4

Propositional Calculus: Resolution and BDDs

4.1 Resolution

One desirable property of a deductive system is that it should be easy to mechanize an efficient proof search. It is very difficult to search for a proof in a Hilbert system because there is no obvious connection between the formula and its proof. Proof search in the propositional calculus is easy and efficient with semantic tableaux and the equivalent (cut-free) Gentzen systems. However, as we shall see in the next chapter, the method of semantic tableaux in the predicate calculus becomes arbitrary and inefficient. The method of resolution, invented by J. A. Robinson in 1965, is frequently an efficient method for searching for a proof. In this section, we introduce resolution for the propositional calculus, though its advantages will not be apparent until it is extended to the predicate calculus.

**CNF and clausal form**

**Definition 4.1** A formula is in *conjunctive normal form (CNF)* iff it is a conjunction of disjunctions of literals.

**Example 4.2** The formula $(\neg p \lor q \lor r) \land (\neg q \lor r) \land (\neg r)$ is in CNF while $(\neg p \lor q \lor r) \land (p \land \neg q) \lor r) \land (\neg r)$ is not in CNF, because of the conjunction within the second disjunction. The formula $(\neg p \lor q \lor r) \land \neg (\neg q \lor r) \land (\neg r)$ is not in CNF because the second disjunction is negated.

**Theorem 4.3** Every formula in the propositional calculus can be transformed into an equivalent formula in CNF.

**Proof:** To convert to an equivalent formula in CNF perform the following steps, each of which preserves logical equivalence:

1. Eliminate all operators except for negation, conjunction and disjunction, using the equivalences in Figure 2.6.
2. Push all negations inward using De Morgan’s laws:

\[ \neg(A \land B) \equiv (\neg A \lor \neg B) \]
\[ \neg(A \lor B) \equiv (\neg A \land \neg B). \]

3. Eliminate double negations using the equivalence \( \neg \neg A \equiv A \).

4. The formula now consists of disjunctions and conjunctions of literals. Use the distributive laws:

\[ A \lor (B \land C) \equiv (A \lor B) \land (A \lor C) \]
\[ (A \land B) \lor C \equiv (A \lor C) \land (B \lor C) \]

to eliminate conjunctions within disjunctions.

Example 4.4 The following sequence of formulas shows the four steps applied to the formula \((\neg p \rightarrow \neg q) \rightarrow (p \rightarrow q)\):

\[
(\neg p \rightarrow \neg q) \rightarrow (p \rightarrow q) \\
\equiv \neg(\neg \neg p \lor \neg q) \lor (\neg p \lor q) \\
\equiv (\neg p \land \neg q) \lor (\neg p \lor q) \\
\equiv (\neg p \lor \neg p \lor q) \land (q \lor \neg p \lor q).
\]

Definition 4.5 A clause is a set of literals which is considered to be an implicit disjunction. A unit clause is a clause consisting of exactly one literal. A formula in clausal form is a set of clauses which is considered to be an implicit conjunction.

Corollary 4.6 Every formula in the propositional calculus can be transformed into an equivalent formula in clausal form.

Proof: In the transformation from a formula to a set of sets of literals, identical literals in a clause and identical clauses in the set will be removed. By idempotence, \( A \equiv A \land A \) and \( A \equiv A \lor A \), logical equivalence is preserved.

Example 4.7 The CNF formula

\[
(\neg q \lor \neg p \lor q) \land (p \lor \neg p \lor q \lor p \lor \neg p)
\]
is equivalent to the clausal form \( \{\neg q, \neg p, q\}, \{p, \neg p, q\} \).
4.1 Resolution

Notation

We sometimes use an abbreviated notation, removing the set delimiters \{ and \} from a clause and denoting negation by a bar over the propositional letter \( \bar{p} \). The formula above is written \( \{ \bar{q} \bar{p}, p \bar{q} \} \) in the abbreviated notation.

The following symbols will be used: \( S \) for a set of clauses (that is, a formula in clausal form), \( C \) for a clause and \( l \) for a literal. The symbols will be subscripted and primed as necessary. If \( l \) is a literal, \( l_c \) is its complement. This means that if \( l = p \) then \( l_c = \bar{p} \) and if \( l = \bar{p} \) then \( l_c = p \). The concept of an assignment is generalized so that it can be defined on literals in the obvious way:

\[ v(l) = T \] means \( v(p) = T \) if \( l = p \) and \( v(p) = F \) if \( l = \bar{p} \).

