SAT Basics

Boolean functions

Let $B = \{0, 1\}$ be the set of values 0 and 1. Call a variable that takes values only from B a binary (or Boolean) variable. A Boolean function $f : B \times B \times \ldots \times B \rightarrow B$ is a mapping from binary values to B.

Binary Operators and the Unary Operator $\neg$

Boolean functions may be expressed in terms of formulas that are built upon binary operators and the unary operator $\neg$. The reason for this is to be able to manipulate a given formula algebraically so as to discover some properties about the corresponding function. If v is a Boolean variable, the value of the expression $\neg v$ is opposite the value of v: that is, if v has value 1 then $\neg v$ has value 0 and if v has value 0, $\neg v$ has value 1. Note that $\neg\neg v = v$. The following is a list of the five non-trivial binary operators. Symbols denoting the operators are shown between their operands in the middle column. English language descriptions of the operators are shown in the right column. Each 4 bit binary pattern on the left represents the values of the expression of the middle column in the same row for all possible input combinations: from left bit to right bit, the bit is the expression value if $v_1 = 0, v_2 = 0$; $v_1 = 1, v_2 = 0$; $v_1 = 0, v_2 = 1$; and $v_1 = 1, v_2 = 1$, respectively.

$\mathcal{O}_x$	**Syntax**	**Operator**
0001	$v_1 \wedge v_2$	logical and
1101	$v_2 \rightarrow v_1$	logical implies
1011	$v_1 \rightarrow v_2$	logical implies
0111	$v_1 \vee v_2$	logical or
1001	$v_1 \equiv v_2$	logical equivalence
0110	$v_1 \oplus v_2$	exclusive-or

Formulas

Formulas are defined recursively. Non-logical symbols '(' and ')' are added to the syntax of a formula to allow its unambiguous evaluation. Any variable v or its complement $\neg v$ is an expression. If ϕ is an expression then $(\neg\phi)$ is an expression. If ϕ_1 and ϕ_2 are expressions and $\mathcal{O}$ is one of the binary operators, then $(\phi_1\ \mathcal{O}\ \phi_2)$ is an expression. A formula is an expression. Say symbol v represents a formula variable. Any occurrence of symbol v at a particular point in a formula is called a positive literal and any occurrence of the string of symbols $\neg v$ is called a negative literal. Below, l will be used to denote a literal that could be either positive or negative.

Given an assignment of values to the variables of a formula ϕ, the value of ϕ is determined by repeatedly replacing an innermost non-evaluated expression either with its value as given by the table above, where variables v_1 and v_2 are values of evaluated expressions, or by negating the value of an evaluated expression.

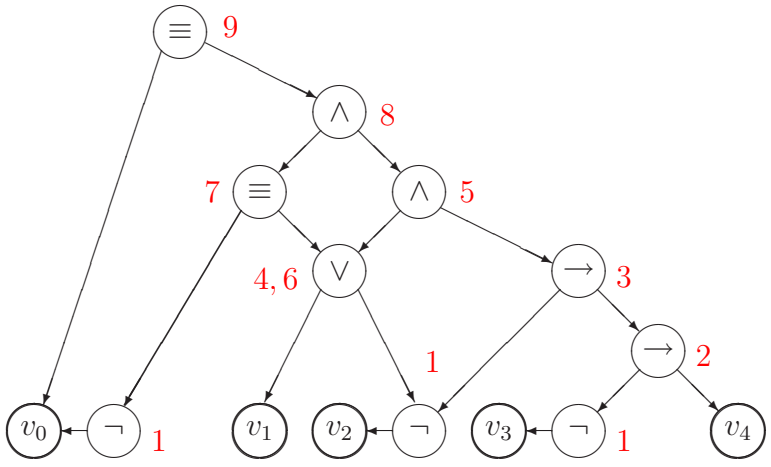
For example, the following is a formula:

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

Under the assignment $v_1 = 1$, $v_2 = 1$, $v_3 = 0$, $v_4 = 0$, $v_5 = 1$ evaluation proceeds step by step as follows where comments are in red and expressions to be evaluated on the next step are in green:

