 0$. 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

IDs	Ab_1	Ab_2	Ab_3	Ab_4	Ab_5
1-16	*	*	*	*	1
17-24	*	1	*	*	0
25-26	1	0	*	1	0

Table 4: Possible stuck-at 0 failures of 1-bit adder gates (see Figure 9) assuming given inputs are $A = 0$ and $B, C = 1$ and observed outputs are $X, Y = 0$. This is the list of assignments to Δ 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'.

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 Δ values that "witness" the observed outputs: these are summarized in Table 4, grouped into three cases. In the first two of these cases the minimum number of gates stuck at 0 is 1.

As before, *it is assumed that the set of malfunctioning gates is as small as possible*, so only those two diagnoses are considered: 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*[99]:

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

This sort of inference is called *non-monotonic* because it is inferred, above, that gate 3 was functioning correctly, since there is no evidence it was not. Later evidence may cause that inference to be withdrawn.

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., [53]). Although worst-case performance of such systems is provably bad, such a system can be useful in many circumstances. Unfortunately, implementations of non-monotonic inference are new enough that there is not yet a sufficiently large body of standard benchmark examples.

3.4 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. Bread-boarding 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 (defined on Page 5) between levels amounts to proving a theorem of the form *implementation I realizes specification S* in a particular, suitable formal proof system. For illustration purposes only, consider the 1-bit full adder of Figure 9. 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:

$$(X \Leftrightarrow (A \wedge \neg B \wedge \neg C) \vee (\neg A \wedge B \wedge \neg C) \vee (\neg A \wedge \neg B \wedge C) \vee (A \wedge B \wedge C)) \wedge$$
$$(Y \Leftrightarrow (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)).$$

A proposed implementation of this specification is given in the dotted region of Figure 9. 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:

$$\begin{aligned}
&(u \Leftrightarrow (A \wedge \neg B) \vee (\neg A \wedge B)) \wedge \\
&(v \Leftrightarrow u \wedge C) \wedge \\
&(w \Leftrightarrow A \wedge B) \wedge \\
&(X \Leftrightarrow (u \wedge \neg C) \vee (\neg u \wedge C)) \wedge \\
&(Y \Leftrightarrow w \vee v).
\end{aligned}$$

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:

$$\psi_S(A, B, C, X, Y) \Leftrightarrow \exists u, v, w \ \psi_I(A, B, C, X, Y, u, v, w),$$

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 values for u, v, w which make the formula have value 1 be found? (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:

$$\forall A, B, C, X, Y(\psi_S(A, B, C, X, Y) \Leftrightarrow \exists u, v, w \ \psi_I(A, B, C, X, Y, u, v, w)),$$

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*.

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 [129]. For illustrative purposes one of these is discussed below, namely the Linear Time Temporal Logic (LTTL). LTTL formulas take value 1 with respect to an infinite sequence of states $S = \{s_0, s_1, s_2, ...\}$. 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 an assignment of values to a set 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 5 along with the standard propositional operators $\neg$, $\wedge$, and $\vee$ (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.

Op. name	$(\mathbf{S}, \mathbf{s_i}) \models$	**if and only if**
	p (p a variable)	$s_i(p) = 1$.
not	$\neg\psi_1$	$(S, s_i) \not\models \psi_1$
and	$\psi_1 \wedge \psi_2$	$(S, s_i) \models \psi_1$ and $(S, s_i) \models \psi_2$
or	$\psi_1 \vee \psi_2$	$(S, s_i) \models \psi_1$ or $(S, s_i) \models \psi_2$
henceforth	$\Box\psi_1$	$(S, s_j) \models \psi_1$ for all states $s_j, j \geq i$.
eventually	$\diamond\psi_1$	$(S, s_j) \models \psi_1$ for some state $s_j, j \geq i$.
next	$\circ\psi_1$	$(S, s_{i+1}) \models \psi_1$.
until	$\psi_1 \mathcal{U} \psi_2$	For some $j \geq i$, $(S, s_i), (S, s_{i+1}), \ldots, (S, s_{j-1}) \models \psi_1$, and $(S, s_j) \models \psi_2$.

Table 5: The operators of temporal logic

Let ψ be a LTTL 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 $\diamond(J \wedge Q)$. The event that ψ has value 1 for state s_i in S is denoted by

$$(S, s_i) \models \psi.$$

Also, say

$$S \models \psi \text{ if and only if } (S, s_0) \models \psi.$$

Finally, two LTTL formulas ψ_1 and ψ_2 are *equivalent* if, for all sequences S,

$$S \models \psi_1 \text{ if and only if } S \models \psi_2.$$

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 11 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 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 6. Observe that the first three expressions of Table 6 represent assumptions needed for the latch to work correctly and do not necessarily reflect

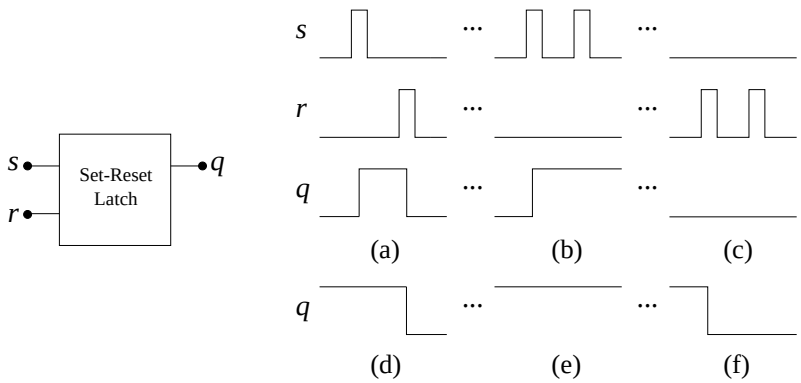


Figure 11: *A set-reset latch and complete description of signal behavior. Inputs are s and r, output is q. The horizontal axis represents time. The vertical axis represents signal value for both inputs and output. Values of all signals are assumed to be either 0 (low) or 1 (high) at any particular time. Two rows for q values are used to differentiate between the cases where the initial value of q is 0 or 1.*

Expressions	**Comments**
$\Box\neg(s \wedge r)$	No two inputs have value 1 simultaneously.
$\Box((s \wedge \neg q) \rightarrow ((s \,\mathcal{U}\, q) \vee \Box s))$	Input s cannot change if s is 1 and q is 0.
$\Box((r \wedge q) \rightarrow ((r \,\mathcal{U}\, \neg q) \vee \Box r))$	Input r cannot change if r is 1 and q is 1.
$\Box(s \rightarrow \Diamond q)$	If s is 1, q will eventually be 1.
$\Box(r \rightarrow \Diamond \neg q)$	If r is 1, q will eventually be 0.
$\Box((\neg q \rightarrow ((\neg q \,\mathcal{U}\, s) \vee \Box \neg q)))$	Output q rises to 1 only if s becomes 1.
$\Box((q \rightarrow ((q \,\mathcal{U}\, r) \vee \Box q)))$	Output q drops to 0 only if r becomes 1.

Table 6: Temporal logic formula for a set-reset latch

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 11, that satisfy the formula of Table 6 are:

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

Clearly, infinitely many sequences satisfy the formula of Table 6, 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 [129], Chapter 6 for details and citations.

3.5 Bounded Model Checking

Section 3.4 considered solving verification problems of the form $S \models \psi_1 \equiv S \models \psi_2$. If it is desired instead to determine whether there exists an S such that $S \models \psi$ temporal operators can be traded for Boolean variables, the sentence can be expressed as a propositional formula, and a SAT solver applied. 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 *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_1^i and v_2^i whose values are intended to be the same as those of variables v_1 and v_2, respectively, on the ith time step. Suppose the starting state is the case where both v_1^0 and v_2^0 have value 0. The transition relation is

Current Output		**Next Output**
00	:	01
01	:	10
10	:	11
11	:	00

the i^{th} line of which can be expressed as the following Boolean function:

$$(v_1^{i+1} \leftrightarrow \neg v_1^i) \wedge (v_2^{i+1} \leftrightarrow v_1^i \oplus v_2^i).$$

Suppose the time-dependent expression to be proved is:

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:**

$$(\neg(v_1^0 \wedge v_2^0) \wedge \neg(v_1^1 \wedge v_2^1) \wedge \neg(v_1^2 \wedge v_2^2) \wedge (v_1^3 \wedge v_2^3))$$

2. **Express the starting state:**

$$(\neg v_1^0 \wedge \neg v_2^0)$$

3. **Force legal transitions (repetitions of the transition relation):**

$$\begin{array}{l}(v_1^1 \leftrightarrow \neg v_1^0) \wedge (v_2^1 \leftrightarrow v_1^0 \oplus v_2^0) \wedge \\ (v_1^2 \leftrightarrow \neg v_1^1) \wedge (v_2^2 \leftrightarrow v_1^1 \oplus v_2^1) \wedge \\ (v_1^3 \leftrightarrow \neg v_1^2) \wedge (v_2^3 \leftrightarrow v_1^2 \oplus v_2^2)\end{array}$$

Since $(a \leftrightarrow b) \Leftrightarrow (a \vee \neg b) \wedge (\neg a \vee b)$, 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_1^i and v_2^i, $0 \leq i \leq 3$, satisfies the above expressions. Information on the use and success of Bounded Model Checking may be found in [12, 26].

3.6 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 [74, 76, 77], by incrementally building a single-output And/Inverter Graph (AIG), representing the mitre [20] 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 [20, 75] and then checked by a SAT 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 [52] and avoids testing all possible input-output combinations, which is $\mathcal{CoNP}$-hard.

In CEC the AIG is incrementally developed from gate level representations. For example, Figure 7(a) shows the AIG for an "and" gate, Figure 7(b) shows the AIG for an "or" gate, Figure 7(c) shows the AIG for "exclusive-or" and Figure 7(d) shows the AIG for the adder circuit of Figure 9. 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 12(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 12(b) shows the first two nodes of the circuit being merged with the AIG and Figure 12(c) shows the completed AIG.

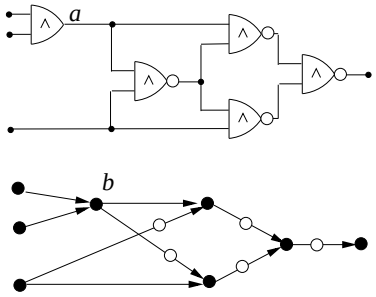
CEC proceeds with the application of many random input vectors to the input vertices of the AIG for the purpose of partitioning the vertices into potential equivalence classes (two vertices are in the same equivalence class if they take the same value under all random input vectors). In the case of Figure 12(c), suppose the potential equivalence classes are $\{1, 2\}, \{3, 4\}, \{5, 6\}$.

The next step is to use a SAT 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.

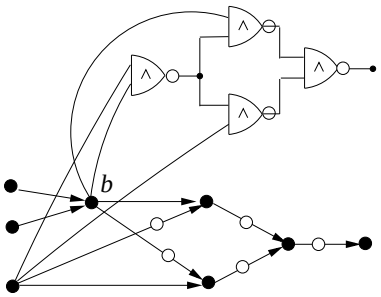
Above, CEC has been shown to check the equivalence of two combinational circuits but it can also be used for checking a specification against a circuit implementation or for reverse engineering a circuit.

3.7 Transformations to Satisfiability

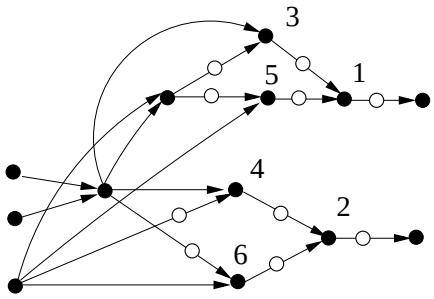
It is routine in the operations research community to transform a given optimization problem into another, solve the new problem, and use the solution to construct a solution or an approximately optimal solution for the given problem. Usual targets of transformations are Linear Programming and Network 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 a successful transformation from Network Steiner Tree Problems to weighted MAX-SAT problems is considered (taken from [68]).



(a) The beginning of the equivalency check - one circuit has been transformed to an AIG.



(b) Node a of the circuit (Figure 12(a)) has been merged with vertex b of the AIG.



(c) The completed AIG.

Figure 12: Creating the AIG.

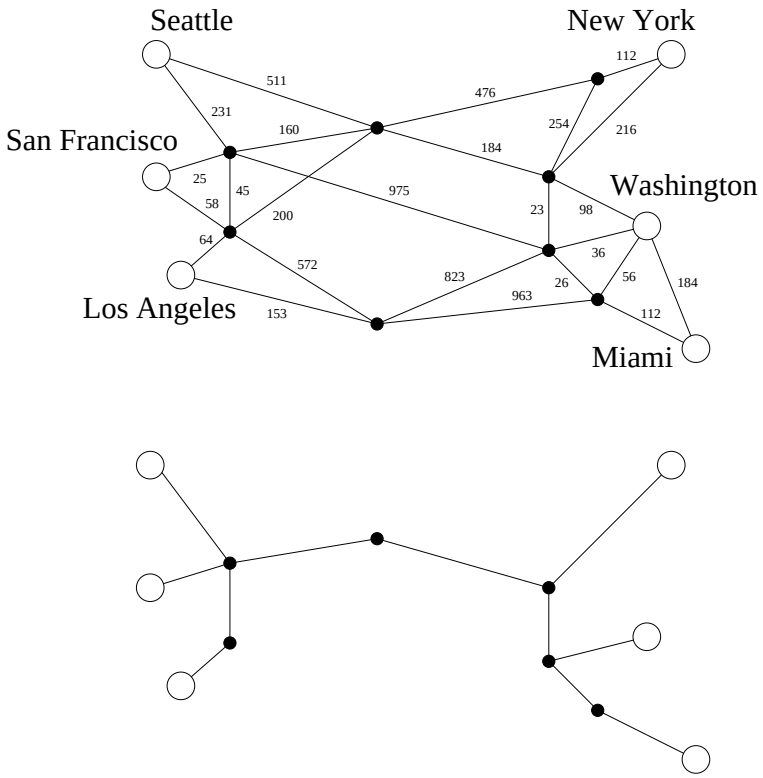


Figure 13: *An example of a Network Steiner Tree Problem (top) and its optimal solution (bottom). The white nodes are terminals and the numbers represent hypothetical costs of laying fiber-optic cables between pairs of cities.*

The Network Steiner Tree Problem originates from the following important network cost problem (see, for example, [1]). 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?*

This problem can be formalized as follows. Let $G(V, E)$ be a graph whose vertex set $V = \{c_1, c_2, c_3, \dots\}$ represents all cities and whose edge set E represents all possible connections between pairs of cities. Let $w : E \mapsto Z^+$ 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 13. This problem is one of the first shown to be $\mathcal{NP}$-complete [67]. The problem appears in many applications and has been extensively studied. Many enumeration algorithms, heuristics and approximation algorithms are known (see [1] 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 ψ with non-negative weights on its clauses. A necessary (feasibility) property of any transformation is that a truth assignment to the variables of ψ 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, consider the linear transformation introduced in [68]. This transformation has a positive integer parameter k which controls the quality of the approximation. 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:

Variable	Subscript Range	Meaning
$e_{i,j}$	edge $\{c_i, c_j\} \in E$	$\{c_i, c_j\} \in$ the Steiner Tree
$p_{i,j,l}$	$\{c_i, c_j\} \in W$, $1 \leq l \leq k$	$P_{i,j,l} \subseteq$ the Steiner Tree

Clause	Subscript Range	Meaning
$(\neg e_{i,j})$	$\{c_i, c_j\} \in E$	$e_{i,j} \notin$ Steiner Tree
$(p_{i,j,1} \vee \ldots \vee p_{i,j,k})$	$\{c_i, c_j\} \in W$	Include at least one of k shortest paths between c_i, c_j.
$(\neg p_{i,j,l} \vee e_{m,n})$	$\{c_m, c_n\} \in P_{i,j,l}$	Include all edges of that path.

Clause	Weight	Significance
$(\neg e_{i,j})$	$w(\{i,j\})$	Maximize weights of edges **not** included
$(p_{i,j,1} \vee \ldots \vee p_{i,j,k})$	I	Force connected subgraph of G containing all vertices of T
$(\neg p_{i,j,l} \vee e_{m,n})$	I	Force connected subgraph of G containing all vertices of T

Table 7: 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 7. 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 7 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 (see Section 4.8.1) or even a branch-and-bound variant (see Section 4.11) can be applied to obtain a solution. The reader is referred to [68] for empirical results showing speed and approximation quality.

3.8 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 cal-

culations. 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:

Vector	Value	Overheating	Coolant Level
00	0	No	Low
01	0	No	Normal
10	0	Yes	Low
11	1	Yes	Normal

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

$$(v_1 \wedge v_2 \wedge \neg v_3) \vee (\neg v_1 \wedge v_4) \vee (v_3 \wedge v_4 \wedge v_5 \wedge v_6) \vee (\neg v_3).$$

DNF formulas have the same form as that of CNF formulas except that the positions of the $\wedge$'s and the $\vee$'s are reversed. In this case the formula is in 4-DNF since each disjunct contains at most 4 conjuncts. Each *term*, e.g., $(v_1 \wedge v_2 \wedge \neg v_3)$ 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.*, [121]). 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 [19]. 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 [18] 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.

4 General Algorithms

In this section some algorithms for solving various SAT problems are presented. 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.

4.1 Efficient Transformation to CNF Formulas

An algorithm due to Tseitin [120] efficiently transforms an arbitrary Boolean formula ϕ to a CNF formula ψ such that ψ has a model if and only if ϕ has a model and if a model for ψ exists, it is a model for ϕ. 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 2.6, any Boolean formula ϕ can be represented as a binary rooted acyclic digraph W_ϕ where each internal vertex represents some operation on one or two operands (an example is given in Figure 6). Associate with each internal vertex x of W_ϕ a new variable v_x not occurring in ϕ. If x represents a binary operator $\mathcal{O}_x$, 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_ϕ write

$$\begin{array}{ll} v_x \Leftrightarrow (v_l \ \mathcal{O}_x \ v_r) & \text{if } \mathcal{O}_x \text{ is binary, or} \\ v_x \Leftrightarrow \neg v_g & \text{if vertex } x \text{ represents } \neg. \end{array}$$

For each equivalence there is a short, functionally equivalent CNF expression. The table of Figure 14 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. The equivalent CNF expression for $\neg$ is $(v_g \vee v_x) \wedge (\neg v_g \vee \neg v_x)$.

The target of the transformation is a CNF formula consisting of all clauses in every CNF expression from Figure 14 that corresponds to an

$\mathcal{O}_x$	**Equivalent CNF Expression**	**Comment**
0000	$(\neg v_x)$	$v_x \Leftrightarrow 0$
1111	(v_x)	$v_x \Leftrightarrow 1$
0011	$(v_l \vee \neg v_x) \wedge (\neg v_l \vee v_x)$	
1100	$(v_l \vee v_x) \wedge (\neg v_l \vee \neg v_x)$	$v_x \Leftrightarrow \neg v_l$
0101	$(v_r \vee \neg v_x) \wedge (\neg v_r \vee v_x)$	
1010	$(v_r \vee v_x) \wedge (\neg v_r \vee \neg v_x)$	
0001	$(v_l \vee \neg v_x) \wedge (v_r \vee \neg v_x) \wedge (\neg v_l \vee \neg v_r \vee v_x)$	$v_x \Leftrightarrow (v_l \wedge v_r)$
1110	$(v_l \vee v_x) \wedge (v_r \vee v_x) \wedge (\neg v_l \vee \neg v_r \vee \neg v_x)$	$v_x \Leftrightarrow (\neg v_l \vee \neg v_r)$
0010	$(\neg v_l \vee \neg v_x) \wedge (v_r \vee \neg v_x) \wedge (v_l \vee \neg v_r \vee v_x)$	
1101	$(v_l \vee v_x) \wedge (\neg v_r \vee v_x) \wedge (\neg v_l \vee v_r \vee \neg v_x)$	$v_x \Leftrightarrow (v_l \rightarrow v_r)$
0100	$(v_l \vee \neg v_x) \wedge (\neg v_r \vee \neg v_x) \wedge (\neg v_l \vee v_r \vee v_x)$	
1011	$(\neg v_l \vee v_x) \wedge (v_r \vee v_x) \wedge (v_l \vee \neg v_r \vee \neg v_x)$	$v_x \Leftrightarrow (v_l \leftarrow v_r)$
1000	$(\neg v_l \vee \neg v_x) \wedge (\neg v_r \vee \neg v_x) \wedge (v_l \vee v_r \vee v_x)$	$v_x \Leftrightarrow (\neg v_l \wedge \neg v_r)$
0111	$(\neg v_l \vee v_x) \wedge (\neg v_r \vee v_x) \wedge (v_l \vee v_r \vee \neg v_x)$	$v_x \Leftrightarrow (v_l \vee v_r)$
1001	$(v_l \vee \neg v_r \vee \neg v_x) \wedge (\neg v_l \vee v_r \vee \neg v_x) \wedge$ $(\neg v_l \vee \neg v_r \vee v_x) \wedge (v_l \vee v_r \vee v_x)$	$v_x \Leftrightarrow (v_l \leftrightarrow v_r)$
0110	$(\neg v_l \vee v_r \vee v_x) \wedge (v_l \vee \neg v_r \vee v_x) \wedge$ $(v_l \vee v_r \vee \neg v_x) \wedge (\neg v_l \vee \neg v_r \vee \neg v_x)$	$v_x \Leftrightarrow (v_l \oplus v_r)$

Figure 14: *CNF expressions equivalent to* $v_x \Leftrightarrow (v_l \ \mathcal{O}_x \ v_r)$ *for* $\mathcal{O}_x$ *as shown.*

equivalence expressed at a non-leaf vertex of W_ϕ plus a unit clause that forces the root expression to evaluate to 1. For example, the expression of Figure 6, namely

$$v_0 \leftrightarrow ((\neg v_0 \leftrightarrow (v_1 \vee \neg v_2)) \wedge (v_1 \vee \neg v_2) \wedge (\neg v_2 \rightarrow \neg v_3 \rightarrow v_4)),$$

transforms to

$(v_0 \vee v_{x_1}) \wedge (\neg v_0 \vee \neg v_{x_1}) \wedge$
$(v_2 \vee v_{x_2}) \wedge (\neg v_2 \vee \neg v_{x_2}) \wedge$
$(v_3 \vee v_{x_3}) \wedge (\neg v_3 \vee \neg v_{x_3}) \wedge$
$(\neg v_1 \vee v_{x_4}) \wedge (\neg v_{x_2} \vee v_{x_4}) \wedge (v_1 \vee v_{x_2} \vee \neg v_{x_4}) \wedge$
$(v_{x_3} \vee v_{x_5}) \wedge (\neg v_4 \vee v_{x_5}) \wedge (\neg v_{x_3} \vee v_4 \vee \neg v_{x_5}) \wedge$
$(v_{x_1} \vee \neg v_{x_4} \vee \neg v_{x_6}) \wedge (\neg v_{x_1} \vee v_{x_4} \vee \neg v_{x_6}) \wedge (\neg v_{x_1} \vee \neg v_{x_4} \vee v_{x_6}) \wedge (v_{x_1} \vee v_{x_4} \vee v_{x_6}) \wedge$
$(v_{x_2} \vee v_{x_7}) \wedge (\neg v_{x_5} \vee v_{x_7}) \wedge (\neg v_{x_2} \vee v_{x_5} \vee \neg v_{x_7}) \wedge$
$(v_{x_4} \vee \neg v_{x_8}) \wedge (v_{x_7} \vee \neg v_{x_8}) \wedge (\neg v_{x_4} \vee \neg v_{x_7} \vee v_{x_8}) \wedge$
$(v_{x_6} \vee \neg v_{x_9}) \wedge (v_{x_8} \vee \neg v_{x_9}) \wedge (\neg v_{x_6} \vee \neg v_{x_8} \vee v_{x_9}) \wedge$
$(v_0 \vee \neg v_{x_9} \vee \neg v_{x_{10}}) \wedge (\neg v_0 \vee v_{x_9} \vee \neg v_{x_{10}}) \wedge (\neg v_0 \vee \neg v_{x_9} \vee v_{x_{10}}) \wedge (v_0 \vee v_{x_9} \vee v_{x_{10}}) \wedge$
$(v_{x_{10}})$

where each line except the last corresponds to an internal vertex x_i of W_ϕ of Figure 6, i increasing from 1 in left-to-right and bottom-to-top order, and the new variables labeling those vertices are $v_{x_1}, \ldots, v_{x_{10}}$ correspondingly. The last line forces the root expression to evaluate to 1. The algorithm of Figure 15 expresses these ideas formally. For simplicity, it is assumed that the input is a formula with syntax consistent with that described on Page 1.

The next lemma and theorems show that the algorithm correctly transforms a given expression to a CNF formula and size of the target formula is linearly related to the size of the original expression.

Lemma 1. *Let $\mathcal{O}$ 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 ψ_1 and ψ_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 $\psi_1 \mid_{v_1=1} \leftrightharpoons_V \psi_1 \mid_{v_1=0}$ and $\psi_2 \mid_{v_2=1} \leftrightharpoons_V \psi_2 \mid_{v_2=0}$,*

$$(v_1 \ \mathcal{O} \ v_2) \wedge \psi_1 \wedge \psi_2 \leftrightharpoons_V \neg(v_1 \ \mathcal{O} \ v_2) \wedge \psi_1 \wedge \psi_2.$$

Proof. It must be shown that for every truth assignment M_V to V: 1) if there is an extension that satisfies ψ_1, ψ_2, and $(v_1 \ \mathcal{O} \ v_2)$, then there is an extension that satisfies ψ_1, ψ_2, and $\neg(v_1 \ \mathcal{O} \ v_2)$; and 2) if there is an extension that satisfies ψ_1, ψ_2, and $\neg(v_1 \ \mathcal{O} \ v_2)$, then there is an extension that satisfies ψ_1, ψ_2, and $(v_1 \ \mathcal{O} \ v_2)$. It is only necessary to take care of 1) since a similar argument applies for 2).

Assume there is an extension M_1 that satisfies ψ_1, ψ_2, and $(v_1 \ \mathcal{O} \ v_2)$. From $\psi_1 \mid_{v_1=1} \leftrightharpoons_V \psi_1 \mid_{v_1=0}$ and $\psi_2 \mid_{v_2=1} \leftrightharpoons_V \psi_2 \mid_{v_2=0}$ and the fact that M_1 satisfies both ψ_1 and ψ_2, all four extensions identical to M_1 except in v_1 and v_2 will satisfy ψ_1 and ψ_2. One of those will also satisfy $\neg(v_1 \ \mathcal{O} \ v_2)$ since $\mathcal{O}$ is non-trivial. Hence, if there is an extension to M_V that satisfies ψ_1, ψ_2, and $(v_1 \ \mathcal{O} \ v_2)$ there is also an extension that satisfies ψ_1, ψ_2, and $\neg(v_1 \ \mathcal{O} \ v_2)$. □

Theorem 2. *Let ϕ be a 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 ϕ represents a CNF formula ψ, written (v_0) if ϕ has no operators and otherwise written $(v_i) \wedge \psi_\phi$, $n \leq i$, where clause (v_i) is due to the line "Set $\psi \leftarrow \{\{Pop\ S\}\}$", and ψ_ϕ is such that $\psi_\phi \mid_{v_i=1} \leftrightharpoons_{V_\phi} \psi_\phi \mid_{v_i=0}$ [4]. In addition, $\psi \leftrightharpoons_{V_\phi} \phi$. That is, any truth assignment $M_{V_\phi} \subset V_\phi$ is a model for ϕ if and only if there is a truth assignment $M_1 \subset \{v_n, v_{n+1}, \ldots, v_i\}$ such that $M_{V_\phi} \cup M_1$ is a model for ψ.*

[4] The meaning of $\leftrightharpoons$ is given on Page 6

Algorithm 1.

Transformation to CNF (ϕ)
/* *Input ϕ: formula consistent with syntax described on Page 1* */
/* *Output ψ: CNF expression functionally equivalent to ϕ* */
/* *Assume variables of ϕ 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 ϕ^q, set of clauses ϕ^c* */
Set $\phi^q \leftarrow$ "ϕ".
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 '$(v \mathcal{O} w)$'
Pop $\mathcal{O} \leftarrow S$.
Pop $v \leftarrow S$.
Pop S.
Set $i \leftarrow i + 1$.
Append $L \leftarrow$ "$v_i \Leftrightarrow (v\ \mathcal{O}\ w)$".
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 ψ^q equals "$v_j \Leftrightarrow \neg v_g$" for some v_j and v_g,
Set $\psi \leftarrow \psi \cup \{\{v_j, v_g\}, \{\neg v_j, \neg v_g\}\}$.
Otherwise, ψ^q equals "$v_j \Leftrightarrow (v_l\ \mathcal{O}\ v_r)$" so do the following:
Build ϕ^c based on $\mathcal{O}$, v_j, v_l, v_r using the table of Figure 14.
Set $\psi \leftarrow \psi \cup \phi^c$.
Output ψ.
□

Figure 15: *Algorithm for transforming a formula to a functionally equivalent CNF formula*

Proof. The output is a set of sets of variables and therefore represents a CNF formula. The line "Set $\psi \leftarrow \{\{\text{Pop } S\}\}$" is reached after all input symbols are read. This happens only after either the "Evaluate '$\neg s$' " block or the "Push $S \leftarrow s$" 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 '$(v \mathcal{O} w)$' block immediately preceding execution of the "Push $S \leftarrow s$" line (otherwise the input is not a formula). Therefore, either v_0, or v_i is the top symbol of S when "Set $\psi \leftarrow \{\{\text{Pop } S\}\}$" is executed so $\{v_0\}$ or $\{v_i\}$ is a clause of ψ.