Properties of clausal form

Definition 4.8 Let \( S, S' \) be sets of clauses. \( S \approx S' \) denotes that \( S \) is satisfiable if and only if \( S' \) is satisfiable.

In the following sequence of lemmas, we show various ways a formula can be transformed without changing its satisfiability.

Lemma 4.9 Suppose that a literal \( l \) appears in (some clause of) \( S \), but \( l_c \) does not appear in (any clause of) \( S \). Let \( S' \) be obtained from \( S \) by deleting every clause containing \( l \). Then \( S \approx S' \).

Proof: If \( S' \) is satisfiable, there is a model \( v \) for \( S' \) such that \( v(C') = T \) for every \( C' \in S' \). (Recall that a set of clauses is an implicit conjunction so that all clauses must be true for \( v \) to be a model.) Extend \( v \) by defining \( v(l) = T \). Then \( v(C) = T \) for every \( C \in S \setminus S' \). (Recall that a clause is an implicit disjunction so that it is sufficient if one literal is true for a clause to evaluate to true.) Conversely, if \( S \) is satisfiable, \( S' \) is obviously satisfiable since \( S' \subseteq S \).

Example 4.10 Let \( S = \{pq\bar{r}, p\bar{q} \} \) and \( S' = \{p\bar{q}, \bar{p}q\} \), where \( S' \) is obtained from \( S \) by deleting the clause \( pq\bar{r} \) containing \( \bar{r} \) since \( \bar{r} = r \) does not appear in \( S \). \( S' \) is satisfied by the interpretation \( v(p) = F \), \( v(q) = F \), which can be extended to an interpretation of \( S \) by defining \( v(r) = T \) so that \( v(pq\bar{r}) = T \). Note that \( S \) is not logically equivalent to \( S' \), since under the interpretation \( v(p) = F \), \( v(q) = F \), \( v(r) = T \), \( S \) evaluates to \( F \) and \( S' \) evaluates to \( T \).

Lemma 4.11 Let \( C = \{l\} \in S \) be a unit clause and let \( S' \) be obtained from \( S \) by deleting every clause containing \( l \) and by deleting \( F \) from every (remaining) clause. Then \( S \approx S' \).
Propositional Calculus: Resolution and BDDs

Proof: Let \( v \) be a model for \( S \). We will prove that \( v \) is also a model for \( S' \) (ignoring the assignment to \( l \)), by showing that for every \( C'_i \) such that \( C'_i = C_i - \{ l \} \), \( v(C'_i) = T \).

\( v(C) = v(l) = T \) since \( v \) is a model for \( S \), so \( v(l') = F \). Since \( v \) is a model for \( S \), \( v(C_i) = T \) and there must be some other literal \( l_i \in C_i \) such that \( v(l_i) = T \). Therefore, \( v(C'_i) = T \). The proof of the converse is similar to the proof of the previous lemma and is left as an exercise.

Example 4.12 Let \( S = \{ r, pq, p \bar{q}, \bar{r} \} \) and \( S' = \{ pq, pq, \bar{q} \} \), where \( \{ r \} \) is the unit clause deleted. \( v(r) = T \) in any model for \( S \), so if \( v(pq) = T \) then either \( v(p) = T \) or \( v(q) = T \). Therefore, \( v(pq) = T \) in \( S' \).

Lemma 4.13 Suppose that both \( l \in C \) and \( l \in C \) for some \( C \in S \), and let \( S' = S - \{ C \} \).

Then \( S \equiv S' \).

Proof: Any interpretation satisfies \( C \).

We will assume that all such valid clauses are deleted from a formula in clausal form.

Definition 4.14 If \( C_1 \subseteq C_2 \), \( C_1 \) subsumes \( C_2 \) and \( C_2 \) is subsumed by \( C_1 \).

Lemma 4.15 Let \( C_1, C_2 \in S \) be clauses such that \( C_1 \) subsumes \( C_2 \), and let \( S' = S - \{ C_2 \} \). Then \( S \equiv S' \), that is, the larger clause can be deleted.