$(1 \equiv ((\neg 1 \equiv (1 \vee \neg 0)) \wedge ((1 \vee \neg 0) \wedge (\neg 0 \rightarrow (\neg 0 \rightarrow 1)))))$ apply values to variables
$(1 \equiv ((0 \equiv (1 \vee 1)) \wedge ((1 \vee 1) \wedge (1 \rightarrow (1 \rightarrow 1)))))$ all negations are innermost
$(1 \equiv ((0 \equiv (1 \vee 1)) \wedge ((1 \vee 1) \wedge (1 \rightarrow 1))))$ $(1 \rightarrow 1)$ has value 1
$(1 \equiv ((0 \equiv (1 \vee 1)) \wedge ((1 \vee 1) \wedge 1)))$ $(1 \rightarrow 1)$ has value 1
$(1 \equiv ((0 \equiv (1 \vee 1)) \wedge (1 \wedge 1)))$ $(1 \vee 1)$ has value 1
$(1 \equiv ((0 \equiv (1 \vee 1)) \wedge 1))$ $(1 \wedge 1)$ has value 1
$(1 \equiv ((0 \equiv 1) \wedge 1))$ $(1 \vee 1)$ has value 1
$(1 \equiv (0 \wedge 1))$ $(0 \equiv 1)$ has value 0
$(1 \equiv 0)$ $(0 \wedge 1)$ has value 0
0 $(1 \equiv 0)$ has value 0: the formula has this value

Evaluation is best visualized by a DAG with operators labeling internal nodes, variables labeling leaves, node degree no greater than 2, and a single root. A non-leaf node represents an expression: each of its edges is incident to a node that represents one of its operands and the operator is given by the node's label. Evaluation proceeds bottom-up through the DAG. The DAG for the above example looks like this:



The numbers in red indicate the order in which expressions were evaluated above.

Some parentheses may be removed without creating ambiguity because most binary operators

are associative. For example

$$
\begin{array}{l}
(v_1 \vee (v_2 \vee (v_3 \vee v_4))) \\
(v_1 \vee ((v_2 \vee v_3) \vee v_4)) \\
(((v_1 \vee v_2) \vee v_3) \vee v_4) \\
(v_1 \vee v_2 \vee v_3 \vee v_4)
\end{array}
$$

are all equivalent.

Shannon Expansion

A Boolean function f of n arguments may be expressed as follows

$$
\begin{array}{rcl}
f(v_1, v_2, \ldots, v_n) & = & (v_i \wedge f(v_1, \ldots, v_{i-1}, 1, v_{i+1}, \ldots, v_n)) \vee \\
& & (\neg v_i \wedge f(v_1, \ldots, v_{i-1}, 0, v_{i+1}, \ldots, v_n))
\end{array}
$$

or as follows

$$
\begin{array}{rcl}
f(v_1, v_2, \ldots, v_n) & = & (\neg v_i \vee f(v_1, \ldots, v_{i-1}, 1, v_{i+1}, \ldots, v_n)) \wedge \\
& & (v_i \vee f(x_1, \ldots, v_{i-1}, 0, v_{i+1}, \ldots, v_n))
\end{array}
$$

for any v_i. It follows by repeated application of the first of the rules above that any Boolean function f may be expressed as follows

$$
\begin{array}{rcl}
f(v_1, v_2, \ldots, v_n) & = & (f(0, 0, \ldots, 0) \wedge \neg v_1 \wedge \neg v_2 \wedge \ldots \wedge \neg v_n) \ \vee \\
& & (f(1, 0, \ldots, 0) \wedge v_1 \wedge \neg v_2 \wedge \ldots \wedge \neg v_n) \ \vee \\
& & (f(0, 1, \ldots, 0) \wedge \neg v_1 \wedge v_2 \wedge \ldots \wedge \neg v_n) \ \vee \\
& & \quad \ldots \\
& & (f(1, 1, \ldots, 1) \wedge v_1 \wedge v_2 \wedge \ldots \wedge v_n)
\end{array}
$$

This is a canonical form known as Disjunctive Normal Form (DNF). By repeated application of the second rule one obtains the following

$$
\begin{array}{rcl}
f(v_1, v_2, \ldots, v_n) & = & (f(0, 0, \ldots, 0) \vee v_1 \vee v_2 \vee \ldots \vee v_n) \ \wedge \\
& & (f(1, 0, \ldots, 0) \vee \neg v_1 \vee v_2 \vee \ldots \vee v_n) \ \wedge \\
& & \quad \ldots \\
& & (f(1, 1, \ldots, 1) \vee \neg x_1 \vee \neg x_2 \vee \ldots \vee \neg x_n)
\end{array}
$$

which is known as Conjunctive Normal Form (CNF).