We show ψ' and ψ have the stated properties by induction on the depth of the input formula ϕ. For improved clarity, $\leftrightharpoons$ is used instead of $\leftrightharpoons_{V_{\psi_{\phi'}}}$ below.

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

For the induction step, suppose ϕ 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 ϕ. Consider first the case $\phi = \neg\phi'$ (ϕ' has depth k). Algorithm 1 stacks $\neg$ in the line "Push $S \leftarrow s$" and then, due to the very next line, proceeds as though it were operating on ϕ'. The algorithm reaches the same point it would have if ϕ' were input. However, in this case, $\neg$ and a variable are on the stack. This requires one pass through the "Evaluate '$\neg s$' " block which adds "$v_i \Leftrightarrow \neg s$" to L and leaves v_i as the only symbol in S. Thus, upon termination,

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

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

$$\begin{aligned}
\psi_\phi \mid_{v_i=1} &= ((v_{i-1} \vee v_i) \wedge (\neg v_{i-1} \vee \neg v_i)) \mid_{v_i=1} \wedge \psi_{\phi'} \\
&= (\neg v_{i-1}) \wedge \psi_{\phi'} \\
&\leftrightharpoons (v_{i-1}) \wedge \psi_{\phi'} \quad \text{(by the induction hypothesis and Lemma 1, Page 36)} \\
&= ((v_{i-1} \vee v_i) \wedge (\neg v_{i-1} \vee \neg v_i)) \mid_{v_i=0} \wedge \psi_{\phi'} \\
&= \psi_\phi \mid_{v_i=0} .
\end{aligned}$$

Next, it is shown that $\psi \leftrightharpoons \phi$ for this case. The expression $(v_i) \wedge \psi_\phi$ can evaluate to 1 only when the value of v_i is 1. Therefore,

$$\begin{aligned}
\psi &= (v_i) \wedge \psi_\phi \leftrightharpoons \psi_\phi \mid_{v_i=1} \\
&= (\neg v_{i-1}) \wedge \psi_{\phi'} \\
&\leftrightharpoons (v_{i-1}) \wedge \psi_{\phi'} \quad \text{(from above)} \\
&\leftrightharpoons \phi' \quad \text{(by } \psi \leftrightharpoons \phi \text{ induction hypothesis)} \\
&\leftrightharpoons \phi.
\end{aligned}$$

Finally, consider the case that $\phi = (\phi_l \ \mathcal{O} \ \phi_r)$ (ϕ_l and ϕ_r have depth at most k). The Algorithm stacks a '(' then, by the inductive hypothesis

and the recursive definition of a formula, completes operations on ϕ_a which results in ψ_{ϕ_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 $\mathcal{O}$ is read next and pushed on S by the line "Push $S \leftarrow s$". Then ψ_{ϕ_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_l, an operator symbol $\mathcal{O}$, and another variable, say v_r. The final ')' is read and the "Evaluate '$(v\mathcal{O}w)$' " section causes the stack to be popped, $v \Leftrightarrow (v_l \mathcal{O} v_r)$ 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,

$$\psi = (v_i) \wedge \psi_\phi = (v_i) \wedge (v_i \Leftrightarrow (v_l \ \mathcal{O} \ v_r)) \wedge \psi_{\phi_l} \wedge \psi_{\phi_r}$$

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

$$\begin{aligned}
\psi_\phi \mid_{v_i=1} &= (v_i \Leftrightarrow (v_l \ \mathcal{O} \ v_r)) \mid_{v_i=1} \wedge \psi_{\phi_l} \wedge \psi_{\phi_r} \\
&= (v_l \ \mathcal{O} \ v_r) \wedge \psi_{\phi_l} \wedge \psi_{\phi_r} \\
&\leftrightharpoons \neg (v_l \ \mathcal{O} \ v_r) \wedge \psi_{\phi_l} \wedge \psi_{\phi_r} \quad \text{(induction hypothesis and Lemma 1)} \\
&= (v_i \Leftrightarrow (v_l \ \mathcal{O} \ v_r)) \mid_{v_i=0} \wedge \psi_{\phi_l} \wedge \psi_{\phi_r} \\
&= \psi_\phi \mid_{v_i=0} .
\end{aligned}$$

It remains to show $\psi \leftrightharpoons \phi$ for this case. By the induction hypothesis, $\psi_{\phi_l} \mid_{v_l=1} \leftrightharpoons \psi_{\phi_l} \mid_{v_l=0}$. Therefore, for a given truth assignment M_{V_ϕ}, if there is an extension such that v_l has value 1 (0) and ψ_{ϕ_l} has value 0, then there is always an extension such that v_l has value 0 (1), respectively, and ψ_{ϕ_l} has value 1. Since, by the induction hypothesis, $\phi_l \leftrightharpoons (v_l) \wedge \psi_{\phi_l}$, there is always an extension such that ψ_{ϕ_l} has value 1 and v_l has the same value as ϕ_l. The same holds for v_r and ϕ_r. Therefore,

$$\begin{aligned}
\psi &= (v_i) \wedge \psi_\phi \leftrightharpoons \psi_\phi \mid_{v_i=1} \\
&= ((v_i) \wedge (v_i \Leftrightarrow (v_l \ \mathcal{O} \ v_r))) \mid_{v_i=1} \wedge \psi_{\phi_l} \wedge \psi_{\phi_r} \\
&\leftrightharpoons (v_l \ \mathcal{O} \ v_r) \wedge \psi_{\phi_l} \wedge \psi_{\phi_r} \\
&\leftrightharpoons (\phi_l \ \mathcal{O} \ \phi_r) \leftrightharpoons \phi
\end{aligned}$$

□

Theorem 3. *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 ψ due to the `Append` of the "Evaluate '$(v\mathcal{O}w)$' block. For each, at most 12 symbols involving literals, 12 involving operators (both $\wedge$ and $\vee$ represented as ","), and 8 involving parentheses (represented as "{" "}") are added to ψ (see Figure 14), for

a total of at most $32m$ symbols. At most $m + n$ new variables associated with negations are added to ψ (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 ψ (from $\{v_i, v_j\}, \{\neg v_i, \neg v_j\}$), 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 ψ to ϕ is not greater than $(56m + 12)/4m = 14 + 3/m < 17$.

□

The next result shows that the transformation is computed efficiently.

Theorem 4. *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 $O(m)$. □

If ϕ contains duplicate subformulas, the output will consist of more variables and CNF blocks than are needed to represent a CNF formula equivalent to ϕ. This is easily remedied by implementing L as a balanced binary search tree keyed on v_l, v_r, and $\mathcal{O}$ and changing the `Append` to a tree insertion operation. If, before insertion in the "Evaluate '$(v\mathcal{O}w)$' " block, it is discovered that '$v_j \Leftrightarrow (v\mathcal{O}w)$' 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 formula to a DNF formula.

4.2 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 [14] and [110] (ca. 1937) and was applied to DNF formulas in a form exemplified by the following:

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

Consensus in DNF was rediscovered in [102] and [98] (ca. 1954) where it was given its name. A few years later its dual, as resolution in propositional logic, was applied to CNF [38] (1958), for example:

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

The famous contribution of Robinson [100] 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 $\mathcal{R}_{c_2}^{c_1}$, 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 5. *Let ψ 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 $\mathcal{R}_{c_2}^{c_1} \notin \psi$ exists. Then $\psi \Leftrightarrow \psi \cup \{\mathcal{R}_{c_2}^{c_1}\}$.*

Proof. Clearly, any assignment that does not satisfy CNF formula ψ cannot satisfy a CNF formula which includes a subset of clauses equal to ψ. Therefore, no satisfying assignment for ψ implies none for $\psi \cup \{\mathcal{R}_{c_2}^{c_1}\}$.

Now suppose M is a model for ψ. 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 $\mathcal{R}_{c_2}^{c_1}$, by definition. Therefore, $\mathcal{R}_{c_2}^{c_1}$ is satisfied by M and so is $\psi \cup \{\mathcal{R}_{c_2}^{c_1}\}$. A similar argument applies if $v \notin M$. □

The *resolution* method makes use of Lemma 5 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 16. It uses the notion of pure literal (Page 3) to help find a model, if one exists. Recall, a literal is pure in formula ψ if it occurs in ψ but its complement does not occur in ψ. If the algorithm outputs "unsatisfiable" then the set of all resolvents generated by the algorithm is a *resolution refutation* for the input formula.

Theorem 6. *Let ψ be a CNF formula represented as a set of sets. The output of Algorithm 2 on input ψ is "unsatisfiable" if and only if ψ is unsatisfiable. If the output is a set of variables, then it is a model for ψ.*

Proof. If the algorithm returns "unsatisfiable" one resolvent is the empty clause. From Lemma 5 and the fact that an empty clause cannot be satisfied, ψ is unsatisfiable.

If the algorithm does not return "unsatisfiable" the `Otherwise` block is entered because new resolvents cannot be generated from ψ. 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 ψ'. Now, consider the final `Repeat` block. If $\psi' = \emptyset$ then all clauses are satisfied by M from the previous `Repeat` block, and $M_{1:i}$ will never falsify a clause for any i. In that case M is returned and is a model for ψ. 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` $M \leftarrow M \cup \{v_i\}$" 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 $M_{1:i}$ 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, it is possible 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 ψ, before the pure literal `Repeat` block, would contain $\mathcal{R}^{c_1})c_2$. Moreover, $\mathcal{R}^{c_1}_{c_2}$ could not be satisfied by $M_{1:i-1}$ because such an assignment would have satisfied c_1 and c_2. But then $\mathcal{R}^{c_1}_{c_2}$ 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 SAT. 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 algo-

Algorithm 2.

Resolution (ψ)
/* *Input ψ is a set of sets of literals representing a CNF formula* */
/* *Output is either "unsatisfiable" or a model for ψ* */
/* *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 $\mathcal{R}_{c_2}^{c_1} \notin \psi$ exists,
Set $\psi \leftarrow \psi \cup \{\mathcal{R}_{c_2}^{c_1}\}$.
Otherwise,
Repeat the following while there is a pure literal l in ψ:
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_ψ 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 ψ, Set $M \leftarrow M \cup \{v_i\}$.
Output M.

□

Figure 16: *Resolution algorithm for CNF formulas.*

rithm may perform poorly or well, depending on the order of resolvents, and one infinite family is presented here. Let there be n "normal" variables $\{v_0, v_1, \ldots, v_{n-1}\}$ and $n-1$ "selector" variables $\{z_1, z_2, \ldots, z_{n-1}\}$. Let each clause contain one normal variable and one or two selector variables. For each $0 \leq i \leq n-1$ let there be 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:

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

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 $\{\neg z_3\}$,...,$\{\neg z_{n-1}\}$ 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 8, 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 chapter deals with this question. However, first consider an extension to resolution that has shown improved ability to admit short refutations.

4.3 Extended Resolution

In the previous section it was shown 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. However, the simple idea of *extended resolution* can yield short refutations in such cases.

Extended resolution is based on a result of Tseitin [120] who showed that, for any pair of variables v, w in a given CNF formula ψ, the following expression may be appended to ψ:

$$(z \vee v) \wedge (z \vee w) \wedge (\neg z \vee \neg v \vee \neg w) \tag{3}$$

where z is a variable that is not in ψ. From Figure 14 this is equivalent to:

$$z \leftrightarrow (\neg v \vee \neg w)$$

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 14 can be used in place of or in addition to (3). 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.

By adding variables not in ψ 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 4.1.

4.4 Davis-Putnam Resolution

The Davis-Putnam procedure (DPP) [39] 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 7. *Let ψ be a CNF formula. Perform the following operations:*

$\psi_1 \leftarrow \psi$.
Choose any variable v from ψ_1.
Repeat the following until no new resolvents with v as pivot can be added to ψ_1:
    If there is a pair of clauses $c_1, c_2 \in \psi_1$ such that $\mathcal{R}_{c_2}^{c_1}$ with v as pivot exists and $\mathcal{R}_{c_2}^{c_1} \notin \psi_1$,
        $\psi_1 \leftarrow \psi_1 \cup \{\mathcal{R}_{c_2}^{c_1}\}$.
$\psi_2 \leftarrow \psi_1$.
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 ψ is satisfiable if and only if ψ_2 is satisfiable.

Proof. By Lemma 5 ψ_1 is functionally equivalent to ψ. Since removing clauses cannot make a satisfiable formula unsatisfiable, if ψ and therefore ψ_1 is satisfiable, then so is ψ_2.

Now, suppose ψ 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 ψ and ψ_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 $\mathcal{R}_{c_2}^{c_1} \in \psi_2$ is falsified by M. Therefore, since every assignment falsifies ψ, ψ_2 is unsatisfiable. □

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.

4.5 Davis-Putnam Loveland Logemann Resolution

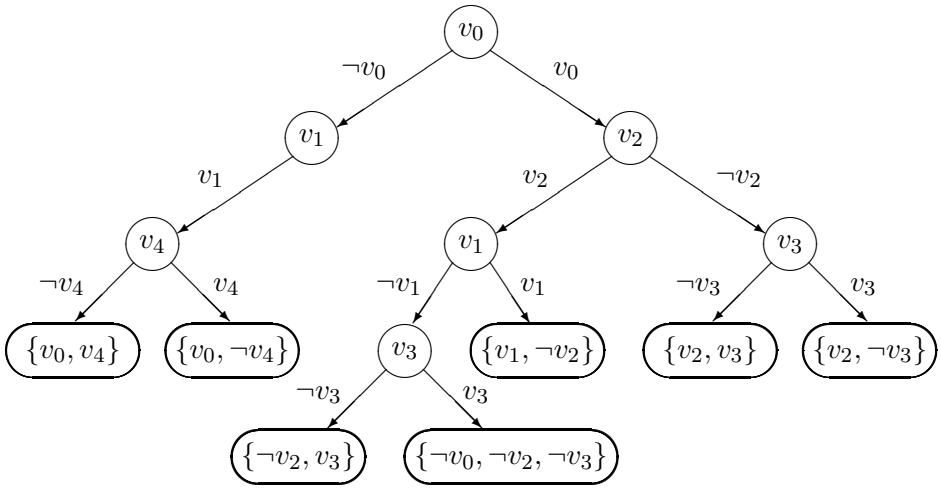
Here a reasonable implementation of the key algorithmic idea in DPP is presented. The idea is to repeat the following: choose a variable, take care of all resolutions due to that variable, and then erase clauses containing it. The algorithm, called DPLL [40], is shown in Figure 17. It was developed when 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.

Algorithm 3.

DPLL (ψ)
/* *Input* ψ *is a set of sets of literals representing a CNF formula* */
/* *Output is either "unsatisfiable" or a model for* ψ */
/* *Locals: integer* d*, variable stack* V_P*, list of formulas* $\psi_0, \psi_1, ...$ */
/* *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:
If $\psi_d = \emptyset$, Output $M_{1:d}$.
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 */
If l is a negative literal then $M_{1:d+1} \leftarrow M_{1:d} \cup \{\neg l\}$.
Otherwise, $M_{1:d+1} \leftarrow M_{1:d}$.
Set $\psi_{d+1} \leftarrow \{c - \{\neg l\} : c \in \psi_d, l \notin c\}$.
Set $d \leftarrow d + 1$.
Otherwise,
If there exists a pure literal in ψ_d,
Choose a pure literal l.
Otherwise, if there is a unit clause in ψ_d,
Choose a literal l in a unit clause and "tag" l.
Otherwise,
Choose a literal l in ψ_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 $\psi_{d+1} \leftarrow \{c - \{\neg l\} : c \in \psi_d, l \notin c\}$.
Set $d \leftarrow d + 1$.

□

Figure 17: *DPLL algorithm for CNF formulas.*



$$\{\{v_0, v_4\}, \{v_0, \neg v_4\}, \{v_0, v_1\}, \{v_1, \neg v_2\}, \{\neg v_2, v_3\},$$
$$\{\neg v_0, \neg v_1, \neg v_2\}, \{\neg v_0, \neg v_2, \neg v_3\}, \{v_2, \neg v_3\}, \{v_2, v_3\}\}$$

Figure 18: *A DPLL refutation tree for the above CNF formula.*

The DPLL algorithm of Figure 17 is written iteratively instead of recursively for better comparison with other algorithms to be discussed later. Omitted is a formal proof of the fact that the output of DPLL on input ψ is "unsatisfiable" if and only if ψ is unsatisfiable, and if the output is a set of variables, then it is a model for ψ. The reader may consult [40] 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 search trees will be used several times in this chapter to assist in making some difficult points clear, this concept is explained 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 ψ 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 ψ, or minimally falsifies at least one clause of ψ. In the latter case, one of the clauses falsified becomes the label for the leaf. If ψ 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 18. With reference to Figure 17, a path from the root to any vertex represents 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 8 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[5]. But, in [4] a family for formulas for which a shortest resolution proof is exponentially smaller than the smallest DPLL refutation is presented. 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 order is often sensitive to structural properties of the given formula. For this reason, quite a number of choice heuristics have been studied and some results on these are presented later in this chapter.

Many modern SAT solvers augment DPLL with other features such as restarts and conflict-driven clause learning. The refutation complexity of a solver so modified is then theoretically similar to that of resolution. More will be said about this in subsequent sections.

This section concludes with the following classic result.

Theorem 8. *Suppose DPLL is applied to a given unsatisfiable CNF formula ψ and suppose the two lines "*Set $\psi_{d+1} \leftarrow \{c - \{\neg l\} : c \in \psi_d, l \notin c\}$*." are together executed p times. Then there is a resolution refutation in which no more than $p/2$ resolvents are generated.*

Proof. Run DPLL on ψ and, in parallel, generate resolvents from clauses of ψ 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 ψ_{d+1} is set. Hence, when DPLL terminates on an unsatisfiable formula, at most $p/2$ resolvents will be generated. Next, it is shown 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 ψ label leaves as discussed on Page 47. 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 ψ 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

[5]See [29] for other results along these lines.

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 ψ and resolvents. Run DPLL on ψ. 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 $S \leftarrow \mathcal{R}_{c_2}^{c_1}$; 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 $c_t = \mathcal{R}_{c_2}^{c_1}$.

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

To prove this claim it is shown 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 $V_P \cup \{\neg l\}$ 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 ψ. 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 $V_P \cup \{\neg l\}$ 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. □

4.6 Conflict-Driven Clause Learning

Early SAT solvers used refinements and enhancements to DPLL to solve SAT instances. Recently, these refinements and enhancements have been standardized and developed into a parameterized algorithm. This algorithm is far enough removed from DPLL that it is commonly referred to as Conflict Driven Clause Learning (CDCL) [111, 101, 112]).

The reasons for the increased solving power and efficiency of modern SAT solvers lie in specialized CDCL techniques tailored to support core CDCL operations such as Binary Constraint Propagation (BCP) and heuristic computation. Each technique on its own is not very impressive but, when all of these techniques are fitted together in the DPLL framework, orders of magnitude improvement can be seen. For example, conflict clause learning helps to create a dynamic search space that is DAG-like instead of tree-like as was the case for early DPLL-based algorithms [10]. This technique, when used in conjunction with dynamic restarts, is as strong as general resolution and exponentially more powerful than DPLL [10]. The combination of such techniques have allowed SAT solvers to be used practically in a wide range of fields such as bioinformatics [89], cryptography [93, 113], and planning [36, 131] as well as on many hard combinatorial problems such as van der Waerden numbers [82, 83]. Also, the recent integration of SAT-based techniques with theory solvers (this technology is named SMT [106] - details are given in Section 4.7) is enabling push-button solutions to industrial-strength problems that previously required expert human interaction to solve (for example, see [62]).

Two chapters in the Handbook of Satisfiability [13] cover CDCL extensively. The reader is referred to [37] and [112] for details. The remainder of this section highlights the main points.

4.6.1 Conflict Analysis

DPLL explores both sides of a choicepoint before backtracking over it. This is called *chronological backtracking*, and is not generally conducive to good SAT solving. It is often the case that the same conflict exists on both sides of a choicepoint. Analyzing the conflict on one side may prevent recomputation of the same conflict on the other side, in which case the choicepoint can be *backjumped* over. Performing conflict analysis in the framework of DPLL enables non-chronological backtracking, which, in effect, works to counteract bad decisions made by the search heuristic.

Conflict analysis is often depicted in terms of an implication graph, though, in practice, the implication graph is not actually built. In modern SAT solvers, new clauses are generated by resolution steps ordered by the current search tree path. Specifically, a conflict is reached when two asserting clauses infer opposing literals during BCP. These two clauses can be resolved together to create a new clause called a *conflict clause*. This clause will either be the empty clause (in which case the formula is unsatisfiable), or it will, if evaluated under the current partial assignment, be falsified. Consider the most recently assigned literal in the conflict clause.

If the literal was assigned by an asserting clause, the conflict clause and the asserting clause are resolved, creating a new conflict clause, and the process is repeated. If the literal was assigned by a choicepoint, the current search tree up to the literal is undone and the value assigned by the choicepoint is flipped. All new conflict clauses may be safely added to the original CNF formula (i.e. learned) because they are logical consequences of the original formula [111]. Learning conflict clauses helps to prune the search space by preventing re-exploration of shared search structure, especially when used in conjunction with restarts. Also, prior to adding newly learned clauses, it may be beneficial to attempt to minimize the clauses by performing a restricted version of resolution with other asserting clauses [11, 125].

There is at least one special case to consider when learning conflict clauses. If a conflict clause contains only one literal at the current decision level, called a Unique Implication Point (UIP), the search state can backjump to the second most recently assigned literal in the clause. This will cause the UIP literal will be naturally inferred via BCP [95] and has the effect of promoting those parts of the search tree relevant to the current conflict.

Conflict analysis is also used to generate resolution proofs of unsatisfiable instances. In the same way that an assignment to a satisfiable instance acts as a polynomial time checkable certificate that an instance is satisfiable, a resolution proof acts as a certificate of unsatisfiability, checkable in linear time and space on the number of learned clauses (though the number of learned clauses can be exponential on the number of variables in the best case). More information on this topic can be found in [130], [11], and [124].

4.6.2 Conflict Clause Memory Management

Not all derived conflict clauses can be stored. Too many conflict clauses increases the overhead of finding an applicable one, too few conflict clauses reduces the chance that a good one can be found to prune the search space. Therefore, the size of the conflict clause data base, what goes into it, and how long entries should stay there has been well studied. The number of conceivable schemes is quite large. A good one was recently discovered and reported in [8]. In that paper the learned clause measure LBD (for Literals Block Distance) is defined to be the number of different levels at which literals of a clause were assigned values as choicepoints up to the current assignment. A small LBD suggests a learned clause that was useful in generating a long sequence of unit resolutions. It was found in experiments on all SAT-Race '06 benchmarks that 40% of unit resolutions occur in clauses of LBD=2 versus 20% on 2-literal clauses and, generally, the lower the LBD, the more significant the contribution to BCP. Due to this, the following "cleaning strategy" was proposed: remove half of the learned clauses, not including a class of easy discovered clauses of least LBD, every $20000+500*x$ conflicts, where x is the number of times this operation has been performed before on the current input. Adding this strategy to existing solvers resulted in a significant improvement in performance.

Many other ideas for determining the life of conflict clauses have been proposed and tried. Several of these are mentioned in [112] near the end of

the chapter.

4.6.3 Lazy Structures

An important aspect of CDCL associated with performance has to do with maintaining data structures that store solver state. The degree to which these structures are managed lazily has an impact on the optimal choice of other solver mechanisms. For example, lazy structures result in faster backtracking so "the authors of Chaff proposed to always stop clause learning at the first UIP" and "always take the backtrack step" [112]. In addition, the implementation of BCP and conflict analysis is influenced by the implementation of lazy structures. Using lazy structures, the fact that a clause becomes satisfied by a state change may not be reported to solver mechanisms immediately or even at all. This presents potential savings by preventing some unnecessary computation and does not affect the correctness of the solver. Next, in this section the classic notion of watched literals [95] is used to illustrate this point.

Suppose a clause is stored as a sequence of literals and suppose the number of literals is at least 2. For each clause, assign two pointers each pointing to one of the literals but both pointing to different literals of the clause. The literals pointed to are called *watched literals* and are distinguished in name as the "first" and "second" watched literal. The following properties are maintained always:

1. A clause is satisfied if and only if its first watched literal has value 1
2. A clause is falsified if and only if its first and second watched literals have value 0
3. A clause is "unit" if and only if one watched literal is unassigned and the other has value 0
4. A clause is unassigned and not "unit" if and only if both watched literals are unassigned

Initially, a clause is unassigned and the pointers are pointing to arbitrary literals. If a clause is unassigned, not "unit," and a literal other than the second watched literal takes value 1, then that literal becomes the first watched literal. If the literal taking value 1 is the second watched literal, the first and second watched literals are swapped. If both watched literals are unassigned and one of them takes value 0, a check must be made to determine whether the clause has another unassigned literal. If it doesn't, the clause becomes a unit clause and the single unassigned literal takes value 1 and becomes the first watched literal swapping title, if necessary, with the watched literal of value 0. Otherwise the pointer referencing the recently assigned watched literal gets set to point to the unassigned literal that was found during the check. The search for an unassigned literal can proceed in any direction among the literals of the clause. Observe that the watch literal pointers do not have to be reset on backtrack. Further savings are obtained by associating a "level" and "value" field with every literal: when a clause is satisfied or

falsified it is the level and value of the first watched literal that is set. Since watched literals do not change during backtrack, this information represents clause status as well. Earlier schemes for managing clause status during backtrack associate level and value fields with clauses and require searching the literals of a clause to determine status. Thus, implementing the watched literal concept results in much less backtracking overhead and significantly increases the number of backtracks handled per second.

As can be observed from the above discussion, watched literals are useful in reacting to single-variable inferences. But, more advanced techniques look at multi-variable inferences, for example two-variable equivalence. Watched literals cannot help in this case. Additional clause references, of course, may be used support these advances [123] at added expense. Another idea [88] is to maintain an order of assigned literals in a clause by non-decreasing level while the unassigned literals remain watched as above. As a variable is assigned a value, it is "sifted" into its proper place in the ordered list. The problem with this idea is that all literals in the ordered list may need to be visited on a backtrack.

4.6.4 CDCL Heuristics

Lazy data structures can drastically increase the speed of BCP, but, since the current state of the search is not readily available, traditional heuristics such as MOMs (see [48] for a complete discussion), Jeroslow-Wang [64], and Johnson [65], cannot be used. The first CDCL heuristic is the Variable State Independent Decaying Sum (VSIDS) [95] heuristic. VSIDS maintains one counter per literal in the SAT instance, initialized to the number of clauses each literal occurs in. VSIDS branches on the literal with the highest count, though this requires sorting the counters, which is expensive, and so is only done periodically. When a new clause is learned, the counts for each literal in the clause are incremented. Also periodically, the counters are halved. This heuristic, and many similar variants, efficiently glean information from learned clauses to guide the solver towards unsatisfiable cores, enabling modern solvers to solve large SAT instances (millions of variables and clauses) that have small resolution proofs.

4.6.5 Restarts

Conflict directed heuristics can sometimes cause a solver to focus too heavily one particular region of the search space, leading to heavy-tailed runtime distribution (where progress seems to exponentially decrease the longer a solver runs). One way to positively modify this behavior [54] is to restart. A *dynamic restart* [69, 61] clears a solver's current partial assignment; heuristic information and learned clauses are kept, and a short preprocessing phase may follow a restart that typically garbage collects the learned clause database. As reported empirically in the literature, "it is advantageous to restart often" [37].

4.7 Satisfiability Modulo Theories

Despite highly significant and broad advances to SAT solver design, there are still many problems for which DPLL and CDCL variants are a challenge. For example, it is well known that verifying correctness of arithmetic circuits is generally difficult for SAT solvers. The problem is that datapath operations are over a fixed bit-width and, beyond modular arithmetic, some operations such as multiplication are not linear and particularly vexing for SAT solvers. An accepted method, called *bit-blasting*, for dealing with small bit-widths is to describe operations as propositional formulas over vectors of Boolean variables, convert to CNF using the results of Section 4.1, and solve with a SAT solver. But, it is doubtful that bit-blasting is going to be universally practical for 64 bit or even 32 bit logic. As another, more universally occurring example, SAT solvers have a difficult time handling cardinality constraints and, more generally, constraints comparing numeric expressions. To mitigate these and other problems a significant body of research has studied the use of a SAT solver to manage a collection of special purpose solvers for first order theories, including bit-vector arithmetic. This extension of SAT has found applications in hardware verification, software model checking, and testing, and vulnerability detection, among other applications, and is called *Satisfiability Modulo Theories* or SMT.