Proof: Since a clause is an implicit disjunction, any interpretation that satisfies \( C_1 \) must also satisfy \( C_2 \).

The concept of subsumption may initially seem non-intuitive. The set of interpretations that satisfy the small clause is contained in the set that satisfies the larger clause, so that once we have found an interpretation satisfying the smaller clause it automatically satisfies the larger one.

Example 4.16 Let \( S = \{ pq, pq, p \bar{q}, \bar{p} \} \) and \( S' = \{ pq, pq, \bar{p} \} \), where \( pq \) is subsumed by \( pq \). If \( S' \) is satisfiable then \( v(pq) = T \), so obviously \( v(pq) = T \).

Definition 4.17 The empty clause is denoted by \( \square \). The empty set of clauses is denoted by \( \emptyset \).

Lemma 4.18 \( \emptyset \) is valid. \( \square \) is unsatisfiable.

Proof: The proof uses reasoning about vacuous sets and may be a bit hard to follow. A set of clauses is valid iff every clause in the set is true under every interpretation. But there are no clauses in \( \emptyset \) that need be true, so \( \emptyset \) is valid. A clause is satisfiable iff there is some interpretation under which at least one literal in the clause is true. Since
there are no literals in □, under any interpretation there are no literals which will be true, so □ is unsatisfiable.

Alternatively, we can use the previous lemmas to give a proof. Consider the set of clauses \{ \{p\}\}. By Lemma 4.9, \{ \{p\}\} \approx \emptyset. Since \{ \{p\}\} is satisfiable, so is \emptyset, say by some interpretation \nu. But by Theorem 2.10, \nu(\emptyset) = T for every interpretation \nu', so \emptyset is valid.

\{ \{p\}, \{\neg p}\}\} is the clausal form of the unsatisfiable formula \p ∧ \neg \p. If we apply Lemma 4.11, the first clause \{ \{p\}\} is deleted from the formula and the literal \pc = \neg \p is deleted from the second clause. Then \{ \}\} \approx \{ \{p\}, \{\neg p\}\}\}, so the set \{ \}\} = \{\}\} is unsatisfiable; since \emptyset is the only clause in the set, it must be unsatisfiable.

**Definition 4.19** Let \(S\) be a set of clauses and \(U\) a set of propositional letters. \(R_U(S)\), the renaming of \(S\) by \(U\), is obtained from \(S\) by replacing each literal \(l\) whose propositional letter is in \(U\) by \(lc = \neg \p\).

**Lemma 4.20** \(S \approx R_U(S)\).

**Proof:** Let \(\nu\) be a model for \(S\). Define an interpretation \(\nu'\) for \(R_U(S)\) by:

\[
\nu'(p) = \nu(\bar{p}), \quad \text{if } p \in U \\
\nu'(p) = \nu(p), \quad \text{if } p \notin U.
\]

Let \(C \in S\) and \(C' = R_U(\{C\})\). Since \(\nu\) is a model for \(S\), \(\nu(C) = T\) and \(\nu(l) = T\) for some \(l \in C\). If the letter \(p\) of \(l\) is not in \(U\) then \(l \in C'\) so \(\nu'(l) = \nu(l) = T\) and \(\nu'(C') = T\). If \(p \in U\) then \(l \in C'\) so \(\nu'(l) = \nu(l) = T\) and \(\nu'(C') = T\). The converse is similar.

**Example 4.21** The set of clauses \(S = \{pqr, \bar{p}q, \bar{q}r, r\}\) is satisfied by the interpretation \(\nu(p) = F, \nu(q) = F, \nu(r) = T\). The renaming \(R_{\{p,q\}}(S) = \{\bar{p}q, r, \bar{q}r\}\) is satisfied by \(\nu(p) = T, \nu(q) = T, \nu(r) = T\).

**The resolution rule**

Resolution is a refutation procedure which is used to show that a formula in clausal form is unsatisfiable. The resolution procedure consists of a sequence of applications of the resolution rule to a set of clauses. The rule maintains satisfiability: if a set of clauses is satisfiable, so is the set of clauses produced by an application of the rule. Therefore, if the unsatisfiable empty clause is ever obtained, the original set of clauses must have been unsatisfiable.