Consensus, Resolution, and Subsumption

A Boolean function that has been Shannon-expanded to an equivalent CNF or DNF formula would generally be too large to work with without some rules that allow reductions to

smaller, equivalent formulas. The most important of these are resolution for CNF formulas, consensus for DNF formulas, and subsumption. Note that a CNF formula ϕ has this form:

$$\phi = \bigwedge_{i=1}^{m} (\bigvee_{j=1}^{k_i} l_{i,j})$$

The expression in parentheses is called a clause. In the above, there are m clauses in ϕ, the number of literals in the i^{th} clause is k_i, and the j^{th} literal of the i^{th} clause is $l_{i,j}$. The k_i may not be equal to the number of variables because, as it is shown below, a CNF formula may be reduced to a smaller, equivalent one.

Let $c_1 = \bigvee_{j=1}^{k_1} l_{1,j}$ and $c_2 = \bigvee_{j=1}^{k_2} l_{2,j}$ be two clauses in ϕ and suppose that for exactly one pair of values for p, q where $1 \leq p \leq k_1$, and $1 \leq q \leq k_2$, $l_{1,p} = \neg l_{2,q}$. Then a new (inferred) clause

$$c' = l_{1,1} \vee \ldots \vee l_{1,p-1} \vee l_{1,p+1} \vee \ldots \vee l_{1,k_1} \vee l_{2,1} \vee \ldots \vee l_{2,q-1} \vee l_{2,q+1} \vee \ldots \vee l_{2,k_2}$$

may be added to ϕ without changing the functionality of ϕ. That is, c' contains all the literals of c_1 and c_2 except for the two literals that were complements of each other. Clause c' is called a resolvent and the operation of adding c' to ϕ is called a resolution step. The variable whose positive literal and negative literal are missing from the resolvent is called the pivot.

Let c_1 and c_2 be clauses in ϕ and suppose c_1 has all the literals that c_2 has. Then c_2 subsumes c_1 and c_1 can be removed from ϕ without affecting the functionality of ϕ.

The following example illustrates how resolution and subsumption can reduce the size of a formula:

	ϕ
before resolution:	$(v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee v_2 \vee v_3) \wedge \ldots$
after resolution:	$(v_1 \vee v_2 \vee v_3) \wedge (\neg v_1 \vee v_2 \vee v_3) \wedge (v_2 \vee v_3) \wedge \ldots$
after resolution & subsumption:	$(v_2 \vee v_3) \wedge \ldots$

The counterpart to resolution in DNF formulas is consensus. There is also a counterpart to subsumption in DNF. These will not be discussed.

The Satisfiability problem (SAT)

SAT is one of the most important problems in computer science. It appears in the design and testing of computer circuits, in theorem proving, in artificial intelligence, in determining a safe shutdown path for complicated, state-of-the-art robotic machine tools, and many other domains of computer science, operations research, and logic. For more information on one of these applications see `http://gauss.ececs.uc.edu/SAT/articles/FAIA185-0457.pdf`.

An instance of SAT is a CNF formula ϕ. The Satisfiability problem is to determine whether there exists an assignment of values to the variables of ϕ which causes ϕ to evaluate to 1.

For a history of SAT see `http://gauss.ececs.uc.edu/SAT/articles/FAIA185-0003.pdf`.

Set of Sets Representation of CNF Formulas

To make algorithms for SAT easier to write and comprehend, a set of sets representation for CNF formulas is used. In this representation a clause is just a set of literals and a formula is a set of clauses. For example, the following is treated as a formula of three clauses, each containing three literals:

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

If c_1 and c_2 are clauses that a resolution step may be applied to with variable v as the pivot, the resolvent c' is given by:

$$c' = \{l \in c_1 \cup c_2 : l \neq v \text{ and } l \neq \neg v\}$$

Davis-Putnam Procedure (DPP)

DPP determines the satisfiability of a CNF formula ϕ by eliminating one variable at a time using resolution.