A formal specification of SMT is not given here; the reader is invited to check [9] for the syntax and semantics of SMT-LIB, the official documented language of SMT solvers. Most importantly, with respect to SAT, an instance of SMT is a formula in first-order logic where uninterpreted functions are admitted and Boolean variables are used to tie predicates from disparate theories and allow a SAT solver to decide the consistency of the constraints of the instance. The scope of SMT is made apparent through the SMT examples shown in Figures 19 and 20. The examples are expressed in a lisp-like language that is native to YICES [43], one of the leading available SMT solvers, and may actually be input to YICES.

```
(set-evidence! true)
(define f::(-> int int))
(define g::(-> int int))
(define a::int)
(assert (forall (x::int) (= (f x) x)))
(assert (forall (x::int) (= (g (g x)) x)))
(assert (/= (g (f (g a))) a))
(check)
```

Figure 19: An SMT instance with uninterpreted functions and quantifiers.

In Figure 19, `f` and `g` are uninterpreted functions: note that only their signatures, namely a single `int` as input and an `int` as output, are defined in the 2nd and 3rd lines. However, the 5th and 6th lines force `f(x)=x`, `g(y)=x`, and `g(x)=y` for all integers. This is inconsistent with the 7th line which requires, for any integer `a`, that `g(f(g(a)))`, which is `g(g(a))` and therefore `a` from the above, is not equal to `a`. On this input YICES returns `unsat`. If `/=` (not equal) is changed to `=` YICES returns `unknown` with the

following interpretation: `a=2, g(2)=1, f(1)=1, g(1)=a2`. The 1st and 8th lines are YICES-specific directives.

Figure 20 shows constraints expressed in terms of bit-vector operations. This instance is satisfied by 8-bit numbers `b1` and `v2` such that multiplying `b1` by 17, shifting right by 2 bits and adding `v2` results in 39.

```
(set-evidence! true)
(define b1::(bitvector 8))
(define b2::(bitvector 8))
(define v1::(bitvector 8))
(assert (= v1 (mk-bv 8 39)))
(define v2::(bitvector 8))

(assert (= (bv-mul b1 (mk-bv 8 17)) b2))
(assert (= (bv-add (bv-shift-right0 b2 2) v2) v1))
(check)
```

Figure 20: An SMT instance with bit-vector expressions

YICES output on this example is:

```
sat
(= v1 0b00100111)
(= b1 0b01110110)
(= b2 0b11010110)
(= v2 0b11110010)
```

That is, `b1` is 118 and `v2` is 242, which is one of many possible solutions.

Clearly, admitting bit-vector operations is important as it makes circuit design and verification problems easier to express and solve: trying to express the constraints of such problems as a CNF formula is unnatural and time-and-space consuming. A special solver for the theory of bit-vector arithmetic and logic may be developed to handle bit-vector predicates such as those in Figure 20. Other theory solvers may be designed to handle other problems that are difficult for CNF solvers. Examples of such theories, including the theory of bit-vectors, are as follows:

1. **Linear arithmetic (LA):** constraints are equations or inequalities involving addition and subtraction over rational variables, constants, or constants times variables.
2. **Uninterpreted functions (UF):** all uninterpreted function, constant and predicate symbols together with Leibniz's law that allows substitution of equals for equals.
3. **Difference arithmetic:** this is linear arithmetic restricted to equations or inequalities of the form $x - y \leq c$ where x and y are variables and c is a constant.
4. **Non-linear arithmetic:** constraints are like linear arithmetic except that variables may be multiplied by other variables.

5. **Arrays:** array constraints ensure that the value of a memory location does not change unless a specific *write* operation on that location with a different value is executed and that a value written to a location may be accessed by a read operation. Symbols of this theory might be `select` and `store` with semantics given by the following:

$$\begin{aligned}
&\texttt{select}(\texttt{store}(v,i,e),j) = \text{ if } i=j \text{ then } e \text{ else } \texttt{select}(v,j)\\
&\texttt{store}(v,i,\texttt{select}(v,i)) = v\\
&\texttt{store}(\texttt{store}(v,i,e),i,f) = \texttt{store}(v,i,f)\\
&i \neq j \Rightarrow \texttt{store}(\texttt{store}(v,i,e),j,f) = \texttt{store}(\texttt{store}(v,j,f),i,e)
\end{aligned}$$

   where $\texttt{select}(v,j)$ means read the j^{th} value from array v and $\texttt{store}(v,i,e)$ means store the value e into the i^{th} element of array v.

6. **Lists:** symbols of this theory might be `car`, `cdr`, `cons`, and `atom` with semantics given by the following:

$$\begin{aligned}
&\texttt{car}(\texttt{cons}(x,y)) = x\\
&\texttt{cdr}(\texttt{cons}(x,y)) = y\\
&\neg\texttt{atom}(x) \Rightarrow \texttt{cons}(\texttt{car}(x),\texttt{cdr}(x)) = x\\
&\neg\texttt{atom}(\texttt{cons}(x,y))
\end{aligned}$$

7. **Fixed width bit-vectors:** the main reason for including a bit-vector theory is to remove the burden of solving bit-level formulas from the SAT solver. As seen in Figure 20, constraints are expressed at word-level and may involve any machine-level operations on words.

8. **Recursive data types:** this is an abstraction for common data structures that may be defined recursively such as linked lists and for common data types that may be defined recursively such as a tree or the natural numbers.

It is not within the scope of this chapter to discuss the construction of decision procedures for theories. The interested reader may begin a search for such information by consulting [107]. The remainder of this section will show how a SAT solver can be made to interact with constraints from several theories.

Some definitions are given now to make the procedures to be discussed later easier to express and analyze. A *first-order formula* is an expression consisting of a set V of variables, symbols $(,), \neg, \vee, \wedge, \rightarrow, \exists, \forall$, a set $\mathcal{P}$ of predicate symbols, a set $\mathcal{F}$ of function symbols, each with associated positive arity, and a set $C \subset \mathcal{F}$ of constant symbols which are just 0-arity function symbols, for example x or f. A *signature*, denoted Σ, is a subset of $\mathcal{F} \cup \mathcal{P}$. A *first-order structure* $\mathcal{A}$ specifies a semantics for the above: that is, a domain A of elements, mappings of every n-ary function symbol $f \in \Sigma$ to a total function $f^{\mathcal{A}} : A^n \mapsto A$ and mappings of every n-ary predicate symbol $p \in \Sigma$ to a relation $p^{\mathcal{A}} \subseteq A^n$. The symbols $\exists$ and $\forall$ are quantifiers and the scope of one of these is the entire formula. A *sentence* is a formula all of whose variables are bound by a quantifier (that is, the formula has no free variables) and therefore evaluates either to *true* or *false*, depending on the structure

it is evaluated over. A formula is a *ground formula* if it has no variables. Observe that $x + 1 \leq y$ has no variables because x and y are constants in the theory of linear arithmetic, and is therefore a ground formula. Denote $(\mathcal{A}, \alpha) \models \phi$ to mean the sentence ϕ evaluates to *true* in structure $\mathcal{A}$ under variable assignment $\alpha : V \mapsto A^{|V|}$. Sentence ϕ is said to be *satisfiable* in $\mathcal{A}$ if $(\mathcal{A}, \alpha) \models \phi$ for some α. Sentence ϕ is said to be *valid* in $\mathcal{A}$ if $(\mathcal{A}, \alpha) \models \phi$ for every α. If $\mathcal{A}$ is finite, the question $(\mathcal{A}, \alpha) \models \phi$ for some α and given ϕ is decidable. However, for infinite structures, the question may be undecidable.

A *theory* T is a set of first-order sentences expressed in the language of some signature Σ. Equality is implicit in the theories mentioned earlier so their signatures do not contain the symbol $=$. A *model* of T is a structure $\mathcal{A}$ in which every sentence of T is valid. Denote by $T \models \phi$ the condition where $\mathcal{A}$ is a model for T and $(\mathcal{A}, \alpha) \models \phi$ for all α. Then ϕ is said to be *T-valid*. Let $\perp$ denote a formula that is not satisfied by any structure. Then ϕ is said to be *T-satisfiable* if $T \cup \{\phi\} \not\models \perp$, also written $T, \phi \not\models \perp$. T-validity and T-satisfiability are seen as dual concepts since $T, \neg\phi \models \perp$ if and only if $T \models \phi$.

One is generally interested in combining theories: that is, proving that statements with terms from more than one theory are valid over those combined theories. Suppose, for two theories T_1 of Σ_1 and T_2 of Σ_2, there exist procedures P_1 and P_2 that determine whether a given formula ϕ_1 expressed in the language of Σ_1 is T_1-valid and whether a given formula ϕ_2 expressed in the language of Σ_2 is T_2-valid, respectively. It is generally not possible to the solve the problem of determining whether $\{\phi_1\} \cup \{\phi_2\}$ in the language of $\Sigma_1 \cup \Sigma_2$ is $(T_1 \cup T_2)$-valid with P_1 and P_2. However, it can be under some modest restrictions that usually do not get in the way in practice. These restrictions will be assumed for the rest of this section and are: theories T_1 and T_2 are decidable, $\Sigma_1 \cup \Sigma_2 = \emptyset$, and T_1 and T_2 are stably infinite: let C be an infinite set of constants not in signature Σ and define a theory T of Σ to be *stably infinite* if every T-satisfiable ground formula of signature $\Sigma \cup C$ is satisfiable in a model of T with an infinite domain of elements.

By the duality noted above, the validity problem can be treated as a satisfiability problem where it is more conveniently solved. Suppose ϕ is a formula that is logically equivalent to a DNF formula with m conjuncts $\phi_{i,1} \wedge \phi_{i,2}$, $i \in \{1, 2, \ldots, m\}$, where $\phi_{i,1}$ and $\phi_{i,2}$ are expressed in the language of $\Sigma_1 \cup C$ and $\Sigma_2 \cup C$, respectively, and C is a set of constants that are shared by $\phi_{i,1}$ and $\phi_{i,2}$. Terms $\phi_{i,1} \wedge \phi_{i,2}$ are said to be in pure form. Clearly, ϕ is $(T_1 \cup T_2)$-unsatisfiable if and only if for all $i \in \{1, 2, \ldots, m\}$, $\phi_{i,1} \wedge \phi_{i,2}$ is $(T_1 \cup T_2)$-unsatisfiable. It is desirable to use P_1 and P_2 individually to determine this. By Craig's interpolation lemma [35], a pure form term $\phi_{i,1} \wedge \phi_{i,2}$ is $(T_1 \cup T_2)$-unsatisfiable if and only if there is a formula ψ, called an *interpolant*, whose free variables are a subset of the free variables contained in both $\phi_{i,1}$ and $\phi_{i,2}$ such that $T_1, \phi_{i,1} \models \psi$ and $T_2, \phi_{i,2}, \psi \models \perp$. Finding an interpolant is a primary goal of an SMT solver and the subject of much of the remainder of this section, the exposition of which is strongly influenced by [72]. Although combining only two theories is considered here, the process generalizes to more than two theories.

The first step in combining theories is to produce a pure-form formula from ϕ. This is accomplished by introducing variables to C which remove the

dependence on objects from an "alien" theory through substitution. More specifically, suppose function $f \in \Sigma_1$ and one of its arguments t is a term, possibly a constant, expressed in Σ_2. Then create a new w, set $C = C \cup \{w\}$, add a new constraint $w = t$, and replace t by w in the argument list of f. Similarly, for predicates $p \in \Sigma_1$, $p \in \Sigma_2$ and function $f \in \Sigma_2$. If there is a constraint $s = t$ where $s \in \Sigma_1$ and $t \in \Sigma_2$ then create a new w, and replace the constraint with $w = s$ and $w = t$. If there is a constraint $s \neq t$ then add a new w_1 and new w_2 and replace the constraint with $w_1 = s$, $w_2 = t$, and $w_1 \neq w_2$. For example, consider

$$\phi = \{x \leq y,\ y \leq x + \texttt{car}(\texttt{cons}(0, x)),\ p(h(x) - h(y)),\ \neg p(0)\}$$

which mixes uninterpreted functions, linear arithmetic, and lists. For this ϕ, $C = \{x, y\}$. The constraint on the left need not be touched because it only contains symbols from linear arithmetic. The subterm $h(x) - h(y)$ is mixed so create new constants $g_1 = h(x)$ and $g_2 = h(y)$ and substitute $g_1 - g_2$ for $h(x) - h(y)$. The term $p(g_1 - g_2)$ is mixed so create constant $g_3 = g_1 - g_2$ and substitute. The remaining term $p(g_3)$ has no "alien" subterms. The symbol 0 is alien to the theory of uninterpreted functions. Hence, create $g_4 = 0$ and substitute making the rightmost constraint $\neg p(g_4)$, and changing the list theory subterm to $\texttt{car}(\texttt{cons}(g_4, x))$, both of which may be left alone. However, the 2nd constraint from the left is still mixed so create $g_5 = \texttt{car}(\texttt{cons}(g_4, x))$ and substitute. This leaves:

$$\begin{array}{ll}
C = \{x, y, g_1, g_2, g_3, g_4, g_5\}, & \text{(Shared constants)} \\
\phi_1 = \{x \leq y,\ y \leq x + g_5,\ g_3 = g_1 - g_2,\ g_4 = 0\}, & \text{(LA)} \\
\phi_2 = \{g_1 = h(x),\ g_2 = h(y),\ p(g_4) = \text{false},\ p(g_3) = \text{true}\}, & \text{(UF)} \\
\phi_3 = \{g_5 = \texttt{car}(\texttt{cons}(g_4, x))\}, & \text{(Lists)}
\end{array}$$

so ϕ has become a pure formula whose constraints are partitioned according to the theories of linear arithmetic, uninterpreted functions, and lists, and are tied by newly created constants and equality constraints.

The next step is to propagate inferred equalities between the partitioned sections. First, this is illustrated using the above example, then a general procedure for propagation is presented. By the first axiom of the theory of lists, the constraint $g_5 = \texttt{car}(\texttt{cons}(g_4, x))$ infers $g_4 = g_5$. This propagates to section LA where $g_5 = 0$ is inferred. Then $y \leq x$ is inferred and $x = y$ is inferred from $x \leq y$ and $y \leq x$. The inferred constraint $x = y$ propagates to section UF where $g_1 = g_2$ is inferred. Propagating back to LA infers $g_3 = 0$ and finally $g_3 = g_4$. But this contradicts UF constraints which say $p(g_3) = \texttt{true}$ and $p(g_4) = \texttt{false}$. Hence, the conclusion is that ϕ is unsatisfiable in the combined theories of linear arithmetic, lists, and uninterpreted functions. Observe that ϕ_1, ϕ_2, and ϕ_3 are all satisfiable in their respective theories.

Figure 21 shows an algorithm for determining satisfiability of a formula $\phi_1 \wedge \phi_2$ that is in pure form with ϕ_1 of signature $\Sigma_1 \cup C$, ϕ_2 of signature $\Sigma_2 \cup C$, C a set of shared constants, $\Sigma_1 \cap \Sigma_2 = \emptyset$, and assumes there exist decision procedures P_1 and P_2 for theories T_1 and T_2 over Σ_1 and Σ_2, respectively. It is straightforward to generalize this to any number of theories; it is also possible to use the generalized algorithm to decide any mixed theory formula by converting to DNF in pure form and applying the algorithm to the conjuncts

Algorithm 4.

CTSolver $(\phi_1, \phi_2, P_1, P_2)$
/* *Input: formulas* ϕ_1 *over* $\Sigma_1 \cup C$, ϕ_2 *over* $\Sigma_2 \cup C$,
where $\Sigma_1 \cap \Sigma_2 = \emptyset$, *and decision procedures*
P_1, P_2 *for theories* T_1 *over* Σ_1 *and* T_2 *over* Σ_2 */
/* *Output: satisfiable if* $\phi_1 \wedge \phi_2$ *is* $(T_1 \cup T_2)$*-satisfiable,*
otherwise unsatisfiable */

1. Apply P_1 to ϕ_1, and apply P_2 to ϕ_2 collecting all inferences of equality made within T_1 and T_2. If ϕ_1 or ϕ_2 is unsatisfiable, return unsatisfiable. Otherwise, if there is an inferred equality between constants $u = v$ that is not inferred by one of ϕ_1 or ϕ_2, then add $u = v$ to the set that does not infer it and repeat this step.

2. If ϕ_1 or ϕ_2 infers a disjunction of equalities between constants without inferring any of the equalities alone, then, for every such equality $u = v$ inferred by ϕ_1, call **CTSolver**$(\phi_1, \phi_2 \cup \{u = v\}, P_1, P_2)$ and for every such equality $u = v$ inferred by ϕ_2, call **CTSolver**$(\phi_1 \cup \{u = v\}, \phi_2, P_1, P_2)$. If, for every inferred equality both calls to **CTSolver** return unsatisfiable, then return unsatisfiable, otherwise return satisfiable.

3. If neither ϕ_1 nor ϕ_2 infers an equality or a disjunction of equalities, then return satisfiable.

□

Figure 21: Combined theory solver for two theories.

separately. Step **1.** applies decision procedures P_1 and P_2 to ϕ_1 and ϕ_2. Both P_1 and P_2 not only decide whether their respective theories are consistent, but also infer the maximum number of equalities from ϕ_1 and ϕ_2 within their respective theories. If no inconsistencies are found, these inferences are propagated out of their theories and the step is repeated, otherwise "unsatisfiable" is returned. Step **2.** does a case split in case a disjunction of equalities is inferred by one of the partitioned sets of constraints. Step **3.** returns "satisfiable" if nothing else can be inferred and no inconsistencies between current sets ϕ_1 and ϕ_2 have been discovered. The algorithm always terminates since the number of shared constants is finite and the number of equalities added is directly related to that number. The algorithm closely follows the algorithm of the original Nelson-Oppen paper [96].

Completeness of Algorithm 4. is not difficult to see. Soundness is proved by observing that the algorithm produces an interpolant for ϕ_1 and ϕ_2: namely the residue of $\phi_1 \wedge \phi_2$. The residue of a formula ϕ, denoted $Res(\phi)$, is the strongest collection of equalities between constants that are inferred by the formula. The important property of residues is that if ϕ_1 and ϕ_2 are formulas whose signatures overlap only in constants, then $Res(\phi_1 \wedge \phi_2)$ is logically equivalent to $Res(\phi_1) \wedge Res(\phi_2)$. Thus, the residue of the con-

junction of all formulas derived within each theory is the conjunction of the residues of formulas for each theory. The algorithm returns "satisfiable" only if all theories are consistent and no additional equalities of constants can be inferred. In this case the residues of all theories are satisfiable. It may be observed that the conjunction of the residues is therefore satisfiable. Hence, the residue of the conjunction of theories is satisfiable. This means if the algorithm returns satisfiable, then $\phi_1 \wedge \phi_2$ is satisfiable.

The purpose of this section is to acquaint the reader with procedures for implements SMT solvers. As such, much has been left out. We have not considered elimination of the purification step by shifting the burden of handling alien terms to the theory solvers themselves, application of theory specific rewrites, Shostak's theories and method, how to lift the requirement of stably infinite theories for completeness by propagating cardinality constraints in addition to equalities. Nelson-Oppen only works for quantifier free formulas but there are some techniques for allowing quantifiers which we will not talk about here. For more information about these topics the reader may start by consulting [107].

4.8 Stochastic Local Search

In DPLL and CDCL variants a search for a satisfying assignment proceeds by extending a partial assignment as long as no constraint is falsified. In Stochastic Local Search (SLS) there is a "current" complete assignment at every point in the search: if some clauses are not satisfied by the current assignment, one of a set of functions, each designed to create a new assignment from the given formula and current assignment, is applied according to some probability schedule to determine a new current assignment. This process continues until either all clauses are satisfied or until some function determines it is time to give up. Thus, SLS algorithms are generally *incomplete* and are not typically designed to provide a certificate for an unsatisfiable formula. A main strength of SLS algorithms is that they admit "long jumps" through the search space, and thereby avoid the condition of becoming mired in an unnecessarily long exploration of a search sub-space, particularly that for which all search trajectories lead to a local minimum number of satisfied clauses: deterministic solvers must take special measures to avoid that condition. On the other hand, it is difficult for SLS solvers to use the systematic tabulation and learning techniques that have made the deterministic variants successful.

4.8.1 Walksat

A successful early implementation of SLS for SAT is Walksat [108]. The Walksat algorithm is shown in Figure 22. It is safe to assume that all clauses in the input formula ψ have at least two literals since elementary and prudent preprocessing of ψ would have assigned a value to any variable in a unit clause to satisfy that clause and would have propagated the effect of that change to the remainder of ψ.

This algorithm has a few interesting features worth pointing out. One is

Algorithm 5.

Walksat (ψ, mt, mf, p)
/* *Input:* ψ *is a set of sets of literals representing a CNF formula,* */
/* mt *is the maximum number of solution attempts,* */
/* mf *is the maximum number of assignment changes,* */
/* p *is a probability* */
/* *Output is either a model for* ψ *or "give up"* */
Repeat the following mt times:
Set $M \leftarrow$ a random truth assignment for ψ.
Repeat the following mf times:
If M is a model for ψ, Output M.
Set $c \leftarrow$ a random falsified clause of ψ.
If $\exists l \in c$ such that flipping $var(l)$ does not
falsify a satisfied clause, Set $v \leftarrow var(l)$.
Otherwise, Set $v \leftarrow var(l)$ where, with probability p, l is
chosen randomly from c; and otherwise is the literal $l \in c$
for which the number of satisfied clauses that become
falsified by flipping $var(l)$ is minimum.
Flip v.
Output "give up"
□

Figure 22: *Walksat.*

the probability p which is used to choose a variable whose value should be reversed (from now on the word 'flipped' is used to mean just this): whether a random literal or the one whose flipped value results in the minimum number of satisfied clauses that become falsified. Experiments suggest the optimal value for p depends on and is sensitive to the particular class of problem the algorithm is applied to [71]. For example, it has been found that $p = 0.57$ is best for random 3-SAT formulas and this point sits at the minimum of an upward facing, cup-shaped curve with steeply increasing slopes on either side of the minimum. A second interesting feature is that a flip is chosen randomly *only if* there is not a variable in c which does not "break" any satisfied clauses. Thus, emphasizing positive movement to the goal of satisfying the most clauses as early as possible, Walksat has weakened some of the protection against loop stagnation that random flips provide. A third feature is the parameter mf which limits exploration of a local search space: if significant progress is not being made for a while then exploration of this local space terminates and a long jump to a different local search space is made and search resumed. This "restart" mechanism compensates for the possible problem non-random flips may get into as mentioned above. Walksat may be applied to MAX-SAT problems. A version for weighted MAX-SAT problems is available [70].

4.8.2 Novelty Family

Novelty [92] is a variant of Walksat that uses a more sophisticated variable selection algorithm. Once a falsified clause c is chosen, a score is computed for each variable v represented in c, based on the total number of satisfied clauses that would result if v were flipped. If the variable with highest score is not the variable that was most recently flipped, it is flipped now. Otherwise, it becomes the flipped variable with probability $1 - p$ or the second highest scoring variable becomes the flipped variable with probability p.

Although Novelty is an improvement over Walksat in many empirical cases it has the problem that its inner loop may not terminate for a given M [58]. Novelty+ is a slight modification that eliminates this problem by introducing a second probability wp: a variable is randomly selected from c and flipped with probability wp and otherwise the Novelty flip heuristic is applied. It is reported that $wp = 0.01$ is sufficient to eliminate the non-termination problem [60] (that is, a solution will be found with arbitrarily high probability, if one exists, even if $mt = 1$, as long as mf is set arbitrarily high).

Adaptive Novelty+ provides one more tweak to the Walksat/Novelty+ approach: the noise parameter p is allowed to vary during search. It has been mentioned above that algorithm performance is sensitive to p: thus, the optimal value for p depends significantly on the particular class of input formulas. Adaptive Novelty+ [59] attempts to mitigate this problem. If p is increased, the probability of inner loop stagnation is reduced. If p is decreased, the ability of the algorithm to follow a gradient to a solution is improved. These two counteracting conditions are thought to explain the sensitivity of p to problem domain. Adaptive Novelty+ begins at $p = 0$ and witnesses a rapid improvement in the number of clauses satisfied. When stagnation is detected p is increased. Such increases continue periodically until stagnation is overcome and then p is reduced. The cycle is continued until termination. Stagnation is detected when it is noticed that an objective function has not changed over a certain number of steps, called the Stagnation Detection Interval. The usual objective function is the number of satisfied clauses. One specific adaptation strategy is the following:

Stagnation Detection Interval:	$m/6$, m=number of clauses
Incremental increase in p:	$p = p + (1 - p) * 0.2$
Incremental decrease in p:	$p = p - 0.4 * p$

The asymmetry between increases and decreases in p is motivated by the fact that detecting search stagnation is computationally more expensive than detecting search progress and that it is advantageous to approximate optimal noise levels from above rather than from below [59]. After p has been reset the current objective function value is stored and becomes the basis for detecting stagnation. Observe that the factors 1/6, 0.4, and 0.2 are parameters that could be adjusted but according to [59] performance appears to be less sensitive to these parameters than it is to p.

4.8.3 Discrete Lagrangian Methods

Discrete Lagrangian Methods are another class of SLS algorithms which control the ability of a search to escape a sub-space that contains a local minimum [109]. Let $\psi = \{c_1, c_2, \ldots, c_m\}$ be a CNF formula with clauses $c_1, \ldots, c_m$, let $V = \{v : \exists\ c \in \psi\ s.t.\ v \in c \text{ or } \neg v \in c\}$, and attach labels to variables to write $V = \{v_1, v_2, \ldots, v_n\}$. Define, for all $c_i \in \psi$ and $v_j \in V$:

$$Q_{i,j}(v_j) = \begin{cases} 1 - v_j & \text{if } v_j \in c_i \\ v_j & \text{if } \neg v_j \in c_i \\ 1 & \text{otherwise.} \end{cases}$$

Let x be an n-dimensional 0-1 vector where x_j holds the value of variable v_j and let $C_i(x) = \prod_{j=1}^{n} Q_{i,j}(v_j)$. Define

$$N(x) = \sum_{i=1}^{m} C_i(x).$$

Then $N(x)$ is the number of unsatisfied clauses in ψ under the assignment x. Consider the following optimization problem:

$$\begin{array}{ll} \text{minimize over } x \in \{0,1\}^n & N(x) = \sum_{i=1}^{m} C_i(x) \\ \text{subject to} & C_i(x) = 0 \quad \forall i \in \{1, 2, \ldots, m\}. \end{array}$$

The objective function (the first line of the above) is enough to specify the SAT problem. The extra constraints are added to guide the search. In particular, the extra constraints help bring the search out of a local minimum.

The above overconstrained optimization problem may be converted to an unconstrained optimization problem by applying a Lagrange multiplier λ_i to each constraint $C_i(x)$ and adding the result to the objective function. Writing λ to represent the vector $\langle \lambda_1, \ldots, \lambda_m \rangle$ of Lagrange multipliers and $\mathbf{C}(\mathbf{x})$ to represent the vector $\langle C_1(x), \ldots, C_m(x) \rangle$, define

$$L(x, \lambda) = N(x) + \lambda^T \mathbf{C}(x)$$

where $x \in \{0,1\}^n$ and λ_i can be real valued. A saddle point (x^*,λ^*) of $L(x, \lambda)$ is defined to satisfy the following condition:

$$L(x^*, \lambda) \leq L(x^*, \lambda^*) \leq L(x, \lambda^*)$$

for all λ sufficiently close to λ^* and for all x whose Hamming distance between x^* and x is 1. The *Discrete Lagrangian Method* (DLM) for solving CNF SAT can be defined as a set of equations which update x and λ:

$$x^{k+1} = x^k - \Delta_x L(x^k, \lambda^k)$$
$$\lambda^{k+1} = \lambda^k + \mathbf{C}(x^k)$$

where $\Delta_x L(x, \lambda)$ is the *discrete gradient operator* with respect to x such that $\Delta_x L(x, \lambda) = \langle \delta_1, \delta_2, \ldots, \delta_n \rangle$, $\forall_{1 \leq i \leq n} \delta_i \in \{-1, 0, 1\}$, $\sum_{1 \leq i \leq n} |\delta_i| = 1$, and $(x - \Delta_x L(x, \lambda)) \in \{0,1\}^n$ (that is, $\Delta_x L(x, \lambda)$ identifies a variable to be flipped and $x - \Delta_x L(x, \lambda)$ is a neighbor of x - that is, at Hamming distance 1

Algorithm 6.