**Rule 4.22 (Resolution rule)** Let \(C_1, C_2\) be clauses such that \(l \in C_1, \bar{l} \in C_2\). The clauses \(C_1, C_2\) are said to be clashing clauses and to clash on the complementary
literals \( l, l' \). C, the \textit{resolvent} of \( C_1 \) and \( C_2 \), is the clause:

\[
\text{Res}(C_1, C_2) = (C_1 - \{l\}) \cup (C_2 - \{l'\}).
\]

\( C_1 \) and \( C_2 \) are the \textit{parent clauses} of \( C \).

\textbf{Example 4.23} The pair of clauses \( C_1 = ab\bar{c} \) and \( C_2 = bc\bar{e} \) clash on the pair \( c, \bar{c} \) and the resolvent is \( C = (ab\bar{c} - \{\bar{c}\}) \cup (bc\bar{e} - \{c\}) = ab \cup b\bar{e} = ab\bar{e} \).

Recall that a clause is a set and duplicate literals are removed in the union.

\textbf{Theorem 4.24} The resolvent \( C \) is satisfiable if and only if the parent clauses \( C_1 \) and \( C_2 \) are (mutually) satisfiable.

\textbf{Proof:} Let \( C_1 \) and \( C_2 \) both be satisfiable under interpretation \( v \). Since \( l, l' \) are complementary, either \( v(l) = T \) or \( v(l') = T \). If \( v(l) = T \), then \( v(l') = F \) so \( C_2 \) is satisfied only if \( v(l') = T \) for some literal \( l' \in C_2 \), \( l' \neq l' \). By construction in the resolution rule, \( l' \in C \), so \( v \) is also a model for \( C \). A symmetric argument holds if \( v(l) = T \).

Conversely, if \( v \) is an interpretation which satisfies \( C \), \( v(l') = T \) for at least one literal \( l' \in C \). By the resolution rule, \( l' \in C_1 \) or \( l' \in C_2 \) (or both). Suppose \( l' \in C_1 \), then \( v(C_1) = T \). Since neither \( l \in C \) nor \( l' \in C \), \( v \) is not defined on \( l \) and we can extend \( v \) to an interpretation \( v' \) by defining \( v'(l') = T \). Then \( v'(C_2) = T \) and \( v'(C_1) = v(C_1) = T \) because \( v' \) is an extension of \( v \). A symmetric argument holds if \( l' \in C_2 \).

\textbf{Algorithm 4.25} (Resolution procedure)

\textbf{Input:} A set of clauses \( S \).

\textbf{Output:} \( S \) is satisfiable or unsatisfiable.

Let \( S \) be a set of clauses and define \( S_0 = S \). Assume that \( S_i \) has been constructed. Choose a pair of clashing clauses \( C_1, C_2 \in S_i \) that has not been chosen before. Let \( C \) be the clause \( \text{Res}(C_1, C_2) \) defined by the resolution rule and let \( S_{i+1} = S_i \cup \{C\} \). If \( C = \Box \), terminate the procedure—\( S \) is unsatisfiable. If \( S_{i+1} = S_i \) for all possible choices of clashing clauses, terminate the procedure—\( S \) is satisfiable.

\textbf{Example 4.26} Let \( S \) be the set of clauses \{1) \( p \), (2) \( \bar{p}q \), (3) \( \bar{r} \), (4) \( \bar{p}\bar{q}r \)\} where the clauses have been numbered for reference. Here is a resolution derivation of \( \Box \) from \( S \), where the justification for each line is the pair of the numbers of the parent clauses that have been resolved to give the resolvent clause in the line.

\begin{align*}
5. \quad \bar{p}\bar{q} & \quad 3, 4 \\
6. \quad \bar{p} & \quad 5, 2 \\
7. \quad \Box & \quad 6, 1
\end{align*}
It is easier to read a resolution derivation if it is presented as a tree (Figure 4.1), where the original clauses label leaves, and resolvents label interior nodes whose children are the clauses used in the resolution.