DPP (ϕ)
/ Input ϕ is a set of sets of literals representing a CNF formula */*
/ Output is either "unsatisfiable" or "satisfiable" */*
If $\phi = \emptyset$ return "satisfiable"
If there is a clause containing a single literal l do the following:
Return **DPP**($\{c \setminus \{\neg l\} : c \in \phi \text{ and } l \notin c\}$).
If there is a variable v which occurs in ϕ only as either a positive literal
or as a negative literal do this:
Return **DPP**($\{c \in \phi : v \notin c \text{ and } \neg v \notin c\}$).
Choose any variable v from ϕ.
Repeat the following as long as there is a pair of clauses $c_1, c_2 \in \phi$
for which a resolution step is possible with v as the pivot:
$\phi_1 \leftarrow \{\{l \in c_1 \cup c_2 : l \neq v \text{ and } l \neq \neg v\}\}$.
If $\phi_1 = \emptyset$ return "unsatisfiable".
$\phi \leftarrow \phi \cup \phi_1$.
Repeat the following while there is a clause $c \in \phi$ such that
$v \in c$ or $\neg v \in c$:
$\phi \leftarrow \phi \setminus \{c\}$.
Return **DPP**(ϕ).

The 2^{nd} and 3^{rd} lines of the algorithm are known as the unit clause rule. If ϕ contains clause $\{l\}$ then all assignments satifying ϕ must have $l = 1$. Hence the question of satisfiability of ϕ can be reduced to the question of satisfiability of ϕ with $l = 1$. But then all clauses of ϕ containing l can be eliminated because they are satisfied by all assignments where $l = 1$ and

all occurrences of $\neg l$ in ϕ can be eliminated because they cannot contribute to satisfying the clauses they are in for any assignment where $l = 1$.

The 4^{th}, 5^{th}, and 6^{th} lines of the algorithm are known as the pure literal rule. If there is a variable that occurs in ϕ either purely as a positive literal or purely as a negative literal then if there is a satisfying assignment for ϕ, there must also be a satisfying assignment for ϕ where v is set to make its literals in ϕ have value 1. Then the satisfiability of ϕ may be reduced to the satisfiability of ϕ minus all clauses containing the pure literal.

Subsumption was not explicitly used in **DPP** but it could be added.

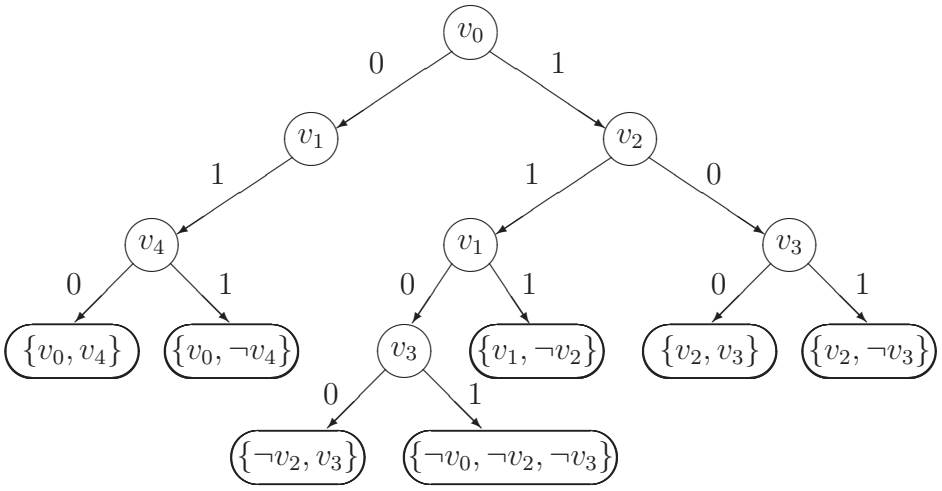
DPP was discovered in 1958. The problem with **DPP** then was that the amount of main memory available on computers of the time was quite small compared to the needs of the algorithm when applied to most interesting inputs. This was improved by a stack-based implementation that is described in the next section.