DLM (ψ)
/* *Input:* ψ *is a set of sets of literals representing a CNF formula,* */
/* *Output is either a model for* ψ *or "give up"* */
Set $x \leftarrow$ a random n-dimensional 0-1 vector.
Set $\lambda \leftarrow \langle 0, 0, \ldots, 0 \rangle$.
Construct $\mathbf{C}(x)$ from ψ.
Repeat the following while $N(x) > 0$:
Set $x \leftarrow x - \Delta_x L(x, \lambda)$.
If condition to update λ is satisfied, do this:
$\lambda \leftarrow \lambda + \gamma \cdot \mathbf{C}(x)$.
Output x
□

Figure 23: *Discrete Lagrangian Method.*

from x). There are two ways to calculate $\Delta_x L(x, \lambda)$. One way, termed *greedy*, replaces the current assignment with a neighboring assignment leading to the maximum decrease in the Lagrangian function value. Another way, termed *hill−climbing*, replaces the current assignment with a neighboring assignment that leads to a decrease in the value of the Lagrangian function. Both involve some local search in the neighborhood of the current assignment but, for every greedy update the complexity of this search is $O(n)$ whereas the hill-climbing update generally is less expensive.

An algorithmic sketch of DLM is shown in Figure 23. The parameter γ controls the magnitude of the changes to λ_i. A value of $\gamma = 1$ is reported to be sufficient for many benchmarks but a smaller γ has resulted in improved performance for some difficult ones. Experiments have shown that λ should not be updated every time the current assignment is replaced. This is because the changes in $L(x, \lambda)$ are usually very small and relatively larger changes in λ can overcompensate causing the search to become lost. Therefore, the DLM algorithm contains a line stating that a condition must be satisfied before λ can be updated. One possibility is that λ is not updated until $\Delta_x L(x, \lambda) = 0$. When that happens the local minimum is reached and the change in λ is needed to jump to a different section of the search space. This strategy may cause a problem if a search-space plateau is entered. In that case, it will be prudent to change λ sooner. For improved variants of DLM see [127, 128].

4.9 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) as graphs were discussed in Section 2.2. 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 2.2. A ROBDD is a BDD such

that: 1) There is no vertex v such that $then(v) = else(v)$; 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, $f(v) = (v \wedge f(then(v))) \vee (\neg v \wedge f(else(v)))$.

Then $f(root(v))$ 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 BDD will be used 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, $then(v)$, and $else(v)$, 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, $then(..)$ and $else(..)$ are empty. Denote by $terminal(1)$ and $terminal(0)$ 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. A hashtable, commonly referred to as a *unique table*, is maintained for the purpose of finding a node's array location given $v, then(v), else(v)$. If there is no corresponding entry in the hashtable a new node is created and recorded in the hashtable with key $v, then(v), else(v)$. A single procedure called **findOrCreateNode** takes $v, then(v), else(v)$ as arguments and returns the BDD array index of a node, either a newly created one or an existing one. This procedure is shown in Figure 24.

The main BDD construction operation is to find and attach two descendant nodes ($then(v)$ and $else(v)$) to a parent node (v). The procedure **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 **buildBDD**, shown in Figure 25. It is assumed that variable indices match the value of *index* applied to that variable (thus, $i = index(v_i)$). The complexity of **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 $\mathbf{reduce}_1$ and $\mathbf{reduce}_0$, shown in Figure 26 are used to describe several important BDD operations in subsequent sections. Assuming v is the root of the BDD representing f, the operation $\mathbf{reduce}_1(v, f)$ returns f constrained by the assignment of 1 to variable v and $\mathbf{reduce}_0(v, f)$ returns f constrained by the assignment of 0 to the variable v.

Details on performing the common binary operations of $\wedge$ and $\vee$ on BDDs will be ignored here. The reader may refer to [6] for detailed descriptions.

Algorithm 7.

findOrCreateNode (v, t, e)
/* *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 $node \leftarrow lookup(\langle v, t, e\rangle)$.
If $node \neq$ `null` then Return $node$.
Set $node \leftarrow createNode(\langle v, t, e\rangle)$.
$insert(\langle v, t, e\rangle, node)$.
Return $node$.
□

Figure 24: Procedure for finding a node or creating and inserting it.

Algorithm 8.

buildBDD (f, i)
/* *Input: Boolean function* f, *index* i */
/* *Output: root* `Node` *of BDD representing* f */
If $f \Leftrightarrow 1$ return $terminal(1)$.
If $f \Leftrightarrow 0$ return $terminal(0)$.
Set $t \leftarrow$ **buildBDD**$(f|_{v_i=1}, i+1)$.
Set $e \leftarrow$ **buildBDD**$(f|_{v_i=0}, i+1)$.
Return **findOrCreateNode**(v_i, t, e).
□

Figure 25: Algorithm for building a BDD: invoked using **buildBDD**(f,1).

Algorithm 9.

reduce$_1$ (v, f)
/* *Input: variable* v, *BDD* f */
/* *Output: reduced BDD* */
If $root(f) == v$ then
Return $then(root(f))$.
Return f.
□

reduce$_0$ (v, f)
/* *Input: variable* v, *BDD* f */
/* *Output: reduced BDD* */
If $root(f) == v$ then
Return $else(root(f))$.
Return f.

Figure 26: Operations **reduce**$_1$ and **reduce**$_0$.

Algorithm 10.

exQuant (f, v)
/* *Input: BDD* f*, variable* v */
/* *Output: BDD* f *with* v *existentially quantified away* */
If $root(f) == v$ then Return $then(root(f)) \vee else(root(f))$.
If $index(v) > index(\text{root}(f))$ then Return 0. // If v is not in f do nothing
Set $h_{f_1} \leftarrow$ **exQuant**$(then(root(f)), v)$.
Let $h_{f_0} \leftarrow$ **exQuant**$(else(root(f)), v)$.
If $h_{f_0} == h_{f_1}$ then Return h_{f_1}.
Return **FindOrCreateNode**$(root(f), h_{f_1}, h_{f_0})$.
□

Figure 27: *Algorithm for existentially quantifying variable* v *away from BDD* f*. The* $\vee$ *denotes the "or" of BDDs.*

Here, it is only mentioned 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 $\wedge$ 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, particularly 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 collection of BDDs, then they may be preprocessed to reveal information that may be exploited later, during search. In particular, inferences[6] may be determined and used to reduce input complexity. The discussion of the next four sections emphasizes this role.

4.9.1 Existential Quantification

A Boolean function which can be written

$$f(v, \vec{x}) = (v \wedge h_1(\vec{x})) \vee (\neg v \wedge h_2(\vec{x}))$$

can be replaced by

$$f(\vec{x}) = h_1(\vec{x}) \vee h_2(\vec{x})$$

where $\vec{x}$ is a list of one or more variables. There is a solution to $f(\vec{x})$ if and only if there is a solution to $f(v, \vec{x})$ so it is sufficient to solve $f(\vec{x})$ to get a solution to $f(v, \vec{x})$. Obtaining $f(\vec{x})$ from $f(v, \vec{x})$ is known as *existentially quantifying* v *away from* $f(v, \vec{x})$. This operation is efficiently handled if $f(v, \vec{x})$ 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

[6]Finding inferences is referred to in the BDD literature as finding *essential* values to variables and a set of inferences (a conjunction of literals) is referred to as a *cube*.

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 27.

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 28), those same inferences are revealed during BDD construction if inference lists for each node are built and maintained. Therefore, a more effective use of existential quantification is in support of other operations, such as strengthening (see Section 4.9.5), to uncover those inferences that cannot be found during BDD construction or in tandem with $\wedge$ 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 29. The speedup can overcome the lost intelligence but it is sometimes better to turn it off.

4.9.2 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 $f \wedge c$ and $g \wedge c$ 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

$$f = (v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee v_2) \text{ and}$$
$$c = (v_1 \vee \neg v_2).$$

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 $f \wedge c$ is the same as $g \wedge c$ but g is smaller than f. As an example of discovering inferences consider

$$f = (v_1 \rightarrow v_2) \wedge (\neg v_1 \rightarrow (\neg v_3 \wedge v_4)) \text{ and}$$
$$c = (v_1 \vee v_3).$$

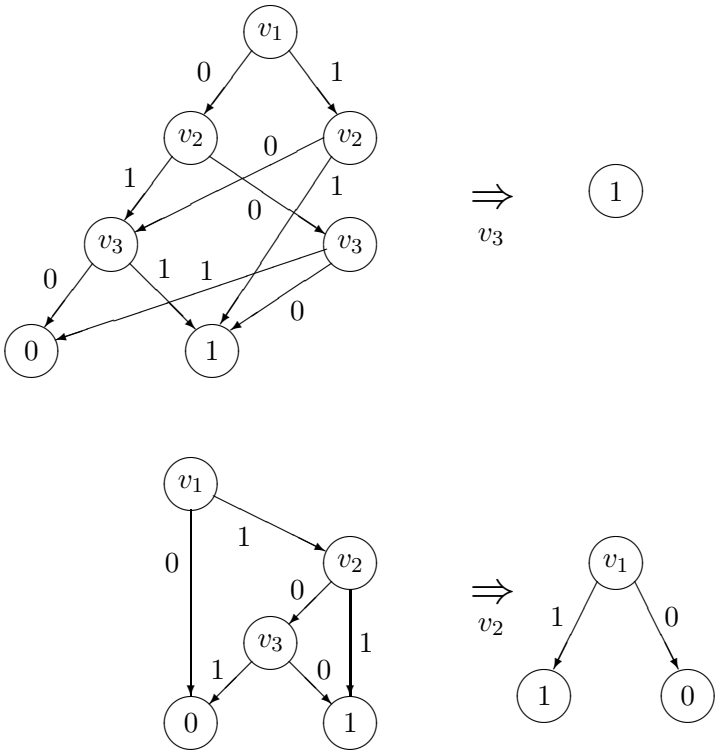


Figure 28: Two examples of existentially quantifying a variable away from a function. Functions are represented as BDDs on the left. Variable v_3 is existentially quantified away from the top BDD leaving 1, meaning that regardless of assignments given to variables v_1 and v_2 there is always an assignment to v_3 which satisfies the function. Variable v_2 is existentially quantified away from the bottom BDD leaving the inference $v_1 = 1$.

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 $g = (v_1) \wedge (v_2)$ which reveals two inferences. The BDDs for f, c, and g of this example are shown in Figure 31. The example showing BDD elimination is deferred to Theorem 9, Section 4.9.4.

Clearly, there are numerous strategies for creating g from f and c and replacing f with g. An obvious one is to have g map to 0 all rows that c maps to 0. This strategy, which will be called zero-restrict in this chapter, 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 33 produces $g = \neg v_3 \wedge (v_1 \vee (\neg v_1 \wedge \neg v_2))$ instead of the inference $g = \neg v_3$ which is obtained from a more intelligent replacement strategy. An 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 $g = \neg v_3$.

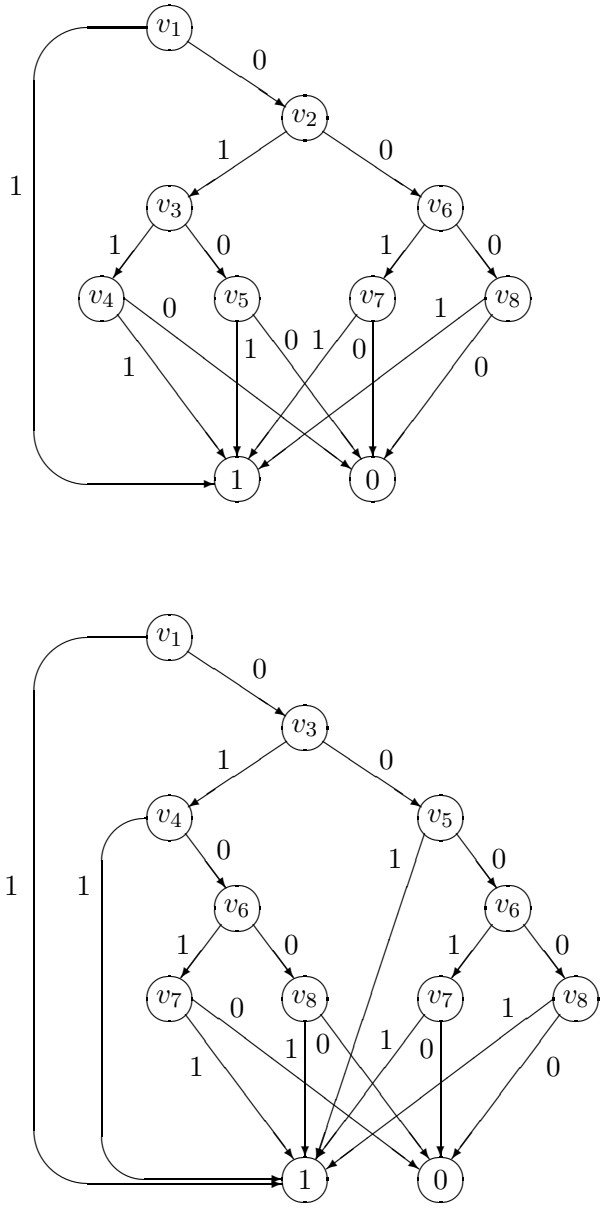


Figure 29: Existential quantification can cause blurring of functional relationships. The top function is seen to separate variables v_6, v_7, and v_8 from v_3, v_4, and v_5 if v_2 is chosen during search first. Existentially quantifying v_2 away from the top function before search results in the bottom 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 $c == \neg f$ return $terminal(0)$.
If $c == f$ return $terminal(1)$.
// f and c have a non-trivial relationship
Set $v_f \leftarrow root(f)$. // v_f is a variable
Set $v_c \leftarrow root(c)$. // 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 $h_{f_1} \leftarrow$ **restrict**(**reduce**$_1(v_f, f)$, **reduce**$_1(v_f, c)$).
Set $h_{f_0} \leftarrow$ **restrict**(**reduce**$_0(v_f, f)$, **reduce**$_0(v_f, c)$).
If $h_{f_1} == h_{f_0}$ then Return h_{f_1}.
Return **findOrCreateNode**(v_f, h_{f_1}, h_{f_0}).
□

Figure 30: Algorithm for restricting a BDD f by a BDD c.

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 32 and Figure 33 where, in the first case, no inference is generated but after f and c are swapped an inference is generated. The next two subsections show two replacement strategies that are among the more commonly used.

4.9.3 Restrict

The original version of restrict is what is called 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 [31]. In this chapter an alternative version which is implemented as Algorithm 11 of Figure 30 is considered. Use the 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 31 to 33 show examples that were referenced in the previous subsection.

Procedure **restrict** is similar to a procedure called generalized co-factor (**gcf**) or constrain (see the next subsection 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

$f = (v_1 \to v_2) \wedge (\neg v_1 \to (\neg v_3 \wedge v_4))$

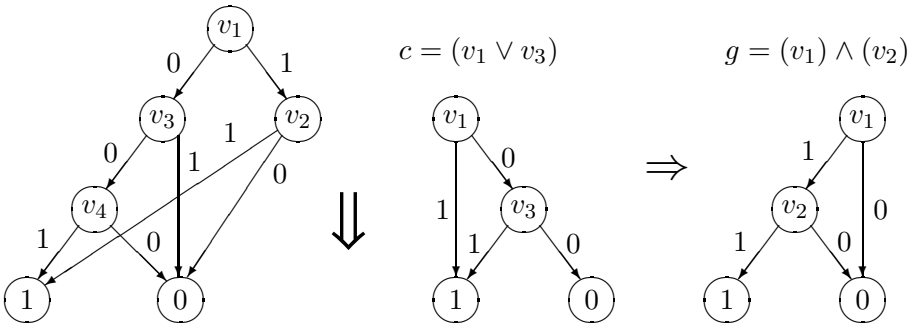


Figure 31: A call to **restrict**(f, c) returns the BDD g shown on the right. In this case inferences $v_1 = 1$ and $v_2 = 1$ are revealed. The symbol $\Downarrow$ denotes the operation.

$f = (v_1 \vee \neg v_2) \wedge (\neg v_1 \vee \neg v_3)$ $\quad$ $c = (v_2 \vee \neg v_3)$

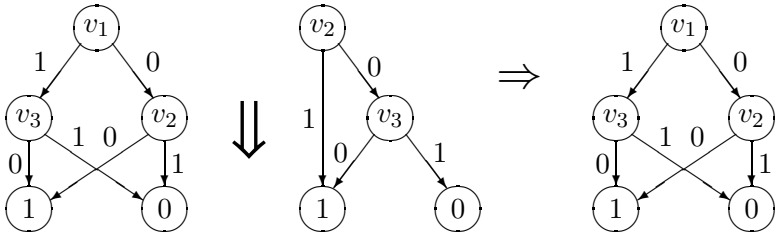


Figure 32: A call to **restrict**(f, c) results in no change.

$f = (v_2 \vee \neg v_3)$ $\quad$ $c = (v_1 \vee \neg v_2) \wedge (\neg v_1 \vee \neg v_3)$ $\quad$ $g = (\neg v_3)$

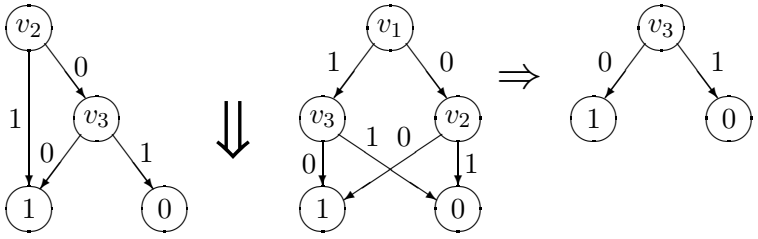


Figure 33: Reversing the roles of f and c in Figure 32, a call to **restrict**(f, c) results in the inference $g = \neg v_3$ as shown on the right. In this case, the large number of 0 truth table rows for c was exploited to advantage.

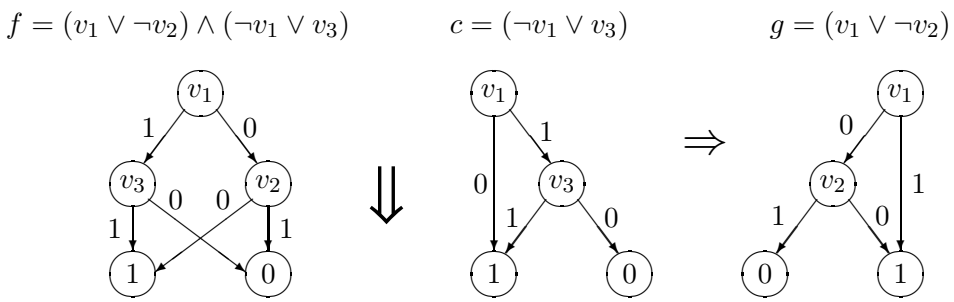


Figure 34: A call to **restrict**(f, c) spreads an inference that is evident in one BDD over multiple BDDs. If v_3 is assigned 0 in f then $v_1 = 0$ and $v_2 = 0$ are inferred. After replacing f with $g =$ **restrict**(f, c), to get the inference $v_2 = 0$ from the choice $v_3 = 0$, visit c to get $v_1 = 0$ and *then* g to get $v_2 = 0$. Thus, restrict can increase work if not used properly. In this case, restricting in the reverse direction leads to a better result.

preprocessing because the BDDs produced from it can never contain more variables than the BDDs they replace.

On the negative side, it can, in odd cases, cause a garbling of local information. Moreover, although **restrict** may reveal some of the inferences that strengthening would (see below), it can still cause the number of search choicepoints to increase. Both these issues are related: **restrict** can spread an inference that is evident in one BDD over multiple BDDs (see Figure 34 for an example).

4.9.4 Generalized Co-factor

The *generalized co-factor* operation, also known as *constrain*, is denoted here by | and implemented as **gcf** (Algorithm 12) in Figure 35. 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 [32].

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 π of the numbers $1, 2, \ldots, n$ 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 $\sum_{i=1}^{n} 2^{n-i}(t'_{\pi_i} \oplus t''_{\pi_i})$. 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 36 shows BDDs f and c under the variable ordering given by $\pi = \langle 1, 2, 3, 4 \rangle$. For assignment vectors $\langle * * 01 \rangle$, $\langle * * 10 \rangle$, $\langle * * 11 \rangle$

Algorithm 12.

gcf (f, c)
/* *Input: BDD* f*, BDD* c */
/* *Output: greatest co-factor of* f *by* c */
If $f == terminal(0)$ or $c == terminal(0)$ return $terminal(0)$.
If $c == terminal(1)$ or $f == terminal(1)$ return f.
Set $v_m \leftarrow index^{-1}(\min\{index(root(c)), index(root(f))\})$.
// v_m is the top variable of f and c
If $\mathbf{reduce}_0(v_m, c) == terminal(0)$ then
  Return $\mathbf{gcf}(\mathbf{reduce}_1(v_m, f), \mathbf{reduce}_1(v_m, c))$.
If $\mathbf{reduce}_1(v_m, c) == terminal(0)$ then
  Return $\mathbf{gcf}(\mathbf{reduce}_0(v_c, f), \mathbf{reduce}_0(v_c, c))$.
Set $h_1 \leftarrow$ $\mathbf{gcf}(\mathbf{reduce}_1(v_m, f), \mathbf{reduce}_1(v_m, c))$.
Set $h_0 \leftarrow$ $\mathbf{gcf}(\mathbf{reduce}_0(v_m, f), \mathbf{reduce}_0(v_m, c))$.
If $h_1 == h_0$ then Return h_1.
Return $\mathbf{FindOrCreateNode}(v_m, h_1, h_0)$.
□

Figure 35: Algorithm for finding a greatest common co-factor of a BDD.

(where $*$ is a wildcard meaning 0 or 1), $\mathbf{gcf}(f, c)$, shown as the BDD at the bottom of Figure 36, 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 $\langle 0001 \rangle$. $\langle 0101 \rangle$, $\langle 1001 \rangle$, and $\langle 1101 \rangle$, respectively. On all these inputs $\mathbf{gcf}(f, c)$ has value 1, which the reader can check in Figure 36.

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

Theorem 9. *Given BDDs* $f_1, ..., f_k$*, for any* $1 \leq i \leq k$*,* $f_1 \wedge f_2 \wedge ... \wedge f_k$ *is satisfiable if and only if* $(f_1|f_i) \wedge ... \wedge (f_{i-1}|f_i) \wedge (f_{i+1}|f_i) \wedge ... \wedge (f_k|f_i)$ *is satisfiable. Moreover, any assignment satisfying the latter can be mapped to an assignment that satisfies* $f_1 \wedge ... \wedge f_k$*.*

Proof. If it can be shown that

$$(f_1|f_i) \wedge \ldots \wedge (f_{i-1}|f_i) \wedge (f_{i+1}|f_i) \wedge \ldots \wedge (f_k|f_i) \tag{4}$$

is satisfiable if and only if

$$(f_1|f_i) \wedge \cdots \wedge (f_{i-1}|f_i) \wedge (f_{i+1}|f_i) \wedge \ldots \wedge (f_k|f_i) \wedge f_i \tag{5}$$

is satisfiable then, since (5) is equivalent to $f_1 \wedge \ldots \wedge f_k$, the first part of the theorem will be proved. Suppose (5) is satisfied by truth assignment t. That t represents a truth table row that f_i maps to 1. Clearly that assignment also satisfies (4). Suppose no assignment satisfies (5). Then all assignments for which f_i maps to 1 do not satisfy (4) since otherwise (5) 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 (4) and (5) 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 (5) because it cannot satisfy (4) by the argument above. Hence, there is no truth assignment falsifying f_i but satisfying (4) so the first part is proved.

For the second part, observe that any truth assignment that satisfies (4) and (5) also satisfies $f_i \wedge \ldots \wedge f_k$ so it is only necessary to consider assignments t that satisfy (4) but not (5). 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 $f_1 \wedge \ldots \wedge f_k$. □

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 $\mathbf{gcf}(f_1, f_i) \wedge \ldots \wedge \mathbf{gcf}(f_k, f_i)$ and then extends the assignment to satisfy f_i, otherwise the solver reports that the instance has no solution. However, unlike **restrict**, generalized co-factoring cannot by itself reduce the number of variables in a given collection of BDDs. Other properties of the **gcf** operation, all of which are easy to show, are:

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

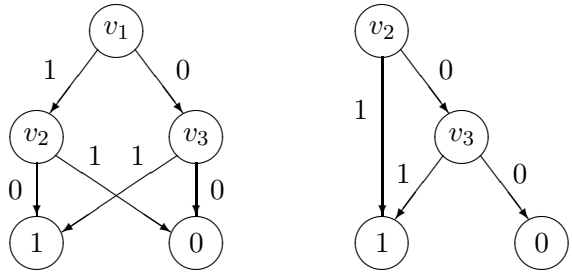
Care must be taken when co-factoring in "both" directions (exchanging f for c). For example, $f \wedge g \wedge h$ cannot be replaced by $(g|f) \wedge (f|g) \wedge h$ since the former may be unsatisfiable when the latter is satisfiable.

Examples of the application of **gcf** are shown in Figures 36 and 37. Figure 36 illustrates the possibility of increasing BDD size. Figure 37 presents the same example after swapping v_1 and v_3 under the same variable ordering and shows that the result produced by **gcf** is sensitive to variable ordering. Observe that the functions produced by **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 **gcf** depends on the variable ordering as well.

4.9.5 Strengthen

This binary operation on BDDs helps reveal inferences that are missed by **restrict** due to its sensitivity to variable ordering. Given two BDDs, b_1 and

$f = (v_1 \rightarrow \neg v_2) \vee (\neg v_1 \rightarrow v_3)$ $\quad$ $c = (v_2 \vee v_3)$



$gcf(f, c) = (v_1 \rightarrow \neg v_2) \vee (\neg v_1 \rightarrow (v_2 \rightarrow v_3))$

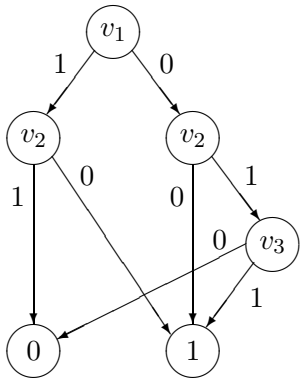


Figure 36: Generalized co-factor operation on f and c as shown. In this case the result is more complicated than f. The variable ordering is $v_1 < v_2 < v_3$.

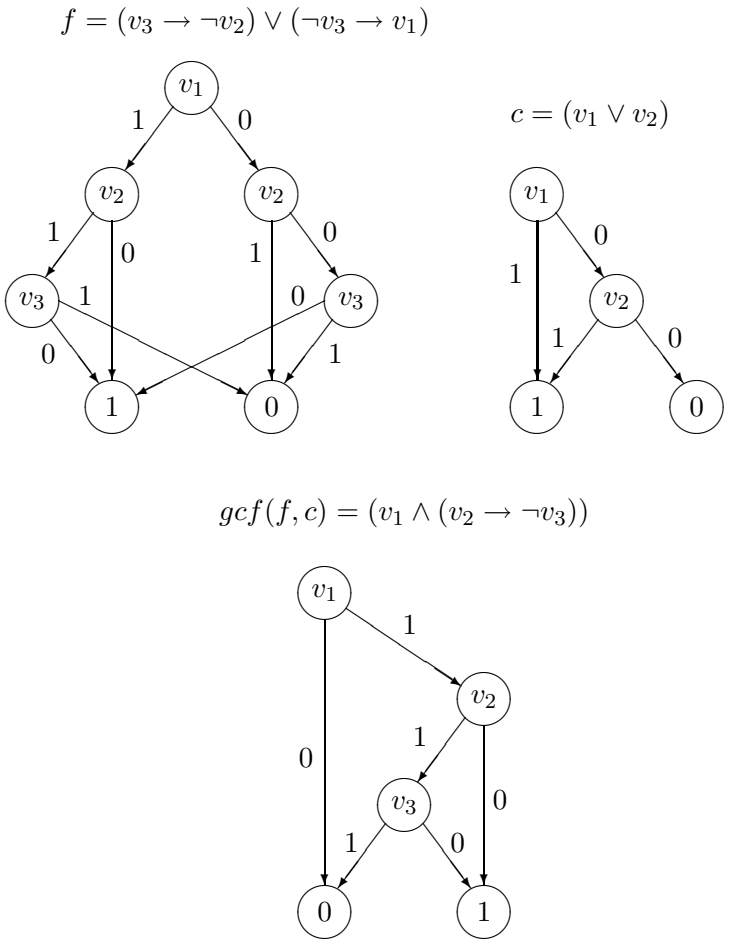


Figure 37: Generalized co-factor operation on the same f and c of Figure 36 and with the same variable ordering but with v_1 and v_3 swapped. In this case the result is less complicated than f and the assignment $\{v_1, v_2\}$ causes the output of **gcf** in this figure to have value 1 whereas the output of **gcf** in Figure 36 has value 0 under the same assignment.