In the example, we have derived the unsatisfiable clause $\square$ so we can conclude that the set of clauses $S$ is unsatisfiable. We leave it to the reader to check that $S$ is the clausal form of $\neg A$ where $A$ is an instance of Axiom 2 of $H$: $(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$. Hence, $\neg A$ is unsatisfiable, that is $A$ is valid and we have used resolution to prove that this instance of Axiom 2 is valid.

**Definition 4.27** A derivation of $\square$ from $S$ is called a refutation of $S$.

**Soundness and completeness of resolution**

**Theorem 4.28** If the set of clauses labeling the leaves of a resolution tree is satisfiable then the clause at the root is satisfiable.

**Proof:** The proof is by induction using Theorem 4.24 and is left as an exercise.

Note that the converse to Theorem 4.28 is not necessarily true because we have no way of ensuring that the extensions made to $v$ on all branches are consistent. In the tree in Figure 4.2, the set of clauses on the leaves $\{r, p\bar{q}, \bar{r}, \bar{p}\bar{q}r\}$ is not satisfiable even though the clause on the root $p$ is satisfiable. In this case, we have made a poor choice of clauses to resolve since we could obtain $\square$ immediately by resolving $r$ and $\bar{r}$.

Since resolution is a refutation procedure, the soundness and completeness are expressed in terms of unsatisfiability, rather than validity.

**Corollary 4.29 (Soundness)** If the empty clause $\square$ is derived from a set of clauses by the resolution procedure, then the set of clauses is unsatisfiable.
Proof: Immediate from Theorem 4.28 and Lemma 4.18.

Theorem 4.30 (Completeness) If a set of clauses is unsatisfiable then the empty clause $\Box$ will be derived by the resolution procedure.

We have to prove that given an unsatisfiable set of clauses, the resolution procedure will eventually terminate producing $\Box$, rather than continuing indefinitely or terminating but failing to produce $\Box$. We defined the resolution procedure to be systematic in that a pair of clauses is never chosen more than once. Since there are only a finite number of distinct clauses on the finite set of propositional letters appearing in the set of clauses, the procedure terminates. Thus, we need only prove that when the procedure terminates, the empty clause is produced.

To prove this, we define a construct called a semantic tree (which must not be confused with a semantic tableau). A semantic tree is a data structure for assignments to the atomic propositions of a formula. If the formula is unsatisfiable, all assignments must falsify the formula. We will associate clauses that are created during a resolution refutation with nodes of the tree called failure nodes, which represent assignments that falsify clauses. Eventually the empty clause $\Box$ must be created and is associated with the root node that is the failure note representing all assignments.

Algorithm 4.31 (Construction of a semantic tree)
Input: A set of clauses $S$.
Output: A semantic tree $T$ for $S$ which is either open or closed.
Let $\{p_1, \ldots, p_n\}$ be the propositional letters appearing in $S$. Form the complete binary tree $T$ of depth $n$ and label the left-branching edges from a node of depth $i - 1$ by $p_i$ and the right-branching edges by $\bar{p}_i$. Each branch $b$ in $T$ defines an interpretation $v_b$ by $v_b(p_i) = T$ if $p_i$ labels the $i$'th edge in $b$, otherwise, $\bar{p}_i$ labels the $i$'th edge in $b$, and $v_b(p_i) = F$. A branch $b$ is closed if $v_b(S) = F$, otherwise $b$ is open. $T$ is closed if all branches are closed, otherwise $T$ is open.
Example 4.32 The semantic tree for $S = \{p, \bar{p}, q, \bar{q}r\}$ is shown in Figure 4.3 where the numbers on the nodes are explained later. The second branch from the left defines the interpretation $v(p) = v(q) = T$, $v(r) = F$, and $v(S) = F$ since the fourth clause $\bar{p}qr$ is false in this interpretation. We leave it to the reader to check that every branch in this tree is closed.

![Figure 4.3 Semantic tree](image)

Lemma 4.33 Let $S$ be a set of clauses and let $T$ a semantic tree for $S$. Every interpretation $v$ for $S$ corresponds to $v_b$ for some $b \in T$, and conversely, every $v_b$ is an interpretation for $S$.

Proof: By construction.

Theorem 4.34 The semantic tree $T$ for a set of clauses $S$ is closed if and only if the set $S$ is unsatisfiable.