Davis-Putnam-Loveland-Logemann Procedure (DPLL)

The improvement to the implementation of **DPP** is in the way variables are eliminated. **DPLL** is a divide-and-conquer algorithm: it executes a search for a satisfying assignment until either all clauses are eliminated or all literals of some clause are eliminated. At any point in the search only some variables have been assigned values. Such partial assignments are extended from a chosen variable v by exploring a subformula assuming $v = 1$ and, if necessary, by exploring a subformula assuming $v = 0$.

DPLL (ϕ)
/* *Input ϕ is a set of sets of literals representing a CNF formula* */
/* *Output is either "unsatisfiable" or "satisfiable"* */
If $\phi = \emptyset$ return "satisfiable"
If there is a clause containing a single literal l do the following:
Return **DPLL**($\{c \setminus \{\neg l\} : c \in \phi$ and $l \notin c\}$).
If there is a variable v which occurs in ϕ only as either a positive literal
or as a negative literal do this:
Return **DPLL**($\{c \in \phi : v \notin c$ and $\neg v \notin c\}$).
Choose any variable v from ϕ.
$\phi_0 \leftarrow \{c \setminus \{v\} : c \in \phi$ and $\neg v \notin c\}$
If **DPLL**(ϕ_0) == "satisfiable" Return "satisfiable".
$\phi_1 \leftarrow \{c \setminus \{\neg v\} : c \in \phi$ and $v \notin c\}$
If **DPLL**(ϕ_1) == "satisfiable" Return "satisfiable".
Return "unsatisfiable".

In the variation above a satisfying assignment is not returned. This is easy to add and has been avoided to keep the algorithm as simple as possible. The algorithm may also be improved by adding a search heuristic to determine a "best" variable to choose for elimination and a "best" order in which to search subformulas created by assigning that chosen



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

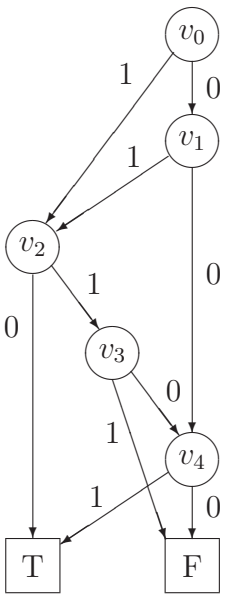
Figure 1: *A DPLL refutation tree for the above CNF formula.*

variable its two possible values. Finally, the collection of improvements known as Conflict Driven Clause Learning (CDCL) has enabled this form of search to solve problems that had been considered intractable not long ago. For more information on CDCL see `http://gauss.ececs.uc.edu/SAT/articles/FAIA185-0131.pdf`.

As a divide-and-conquer algorithm, the execution of **DPLL** on a particular input may be traced graphically as a search tree. An example is shown in Figure 1. Each non-leaf of the tree is labeled with a variable name and may have one or two descendents. The edges connecting those descendents are labeled 0 or 1. A non-leaf node with two descendents corresponds to a variable that was chosen in the 7^{th} line of the algorithm. A non-leaf node with one descendent corresponds to an application of either the unit clause rule or the pure literal rule. A path from the root to a leaf indicates a partial truth assignment to the variables of the input formula. Thus, the partial assignment corresponding to the extreme left path in the figure is $v_0 = v_4 = 0$ and $v_1 = 1$. Each leaf is labeled with a clause that is falsified by the assignment of the path to the leaf - the set-of-sets representation is used to save space. The example of the figure is unsatisfiable, hence there is a clause labeling every leaf. If a formula is satisfiable, one of the paths would lead to a leaf that was unlabeled and the path to that leaf would be a satisfying assignment.