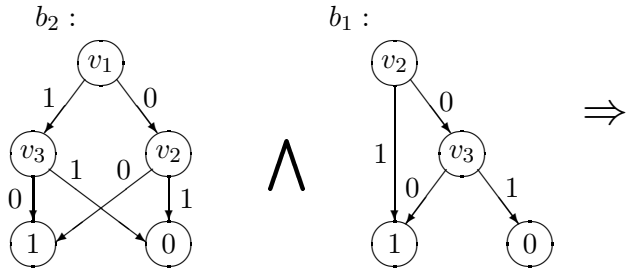
Algorithm 13.

Strengthen (b_1, b_2)
/* *Input: BDD* b_1*, BDD* b_2 */
/* *Output: BDD* b_1 *strengthened by* b_2 */
Set $\vec{x} \leftarrow \{x : x \in b_2, x \notin b_1\}$.
Repeat the following for all $x \in \vec{x}$:
Set $b_2 \leftarrow$ **exQuant**(b_2, x).
Return $b_1 \wedge b_2$.
□

Figure 38: Algorithm for strengthening a BDD by another.

b_2, strengthening conjoins b_1 with the *projection* of b_2 onto the variables of b_1: that is, $b_1 \wedge \exists \vec{v}\, b_2$, 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_js sometimes reveals additional inferences or equivalences. Algorithm **strengthen** is shown in Figure 38. Figure 39 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 **strengthen** shares information between BDDs, it can be thought of as sharing intelligence and *strengthening* the relationships between functions; the added intelligence in these strengthened functions can be exploited by a smart search heuristic. We have found that **strengthen** usually decreases the number of choicepoints when a particular search heuristic is employed, but sometimes it causes more choicepoints. It may be conjectured this is due to the delicate nature of some problems where duplicating information in the BDDs leads the heuristic astray.



Strengthening example: Existentially quantify v_1 away from b_2...

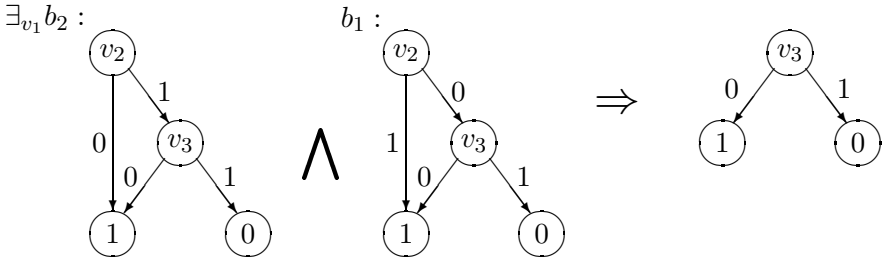


Figure 39: ...then conjoin the two BDDs. Inference $v_3 = 0$ is revealed.

Procedure **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.

We close this section by mentioning that for some classes of problems res-

olution 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.

4.10 Decompositions

The variable elimination methods of Sections 4.5 and 4.11 recursively *decompose* a given formula ψ into overlapping subformulas ψ_1 and ψ_2 such that the solution to ψ can be inferred from the solutions to ψ_1 and ψ_2. The decompositions are based on occurrences of a selected variable v in clauses of ψ and each subformula has at least as many clauses as those of ψ 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 ψ to ψ_1 and ψ_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: so, if $k = 3$ and m/n is, say 4, then the average number of occurrences of a randomly chosen variable is only 12. Hence, the methods often suffer computational inadequacies 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 generality for the sake of producing subformulas of relatively small size. Truemper's book [119] 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 chapter.

4.10.1 Monotone Decomposition

Let CNF formula ψ of m clauses and n variables be represented as a $m \times n$ $(0, \pm 1)$-matrix $\mathcal{M}_\psi$ (defined in Section 2.1). A *monotone decomposition* of $\mathcal{M}_\psi$, is a permutation of rows and columns of $\mathcal{M}_\psi$ 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:

$$\left(\begin{array}{c|c} \mathcal{A}^1 & \mathcal{E} \\ \hline \mathcal{D} & \mathcal{A}^2 \end{array}\right) \tag{6}$$

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.

Algorithm 14.

Monotone Decomposition Solver (ψ)
/* *Input: CNF formula* ψ *as* $(0, \pm 1)$ $\mathcal{M}_\psi$ *monotone decomposition* */
/* *Output: "unsatisfiable" or a model for* ψ */
/* *Locals: set of variables* M_1, M_2 */
Let $\mathcal{M}_\psi$ be partitioned according to (6).
If Horn formula $\mathcal{A}^1$ is unsatisfiable, Output "unsatisfiable."
Let M_1 be a unique minimum model for the Horn formula $\mathcal{A}^1$.
Remove from $\mathcal{A}^2$ all rows common to $\mathcal{D}$'s that are satisfied by M_1.
If $\mathcal{A}^2$ is unsatisfiable, Output "unsatisfiable."
Let M_2 be a model for $\mathcal{A}^2$.
Output $M_1 \cup M_2$.
□

Figure 40: *Algorithm for determining satisfiability of a Monotone Decomposition.*

The submatrix $\mathcal{A}^1$ represents a *Horn Formula*. In Section 5.2 Horn formulas are shown to have the following two important properties: they are solved efficiently, for example by Algorithm 20 of Figure 46, and, by Theorem 20, 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, $M \subset M'$). This property, plus the nature of submatrices $\mathcal{D}$ and $\mathcal{E}$, effectively allow a split of the problem of determining the satisfiability of ψ into two independent problems: namely, determine satisfiability for the Horn formula represented by $\mathcal{A}^1$ and determine satisfiability for the subformula represented by $\mathcal{A}^2$. The algorithm of Figure 40 shows this in more detail. The following theorem proves correctness of this algorithm.

Theorem 10. *Let CNF formula* ψ *be represented as a monotone decomposition* $(0, \pm 1)$*-matrix. On input* ψ*, Algorithm 14 outputs "unsatisfiable" if and only if* ψ *is unsatisfiable and if* ψ *is satisfiable, then the output set* $M_1 \cup M_2$ *is a model for* ψ*.*

Proof. Clearly, if Horn formula $\mathcal{A}^1$ is unsatisfiable then so is ψ. So, suppose there is a model M_1 for $\mathcal{A}^1$ and consider the rows of $\mathcal{A}^2$ remaining after rows common to those of $\mathcal{D}$ which are satisfied by M_1 are removed. Since M_1 is a unique minimum model for $\mathcal{A}^1$, no entries of $\mathcal{D}$ are $+1$, and variables of $\mathcal{A}^1$ are distinct from variables of $\mathcal{A}^2$, no remaining row of $\mathcal{A}^2$ can be satisfied by any model for $\mathcal{A}^1$. Therefore, if these rows of $\mathcal{A}^2$ are unsatisfiable, then so is ψ. On the other hand, if these rows are satisfied by model M_2, then clearly, $M_1 \cup M_2$ is a model for ψ. □

A $(0, \pm 1)$ matrix $\mathcal{M}_\psi$ representing CNF formula ψ may have more than one monotone decomposition. Of particular interest is the *maximum monotone decomposition* of $\mathcal{M}_\psi$. That is, the monotone decomposition of ψ such that $\mathcal{A}^1$ has the greatest number of rows and columns. A monotone decomposition is said to be maximal with respect to the dimensions of $\mathcal{A}^1$. The

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

Theorem 11. *Any* $(0, \pm 1)$ *matrix* $\mathcal{M}$ *has a unique maximal monotone decomposition.*

Proof. Suppose $\mathcal{M}$ has two distinct maximal monotone decompositions, say $\mathcal{M}_1$ and $\mathcal{M}_2$. Let $\mathcal{A}_i^1$, $\mathcal{A}_i^2$, and $\mathcal{D}_i$, $i \in \{1, 2\}$, be the partition of $\mathcal{M}$, after column scaling, corresponding to $\mathcal{M}_i$ (see the partition (6) on page 79). Construct a new partition $\mathcal{M}'$ of $\mathcal{M}$ into $\mathcal{A}'^1$, $\mathcal{A}'^2$ and $\mathcal{D}'$ such that $\mathcal{A}'^1$ includes all rows and columns of $\mathcal{A}_1^1$ and $\mathcal{A}_2^1$. For those columns of $\mathcal{M}'$ that are also columns of $\mathcal{A}_1^1$ use a column scaling that is exactly the same as the one used in $\mathcal{M}_1$. For all other columns use the same scaling as in $\mathcal{M}_2$. The submatrix of $\mathcal{A}'^1$ that includes rows and columns of $\mathcal{A}_1^1$ is the same as $\mathcal{A}_1^1$ because the scaling of those columns is the same as for $\mathcal{M}_1$. The submatrix of $\mathcal{A}'^1$ including rows of $\mathcal{A}_1^1$ and columns not in $\mathcal{A}_1^1$ must be a 0 submatrix by the monotone decomposition $\mathcal{M}_1$. The submatrix of $\mathcal{A}'^1$ including columns of $\mathcal{A}_1^1$ and no rows of $\mathcal{A}_1^1$ must contain only 0 or -1 entries due to the $\mathcal{M}_1$ scaling and the submatrix including neither columns or rows of $\mathcal{A}_1^1$ must be Horn due to $\mathcal{M}_2$ column scaling. It follows that submatrix $\mathcal{A}'^1$ is Horn (at most one +1 in each row). It is similarly easy to check that the submatrix of $\mathcal{M}'$ consisting of rows of $\mathcal{A}'^1$ and columns other than those of $\mathcal{A}'^1$ is 0 and that the submatrix of $\mathcal{M}'$ consisting of columns of $\mathcal{A}'^1$ and rows other than those of $\mathcal{A}'^1$ contains no +1 entries. It follows that $\mathcal{M}'$ is a monotone decomposition. Since $\mathcal{A}_1^1 \supset \mathcal{A}'^1$ and $\mathcal{A}_2^1 \supset \mathcal{A}'^1$ neither $\mathcal{M}_1$ nor $\mathcal{M}_2$ is a maximal monotone decomposition in violation of the hypothesis. The theorem follows. □

From Theorem 11 there is always a maximum monotone decomposition for $\mathcal{M}_\psi$.

A maximum monotone decomposition is useful because: 1) $\mathcal{A}^1$, representing a Horn formula, is as large as possible so $\mathcal{A}^2$ is as small as possible; 2) Horn formulas may be efficiently solved by Algorithm 20; and 3) a maximum monotone decomposition can be found efficiently, as will now be shown.

A maximum monotone decomposition can be found using Algorithm 15 of Figure 41. 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 primes are used as in $\mathcal{E}'$ to refer to the current incantation of corresponding submatrices. Initially, that matrix is $\mathcal{M}_\psi$. At the end of a stage, if the algorithm needs another stage to produce a bigger decomposition, $\mathcal{A}'^2$ 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 $\mathcal{A}'^1$ and $\mathcal{D}'$ matrices of previous stages. The important operation is to move a non-positive column that intersects $\mathcal{A}'^2$ to just right of the border of the current stage's $\mathcal{A}'^1$ matrix, move the border of $\mathcal{A}'^1$ and $\mathcal{D}'$ to the right by one column, tag and move the rows containing 1 on the right boundary of the changed $\mathcal{D}'$ up to just below the border of $\mathcal{A}'^1$, and finally lower the border of $\mathcal{A}'^1$ and $\mathcal{E}'$ down to include the tagged rows. Doing so keeps $\mathcal{A}'^1$ Horn and $\mathcal{D}'$ non-positive

and enlarges $\mathcal{A}'^1$. If no non-positive column exists through $\mathcal{A}'^2$, no column can be made non-positive through $\mathcal{A}'^2$ 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 12. *Refer to Algorithm 15 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 a 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 $\mathcal{A}'^1$ except v by -1 blocks a monotone decomposition as well.*

Proof.

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

2. Consider columns in $\mathcal{A}'^1$ first. The proof is by induction on the number of columns processed in $\mathcal{A}'^1$. The base case has no such column: that is, $\mathcal{A}'^1$ 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 $\mathcal{A}'^1$ 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 $\mathcal{A}^1$ 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 $\mathcal{A}^1$ 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 $\mathcal{A}'^1$ 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.

A formula ψ which has a maximum monotone decomposition where $\mathcal{A}^2$ is a member of an efficiently solved subclass of Satisfiability obviously may be solved efficiently by Algorithm 14 if $\mathcal{A}^2$ can efficiently be recognized as belonging to such a subclass. Chapter 5 discusses several efficiently solved subclasses of Satisfiability problems which may be suitable for testing. If $\mathcal{A}^2$ represents a 2-SAT formula (See Section 5.1) then ψ is said to be q-Horn. The class of q-Horn formulas was discovered and efficiently solved in [15, 16] and it was the results of that that work led to the development of maximum monotone decompositions [118].

4.10.2 Autarkies and Safe Assignments

Definition 13. *An assignment to a set of variables is said to be* autark *or an* autarky *if all clauses that contain at least one of those variables are satisfied by the assignment. We will call a set of variables that is associated with an autarky an* autark set*.*

If all clauses satisfied by an autarky are removed from a CNF formula ψ, then the resulting formula is equivalent in satisfiability to ψ. A simple example of an autark set is a collection of one or more pure literals. A simple decomposition is to remove all clauses that contain pure literals. One can do the same for any autarky. However, discovering autarkies can be expensive. In some cases, though, an autarky can be found in polynomial time. This is treated in Section 5.9.

A similar decomposition exists for BDDs. Let f be a Boolean function and let $f\mid_v$ ($f\mid_{\neg v}$) denote the function obtained by setting variable v to 1 (respectively, 0).

Lemma 14. ([126]) *Given a conjunction of BDDs $f = f_1 \wedge ... \wedge f_m$ and variable v occurring in one or more BDDs of f, let f' be the conjunction of all BDDs in f which contain v. Let $f'_v = \neg(f'\mid_v) \wedge (f'\mid_{\neg v})$. Let $f'_{\neg v} = (f'\mid_v) \wedge \neg(f'\mid_{\neg v})$.*

1. *If f'_v has value 0, then $f\mid_v$ is satisfiable if and only if f is satisfiable.*
2. *If $f'_{\neg v}$ has value 0, then $f\mid_{\neg v}$ is satisfiable if and only if f is satisfiable.*

Lemma 14 states that if any of the BDDs in f are falsified or if all of the BDDs in f are satisfied by setting v to 1 (respectively, 0) then it is safe to make that assignment because the satisfiability of f does not change by doing so. We emphasize that the safe value for v is *not necessarily inferred*. This lemma provides a way to test whether a safe value exists for a variable, i.e. if $f'_v(f'_{\neg v})$ has value 0 then it is safe to set v to 1 (0) in f. However, to use this lemma directly requires conjoining all BDDs containing v and

Algorithm 15.

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

Figure 41: *Algorithm for finding the maximum monotone decomposition of a* $(0, \pm 1)$ *matrix.*

from Section 4.9 this could be expensive. In pursuit of an efficient method for finding safe assignments Lemma 14 may be used to derive the following weaker result:

Theorem 15. ([126]) *Given a conjunction of BDDs $f = f_1 \wedge ... \wedge f_m$ and variable v occurring in one or more BDDs of f, let f' be the conjunction of all BDDs in f which contain v. Without loss of generality, suppose $f' = f_1 \wedge ... \wedge f_n$ where $n \leq m$. Let $f'_v = (\neg(f_1|_v) \wedge f_1|_{\neg v}) \vee ... \vee (\neg(f_n|_v) \wedge f_n|_{\neg v})$. Let $f'_{\neg v} = (f_1|_v \wedge \neg(f_1|_{\neg v})) \vee ... \vee (f_n|_v \wedge \neg(f_n|_{\neg v}))$.*

1. *If f'_v has value 0, then $f\,|_v$ is satisfiable if and only if f is satisfiable.*
2. *If $f'_{\neg v}$ has value 0, then $f\,|_{\neg v}$ is satisfiable if and only if f is satisfiable.*

According to Theorem 15 a safe assignment may be found without having to conjoin BDDs containing v. This is practical when BDDs are fairly small and is more practical than conjoining BDDs if v appears in many of them. However, it is possible than a safe assignment discovered using Lemma 14 may be undiscoverable using Theorem 15.

The following is the corresponding theorem for safe assignments involving more than one variable:

Theorem 16. ([126]) *Given a conjunction of BDDs $f = f_1 \wedge ... \wedge f_m$ and a set of variables $V = \{v_1, ..., v_k\}$ each occurring in one or more BDDs of f, let f' be the conjunction of all BDDs in f which contain at least one of the variables in V. Let $\mathcal{M} = \{M_1, ..., M_{2^k}\}$ be the set of all possible truth assignments to the variables in V. Without loss of generality, suppose $f' = f_1 \wedge ... \wedge f_n$ where $n \leq m$. $\forall_{1 \leq i \leq 2^k}$, if $(\neg(f'\,|_{M_i}) \wedge (f'\,|_{M_1} \vee ... \vee f'\,|_{M_{2^k}}))$ has value 0 then $f\,|_{M_i}$ is satisfiable if and only if f is satisfiable.*

It is straightforward to turn Theorems 15 and 16 into an algorithm but this is not done here because numerous variants to speed up the search are possible and it is impractical to list all of them.

4.11 Branch-and-bound

The DPLL algorithm of Figure 17 is sequential in nature. At any point in the search only 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 potentially 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. A variable v is chosen for assignment from the subformula of the active node of lowest l value and that node is expanded. The 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 SAT, one of the most important being MAX-SAT (Page 7). It requires a function $g_\psi(M_{1:})$ which maps a given formula ψ 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 ψ 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_ψ 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_ψ. 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 42 as Algorithm 16. 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 4.12.2 shows how Linear Programming relaxations of Integer Programming representations of search nodes can be used as heuristic functions. This section concludes with an alternative to illustrate what else is possible.

Algorithm 16.

Branch-and-bound (ψ, h, g_ψ)
/* *Input: n variable CNF formula* ψ*, heuristic function* h*,* */
/* *objective function* g_ψ *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 $P \leftarrow \langle\langle \psi, M_{1:0}\rangle, 0\rangle$.
Repeat the following while $P \neq \emptyset$:
Pop $\langle \psi', M_{1:i}\rangle \leftarrow P$.
If $\psi' = \emptyset$ then Output M.
Choose variable v from ψ'.
Set $\psi'_1 \leftarrow \{c \setminus \{v\} : c \in \psi' \text{ and } \neg v \notin c\}$.
Set $\psi'_2 \leftarrow \{c \setminus \{\neg v\} : c \in \psi' \text{ and } v \notin c\}$.
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 $P \leftarrow \langle\langle \psi'_j, M_{1:i+1}\rangle, h(\psi'_j) + g_\psi(M_{1:i+1})\rangle$.
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 42: Classic branch-and-bound procedure adapted to Satisfiability.

Recall the problem of Variable Weighted Satisfiability which was defined in Section 1: given CNF formula ψ and positive weights on variables, find a satisfying assignment for ψ, 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:

$$(v_1 \vee v_3 \vee v_7) \wedge (v_2 \vee v_6) \wedge (v_4 \vee v_5 \vee v_8).$$

A strictly lower bound on the minimum weight solution over all extensions to $M_{1:}$ is $g_\psi(M_{1:}) + h(\psi_{M_{1:}})$, 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 [49], 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 [91]. The interested reader can consult [91] and [33] for additional ideas.

4.12 Algebraic Methods

A collection of Boolean constraints may be expressed as a system of algebraic equations or inequalities which has a solution if and only if the constraints are satisfiable. 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 non-optimal choices.

4.12.1 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 [23]. 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* [27]. 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 in CNF 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:

$$\sum_{i=0}^{2^n-1} c_i v_1^{b_{i,0}} v_2^{b_{i,1}} \ldots v_n^{b_{i,n-1}} = 0 \tag{7}$$

where all variables v_i can take values 0 or 1, and addition is taken modulo 2. An equation of the form (7) is said to be multi-linear. A product $t_i = v_1^{b_{i,0}} v_2^{b_{i,1}} \ldots v_n^{b_{i,n-1}}$, 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, $v_1v_2 + v_1v_2$ 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 (7) may be multiplied by a term and the resulting equation may be reduced to the form (7) by rule 2. above. Thus, $v_3v_4(v_1 + v_3 = 0)$ becomes $v_1v_3v_4 + v_3v_4 = 0$.
4. Two equations may be added to produce an equation that may be reduced by rule 1. above to the form (7). 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

$$v_1v_2v_3 + v_1v_2 + v_2v_3 + v_1 + 1 = 0 \text{ and}$$
$$v_1v_2v_3 + v_1v_2 + v_2v_3 + v_1 = 0$$

are complementary.

It is left 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 17. *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 (0^{th}) 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 ψ and create set B, initially empty. Repeat the following until ψ is empty. Pick an equation e of ψ 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 ψ that have never been in ψ. This sets up the addition of e with all other equations in B. When ψ 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

$$(v_1 \vee v_2 \vee v_3)$$

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

$$(\neg v_1 \vee v_2 \vee v_3)$$

is represented by

$$(1 + v_1)(1 + v_2)(1 + v_3) + v_2(1 + v_3) + v_3 + 1 = 0$$

which reduces to

$$v_1v_2v_3 + v_1v_2 + v_1v_3 + v_1 = 0. \tag{8}$$

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 4.9) 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 28 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:

$$(v_1 \vee \neg v_2) \wedge (v_2 \vee \neg v_3) \wedge (v_3 \vee \neg v_1) \tag{9}$$

The equations corresponding to (9) are expressed below as (1), (2), and (3). All equations following those equations are derived as stated on the right.

v_1v_2			$+v_2$		$= 0$	(1)
	v_2v_3			$+v_3$	$= 0$	(2)
v_1v_3		$+v_1$			$= 0$	(3)
$v_1v_2v_3$		$+v_2v_3$			$= 0$	$(4) \Leftarrow v_3 \cdot (1)$
$v_1v_2v_3$				$+v_3$	$= 0$	$(5) \Leftarrow (4) + (2)$

Algorithm 17.

An Algebraic Solver (ψ, d)
/* *Input: List of equations* $\psi = \langle e_1, ..., e_m \rangle$*, 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 $\exists e' \in B : first\_non\text{-}zero(e) = first\_non\text{-}zero(e')$:
Set $e \leftarrow reduce(e + e')$. /* 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 $reduce(ve)$ has not been in ψ:
Append $\psi \leftarrow reduce(ve)$. /* Rule 3. */
Output "satisfiable".
□

Figure 43: *Simple algebraic algorithm for SAT.*

$$
\begin{array}{llll}
v_1v_2v_3 \quad +v_1v_3 & & = 0 & (6) \Leftarrow v_1 \cdot (2) \\
v_1v_2v_3 \quad +v_1 & & = 0 & (7) \Leftarrow (6) + (3) \\
v_1v_2v_3 + v_1v_2 & & = 0 & (8) \Leftarrow v_2 \cdot (3) \\
v_1v_2v_3 \quad +v_2 & & = 0 & (9) \Leftarrow (8) + (1) \\
v_1 \quad +v_2 & & = 0 & (10) \Leftarrow (9) + (7) \\
v_1 \quad +v_3 & & = 0 & (11) \Leftarrow (5) + (7)
\end{array}
$$

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 (9)

$$(\neg v_1 \vee \neg v_2) \wedge (v_3 \vee v_1)$$

the equation $v_1 + v_2 + 1 = 0$ 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 17 and the degree of an equation is $degree(e) = \max\{deg(t) : t \text{ is a non-zero term in } e\}$. An example is Algorithm 17 of Figure 43 which is adapted from [27]. In the algorithm terms are re-indexed as in Theorem 17. Then *first_non-zero*(e_i) is used to mean the highest index of a non-zero term of e_i. The function $reduce(e)$ 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 4.9. Equations corresponding to f and c in Figure 32 are

$$\begin{aligned} f: &\quad v_1v_3 + v_2 + v_1v_2 = 0 \\ c: &\quad v_2v_3 + v_3 = 0 \end{aligned}$$

Multiply f by v_2v_3 to get $v_2v_3 = 0$ which adds with c to get $v_3 = 0$, the inference that is missed by **restrict**(f, c) in Figure 32. The inference can be derived from BDDs by reversing the role of f and c as shown in Figure 33. Consider what multiplying f by v_2v_3 and adding to c means in the BDD world. The BDD representing v_2v_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, $d \wedge c$ can safely be added as a BDD as long as f remains. But it is easy to check that $d \wedge c$ 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 4.9.4, 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 $g = \mathbf{gcf}(f, c)$ turns out to be $(v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_2 \vee \neg v_3)$ which is different from d in the leading clause but is sufficient to derive the inference when conjoined with c. By the definition of **gcf**, since $f \wedge c = g \wedge c$, 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 28 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 39 have equation representations

$$\begin{aligned} b2: &\quad v_1v_3 + v_2 + v_1v_2 = 0 \\ b1: &\quad v_3 + v_2v_3 = 0. \end{aligned}$$

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

4.12.2 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:

$$\begin{array}{rl} \text{maximize or minimize} & \boldsymbol{c}\,\boldsymbol{\alpha} \\ \text{subject to} & \mathcal{M}\boldsymbol{\alpha} \;\leq\; \boldsymbol{b} \\ & l \leq \boldsymbol{\alpha} \;\leq u \\ & \alpha_i \text{ integral}, 1 \leq i \leq n \end{array} \tag{10}$$

where $\mathcal{M}$ is a constraint matrix, $\boldsymbol{c}$ is a linear objection function, $\boldsymbol{b}$ is a constant vector, and $\boldsymbol{\alpha}$ 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 (10). They are divided into the categories of preprocessing and solving. However, an important third aspect concerns the matrix $\mathcal{M}$ 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. 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:

$$\sum_{i=1}^{n} v_{i,j} = 1, \qquad 1 \leq j \leq n+1, \tag{11}$$

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

$$v_{i,j} + v_{i,k} \leq 1, \qquad 1 \leq i \leq n, 1 \leq j < k \leq n+1. \tag{12}$$

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. If running the algorithm to be shown later, practically complete enumeration is necessary before finally it is determined that no solution exists [66]. However, the requirement that at most one pigeon is in a hole may alternatively be represented by

$$\sum_{j=1}^{n+1} v_{i,j} \leq 1, \qquad 1 \leq i \leq n. \tag{13}$$

which may be used instead of (12). The new constraints are much tighter than (12) and the system (11) and (13) is easily solved [66].

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.

5 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 much space in this section is devoted to a few special cases. Hopefully this will help to fully appreciate the possibilities. In particular, results on minimally unsatisfiable and nested formulas are greatly detailed.

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 [103]. 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 several others including q-Horn, extended Horn, CC-balanced and SLUR that cannot be so defined will be described. The reason is that Schaefer's scheme is limited to classes that can be recognized in log space.

Below, some of the more notable easy classes and present algorithms for their solution are identified. A crucial component of many of these algorithms is *unit resolution*. An implementation is given in Figure 44.

5.1 2-SAT

All clauses of a 2-SAT formula contain at most two literals. A given 2-SAT formula ψ may be solved efficiently by constructing the implication graph $\vec{G}_\psi$ of ψ (see Section 2.3) and traversing its vertices, ending either at a cycle containing complementary literals or with no additional vertices to explore. The algorithm of Figure 45 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

Algorithm 18.

Unit Resolution (ψ)
/* *Input: set of sets CNF formula* ψ */
/* *Output: pair* $\langle$*CNF formula* ϕ*, partial assignment* $P\rangle$ */
/* ψ *is satisfiable if and only if* ϕ *is satisfiable* */
/* ϕ *has no unit clauses* */
/* *Locals: set of variables* P*, set of sets CNF formula* ϕ */
Set $\phi \leftarrow \psi$; Set $P \leftarrow \emptyset$.
Repeat the following while $\emptyset \notin \phi$ and there is a unit clause in ϕ:
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 44: *Unit resolution for CNF formulas.*

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, ϕ 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 variables during traversal of a component, and the variables ϕ' and l' save the point to return to if a contradiction is found. This algorithm is adapted from [44].

The next two theorems are stated without proof.

Theorem 18. *On input CNF formula* ψ*, Algorithm* **2-SAT Solver** *outputs "unsatisfiable" if and only if* ψ *is unsatisfiable and if it outputs a set* M*, then* M *is a model for* ψ*.* □

Theorem 19. *On input CNF formula* ψ *containing* m *clauses and* n *variables, Algorithm* **2-SAT Solver** *has* $O(m+n)$ *worst case complexity.* □

5.2 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 $(\neg v_1 \vee \neg v_2 \vee \ldots \vee \neg v_i \vee v)$ is equivalent to the rule $v_1 \wedge v_2 \wedge \ldots \wedge v_i \rightarrow v$ or the implication $v_1 \rightarrow$

Algorithm 19.