Proof: Suppose that $T$ is closed and let $v$ be an arbitrary interpretation for $S$. $v$ is $v_b$ for some branch by Lemma 4.33. Since $T$ is closed, $v_b(S) = F$. But $v = v_b$ is arbitrary so $S$ is unsatisfiable. Conversely, let $S$ be an unsatisfiable set of clauses and let $T$ be a semantic tree for $S$. Let $b$ be an arbitrary branch. Then $v_b$ is an interpretation for $S$ by Lemma 4.33, and $v_b(S) = F$ since $S$ is unsatisfiable. Since $b$ was arbitrary, $T$ is closed.

Traversing a branch of the semantic tree top-down, at each node there is a (partial) interpretation defined by the edges already traversed. It is possible that this interpretation is sufficiently defined to evaluate some of the clauses. In particular, some clause might evaluate to $F$. Since a set of clauses is an implicit conjunction, if one clause evaluates to $F$, the partial interpretation is sufficient to conclude that the entire set of clauses is false. In a closed semantic tree, there must be such a node on every branch: the node may be a leaf or it may be an interior node.

Definition 4.35 Let $T$ be a closed semantic tree for a set of clauses $S$ and let $b$ be a branch in $T$. The node in $b$ closest to the root which falsifies $S$ is a failure node.
**Example 4.36** In Figure 4.3, the node numbered 2 defines a partial interpretation \( v(p) = T, v(q) = F \), which falsifies the clause \( \overline{pq} \) and thus the entire set of clauses \( S \). Neither the parent node (which defines the partial interpretation \( v(p) = T \)) nor the root itself falsify any of the clauses in the set, so node 2 is the node closest to the root on its branches which falsifies \( S \). Hence, node 2 is a failure node.

**Lemma 4.37** Let \( T \) be a closed tree for \( S \). Then each failure node in \( T \) falsifies at least one clause in \( S \).

**Proof:** Immediate.

**Definition 4.38** A clause falsified by a failure node is called a clause associated with the node.

**Example 4.39** In the semantic tree in Figure 4.3, each failure node is circled and labeled with the number of the clause associated with it. It is possible that more than one clause is associated with a failure node; for example, if \( q \) is added to the set of clauses, then \( q \) is another clause associated with failure node numbered 2.

We can be explicit on the relationship between failure nodes and clauses in \( S \).

**Lemma 4.40** A clause \( C \) associated with a failure node \( n \) is a subset of the complements of the literals appearing on the branch \( b \) to \( n \).

**Proof:** Intuitively, for \( C \) to be falsified at a failure node \( n \), all the literals in \( C \) must be assigned to in the partial interpretation and furthermore they must all be assigned \( F \) because \( C \) is a disjunction. Since the partial interpretation is defined by assigning \( T \) to the edge labels, the lemma follows.

Formally, let \( C = \overline{l_1} \cdots \overline{l_k} \) and let \( \{e_1, \ldots, e_m\} \) be the set of literals labeling edges in the branch. Since \( C \) is the clause associated with the failure node \( n \), for each \( i \), \( v(l_i) = F \) where \( v \) is defined by \( v(e_i) = T \) on the corresponding edges. Therefore, \( l_i = e'_i \in \{e'_1, \ldots, e'_m\} \) and \( C = \bigcup_{i=1}^{m} \{l_i\} \subseteq \{e'_1, \ldots, e'_m\} \).

**Definition 4.41** Let \( n_1, n_2 \) be failure nodes which are children of node \( n \). \( n \) is called an inference node:
4.1 Resolution

Example 4.42 In Figure 4.3, $\bar{r}$ is a clause associated with the failure node numbered 3. $\{\bar{r}\}$ is a (proper) subset of $\{\bar{p}, \bar{q}, \bar{r}\}$, the set of complements of the literals assigned to on the branch.