Binary Decision Diagrams

An alternative representation of Boolean functions is the Reduced Ordered Binary Decision Diagram (BDD). An example is shown in Figure 2. A BDD is a rooted, binary directed acyclic graph with two leaves labeled `T` and `F`. Internal nodes are labeled with variable names as in DPLL search trees such as the one depicted in Figure 1. Out of each non-leaf, one edge is labeled 0 and one edge is labeled 1 with the same meaning as in the DPLL



v_0	v_1	v_2	v_3	v_4	f
0	0	0	0	0	0
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	1	1
0	0	1	0	0	0
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	1	1
0	1	1	1	0	0
0	1	1	1	1	0
1	0	0	0	0	1
1	0	0	0	1	1
1	0	0	1	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	0	1	0	1	1
1	0	1	1	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	0	0	1	1
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	0	1	1
1	1	1	1	0	0
1	1	1	1	1	0

$$
\begin{aligned}
&(v_0 \vee v_1 \vee \neg v_4) \wedge \\
&(v_0 \vee \neg v_1 \vee \neg v_2 \vee v_3 \vee \neg v_4) \wedge \\
&(v_0 \vee \neg v_1 \vee \neg v_2 \vee \neg v_3) \wedge \\
&(\neg v_0 \vee \neg v_2 \vee v_3 \vee \neg v_4) \wedge \\
&(\neg v_1 \vee \neg v_2 \vee \neg v_3)
\end{aligned}
$$

Figure 2: A BDD on the left that compactly represents the Boolean function whose truth table is displayed in the center. An equivalent CNF representation is obtained directly from the paths connecting the root to the `F` leaf and is shown on the right.

search tree but in BDDs there is no node of degree 1. Unlike DPLL search trees there is an order on the variables of the BDD and variables on any path from root to leave obey it. A single BDD compactly represents a Boolean function and even a DPLL-like search tree except that in BDDs equivalent subfunctions may share structure. This is one feature that has given BDDs an edge over other representations in some applications. Other performance enhancing features are primarily the BDD operations such as existential quantification and greatest common co-factor that do not have a direct counterpart in DPLL-like algorithms. For more information on BDDs see `http://gauss.ececs.uc.edu/SAT/articles/bdd97.pdf`.

It is straightforward to obtain an equivalent CNF representation of the Boolean function that is represented by a BDD by traversing all paths from root to the `F` leaf: each path represents one clause whose literals are the reverse of the assignment indicated by the path. An example is shown in Figure 2.

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 this representation is that algorithms for solving such systems incorporate operations that 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.

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:

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

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

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

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

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

3. A multi-linear equation may be multiplied by a term and the resulting equation may be reduced to a multi-linear equation by rule 2. above. Thus, $v_3 v_4 (v_1 + v_3 = 0)$ becomes $v_1 v_3 v_4 + v_3 v_4 = 0$.

4. Two equations may be added to produce an equation that may be reduced by rule 1. above to a multi-linear equation. 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. 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.

Now for some examples. 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

$$v_1 v_2 v_3 + v_1 v_2 + v_1 v_3 + v_2 v_3 + v_1 + v_2 + v_3 + 1 = 0.$$

This may be verified from the truth table for the clause. 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

$$v_1v_2v_3 + v_1v_2 + v_1v_3 + v_1 = 0.$$

Exclusive-or and equivalence functions yield linear equations. Thus, $(v_1 \oplus v_2 \oplus v_3 \oplus v_4)$ is represented by

$$v_1 + v_2 + v_3 + v_4 + 1 = 0.$$

An equation representing a BDD 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 in Figure 2 is

$$v_0(1+v_2) + v_0v_2(1+v_3)v_4 + (1+v_0)v_1(1+v_2) + (1+v_0)v_1v_2(1+v_3)v_4 + (1+v_0)(1+v_1)v_4 + 1 = 0$$

Addition of equations and the Gaussian-elimination nature of algebraic proofs is illustrated by showing steps that solve the following simple formula:

$$(v_1 \vee \neg v_2) \wedge (v_2 \vee \neg v_3) \wedge (v_3 \vee \neg v_1) \tag{1}$$

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

$$\begin{array}{llll}
v_1v_2 + v_2 = 0 & (1) \\
v_2v_3 + v_3 = 0 & (2) \\
v_1v_3 + v_1 = 0 & (3) \\
\hline
v_1v_2v_3 + v_2v_3 = 0 & (4) \Leftarrow v_3 \cdot (1) \\
v_1v_2v_3 + v_3 = 0 & (5) \Leftarrow (4) + (2) \\
v_1v_2v_3 + v_1v_3 = 0 & (6) \Leftarrow v_1 \cdot (2) \\
v_1v_2v_3 + v_1 = 0 & (7) \Leftarrow (6) + (3) \\
v_1v_2v_3 + v_1v_2 = 0 & (8) \Leftarrow v_2 \cdot (3) \\
v_1v_2v_3 + v_2 = 0 & (9) \Leftarrow (8) + (1) \\
v_1 + v_2 = 0 & (10) \Leftarrow (9) + (7) \\
v_1 + v_3 = 0 & (11) \Leftarrow (5) + (7)
\end{array}$$