2-SAT Solver (ψ)
/* *Input: set of sets 2-CNF formula* ψ */
/* *Output: "unsatisfiable" or a model for* ψ */
/* *Locals: variable* s*, set of variables* M, M'*, set of sets formula* ϕ */
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 $\langle \phi, M' \rangle \leftarrow$ **Unit Resolution** (ϕ).
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 ϕ.
Set $M \leftarrow M \cup M'$.
Set $M' \leftarrow \emptyset$.
Set $\phi' \leftarrow \phi$.
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 45: *Algorithm for determining satisfiability of 2-CNF formulas.*

Algorithm 20.

Horn Solver (ψ)
/* *Input: set of sets Horn formula* ψ */
/* *Output: "unsatisfiable" or a model for* ψ */
/* *Locals: set of variables* M */
Set $M \leftarrow \emptyset$.
Repeat the following until no positive literal unit clauses are in ψ:
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.
□

Figure 46: *Algorithm for determining satisfiability of Horn formulas.*

$v_2 \rightarrow \ldots \rightarrow v_i \rightarrow v$. 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 20. *Every Horn formula has a unique minimum model.*

Proof. Let ψ be a Horn formula. Let M be a minimal model for ψ, that is, a smallest subset of variables of value 1 that satisfies ψ. Choose any $v \in M$. Since $M \setminus \{v\}$ is not a model for ψ, 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 ψ. It follows that all models other than M must have cardinality greater than $|M|$. Hence M is a unique minimum model for ψ. □

The satisfiability of Horn formulas can be determined in linear time using unit resolution [41, 63, 105]. One of several possible variants is shown in Figure 46.

Theorem 21. *Given Horn formula* ψ *as input, Algorithm* **Horn Solver** *outputs "unsatisfiable" if and only if* ψ *is unsatisfiable and if it outputs a set of variables* M*, then* M *is a unique minimum model for* ψ*.*

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 ψ.

Suppose M is not the unique minimum model for ψ. Then, for some variable v, the assignment $M \setminus \{v\}$ satisfies ψ. 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 ψ. 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 ψ' 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, ψ' 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 ψ. 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.

5.3 Renamable Horn Formulas

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

Renamable Horn formulas can be recognized and solved in $O(|\psi|)$ time [7, 86]. Algorithm **SLUR** on Page 102 solves renamable Horn formulas in linear time.

5.4 Linear Programming Relaxations

Let $\mathcal{M}_\psi$ be a $(0, \pm 1)$ matrix representing a CNF formula ψ. If $\mathcal{M}_\psi$ has a particular structure it is possible to solve the inequalities 1 with non-integer constraints $0 \leq \alpha_i \leq 1$ to obtain a solution to ψ either directly or by rounding. Notable classes based on particular matrix structures are the extended Horn formulas and what in this chapter are called the CC-balanced formulas.

The class of *extended Horn formulas* was introduced by Chandru and Hooker [25] 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 [24] 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 ψ 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 ψ

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* [114]. Extended Horn formulas can be solved in polynomial time by Algorithm **SLUR** on Page 102 [104].

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

Chandru and Hooker proposed an algorithm that finds a model for a satisfiable extended Horn 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 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 **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 [28] 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 ψ with $(0, \pm 1)$ matrix representation $\mathcal{M}_\psi$ is *CC-balanced* if in every submatrix of $\mathcal{M}_\psi$ 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 [119]). Recognizing that a formula is CC-balanced takes linear time. However, the recognition problem is moot because Algorithm **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 *Single*

Lookahead Unit Resolution [104] (SLUR). The SLUR class is peculiar in that it is defined based on an algorithm rather than on properties of formulas. Algorithm **SLUR** of Figure 47), 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 **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 **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 **SLUR** since once branch of execution is always cut. This causes quadratic worst case complexity.

A simple modification to Algorithm **SLUR** brings the complexity down 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.

Theorem 22. *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.*

Proof. For reasons mentioned above, only the number of steps used by unit resolution is considered. 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 $O(|\psi|)$. □

All Horn, renamable 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

Algorithm 21.

SLUR (ψ)
/* *Input: a set of sets CNF formula* ψ */
/* *Output: "unsatisfiable" or a model for* ψ */
/* *Locals: set of variables* M*, flag* d*, formula* ψ' */
Set $\langle \psi', M \rangle \leftarrow$ **Unit Resolution** (ψ).
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 $\langle \psi_1, M_1 \rangle \leftarrow$ **Unit Resolution** ($\{c - \{v\} : c \in \psi', \neg v \notin c\}$).
Set $\langle \psi_2, M_2 \rangle \leftarrow$ **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

□

Figure 47: *Algorithm for determining satisfiability of SLUR formulas.*

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**.

5.5 q-Horn Formulas

This class of propositional formulas was developed in [16] and [17]. The class of q-Horn formulas may be characterized as a special case of maximum monotone decomposition of matrices [118, 119]. Express a *CNF* formula of m clauses and n variables as an $m \times n$ $(0, \pm 1)$-matrix $\mathcal{M}$. In the monotone decomposition of $\mathcal{M}$, columns are scaled by -1 and the rows and columns are partitioned into submatrices as follows:

$$\left(\begin{array}{c|c} \mathcal{A}^1 & \mathcal{E} \\ \hline \mathcal{D} & \mathcal{A}^2 \end{array}\right)$$

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 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 [119] 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 4.10.1). Once a q-Horn formula is in its decomposed form it can be solved in linear time by Algorithm **q-Horn Solver** of Figure 48.

Theorem 23. *Given q-Horn formula ψ, Algorithm* **q-Horn Solver** *outputs "unsatisfiable" if and only if ψ is unsatisfiable and if ψ is satisfiable, then the output set $M_1 \cup M_2$ is a model for ψ.*

Algorithm 22.

q-Horn Solver ($\mathcal{M}_\psi$)
/* *Input: a* $(0, \pm 1)$ *matrix representation for q-Horn formula* ψ */
/* *Output: "unsatisfiable" or a model for* ψ */
/* *Locals: set of variables* M_1, M_2 */
Find the maximum monotone decomposition of $\mathcal{M}_\psi$ (Page 84).
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 48: *Algorithm for determining satisfiability of q-Horn formulas.*

Proof. Clearly, if Horn formula $\mathcal{A}^1$ is unsatisfiable then so is ψ. 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, ψ is unsatisfiable in this case. The set $M_1 \cup M_2$ is a model for ψ 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 24. *A CNF formula is q-Horn if and only if its satisfiability index is no greater than 1.*

Proof. Let ψ be a q-Horn formula with variable set V_ψ and suppose $|V_\psi| = n$. Let $\mathcal{M}_\psi$ be a monotone decomposition for ψ. 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 from $\mathcal{M}_\psi$. Assign value 1 to all α_i, $0 \leq i < n_a$, and value 1/2 to all α_i, $n_a \leq i < n$. This satisfies $0 \leq \alpha_i \leq 1$ of system 2. Since rows of $\mathcal{A}^1$ 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 is 1. Since all rows of $\mathcal{D}$ and $\mathcal{A}^2$ 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 has maximum value 1 too. Thus, inequality 2 is satisfied with $z = 1$.

Now, suppose ψ has satisfiability index I_ψ which is no greater than 1. Choose values for all α_i terms such that inequality 2 is satisfied for $z = I_\psi$. Let π 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 π. 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, 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 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 ρ 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 ρ 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 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 ψ. □

The following result from [17] is also interesting in light of Theorem 24. It is stated without proof.

Theorem 25. *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 $1 + a\ln(n)/n$, where a is any positive constant. The $\mathcal{M}$ matrix for any formula in this class can be scaled by -1 and partitioned as follows:

$$\left(\begin{array}{c|c|c} \mathcal{A}^1 & \mathcal{E} & \mathcal{B}^1 \\ \hline \mathcal{D} & \mathcal{A}^2 & \mathcal{B}^2 \end{array}\right)$$

where submatrices $\mathcal{A}^1$, $\mathcal{A}^2$, $\mathcal{E}$, and $\mathcal{D}$ have the properties required for q-Horn formulas and the number of columns in $\mathcal{B}^1$ and $\mathcal{B}^2$ 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 $\mathcal{B}^1$ and $\mathcal{B}^2$.

5.6 Matched Formulas

The matched formulas have been considered in the literature (see [117]) 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 2.5) for CNF formula ψ, 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 $E' \subset E$ 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 ψ 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 Section 5.12.

5.7 Generalized Matched Formulas

The class of matched formulas has been generalized by Szeider [116]. Let $G_\psi(V_1, V_2, E)$ be the variable-clause matching graph of ψ as above. Let $V_1' \subset V_1$ and $V_2' \subset V_2$ and suppose the subgraph $G'_\psi(V_1', V_2', E')$ induced by V_1' and V_2' is complete: that is, for every pair of vertices $v_1 \in V_1'$ and $v_2 \in V_2'$ there is an edge $e \in E'$. Call such a subgraph a *biclique*.

Theorem 26. *Let $G_\psi(V_1, V_2, E)$ be the variable-clause graph for ψ. Let $\{X_1, X_2, \ldots, X_r\}$ be a collection of bicliques in $G_\psi(V_1, V_2, E)$ and suppose (1) the number of clause vertices in X_1 is less than 2 raised to the number of variable vertices in X_i, $1 \leq i \leq r$, (2) every $v_1 \in V_1$ (representing a clause) is in some X_i, $1 \leq i \leq r$, and (3) every $v_2 \in V_2$ (representing a variable) is in at most one X_i, $1 \leq i \leq r$. Then ψ is satisfiable.*

Proof. Let V^{X_i} be the variables represented by vertices of X_i and let C^{X_i} be the clauses represented by the remaining vertices of X_i. Remove from all $c \in C^{X_i}$ all literals not associated with variables of V^{X_i}. Since X_i is complete, the number of literals in c is $|V^{X_i}|$ and the number of assignments to the variables of V^{X_i} which are falsified by c is 1. By condition (1), the total number of falsifying assignments for C^{X_i} is less than all possible assignments to V^{X_i}, hence some assignment satisfies C^{X_i}. By condition (3), $V^{X_i} \cap V^{X_j} = \emptyset$, $i \neq j$, so satisfying assignments for both C^{X_i} and C^{X_j} cannot conflict. By condition (2), every clause in ψ is in some biclique and is therefore satisfied by some assignment. □

Clause width is typically fixed. In that case, a model for a formula satisfying the conditions of Theorem 26 can be found in time linear in the number of clauses of ψ. If the bicliques of G_ψ are all single edges, then ψ is a matched formula.

Unfortunately, the problem of recognizing a generalized matched formula is $\mathcal{NP}$-complete, even for 3-CNF formulas.

5.8 Nested and Extended Nested Satisfiability

The complexity of nested satisfiability, inspired by Lichtenstein's theorem of planar satisfiability [87], has been studied in [73]. 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

$$(v_6 \vee \neg v_7 \vee v_8) \wedge (v_2 \vee v_4) \wedge (\neg v_6 \vee \neg v_9) \wedge (v_1 \vee \neg v_5 \vee v_{10}).$$

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 5.13. Despite this, the class of nested formulas is not contained in either of the SLUR or q-Horn classes as shown in Section 5.12. 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 [73] 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 $(v_0 \vee v_{n+1})$ 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 49. The following lemma is needed to prove correctness. The actual dependence of h values on i is not shown to prevent the notation from going out of control. However, from the context, this dependence should be clear.

Lemma 27. *Assume, at the start of any iteration* $0 \leq i \leq m$ *of the outer*

Algorithm 23.

Nested Solver (ψ)
/* *Input: set of sets CNF formula* ψ*, 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* $\mathcal{D}$ *of Boolean variables.* */
Apply unit resolution to eliminate all unit clauses in ψ.
Topologically sort the clauses of ψ 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 $\mathcal{D} \leftarrow \{D_{j,j+1}(s,t) \leftarrow 1 : \quad 0 \le j \le n, \quad s,t \in \{0,1\}\}$.
Repeat the following for $0 \le i \le m$:
Let $0 < h_1 < h_2 < \ldots < h_k < n+1$ be the intervals for $D \in \mathcal{D}$.
// That is, $\mathcal{D} = \{D_{0,h_1}(*,*), D_{h_1,h_2}(*.*), \ldots, D_{h_k,n+1}(*,*)\}$.
Let $c_i = \{l_{h_p}, \ldots, l_{h_q}\}$, where $l_{h_r} \in \{v_{h_r}, \neg v_{h_r}\}$, $p \le r \le q$.
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 $l_{h_p} = v_{h_p}$ 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 $l_{h_p} = \neg v_{h_p}$ 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 $0 \le j < q-p$, $s \in \{0,1\}$, $t \in \{0,1\}$:
If $t = 1$ then Set $l \leftarrow v_{h_{p+j+1}}$, Otherwise Set $l \leftarrow \neg v_{h_{p+j+1}}$.
Set $E_{h_p,h_{p+j+1}}(s,t) \leftarrow$
$(E_{h_p,h_{p+j}}(s,1) \wedge D_{h_{p+j},h_{p+j+1}}(1,t)) \vee$
$(E_{h_p,h_{p+j}}(s,0) \wedge D_{h_{p+j},h_{p+j+1}}(0,t)) \vee$
$(G_{h_p,h_{p+j}}(s,1) \wedge D_{h_{p+j},h_{p+j+1}}(1,t) \wedge l \in c_i) \vee$
$(G_{h_p,h_{p+j}}(s,0) \wedge D_{h_{p+j},h_{p+j+1}}(0,t) \wedge l \in c_i)$.
Set $G_{h_p,h_{p+j+1}}(s,t) \leftarrow$
$(G_{h_p,h_{p+j}}(s,1) \wedge D_{h_{p+j},h_{p+j+1}}(1,t) \wedge \neg(l \in c_i)) \vee$
$(G_{h_p,h_{p+j}}(s,0) \wedge D_{h_{p+j},h_{p+j+1}}(0,t) \wedge \neg(l \in c_i))$.
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 $\mathcal{D} \leftarrow \mathcal{D} \setminus \{D_{h_p,h_{p+1}}(*,*) \ldots D_{h_{q-1},h_q}(*,*)\} \cup \{D_{h_p,h_q}(*,*)\}$.
If $D_{0,n+1}(1,1) = 1$, Output "satisfiable,"
Otherwise Output "unsatisfiable."
□

Figure 49: *Algorithm for determining satisfiability of nested formulas.*

Repeat loop of Algorithm **Nested Solver**, *that*

$$\mathcal{D} = \{D_{h_0,h_1}(*,*), D_{h_1,h_2}(*.*), \ldots, D_{h_k,h_{k+1}}(*,*)\},$$

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_{h+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* $\psi^i_{j,j+1}$ *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* $\psi^i_{p,p+j} \cup \{\{l_x : l_x \in c_i, x \leq h_{p+j}\}\}$ *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* $\psi^i_{p,p+j} \cup \{\{l_x : l_x \in c_i, x \leq h_{p+j}\}\}$ *is not satisfiable but* $\psi^i_{p,p+j}$ *is satisfiable with variable* h_p *set to value* s *and variable* $v_{h_{p+j}}$ *set to value* t.

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 $D_{h_p,h_{p+1}} \ldots D_{h_{q-1},q}$ 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 $D_{j,j+1}(*,*)$, $0 \leq j \leq n$, and these have value 1. From the definition of $\psi^i_{p,j}$, $\psi^0_{j,j+1} = \emptyset$. 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 $D_{h_r,h_{r+1}}(*,*)$ 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 $E_{h_p,h_q}(s,t)$ has value 1 if and only if $\psi^i_{p,q} \cup \{c_i\}$ 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, $\psi^0_{*,*} = \emptyset$. Moreover, by Point 1, $\{l_x : l_x \in c_0, x \leq h_p\} = \{l_{h_p}\}$ so $\psi^0_{p,p} \cup \{\{l_{h_p}\}\} = \{\{l_{h_p}\}\}$. 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 if $l_{h_p} = v_{h_p}$, then $\psi^0_{p,p} \cup \{\{v_{h_p}\}\}$ 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 $\psi^0_{p,p} \cup \{\{\neg v_{h_p}\}\}$ 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 $c_i^{j-1} = \{l_x : l_x \in c_i, x \leq p + j - 1\}$ 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

$$\psi^i_{p,p+j} \cup \{c_i^{j-1}\} = (\psi^i_{p,p+j-1} \cup \{c_i^{j-1}\}) \cup \psi^i_{p+j-1,p+j},$$

and $(\psi^i_{p,p+j-1} \cup \{c_i^{j-1}\}) \cap \psi^i_{p+j-1,p+j} = \emptyset$. For the sake of visualization, define $\psi_1 = \psi^i_{p,p+j-1}$ and $\psi_2 = \psi^i_{p+j-1,p+j}$. At most one variable, namely $v_{h_{p+j-1}}$, is common to both ψ_1 and ψ_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 ψ_2 are satisfiable if and only if $\psi^i_{p,p+j} \cup \{c_i^{j-1}\}$ is satisfiable. Now consider $\psi^i_{p,p+j} \cup \{c_i^j\}$. Since every variable of c_i has an index matching one of $\{h_p, h_{p+1}, \ldots, h_q\}$, $c_i^j \setminus c_i^{j-1}$ either is the empty set or $\{v_{h_{p+j}}\}$ or $\{\neg v_{h_{p+j}}\}$. Therefore, given $v_{h_{p+j-1}}$, $\psi^i_{p,p+j-1} \cup \{c_i^j\}$ is satisfiable if and only if either both $\psi_1 \cup \{c_i^{j-1}\}$ and ψ_2 are satisfiable or ψ_1 and ψ_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 ψ_1, $\psi_1 \cup \{c_i^{j-1}\}$ can be unsatisfiable with ψ_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 $\psi^i_{p,p+j-1}$ and $\psi^i_{p+j-1,p+j}$, and the assignments to E and G in the main inner loop of Algorithm 23, the values of E and G match Points 3 and 4 for the jth 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 $\psi^i_{h_p,h_q} \cup \{c_i\}$ is satisfiable. This matches the hypothesis of Point 2 thereby completing the proof that Point 2 holds. □

Corollary 28. *Algorithm* **Nested Solver** *correctly determines satisfiability for a given nested formula.*

Proof. Algorithm **Nested Solver** determines ψ is satisfiable if and only if $D_{0,n+1}(1,1)$ has value 1. But, from Lemma 27, $D_{0,n+1}(1,1)$ 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 [73]. Suppose the algorithm is applied to a nested formula containing the following clauses which are shown in proper order after the topological sort:

$$(v_1 \vee v_2) \wedge (v_2 \vee v_3) \wedge (\neg v_2 \vee \neg v_3) \wedge (v_3 \vee v_4) \wedge (v_3 \vee \neg v_4) \wedge (\neg v_3 \vee v_4) \wedge (\neg v_1 \vee v_2 \vee v_4)$$

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.

$D_{*,*}(s,t)$	0, 0	0, 1	1, 0	1, 1
$1,2$	0	1	1	1
$2,3$	0	1	1	0
$3,4$	0	0	0	1

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.

$E_{p,p+j}(s,t)$	0, 0	0, 1	1, 0	1, 1
$1,1$	1	0	0	0
$1,2$	0	1	0	1
$1,3$	1	0	1	0
$1,4$	0	0	0	1

$G_{p,p+j}(s,t)$	0, 0	0, 1	1, 0	1, 1
$1,1$	0	0	0	1
$1,2$	0	0	1	0
$1,3$	0	0	0	1

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.

$D_{*,*}(s,t)$	0, 0	0, 1	1, 0	1, 1
$1,4$	0	0	0	1

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 29. *Algorithm* **Nested Solver** *has worst-case time complexity that is linear in the size of* ψ. □

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 [57]. The details are skipped. This extension can be recognized and solved in linear time. For details, the reader is referred to [57].

5.9 Linear Autark Formulas

This class is based on the notion of an autark assignment which was introduced in [94]. 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.

The following shows how to find an autark assignment in polynomial time. Let CNF formula ψ of m clauses and n variables be represented as a $(0, \pm 1)$ matrix $\mathcal{M}_\psi$. Let $\boldsymbol{\alpha}$ be an n dimensional real vector with components $\alpha_1, \alpha_2, \ldots, \alpha_n$. Consider the following system of inequalities:

$$\begin{aligned} \mathcal{M}_\psi \boldsymbol{\alpha} &\geq 0, \\ \boldsymbol{\alpha} &\neq 0. \end{aligned} \tag{14}$$

Theorem 30. ([90]) *A solution to* (14) implies an autark assignment for ψ.

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. It is shown that either a clause is satisfied by this assignment or it contains only unassigned variables.

For every clause, by (1), it is necessary to satisfy the following:

$$a_1 v_1 + a_2 v_2 + \ldots + a_n v_n \geq 1 - b \tag{15}$$

where a_i factors are 0,-1, or +1, and the number of negative a_i terms is b. Suppose $\boldsymbol{\alpha}$ is a solution to (14). Write an inequality of (14) as follows:

$$a_1 \alpha_1 + a_2 \alpha_2 + \ldots + a_n \alpha_n \geq 0 \tag{16}$$

Suppose at least one term, say $a_i \alpha_i$, is positive. If α_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 (15) must be at least $1 - b$. On the other hand, if α_i is negative, then $a_i = -1$, $v_i = 0$ and the product $a_i v_i$ in (15) is 0. Since there are $b - 1$ negative factors left in (15), the left side must be at least $1 - b$. Therefore, in either case, inequalities of the form (16) with at least one positive term represent clauses that are satisfied by the assignment corresponding to $\boldsymbol{\alpha}$.

Now consider the case where no terms in (16) are positive. Since the left side of (16) is at least 0, there can be no negative terms either. Therefore,

all the variables for the represented clause are unassigned. □

A formula ψ for which the only solution to (14) is $\boldsymbol{\alpha}=0$ is said to be *linear autarky-free*.

System (14) 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 50 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 (14). In a departure from [90] 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 31. *Let ψ be a CNF formula that is input to Algorithm* **LinAut** *and let ψ' be the formula that is output. If $\psi' = \emptyset$ then ψ is satisfiable and $\boldsymbol{\alpha}^S$ transforms to a satisfying assignment for ψ.*

Proof. An autark assignment t to ψ' induces a partition of clauses of ψ' 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 ψ'. If there is no satisfying assignment for either group then ψ' cannot be satisfiable. Therefore, since each iteration finds and applies an autark assignment (via $\boldsymbol{\alpha}$), if $\psi' = \emptyset$ then the composition of the autark assignments of each iteration is a satisfying assignment for ψ. □

The algorithm has polynomial time complexity. In some cases it solves its input formula.

Theorem 32. *Let ψ be Horn or renamable Horn. If ψ is satisfiable then* **LinAut** *returns an $\boldsymbol{\alpha}^S$ that transforms to a solution of ψ as described in the proof of Theorem 30 and the formula returned by* **LinAut** *is $\emptyset$. If ψ is unsatisfiable* **LinAut** *returns "unsatisfiable."*

Proof. Horn or renamable Horn formula ψ is unsatisfiable only if there is at least one positive unit clause in ψ. In this case **Unit Resolution**(ψ) will output a formula containing $\emptyset$ and **LinAut** will output "unsatisfiable." Otherwise **Unit Resolution**(ψ) outputs a Horn or renamable Horn formula ψ' with no unit clauses and a partial assignment P which is recorded in $\boldsymbol{\alpha}^S$. In the next paragraph we will show that any such Horn or renamable 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 renamable Horn. Therefore, the Repeat loop of **LinAut** must continue until there are no clauses left. Then, by Lemma 31, $\boldsymbol{\alpha}^S$ transforms to a satisfying assignment for ψ.

Now it is shown that there is always an $\boldsymbol{\alpha} \neq 0$ that solves (14). Observe that any Horn formula without unit clauses has at least one negative literal

Algorithm 24.

LinAut (ψ)
/* *Input: a set of sets CNF formula* ψ */
/* *Output: pair* $\langle$*real vector, subformula of* $\psi\rangle$ *or "unsatisfiable"* */
Set $\langle \psi', P\rangle \leftarrow$ **Unit Resolution**(ψ).
If $\emptyset \in \psi'$ then Output "unsatisfiable".
Set $\boldsymbol{\alpha}^S \leftarrow 0$.
For each $v_i \in P$ Set $\alpha_i^S \leftarrow 1$.
For each v_i such that $v_i \notin P$ and $v_i \in V_P$ Set $\alpha_i^S \leftarrow -1$.
If $\psi' = \emptyset$ then Output $\langle \boldsymbol{\alpha}^S, \emptyset\rangle$.
Repeat the following until some statement generates output:
Solve for $\boldsymbol{\alpha}$: $\mathcal{M}_{\psi'}\boldsymbol{\alpha} \geq 0$ and $\alpha_i^S = 0$ if column i in $\mathcal{M}_{\psi'}$ is 0.
If $\boldsymbol{\alpha}$=0 then Output $\langle \boldsymbol{\alpha}^S, \psi'\rangle$.
Otherwise, do the following:
Set $\boldsymbol{\alpha}^S \leftarrow \boldsymbol{\alpha}^S + \boldsymbol{\alpha}$.
For every $\alpha_i^S \neq 0$ do the following:
Zero out rows of $\mathcal{M}_{\psi'}$ with a non-zero entry in column i.
Set $\psi' \leftarrow \{c : c \in \psi', v_i \notin c, \neg v_i \notin c\}$.

□

Figure 50: *Repeated decomposition of a formula using autark assignments. V_P is the set of variables whose values are set in partial assignment P. Output real vector $\boldsymbol{\alpha}^S$ can be transformed to a truth assignment as described in the proof of Theorem 30.*

in every clause. Therefore some all negative vector $\boldsymbol{\alpha}$ of equal components solves (14). In the case of renamable Horn, the polarity of $\boldsymbol{\alpha}$ 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 renamable Horn formula. Note that Algorithm **LinAut** may find an $\boldsymbol{\alpha}$ that does not zero out all rows so the Repeat loop may have more than 1 iteration but, since the reduced formula ψ is Horn or renamable Horn, there is always an $\boldsymbol{\alpha} \neq 0$ that solves (14) for the clauses that are left. □

The following Horn formula would violate Theorem 32 if the call to **Unit Resolution** had not been added to **LinAut**:

$$(v_1) \wedge (v_2) \wedge (v_3) \wedge (v_1 \vee \neg v_2 \vee \neg v_3) \wedge (\neg v_1 \vee v_2 \vee \neg v_3) \wedge (\neg v_1 \vee \neg v_2 \vee v_3).$$

This formula is satisfied with variables set to 1 but is linear autarky-free.

Theorem 33. *If ψ is a 2-SAT formula that is satisfiable, then* **LinAut** *outputs an $\boldsymbol{\alpha}^S$ that represents a solution of ψ and the formula output by* **LinAut** *is $\emptyset$.*

Proof. In this case **Unit Resolution**(ψ) outputs ψ and causes no change to $\boldsymbol{\alpha}^S$. The only non-zero autark solutions to $\mathcal{M}_\psi \boldsymbol{\alpha} \geq 0$ 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 ψ 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 32, $\boldsymbol{\alpha}^S$ transforms to a satisfying assignment for ψ. □

If ψ is an unsatisfiable 2-SAT formula then **LinAut**(ψ) 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 34. *All q-Horn formulas are linear autark formulas*

Proof. Apply **LinAut** to q-Horn formula ψ. Then a linear autarky-free q-Horn formula is output. Assume that all clauses in the output formula have at least 2 literals - this can be done because all clauses of ψ' 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 ψ'. The rows and columns of $\mathcal{M}_{\psi'}$ representing the output q-Horn formula ψ' may be permuted and columns may be scaled by -1 to get a form as shown on Page 79 where $\mathcal{A}^1$ represents a Horn formula, $\mathcal{D}$ is non-positive, and $\mathcal{A}^2$ represents a 2-SAT formula. Construct vector $\boldsymbol{\alpha}$ as follows. Assign equal negative values to all α_i where i is the index of a column through $\mathcal{A}^1$ (reverse the value if the column had been scaled by -1) and 0 to all other α_i. Then,

since there are at least as many positive as negative entries in every row through $\mathcal{A}^1$, the product of any of those rows, unscaled, and $\boldsymbol{\alpha}$ is greater than 0. Since there are only negative entries in rows through $\mathcal{D}$ and all the entries through columns of $\mathcal{A}^2$ multiply by α_i values that are 0, the product of any of the rows through $\mathcal{D}$, unscaled, and $\boldsymbol{\alpha}$ are also positive. Therefore, $\boldsymbol{\alpha}$ shows that ψ' is not linear autark-free, a contradiction. It follows that $\mathcal{A}^1 = \mathcal{D} = \emptyset$ and the output formula is either 2-SAT or $\emptyset$. If it's 2-SAT, it must be unsatisfiable as argued in Theorem 33. □

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 35. *If the shortest clause of a CNF formula ψ has width k and if the satisfiability index of ψ is less than $k/2$, then ψ is satisfiable.*

Proof. This is seen more clearly by transforming variables as follows. Define real n dimensional vector $\boldsymbol{\beta}$ with components

$$\beta_i = 2\alpha_i - 1 \quad \text{for all} \quad 0 \leq i < n.$$

The satisfiability index (2) for the transformed variables may be expressed, by simple substitution, as follows:

$$\mathcal{M}_\psi \boldsymbol{\beta} \ \leq 2\boldsymbol{Z} \ - \ \boldsymbol{1} \tag{17}$$

where $\boldsymbol{1}$ is an n dimensional vector expressing the number of literals in all clauses and $\boldsymbol{Z} = \langle z, z, \ldots, z \rangle$. The minimum z that satisfies (17) is the satisfiability index of ψ. Apply Algorithm **LinAut** to ψ 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 ψ, it also holds for the ψ' that is output by **LinAut**. Suppose $\psi' \neq \emptyset$. Formula ψ' is linear autarky-free so the only solution to $0 \leq \mathcal{M}_{\psi'} \boldsymbol{\beta}$ is $\boldsymbol{\beta} = 0$ which implies $0 \leq 2\boldsymbol{Z} \ - \ \boldsymbol{1}$. Since the shortest clause of ψ' 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 31, the input formula is satisfiable. □

The effectiveness of this test on random k-CNF formulas will be discussed in Section 5.13.