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 (1)

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

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 an equation is

$$degree(e) = \max\{degree(t) : t \text{ is a non-zero term in } e\}.$$

An example, adapted from Clegg et.al. (1996) is given below. In the algorithm terms are re-indexed in a particular way. 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.

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 */
    $B \leftarrow \emptyset$.
    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 $reduce(v \cdot e)$ has not been in ψ:
                            Append $\psi \leftarrow reduce(v \cdot e)$. /* Rule 3. */
    Output "satisfiable".

The reason for considering such solvers is that

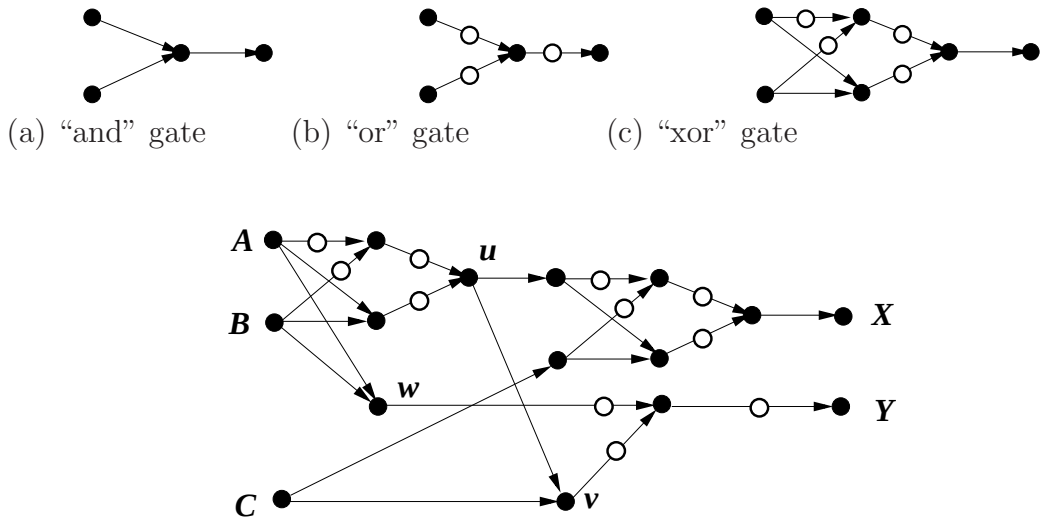
1. The number of derivations used by the algebraic solver above is within a polynomial factor of the minimum number possible.

2. The minimum number of derivations used by the algebraic solver cannot be much greater than, and may sometimes be far less than the minimum number needed by resolution.

And-Inverter Graphs

An And-Inverter Graph (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 3.

AIGs are effective in dealing with the Combinational Equivalence Checking (CEC) problem which is to verify that two given *combinational circuit* implementations are functionally



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

(d) one-bit "adder" circuit

Figure 3: Examples of AIGs. Edges with white circles are negated. Edges without a white circle are not negated.

equivalent. CEC can be solved very efficiently, in general, by incrementally building a single-output AIG, representing the mitre 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 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 and avoids testing all possible input-output combinations.

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). Then a SAT solver is used 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.

Satisfiability Modulo Theories

There are many commonly occurring families of clauses that are known to cause difficulty for resolution-based solvers. Many of these families arise from the need to reason about bit vector operations or memory access and so on. Such reasoning is best left to tailored decision procedures - such procedures are able to solve a system of constraints in a particular theory (for example, bit vector logic) only. Typically, a system of constraints mixes terms that arise in several theories. A SAT solver may be used to tie all the theories together to provide a sound decision procedure for mixed theories. This is not an easy task - combining the results of more than one decision procedure may not be sound. For details about SMT solvers see

`http://jsat.ewi.tudelft.nl/content/volume3/JSAT3_9_Sebastiani.pdf`