5.10 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*. This section begins with a discussion of the special case where deficiency is 1, then considers 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.

The following result was proved in one form or another by several people (for example, [2, 81]). A simple argument due to Truemper is presented.

Theorem 36. *If ψ is a minimally unsatisfiable CNF formula with $|V_\psi| = n$ variables, then the number of clauses in ψ must be at least $n + 1$.*

Proof. Let ψ 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 ψ (see Section 2.5) 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 ψ is satisfiable, a contradiction. Therefore, by Hall's Theorem [55], there is a subset $V'_\psi \subset V_1$ with neighborhood smaller than $|V'_\psi|$. Let V'_ψ be such a subset of maximum cardinality. Define ψ_1 to be the CNF formula consisting of the clauses of ψ corresponding to the neighborhood of V'_ψ. By the minimality property of ψ, ψ_1 must be satisfiable. Delete from ψ the clauses of ψ_1 and from the remaining clauses all variables occurring in ψ_1. Call the resulting CNF formula ψ_2. There must be a matching of the clauses of ψ_2 into the variables of ψ_2 since otherwise V'_ψ and therefore ψ_1 was not maximal in size. Hence ψ_2 is satisfiable. But if ψ_1 and ψ_2 are satisfiable then so is ψ, a contraction. □

Most of the remaining ideas of this section have been inspired by Oliver Kullmann [78].

A *saturated minimally unsatisfiable* formula ψ is a minimally unsatisfiable formula such that adding any literal l, existing in a clause of ψ, to any clause of ψ 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 37. *Let ψ be a saturated minimally unsatisfiable formula and let l be a literal from ψ. 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 ψ, what's left is minimally unsatisfiable.

Proof. Formula ψ' is unsatisfiable if ψ is, otherwise there is an assignment which sets l to 1 and satisfies ψ. 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 ψ'' by adding l' to all clauses of ψ containing $\neg l$. Since ψ is saturated, there is a truth assignment M satisfying ψ'' and therefore ψ' 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 ψ', a contradiction. □

The following is a simple observation that is needed to prove Theorem 39 below.

Lemma 38. *Every variable of a minimally unsatisfiable formula occurs positively and negatively in the formula.*

Proof. Let ψ be any minimally unsatisfiable formula. Suppose there is a positive literal l such that $\exists c \in \psi : l \in c$ and $\forall c \in \psi, \neg l \notin c$. Let $\psi_1 = \{c :$

$c \in \psi, l \notin c\}$. Let $\psi_2 = \psi \setminus \psi_1$ denote the clauses of ψ which contain literal l. Clearly, $|\psi_2| > 0$. Then, by definition of minimal unsatisfiability, ψ_1 is satisfiable by some truth assignment M. Hence $M \cup \{l\}$ satisfies $\psi_1 \cup \psi_2 = \psi$. This contradicts the hypothesis that ψ is unsatisfiable. A similar argument applies if literal l is negative, proving the lemma. □

Theorem 39. *Let ψ 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 ψ and the literal $\neg v$ occurs exactly one time in ψ.*

Proof. If ψ is not saturated, there is some set of literals already in ψ that may be added to clauses in ψ to make it saturated. Doing so does not change the number of variables and clauses of ψ. So, suppose from now on that ψ 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 ψ is minimum. By Lemma 38, the number of occurrences of literal v in ψ is at least one and the number of occurrences of literal $\neg v$ in ψ is at least one. By Lemma 37,

$$\psi_1 = \{c \setminus \{\neg v\} : c \in \psi, v \notin c\} \quad \text{and} \quad \psi_2 = \{c \setminus \{v\} : c \in \psi, \neg v \notin c\}$$

are minimally unsatisfiable. Clearly, $|V_{\psi_1}| = |V_{\psi_2}| = n - 1$ or else the minimality of variable v is violated. By Theorem 36, the fact that ψ_1 and ψ_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 ψ is one and the number of occurrences of literal $\neg v$ in ψ is one. □

Lemma 40. *Let ψ be a minimally unsatisfiable CNF formula with $n > 0$ variables and $n+1$ clauses. Let v be the variable of Theorem 39 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 $\neg w \in c_{\bar{v}}$. By the minimality of ψ 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 ψ and in the latter case $M \cup \{v\}$ satisfies ψ, 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 41. *Let ψ be a minimally unsatisfiable formula with $n > 1$ variables and $n+1$ clauses. Let v be the variable of Theorem 39, let c_v be the clause of ψ containing the literal v, and let $c_{\neg v}$ be the clause of ψ containing the literal $\neg v$. Then $\psi' = \psi \setminus \{c_v, c_{\neg v}\} \cup \{\mathcal{R}^{c_v}_{c_{\neg v}}\}$ is a minimally unsatisfiable formula with $n-1$ variables and n clauses.*

Proof. By Lemma 40 the resolvent $\mathcal{R}^{c_v}_{c_{\neg v}}$ of c_v and $c_{\neg v}$ exists. Moreover, $\mathcal{R}^{c_v}_{c_{\neg v}}! = \emptyset$ if $n > 1$ or else ψ is not minimally unsatisfiable. Therefore,

ψ' 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 ψ, there is an assignment M which satisfies $\psi \setminus \{c\}$. By Lemma 5, M also satisfies $\psi \setminus \{c, c_v, c_{\neg v}\} \cup \{\mathcal{R}_{c_{\neg v}}^{c_v}\}$. Hence M satisfies $\psi' \setminus \{c\}$. Now remove c_v and $c_{\neg v}$ from ψ. Again, by the minimality of ψ, there is an assignment M which satisfies $\psi \setminus \{c_v, c_{\neg v}\}$ and hence $\psi' \setminus \{\mathcal{R}_{c_{\neg v}}^{c_v}\}$. Thus, removing any clause from ψ' results in a satisfiable formula and the theorem is proved. □

Theorem 41 implies the correctness of the polynomial time algorithm of Figure 51 for determining whether a CNF formula with one more clause than variable is minimally unsatisfiable. If the output is "munsat(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 42. *Let ψ 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 ψ into two disjoint sets ψ_v and $\psi_{\neg v}$ such that literal v only occurs in clauses of ψ_v, literal $\neg v$ only occurs in clauses of $\psi_{\neg v}$ and no variable other than v that is in ψ_v is also in $\psi_{\neg v}$ and no variable other than v that is in $\psi_{\neg v}$ is also in ψ_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 $(v) \wedge (\neg v)$. Suppose the hypothesis is true for all minimally unsatisfiable CNF formulas containing $k > 1$ or fewer variables and having deficiency 1. Let ψ be a minimally unsatisfiable CNF formula with $k+1$ variables and $k+2$ clauses. From Theorem 39 there is a variable v in ψ 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 40 the resolvent $\mathcal{R}_{c_{\neg v}}^{c_v}$ of c_v and $c_{\neg v}$ exists and $\mathcal{R}_{c_{\neg v}}^{c_v} \neq \emptyset$ or else ψ is not minimally unsatisfiable. Let $\psi' = (\psi \setminus \{c_v, c_{\neg v}\}) \cup \{\mathcal{R}_{c_{\neg v}}^{c_v}\}$. That is, ψ' is obtained from ψ by resolving c_v and $c_{\neg v}$ on variable v. By Theorem 41 ψ' is minimally unsatisfiable with k variables and $k+1$ clauses. Then, by the induction hypothesis, there is a partition $\psi'_{v'}$ and $\psi'_{\neg v'}$ of clauses of ψ' 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 $\mathcal{R}_{c_{\neg v}}^{c_v} \in \psi'_{\neg v'}$. Then $\psi'_{v'}$ and $(\psi'_{\neg v'} \setminus \{\mathcal{R}_{c_{\neg v}}^{c_v}\}) \cup \{c_v, c_{\neg v}\}$ (denoted ψ_v and $\psi_{\neg v}$, respectively) form a non variable overlapping partition of clauses of ψ (excluding v' and $\neg v'$), literal v' occurs once in a clause of ψ_v, and literal $\neg v'$ occurs once in a clause of $\psi_{\neg v}$. A similar statement holds if $\mathcal{R}_{c_{\neg v}}^{c_v} \in \psi'_{v'}$. Therefore, the hypothesis holds for ψ or any other minimally unsatisfiable formula of $k+1$ variables and $k+2$ clauses. The theorem follows. □

Theorem 43. *A CNF formula ψ with n variables and $n+1$ clauses is minimally unsatisfiable if and only if there is a refutation tree for ψ in which every clause of ψ labels a leaf exactly one time, every variable of ψ 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 ψ.*

Algorithm 25.

Min Unsat Solver (ψ)
/* *Input: a set of sets CNF formula* ψ */
/* *Output: either "munsat(1)" or "not munsat(1)"* */
Repeat the following until all variables of ψ are eliminated:
If the difference between clauses and variables is not 1,
Output "not munsat(1)."
If no variable of ψ occurs once positively and once negatively,
Output "not munsat(1)."
If, for some v, ψ includes $(v) \wedge (\neg v)$ 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 $\mathcal{R}^{c_v}_{c_{\neg v}}$ of clauses c_v and $c_{\neg v}$.
Set $\psi \leftarrow \psi \setminus \{c_v, c_{\neg v}\} \cup \{\mathcal{R}^{c_v}_{c_{\neg v}}\}$.
Output "munsat(1)."

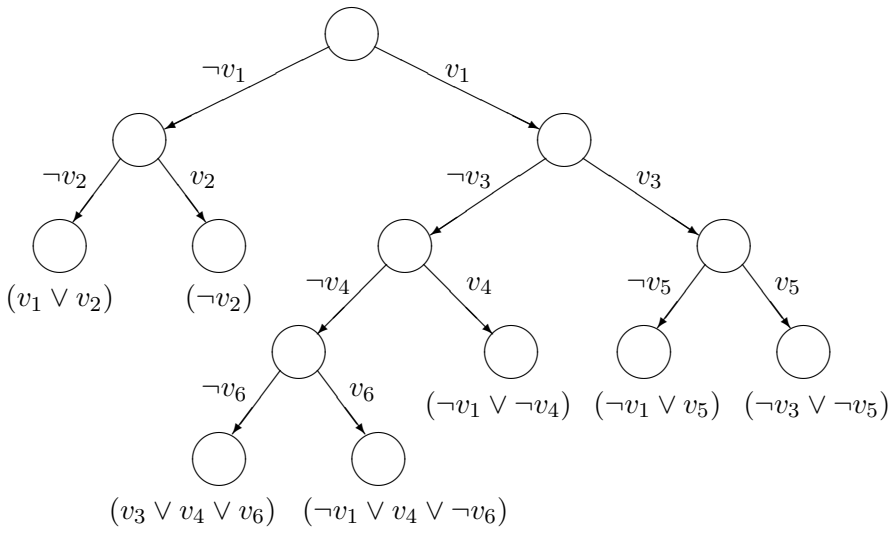
□

Figure 51: *Algorithm for determining whether a CNF formula with one more clause than variable is minimally unsatisfiable.*

Proof. ($\leftarrow$) By induction on the size of the refutation tree. Suppose there is such a refutation tree $\mathcal{T}_\psi$ for ψ. If $\mathcal{T}_\psi$ consists of two edges, they must be labeled v and $\neg v$ for some variable v. In addition, each clause labeling a leaf of $\mathcal{T}_\psi$ must consist of one literal that is opposite to that labeling the edge the leaf is the endpoint of. Hence, ψ must be $(v) \wedge (\neg v)$ which is minimally unsatisfiable.

Now suppose the theorem holds for refutation trees of $k > 2$ or fewer edges and suppose $\mathcal{T}_\psi$ has $k+1$ edges. Let v be the variable associated with the root of $\mathcal{T}_\psi$, and let ψ_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 $\mathcal{T}_\psi$ and ψ_1 and ψ_2 are as described in the statement of the theorem. Hence, by the induction hypothesis, ψ_1 and ψ_2 are minimally unsatisfiable. Let c be a clause in ψ_1. Let M_1 be an assignment to variables of ψ_1 satisfying $\psi_1 \setminus \{c\}$. Let $M_2 = M_1 \cup \{v\}$. Clearly, M_2 satisfies ψ_v 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 ψ_2 and are satisfied by some truth assignment M_3 to variables of ψ_2 since ψ_2 is minimally unsatisfiable. Since the variables of ψ_1 and ψ_2 do not overlap, $M_2 \cup M_3$ is an assignment satisfying $\psi_{\bar{v}} \cup \psi_v \setminus \{c\}$ and therefore $\psi \setminus \{c\}$. The same result is obtained if c is removed from ψ_2. Thus, ψ is minimally unsatisfiable.

($\rightarrow$) 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 ψ. 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



$$\begin{aligned}\psi \;=\; & (v_1 \vee v_2) \wedge (\neg v_2) \wedge (v_3 \vee v_4 \vee v_6) \wedge (\neg v_1 \vee v_4 \vee \neg v_6) \wedge \\ & (\neg v_1 \vee \neg v_4) \wedge (\neg v_1 \vee v_5) \wedge (\neg v_3 \vee \neg v_5)\end{aligned}$$

Figure 52: *A minimally unsatisfiable CNF formula ψ 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.*

resolvent. Construct two edges from the new node to the two nodes which 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 complements 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 52.

Now, attention is turned to the case of minimally unsatisfiable formulas

with deficiency $\delta > 1$. Algorithms for recognizing and solving minimally unsatisfiable formulas in time $n^{O(k)}$ were first presented in [80] and [46]. The fixed-parameter tractable algorithm that is presented in [115] is discussed here.

Let ψ be a CNF formula with m clauses and n variables. The deficiency of ψ is the difference $m - n$. Each subformula of ψ has its own deficiency, namely the difference between the number of clauses of the subformula and the number of variables it contains. The *maximum deficiency* of ψ is then the maximum of the deficiencies of all its subformulas. If, for every non-empty subset V' of variables of ψ there are at least $|V'| + q$ clauses $C \subset \psi$ such that some variable of V' is contained in C, then ψ is said to be *q-expanding*. The following four lemmas are stated without proof.

Lemma 44. *The maximum deficiency of a formula can be determined in polynomial time.* □

Lemma 45. *Let ψ be a minimally unsatisfiable CNF formula which has δ more clauses than variables. Then the maximum deficiency of ψ is δ.* □

Lemma 46. *Let ψ be a minimally unsatisfiable formula. Then for every nonempty set V' of variables of ψ there is at least one clause $c \in \psi$ such that some variable of V' occurs in c.* □

Lemma 47. *Let ψ be a CNF formula with maximum deficiency δ. A maximum matching of the bipartite variable-clause graph G_ψ of ψ does not cover δ clause vertices.* □

Lemma 47 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 48. *Let ψ be a 1-expanding CNF formula with maximum deficiency δ. Let $\psi' \subset \psi$ be a subformula of ψ. Then the maximum deficiency of ψ' is less than δ.*

Proof. We need to show that for any subset $\psi' \subset \psi$, if the number of variables contained in ψ' is n', then $|\psi'| - n' < \delta$. Let G_ψ be the bipartite variable-clause graph of ψ. In what follows symbols will be used to represent clause vertices in G_ψ and clauses in ψ interchangeably. Choose a clause $c \in \psi \setminus \psi'$. Let M_{G_ψ} be a maximum matching on G_ψ 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_ψ that are not covered by M_{G_ψ}. By Lemma 47 $|C| = \delta$ so, since $c \notin \psi'$, $|C \cap \psi'| < \delta$. Since M_{G_ψ} matches every clause vertex in $\psi' \setminus C$ to a variable that is in ψ', 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 δ. □

Theorem 49. ([115]) *Let ψ be a CNF formula with maximum deficiency δ. The satisfiability of ψ can be determined in time $O(2^\delta n^3)$ where n is the number of variables in ψ.*

Proof. Let m be the number of clauses in ψ. By hypothesis, $m \leq n + \delta$. Let $G = (V_1, V_2, E)$ be the variable-clause matching graph for ψ. 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 matching algorithm for bipartite graphs [30]. The next step is to build a refutation tree for ψ of depth δ. □

Theorem 50. ([115]) *Let ψ be a minimally unsatisfiable CNF formula with δ more clauses than variables. Then ψ can be recognized as such in time $O(2^\delta n^4)$ where n is the number of variables in ψ.*

Proof. By Theorem 45 the maximum deficiency of ψ is δ which, by Lemma 47, can be checked in $O(n^3)$ time. Since, by Lemma 48, the removal of a clause from ψ reduces the maximum deficiency of ψ, the algorithm inferred by Theorem 49 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 ψ 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. □

5.11 Bounded Resolvent Length Resolution

This class is considered here because it is a counterpoint to minimally unsatisfiable formulas. A CNF formula ψ is k-BRLR if Algorithm **BRLR** of Figure 53 either generates all resolvents of ψ or returns "unsatisfiable."

Algorithm **BRLR** implements a simplified form of *k-closure* [122]. 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 $\mathcal{T}$ of matrices of dimension $n \times 2, \binom{n}{2} \times 4, \ldots, \binom{n}{k} \times 2^k$ such that each cell has value 1 if and only if a corresponding original clause of ψ 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 $\mathcal{T}$ in constant time by setting a link just one time as the clause is scanned the first time or created as a resolvent.

Algorithm 26.

BRLR(ψ, k)
/* *Input: a set of sets CNF formula* ψ */
/* *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 $\mathcal{R}_{c_2}^{c_1} \notin \psi$, $|\mathcal{R}_{c_2}^{c_1}| \leq k$
    Set $\psi \leftarrow \psi \cup \mathcal{R}_{c_2}^{c_1}$.
  Otherwise, if $\emptyset \notin \psi$ do the following:
    If all resolvents have been generated then Output "satisfiable".
    Otherwise, Output "give up".
  If $\emptyset \in \psi$, Output "unsatisfiable."

□

Figure 53: Finding a k-bounded refutation

Initially, set to 1 all the cells of $\mathcal{T}$ which correspond to a clause of ψ 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 $\mathcal{T}$ 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 $\mathcal{T}$ 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 $\mathcal{T}$, 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.

5.12 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.

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:

$$(v_1 \vee \neg v_2 \vee v_4) \wedge (v_1 \vee v_2 \vee v_5) \wedge (\neg v_1 \vee \neg v_3 \vee v_6) \wedge (\neg v_1 \vee v_3 \vee v_7).$$

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 α's to 0 in 2.

The following formula is matched and SLUR but is not q-Horn:

$$(\neg v_2 \vee v_3 \vee \neg v_5) \wedge (\neg v_1 \vee \neg v_3 \vee v_4) \wedge (v_1 \vee v_2 \vee \neg v_4).$$

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):

$$(\neg v_3 \vee \neg v_4) \wedge (v_3 \vee v_4 \vee \neg v_5) \wedge (\neg v_1 \vee v_2 \vee \neg v_3) \wedge (v_1 \vee v_3 \vee v_5).$$

The following formula is nested but not SLUR (choose v_3 first, use the branch where v_3 is 0 to enter a situation where satisfaction is impossible):

$$(v_1 \vee v_2) \wedge (v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee v_4) \wedge (\neg v_1 \vee \neg v_4).$$

The following is a linear autark formula that is not q-Horn ($\boldsymbol{\alpha}=\langle 0.5, -0.5, 0\rangle$ and the minimum Z is 3/2):

$$(v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee \neg v_2 \vee \neg v_3).$$

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 $\boldsymbol{\alpha}=\langle 0, 0, 0\rangle$ satisfies the inequality of (14)):

$$(\neg v_1 \vee v_2 \vee v_3) \wedge (v_1 \vee \neg v_2 \vee v_3) \wedge (v_1 \vee v_2 \vee \neg v_3) \wedge (\neg v_1 \vee \neg v_2 \vee \neg v_3).$$

The following formula is minimally unsatisfiable with one more clause than variable, but is not 3-BRLR:

$$(v_0 \vee v_1 \vee v_{48}) \wedge (v_0 \vee \neg v_1 \vee v_{56}) \wedge (\neg v_0 \vee v_2 \vee v_{60}) \wedge (\neg v_0 \vee \neg v_2 \vee v_{62}) \wedge$$
$$(v_3 \vee v_4 \vee \neg v_{48}) \wedge (v_3 \vee \neg v_5 \vee v_{56}) \wedge (\neg v_3 \vee v_5 \vee v_{60}) \wedge (\neg v_3 \vee \neg v_5 \vee v_{62}) \wedge$$
$$(v_6 \vee v_7 \vee v_{49}) \wedge (v_6 \vee \neg v_7 \vee \neg v_{56}) \wedge (\neg v_6 \vee v_8 \vee v_{60}) \wedge (\neg v_6 \vee \neg v_8 \vee v_{62}) \wedge$$
$$(v_9 \vee v_{10} \vee \neg v_{49}) \wedge (v_9 \vee \neg v_{10} \vee \neg v_{56}) \wedge (\neg v_9 \vee v_{11} \vee v_{60}) \wedge (\neg v_9 \vee \neg v_{11} \vee v_{62}) \wedge$$
$$(v_{12} \vee v_{13} \vee v_{50}) \wedge (v_{12} \vee \neg v_{13} \vee v_{57}) \wedge (\neg v_{12} \vee v_{14} \vee \neg v_{60}) \wedge (\neg v_{12} \vee \neg v_{14} \vee v_{62}) \wedge$$
$$(v_{15} \vee v_{16} \vee \neg v_{50}) \wedge (v_{15} \vee \neg v_{16} \vee v_{57}) \wedge (\neg v_{15} \vee v_{17} \vee \neg v_{60}) \wedge (\neg v_{15} \vee \neg v_{17} \vee v_{62}) \wedge$$
$$(v_{18} \vee v_{19} \vee v_{51}) \wedge (v_{18} \vee \neg v_{19} \vee \neg v_{57}) \wedge (\neg v_{18} \vee v_{20} \vee \neg v_{60}) \wedge (\neg v_{18} \vee \neg v_{20} \vee v_{62}) \wedge$$
$$(v_{21} \vee v_{22} \vee \neg v_{51}) \wedge (v_{21} \vee \neg v_{22} \vee \neg v_{57}) \wedge (\neg v_{21} \vee v_{23} \vee \neg v_{60}) \wedge (\neg v_{21} \vee \neg v_{23} \vee v_{62}) \wedge$$
$$(v_{24} \vee v_{25} \vee v_{52}) \wedge (v_{24} \vee \neg v_{25} \vee v_{58}) \wedge (\neg v_{24} \vee v_{26} \vee v_{61}) \wedge (\neg v_{24} \vee \neg v_{26} \vee \neg v_{62}) \wedge$$
$$(v_{27} \vee v_{28} \vee \neg v_{52}) \wedge (v_{27} \vee \neg v_{28} \vee v_{58}) \wedge (\neg v_{27} \vee v_{29} \vee v_{61}) \wedge (\neg v_{27} \vee \neg v_{29} \vee \neg v_{62}) \wedge$$
$$(v_{30} \vee v_{31} \vee v_{53}) \wedge (v_{30} \vee \neg v_{31} \vee \neg v_{58}) \wedge (\neg v_{30} \vee v_{32} \vee v_{61}) \wedge (\neg v_{30} \vee \neg v_{32} \vee \neg v_{62}) \wedge$$
$$(v_{33} \vee v_{34} \vee \neg v_{53}) \wedge (v_{33} \vee \neg v_{34} \vee \neg v_{58}) \wedge (\neg v_{33} \vee v_{35} \vee v_{61}) \wedge (\neg v_{33} \vee \neg v_{35} \vee \neg v_{62}) \wedge$$
$$(v_{36} \vee v_{37} \vee v_{54}) \wedge (v_{36} \vee \neg v_{37} \vee v_{59}) \wedge (\neg v_{36} \vee v_{38} \vee \neg v_{61}) \wedge (\neg v_{36} \vee \neg v_{38} \vee \neg v_{62}) \wedge$$
$$(v_{39} \vee v_{40} \vee \neg v_{54}) \wedge (v_{39} \vee \neg v_{40} \vee v_{59}) \wedge (\neg v_{39} \vee v_{41} \vee \neg v_{61}) \wedge (\neg v_{39} \vee \neg v_{41} \vee \neg v_{62}) \wedge$$
$$(v_{42} \vee v_{43} \vee v_{55}) \wedge (v_{42} \vee \neg v_{43} \vee \neg v_{59}) \wedge (\neg v_{42} \vee v_{44} \vee \neg v_{61}) \wedge (\neg v_{42} \vee \neg v_{44} \vee \neg v_{62}) \wedge$$
$$(v_{45} \vee v_{46} \vee \neg v_{55}) \wedge (v_{45} \vee \neg v_{46} \vee \neg v_{59}) \wedge (\neg v_{45} \vee v_{47} \vee \neg v_{61}) \wedge (\neg v_{45} \vee \neg v_{47} \vee \neg v_{62})$$

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 ψ that is unsatisfiable, add another clause arbitrarily using the variables of ψ. 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 5.13 using probabilistic measures to determine relative sizes of the classes.

5.13 Probabilistic comparison of incomparable classes.

Formulas that are members of certain polynomial time solvable classes that have been considered in this section appear to be much less frequent than formulas that can usually be solved efficiently by some variant of DPLL. This statement is supported by a probabilistic analysis of these classes where a random formula is an instance of k-SAT with m clauses chosen uniformly and without replacement from all possible width k clauses taken from n variables. A few examples may help to illuminate. In what follows ψ is used to denote a random k-SAT formula.

Consider the class of Horn formulas (Section 5.2) first. The probability that a randomly generated clause is Horn is $(k+1)/2^k$ so the probability that ψ is Horn is $((k+1)/2^k)^m$. This tends to 0 as m tends to ∞ for any fixed k. For a hidden Horn formula (Section 5.3), 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 $2^n((k+1)/2^k)^m$. This tends to 0 for increasing m and n if $m/n > 1/(k - \log_2(k+1))$. Therefore, by Markov's inequality, ψ 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 5.5) is taken from [47]. 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

$$\begin{aligned}&(v_1 \vee \neg v_2) \wedge (v_2 \vee \neg v_3) \wedge \ldots \wedge (v_i \vee \neg v_{i+1} \vee v_{p+1}) \wedge \\ &\ldots \wedge (v_j \vee \neg v_{j+1} \vee \neg v_{p+1}) \wedge \ldots \wedge (v_p \vee \neg v_1)\end{aligned} \tag{18}$$

where $v_i \neq v_j$ if $i \neq j$. 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 $\mathcal{C} \subset \psi$, if no two literals removed from $\mathcal{C}$ are the same or complementary, then $\mathcal{C}$ is called a *q-blocked c-cycle*.

If ψ has a q-blocked c-cycle then it is not q-Horn. Let a q-blocked c-cycle in ψ be represented as above. Develop satisfiability index inequalities (2) for ψ. After rearranging terms in each, a subset of these inequalities is as follows

$$\begin{aligned}
&\alpha_1 \leq Z - 1 + \alpha_2 - \ldots && (19)\\
&\ldots\\
&\alpha_i \leq Z - 1 + \alpha_{i+1} - \alpha_{p+1} - \ldots\\
&\ldots\\
&\alpha_j \leq Z - 1 + \alpha_{j+1} - (1 - \alpha_{p+1}) - \ldots\\
&\ldots\\
&\alpha_p \leq Z - 1 + \alpha_1 - \ldots . && (20)
\end{aligned}$$

From inequalities (19) to (20) it can be deduced that

$$\alpha_1 \leq pZ - p + \alpha_1 - (1 - \alpha_{p+1} + \alpha_{p+1}) - \ldots$$

or

$$0 \leq pZ - p - 1 + \ldots$$

where all the terms in $\ldots$ are non-positive. Thus, all solutions to (19) through (20) 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 24) that $Z \leq 1$ in order for ψ 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 51. *A random k-SAT formula is not q-Horn, with probability tending to 1, if* $m/n > 4/(k^2 - k)$. □

For $k = 3$ this is $m/n > 2/3$.

A similar analysis yields the same results for hidden Horn, SLUR, CC-balanced, or extended Horn classes. The critical substructure which causes ψ not to be SLUR is called a criss-cross loop. An example is shown is Figure 54 as a propositional connection graph. Just one criss-cross loop in ψ prevents it from being SLUR. Comparing Figure 54 and expression (18) it is evident 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-SAT formulas are constructed without consideration of such cyclic structures: at some point as m/n is increased cycles begin to appear in ψ 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-SAT formula is *not* a member of one of the above named polynomial time solvable classes,

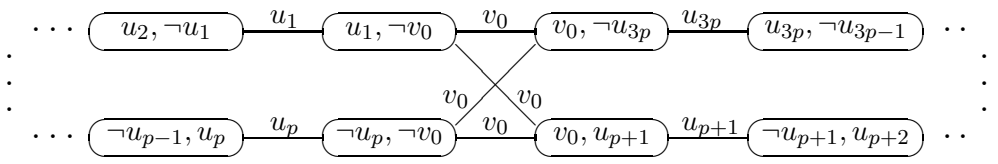


Figure 54: A "criss-cross loop" of $t = 3p + 2$ clauses represented as a propositional connection graph. Only "cycle literals" are shown in the nodes; "padding literals," required for $k \geq 3$ and different from cycle literals, are present but are not shown.

with probability tending to 1, when $m/n > 1/O(k^2)$. By contrast, simple polynomial time procedures will solve a random k-SAT formula with high probability when $m/n < 2^k/O(k)$. The disappointing conclusion that many polynomial time solvable classes are relatively rare among random k-SAT 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 105) 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-SAT generator produces "many more" matched formulas than SLUR or q-Horn formulas. Let $Q \subset \psi$ be any subset of clauses of ψ. Denote by $\mathbf{V}(Q)$ the neighborhood of Q: that is, the set of variables that occur in Q. Then $\mathbf{V}(Q)$ is the set variables corresponding to a subset of vertices in V_2 of G_ψ that are adjacent to the vertices in V_2 that correspond to clauses in Q. Denote the deficiency of Q by $\delta(Q)$. From the definition of deficiency (Page 122) $\delta(Q) = |Q| - |\mathbf{V}(Q)|$. A subset $Q \subset \psi$ is said to be *deficient* if $\delta(Q) > 0$. The following theorem is well known.

Theorem 52. (Hall's Theorem [55])
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 is deficient. □

Theorem 53. *Random k-SAT formulas are matched formulas with probability tending to 1 if $m/n < r(k)$ where $r(k)$ is given by the following table [47].*

k	$r(k)$
3	*.64*
4	*.84*
5	*.92*
6	*.96*
7	*.98*
8	*.990*
9	*.995*
10	*.997*

□

Theorem 53 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 52 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 in this subsection up to this point are interesting for at least two reasons. First, Theorem 53 says that random k-SAT formulas are matched formulas with high probability if $m/n < r(k)$ which is approximately 1. But, by Theorem 51 and similar results, a random k-SAT 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, mostly 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 has been generalized considerably to larger polynomial time solvable classes such as Linear Autarkies [79, 90] which are described in Section 5.9 and biclique satisfiable formulas of Section 5.7.

But, there is disappointing news concerning Linear Autarkies. Consider the test for satisfiability that is implied by Theorem 35: 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 114, will output $\psi' = \emptyset$ on any formula for which $z < k/2$.

Theorem 54. *The above test for satisfiability does not succeed on a random k-SAT formula ψ with probability tending to 1 as $m, n \to \infty$ and $m/n > 1$.* □

6 Other Topics

Several significant topics are not treated in this chapter. There are experimental algorithms that control the space of solutions with linear or quadratic cuts. The Handbook on Satisfiability [13] is a good source of information about these. Several probabilistic algorithms other than those presented here have been proposed, for example Simulated Annealing and Genetic Algorithm variants. These were omitted in favor of more influential varieties. Message passing algorithms such as Survey Propagation have been omitted as they seem to have a special niche. More general constraint programming algorithms have been omitted as the focus here is strictly on SAT. The use of SAT in non-monotonic settings was mentioned in the applications sec-

tion but the topic is too broad to cover adequately in this chapter so Stable Models, Well Founded Semantics, and Answer Set Programming algorithms, among others, have been omitted.

7 Acknowledgment

We thank John S. Schlipf for help in writing some of the text, particularly the sections on the diagnosis of circuit faults, and Binary Decision Diagrams. We especially appreciate John's attention to detail which was quite valuable to us.

References

[1] A. Agrawal, P. Klein, and R. Ravi. When trees collide: an approximation algorithm for the generalized Steiner problem on networks. *SIAM Journal on Computing*, **24**:440–456, 1995.

[2] R. Aharoni, and N. Linial. Minimal Non-Two-Colorable Hypergraphs and Minimal Unsatisfiable Formulas. *Journal of Combinatorial Theory, Series A*, **43**:196–204, 1986.

[3] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, **C-27**:509–516, 1978.

[4] M. Alekhnovich, J. Johannsen, T. Pitassi, A. Urquhart. An exponential separation between regular and general resolution. Theory of Computing, **3**:81-102, 2007. http://theoryofcomputing.org

[5] R. Amara, and A.J. Lipinski. Business Planning for an Uncertain Future: Scenarios & Strategies. *Pergammon Press*, 1983.

[6] H. Andersen. An Introduction to Binary Decision Diagrams. Technical report, Department of Information Technology, Technical University of Denmark, 1998.

[7] B. Aspvall. Recognizing disguised NR(1) instances of the satisfiability problem. *Journal of Algorithms*, **1**:97–103, 1980.

[8] G. Audemard, and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 399–404, Pasadena, CA, 2009.

[9] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 1.2. Linked from `http://combination.cs.uiowa.edu/smtlib/`.

[10] P. Beame, H. Kautz, and A. Sabharwal. Understanding the Power of Clause Learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, 1194–1201, Hyderabad, India, 2007.

[11] P. Beame, H. Kautz, and A. Sabharwal. Towards Understanding and Harnessing the Potential of Clause Learning. *Journal of Artificial Intelligence Research*, **22**:319–351, 2004.

[12] A. Biere, A. Cimatti, E. Clarke, Y. Zhu. Symbolic Model Checking without BDDs. *Lecture Notes in Computer Science*, **1579**:193–207, 1999.

[13] A. Biere, M. Heule, H. van Maaren, and T. Walsh (eds.) *Handbook of Satisfiability*, IOS Press, 2009.

[14] A. Blake. Canonical Expressions in Boolean Algebra. Ph.D. Dissertation, Department of Mathematics, University of Chicago, 1937.

[15] E. Boros, Y. Crama, and P.L. Hammer. Polynomial-time inference of all valid implications for Horn and related formulae. *Annals of Mathematics and Artificial Intelligence*, **1**:21–32, 1990.

[16] E. Boros, P.L. Hammer, and X. Sun. Recognition of q-Horn formulae in linear time. *Discrete Applied Mathematics*, **55**:1–13, 1994.

[17] E. Boros, Y. Crama, P.L. Hammer, and M. Saks. A complexity index for satisfiability problems. *SIAM Journal on Computing*, **23**:45–49, 1994.

[18] E. Boros, P.L. Hammer, T. Ibaraki, A. Kogan, E. Mayoraz, and I. Muchnik. An implementation of Logical Analysis of Data. RUTCOR Research Report RRR 04-97, RUTCOR, Rutgers University, 1996.

[19] E. Boros, P.L. Hammer, T. Ibaraki, and A. Kogan. Logical analysis of numerical data. *Mathematical Programming*, **79**:163–190, 1997.

[20] D. Brand. Verification of large synthesized designs. In *Proceeding of the International Conference on Computer Aided Design*, 534–537, IEEE Computer Society Press, Los Alamitos, 1993.

[21] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, **C-35**:677–691, 1986.

[22] R.E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, **24**:293–318, 1992.

[23] B. Buchberger. An Algorithm for Finding a Basis for the Residue of a Class Ring of a Zero-Dimensional Polynomial Ideal. Ph.D. Thesis, Universität Innsbruck, Institut für Mathematik, 1965.

[24] R. Chandrasekaran. Integer programming problems for which a simple rounding type of algorithm works. In *Progress in Combinatorial Optimization*, W. Pulleyblank (ed.), Academic Press Canada, Toronto, Ontario, Canada, 101–106, 1984.

[25] V. Chandru, and J.N. Hooker. Extended Horn sets in propositional logic. *Journal of the Association for Computing Machinery*, **38**:205–221, 1991.

[26] E. Clarke, A. Biere, R. Raimi, Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, **19**(1):7–34, 2001.

[27] M. Clegg, J. Edmonds, and R. Impagliazzo. Using the Groebner basis algorithm to find proofs of unsatisfiability. *28th ACM Symposium on the Theory of Computing*, 174–183, ACM, Philadelphia, Pennsylvania, United States, 1996.

[28] M. Conforti, G. Cornuéjols, A. Kapoor, K. Vušković, and M.R. Rao. Balanced Matrices. In *Mathematical Programming: State of the Art*, J.R. Birge, and K.G. Murty (eds.), Braun-Brumfield, United States. Produced in association with the 15th International Symposium on Mathematical Programming, University of Michigan, 1994.

[29] S.A. Cook, and R.A. Reckhow. Corrections for "On the Lengths of Proofs in the Propositional Calculus." *SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, **6**, 1974.

[30] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms, 2nd edition. MIT Press and McGraw-Hill, 2001.

[31] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. *Lecture Notes in Computer Science*, **407**:365–373, Springer, 1990.

[32] O. Coudert, and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the 1990 International Conference on Computer-Aided Design* (ICCAD '90), 126–129, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[33] O. Coudert. On solving covering problems. In *Proceedings of the* 33^{rd} *Design Automation Conference*, 197–202, IEEE Computer Society Press, Los Alimitos, 1996.

[34] Y. Crama, P.L. Hammer, and T. Ibaraki. Cause-effect relationships and partially defined Boolean functions. *Annals of Operations Research* **16**:299–326, 1998.

[35] W. Craig. Linear reasoning: A new form of the Herbrand-Gentzen theorem. *Journal of Symbolic Logic*, **22**(3):250-268, 1957.

[36] J.M. Crawford, and A.B. Baker. Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In *Proceedings of the National Conference on Artificial Intelligence*, 1092–1097, AAAI Press/The MIT Press, Menlo Park, CA, 1994.

[37] A. Darwiche, and K. Pipatsrisawat. Complete Algorithms In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh (eds.), IOS Press, 99–130, 2009.

[38] M. Davis, and H. Putnam. Computational methods in the propositional calculus. Unpublished report, Rensselaer Polytechnic Institute, 1958.

[39] M. Davis, and H. Putnam. A computing procedure for quantification theory. *J. of the ACM*, **7**:201–215, 1960.

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

[41] W.F. Dowling, and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, **1**:267–284, 1984.

[42] C. Ducot, and G.J. Lubben. A typology for Scenarios. *Future*, **12**:51–57, 1980.

[43] B. Dutertre, and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T)*. In *Lecture Notes in Computer Science*, **4144**:81-94, Springer, Berlin/Heidelberg, 2006.

[44] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multi-commodity flow problems. *SIAM J. on Computing*, **5**:691–703, 1976.

[45] R. Feldmann, and N. Sensen. Efficient algorithms for the consistency analysis in szenario projects. Technical Report, Universität Paderborn, Germany, 1997.

[46] H. Fleischner, O. Kullmann, and S. Szeider. Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference. *Theoretical Computer Science*, **289**(1):503-516, 2002.

[47] J. Franco, and A. Van Gelder. A Perspective on Certain Polynomial Time Solvable Classes of Satisfiability. *Discrete Applied Mathematics*, **125**(2-3):177–214, 2003.

[48] J.W. Freeman. Improvements to Propositional Satisfiability Search Algorithms. Ph.D. Thesis, University of Pennsylvania, Computer and Information Science, 1995.

[49] Z. Fu, and S. Malik. Solving the minimum-cost Satisfiability problem using SAT based branch-and-bound search. In *Proceedings of the International Conference on Computer Aided Design*, 852–859, Association for Computing Machinery, New York, 2006.

[50] J. Gausemeier, A. Fink, and O. Schlake. Szenario-Management. *Carl Hanser Verlag*, München-Wien, 1995.

[51] M. Godet. Scenarios and Strategic Management. *Butterworths Publishing (Pty) Ltd.*, South Africa, 1987.

[52] E. Goldberg, and Y. Novikov. How good can a resolution based SAT-solver be? *Lecture Notes in Computer Science*, **2919**:35–52, Springer, 2003.

[53] E. Grégoire, B. Mazure, and L. Saïs. Logically-complete local search for propositional non-monotonic knowledge bases.

[54] S. Haim, and T. Walsh. Restart Strategy Selection Using Machine Learning Techniques. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, 312–325, Swansea, UK, 2009.

[55] P. Hall. On representatives of subsets. *Journal of London Mathematical Society*, **10**:26–30, 1935.

[56] P.L. Hammer. Partially defined Boolean functions and cause-effect relationships. In *Proceedings of the International Conference on Multi-Attribute Decision Making Via OR-Based Expert Systems*, University of Passau, Passau, Germany, 1986.

[57] P. Hansen, B. Jaumard, and G. Plateau. An extension of nested satisfiability. Les Cahiers du GERAD, G-93-27, 1993.

[58] H.H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the 16th National Conference on Artificial Intelligence*, 661–666, AAAI Press/The MIT Press, Menlo Park, CA, 1999.

[59] H.H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the National Conference on Artificial Intelligence*, 655–660, AAAI Press/The MIT Press, Menlo Park, CA, 2002.

[60] H.H. Hoos, and D.A.D. Tompkins. Adaptive Novelty+. Technical Report, Computer Science Department, University of British Columbia, 2007. Available from: `http://www.satcompetition.org/2007/adaptnovelty.pdf`

[61] J. Huang. The Effect of Restarts on the Efficiency of Clause Learning. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 2318–2323, Acapulco, Mexico, 2003.

[62] W. Hunt, and S. Swords. Centaur Technology Media Unit Verification. In *Proceedings of the International Conference on Computer Aided Design*, 353–367, Association for Computing Machinery, Grenoble, France, 2009.

[63] A. Itai, and J. Makowsky. On the complexity of Herbrand's theorem. Working paper 243, Department of Computer Science, Israel Institute of Technology, 1982.

[64] R.E. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of Mathematics and AI*, **1**:167–187, 1990.

[65] D.S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and Systems Sciences*, **9**:256–278, 1974.

[66] E.L. Johnson, G.L. Nemhauser, and M.W.P. Savelsbergh. Progress in linear programming-based algorithms for Integer Programming: an exposition. *INFORMS Journal on Computing*, **12**(1):2–23, 2000.

[67] R.M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, R.E. Miller, and J.W. Thatcher (eds.), Plenum Press, New York, 85–103, 1972.

[68] H. Kautz, B. Selman, and Y. Jiang. Stochastic search for solving weighted MAX-SAT encodings of constraint satisfaction problems. Preprint, AT&T Laboratories, 600 Mountain Ave., Murray Hill, NJ, 1998.

[69] H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, and B. Selman Dynamic Restart Policies. In *Proceedings of the National Conference on Artificial Intelligence*, 674–681, AAAI Press/The MIT Press, Menlo Park, CA, 2002.

[70] H. Kautz, and B. Selman. MAXWalkSat. Available from: `http://www.cs.rochester.edu/ kautz/walksat/`

[71] H. Kautz, A. Sabharwal, and B. Selman. Incomplete Algorithms. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh (eds.), IOS Press, 185–204, 2009.

[72] F. Klaedtke. Decision Procedures for Logical Theories, Lecture 12: Combining Decision Procedures: Nelson-Oppen. Department of Computer Science, ETH Zurich, 2006.

[73] D. Knuth. Nested satisfiability. *Acta Informatica*, **28**:1-6, 1990.

[74] F. Krohm, and A. Kuehlmann. Equivalence checking using cuts and heaps. In *Proceedings of the 34th Design Automation Conference*, 263–268, IEEE Computer Society Press, Los Alimitos, 1997.

[75] F. Krohm, A. Kuehlmann, and A. Mets. The use of random simulation in formal verification. In *Proceedings of the IEEE International Conference on Computer Design*, 371–376, IEEE Computer Society Press, Los Alimitos, 1997.

[76] A. Kuehlmann, V. Paruthi, F. Krohm, and M.K. Ganai. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer Aided Design*, **21**(12):1377–1394, 2002.

[77] A. Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *Proceeding of the IEEE International Conference on Computer Aided Design*, 50–57, IEEE Computer Society Press, Los Alamitos, 2004.

[78] O. Kullmann. A first summary on minimal unsatisfiable clause-sets. Technical report, June, 1998.

[79] O. Kullmann. Investigations on autark assignments. *Discrete Applied Mathematics*, **107**:99-137, 2000.

[80] O. Kullmann. An application of Matroid theory to the SAT problem. In *Proceedings of the* 15th *Annual IEEE Conference on Computational Complexity*, 116–124, 2000.

[81] H. Kleine Büning. On the minimal unsatisfiability problem for some subclasses of CNF. In *Abstracts of 16th Int'l Symposium on Mathematical Programming*, Lausanne, 1997.

[82] M. Kouril, and J. Franco. Resolution Tunnels for Improved SAT Solver Performance. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, 136–141, St. Andrews, UK, 2005.

[83] M. Kouril, and J. Paul. The van der Waerden Number W(2, 6) Is 1132. *Experimental Mathematics*, **17**(1):53–61, 2008.

[84] R.A. Kowalski. A proof procedure using connection graphs. *Journal of the ACM*, **22**:572–595, 1974.

[85] C.Y. Lee. Representation of switching circuits by binary decision programs. *Bell Systems Technical Journal*, **38**:985–999, 1959.

[86] H.R. Lewis. Renaming a set of clauses as a Horn set. *Journal of the Association for Computing Machinery*, **25**:134–135, 1978.

[87] D. Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, **11**:329–343, 1982.

[88] I. Lynce, and J.P.M. Silva. Efficient data structures for backtrack search SAT solvers *Annals of Mathematics and Artificial Intelligence*, **43**(1):137–152, 2005.

[89] I. Lynce, and J.P.M. Silva. SAT in Bioinformatics: Making the Case with Haplotype Inference. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, 136–141, Seattle, WA, 2006.

[90] H. van Maaren. A short note on some tractable classes of the Satisfiability problem. *Information and Computation*, **158**(2):125–130, 2000.

[91] V.M. Manquinho, and J.P.M. Silva. Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem. *IEEE Transactions on CAD of Integrated Circuits and Systems*, **21**(5):505–516, 2002.

[92] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the 14th National Conference on Artificial Intelligence*, 321-326, AAAI Press/The MIT Press, Menlo Park, CA, 1997.

[93] I. Mironov, and L. Zhang. Applications of SAT Solvers to Cryptanalysis of Hash Functions. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing*, 102–115, Seattle, WA, 2006.

[94] B. Monien, and E. Speckenmeyer. Solving Satisfiability in less than 2^n steps. *Discrete Applied Mathematics*, **10**:287–295, 1985.

[95] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference*, 530–535, Las Vegas, NV, USA, 2001.

[96] G. Nelson, and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, **1**(2):245-257, 1979.

[97] D.C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, **12**:291-302, 1980.

[98] W.V.O. Quine. A Way To Simplify Truth Functions. *American Mathematical Monthly*, **62**:627–631, 1955.

[99] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, **32**:57–95, 1987.

[100] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, **12**:23–41, 1965.

[101] L. Ryan. Efficient Algorithms For Clause-Learning SAT Solvers. Masters Thesis, Simon Fraser University, 2004.

[102] E.W. Samson, and B.E. Mills. Circuit minimization: algebra and algorithms for new Boolean canonical expressions. AFCRC Technical Report 54-21, 1954.

[103] T.S. Schaefer. The complexity of satisfiability problems. In *Proceedings of* 10^{th} *Annual ACM Symposium on Theory of Computing*, 216–226, ACM, New York, 1978.

[104] J.S. Schlipf, F. Annexstein, J. Franco, and R. Swaminathan. On finding solutions for extended Horn formulas. *Information Processing Letters*, **54**:133–137, 1995.

[105] M.G. Scutella. A note on Dowling and Gallier's top-down algorithm for propositional Horn satisfiability. *Journal of Logic Programming*, **8**:265–273, 1990.

[106] R. Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation*, **3**:(3-4)141–224, 2007.

[107] C. Barrett, R. Sebastiani, S.A. Sanjit, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh (eds.), IOS Press, 825–886, 2009.

[108] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Cliques, Coloring, and Satisfiability: the second DIMACS implementation challenge*, D.S. Johnson and M.A. Trick (eds.), American Mathematical Society, 521–532, 1996.

[109] Y. Shang, and B.W. Wah. A discrete Lagrangian-based global search method for solving satisfiability problems. *Journal of Global Optimization*, **12**(1):61–100, 1998.

[110] C.E. Shannon. A symbolic analysis of relay and switching circuits. Masters Thesis, Massachusetts Institute of Technology, 1940. Available from http://hdl.handle.net/1721.1/11173.

[111] J.P.M. Silva, and K.A. Sakallah. GRASP - A New Search Algorithm For Satisfiability. In *Proceedings of the 1996 International Conference on Computer-Aided Design* (ICCAD '96), 220–227, IEEE Computer Society Press, Los Alamitos, CA, 1996.

[112] J.P.M. Silva, I. Lynce, and S. Malik. Conflict-Driven Clause Learning SAT Solvers. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsh (eds.), IOS Press, 131–153, 2009.

[113] E.W. Smith, and D.L. Dill. Automatic Formal Verification of Block Cipher Implementations. In *Proceedings of the Formal Methods in Computer-Aided Design Conference*, 1-7, Portland, Oregon, 2008.

[114] R.P. Swaminathan, and D.K. Wagner. The arborescence–realization problem. *Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, **59**(3):267–283, 1995.

[115] S. Szeider. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. *Journal of Computer and System Sciences*, **69**:656–674, 2004.

[116] S. Szeider. Generalizations of matched CNF formulas. *Annals of Mathematics and Artificial Intelligence*, **43**(1):223–238, 2005.

[117] C.A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, **8**:85–89, 1984.

[118] K. Truemper. Monotone Decomposition of Matrices. Technical Report UTDCS-1-94, University of Texas at Dallas. 1994.

[119] K. Truemper. Effective Logic Computation. John Wiley, 1998.

[120] G.S. Tseitin. On the Complexity of Derivations in Propositional Calculus. In *Structures in Constructive Mathematics and Mathematical Logic, Part II*, A.O. Slisenko (ed.), Nauka, Moscow, 115–125, 1968 (translated from Russian).

[121] L.G. Valiant. A Theory of the Learnable. *Communications of the ACM*, **27**:1134–1142, 1984.

[122] A. Van Gelder, and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, D.S. Johnson, and M. Trick (eds.), American Mathematical Society, 1996.

[123] A. Van Gelder. Generalizations of Watched Literals for Backtracking Search. In *Seventh International Symposium on Artificial Intelligence and Mathematics*, 2002.

[124] A. Van Gelder. Verifying propositional unsatisfiability: Pitfalls to avoid. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, 328–333, Lisbon, Portugal, 2007.

[125] A. Van Gelder. Improved Conflict-Clause Minimization Leads to Improved Propositional Proof Traces. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, 141–146, Swansea, UK, 2009.

[126] S. Weaver, J. Franco, and J. Schlipf. Extending existential quantification in conjunctions of BDDs. *Journal on Satisfiability, Boolean Modeling and Computation*, **1**:89–110, 2006.

[127] Z. Wu, and B.W. Wah. Trap escaping strategies in discrete Lagrangian methods for solving hard satisfiability and maximum satisfiability problems. In *Proceedings of the 16th National Conference on Artificial Intelligence*, 673–678, AAAI Press/The MIT Press, Menlo Park, CA, 1999.

[128] Z. Wu, and B.W. Wah. An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In *Proceedings of the 17th National Conference on Artificial Intelligence*, 310–315, AAAI Press/The MIT Press, Menlo Park, CA, 2000.

[129] M. Yoeli. Formal Verification of Hardware Design. IEEE Computer Society Press, Los Alamitos, California, 1988.

[130] L. Zhang, and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Proceedings of the conference on Design, Automation and Test in Europe*, page 880-885, Washington, DC, USA, 2003.

[131] H. Zhang, D. Li, and H. Shen. A SAT Based Scheduler for Tournament Schedules. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, Vancouver, BC, 2004.

A Glossary

Algorithm: (16, 25, 30)

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: (4)

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: (3, also see **Formula, CNF**)

In CNF formulas a clause is a disjunction of literals such as the following: $(\bar{a} \vee b \vee c \vee d)$. A clause containing only negative (positive) literals is called a *negative clause* (alternatively, a *positive clause*). In this chapter a disjunction of literals is also written as a set such as this: $\{\bar{a}, b, c, d\}$. 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: $(a \wedge b \wedge c \rightarrow g)$. In this chapter, 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: (26, 27, 34, 105, 107, 120)

One of two vertices spanned by an edge. See **Graph**.

Formula, DNF: (33)

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 $C_1 \vee C_2 \vee \ldots C_m$ 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: (15, 18, 30, 1)

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 $C_1 \wedge C_2 \wedge \ldots C_m$ 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 chapter a disjunctive clause, sometimes called a clause when the context is clear, is regarded 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 chapter.

$$\{\{\bar{a}, b\}, \{a, c, d\}, \{c, \bar{d}, \bar{e}\}\}$$

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: (3, 17)

A CNF formula in which every clause has at most one positive literal. Satisfiability of Horn formulas can be determined in linear time [41, 63]. 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: (5)

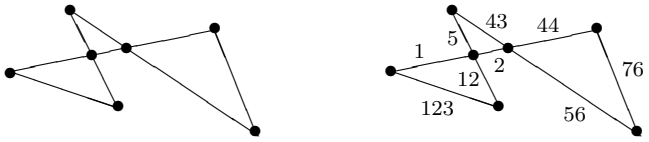
An unsatisfiable CNF formula such that removal of any clause makes it satisfiable.

Formula, Propositional or Boolean: (15, 24, 30, 32, 3)

A Boolean variable is a formula. If ψ is a formula, then (ψ) is a formula. If ψ is a formula then $\neg\psi$ is a formula. If ψ_1 and ψ_2 are formulas and $\mathcal{O}_b$ is a binary Boolean operator, then $\psi_1 \mathcal{O}_b \psi_2$ is a formula. In some contexts other than Logic Programming, we use $\bar{a}$ or $\overline{\psi}$ instead of $\neg a$ or $\neg\psi$ to denote negation of a variable or formula. Formulas evaluate to *true* or *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: (29)

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 *weighted graph*. Below is an example of a graph and a weighted graph.



Graph, Directed: (10)

A graph in which some orientation is given to each edge.

Graph, Directed Acyclic: (13, 9, 9)

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: (13, 9, 9)

A connected, directed graph with no cycles such that there is exactly one vertex, known as the *root*, whose incident edges are all oriented away from it.

Literal: (1, see also **Formula, CNF**)

A Boolean variable or the negation of a Boolean variable. In the context of a formula, a negated variable is called a *negative* literal and an unnegated variable is called a *positive* literal.

Logic Program, Normal: (3)

A formula consisting of implicational clauses. That is, clauses have the form $(a \wedge b \wedge c \rightarrow g)$ 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.

Model, Minimal Model: (4)

For the purposes of this chapter, 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**.

$\mathcal{NP}$-hard: (15)

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: (22, 1, 34)

A mapping from binary Boolean vectors to $\{0, 1\}$. Most frequently used binary operators are

Or	$\vee$:	$\{00 \mapsto 0;\ 10, 01, 11 \mapsto 1\}$
And	$\wedge$:	$\{00, 01, 10 \mapsto 0;\ 11 \mapsto 1\}$
Implies	$\rightarrow$:	$\{01 \mapsto 0;\ 00, 10, 11 \mapsto 1\}$
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 $(v_l \wedge v_r)$, the operator 0111 has the same functionality as $(v_l \vee v_r)$, 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 ψ is a formula.

Operator, Temporal: (22, 22)

An operator used in temporal logics. Basic operators include *henceforth* $(\Box\psi_1)$, *eventually* $(\Diamond\psi_1)$, *next* $(\circ\psi_1)$, and *until* $(\psi_1\ \mathcal{U}\ \psi_2)$.

Satisfied Clause: (see **Formula, CNF**)

Satisfiability (SAT): (15, 18, 27)

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: (22, 22, 24)

A particular assignment of values to the parameters of a system.

Subgraph: (29, 30, 32)

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: (32)

A clause consisting of one literal. See also **Clause** in the glossary.

Variable, Propositional or Boolean: (19, 30, 32)

An object taking one of two values $\{0, 1\}$.

Vertex: (see **Graph**)

Vertex Weighted Satisfiability: (16)

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: (27, 32)

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.