\[\begin{array}{c}
\text{Lemma 4.43} \\
\text{In a closed semantic tree $T$ for a set of clauses $S$, there is at least one inference node.}
\end{array}\]

\[\begin{array}{c}
\text{Proof: Suppose that $n_1$ is a failure node and that its sibling $n_2$ is not. Then no ancestor of $n_2$ is a failure node, because its ancestors are also ancestors of $n_1$ which is, by assumption, the node closest to the root on its branch which falsifies the set of clauses. But every branch through $n_2$ defines an interpretation and they must all falsify $S$ by the assumption that $T$ is closed. So the assignments defined at the leaves of the subtree rooted at $n_2$ falsify $S$; at least one node $n'$ which is a descendant of $n_2$ (a leaf or an ancestor of a leaf) must be a failure node. By induction, either there is an inference node (a pair of sibling failure nodes), or we obtain an infinite sequence of failure nodes $n_1, n', n''$... of increasing depth which is impossible in a finite tree.}
\end{array}\]

Example 4.44 In Figure 4.3, the parent of nodes 3 and 4 is an inference node.

\[\begin{array}{c}
\text{Lemma 4.45} \\
\text{In a closed semantic tree, let $b$ be the branch from the root terminating at inference node $n$. The children $n_1$ and $n_2$ of $n$ are failure nodes, so let $C_1$ and $C_2$ be any clauses associated with them, respectively. Then $C_1, C_2$ clash, and $v_b$, the partial interpretation associated with $n$, falsifies $C$ their resolvent clause.}
\end{array}\]

\[\begin{array}{c}
\text{Proof: Let $b_1$ and $b_2$ be the branches that terminate at failure nodes $n_1$ and $n_2$, respectively. Since $b_1$ and $b_2$ are identical except for the edges from $n$ to $n_1$ and $n_2$, by Lemma 4.40 the nodes are associated with clauses $C_1, C_2$ which clash on the literals $l, l'$ of the atom $p$. The partial interpretation $v_{b_1}$ is the same as $v_{b_2}$, except that it does not assign to $p$. But $v_{b_1}(C_1) = v_{b_2}(C_2) = F$ since $n_1$ and $n_2$ are failure nodes, so certainly, $v_{b_1}(C_1 - \{l\}) = F$ and $v_{b_2}(C_2 - \{l'\}) = F$. Clearly, $v_b(C) = v_b((C_1 - \{l\}) \cup (C_2 - \{l'\})) = F.$}
\end{array}\]

Example 4.46 In Figure 4.3, $\bar{r}$ and $\bar{p}qr$ are clauses associated with failure nodes 3 and 4, respectively. The resolvent $\bar{p}q$ is falsified by $v(p) = T, v(q) = T$, the partial interpretation associated with the parent of 3 and 4. Note that if we add the resolvent $\bar{p}q$ to $S$ to obtain $S_1$, the parent node is a failure node for $S_1$.

There is one more technicality that must be overcome in the general case. The same semantic tree (Figure 4.3) is also a semantic tree for the set of clauses $\{p, \bar{p}q, \bar{r}, \bar{pr}\}$, where 3 is a failure node associated with $\bar{r}$ and 4 is a failure node associated with $\bar{pr}$. However, their parent is not a failure node, since the resolvent $\bar{p}$ is already falsified by
a node higher up in the tree, while a failure node was defined to be the node closest to
the root which falsifies the set of clauses.

**Lemma 4.47** Let \( n \) be an inference node, let \( C_1, C_2 \in S \) be clauses associated with
the failure nodes that are the children of \( n \), and let \( C \) be their resolvent. Then \( S \cup \{ C \} \)
has a failure node that is either \( n \) or an ancestor of \( n \) and \( C \) is a clause associated
with the failure node.

**Proof:** By Lemma 4.45, \( v(C) = F \), where \( v \) is the partial interpretation associated
with the inference node. By Lemma 4.40, \( C \subseteq \{ \ell_{j1}^1, \ldots, \ell_{jn}^j \} \), the complements of the
literals labeling \( b \), the path to the inference node. Let \( j \) be the smallest index such
\( C \cap \{ \ell_{j1}^1, \ldots, \ell_{jn}^j \} \) is empty. Then \( C \subseteq \{ \ell_{j1}^1, \ldots, \ell_{jn}^j \} \) and \( v_j(C) = F \) where \( v_j \) is the partial
interpretation defined at node \( j \). \( j \) is a failure node and \( C \) is a clause associated with
it.

**Proof of the completeness of resolution:** If \( S \) is an unsatisfiable set of clauses, there
is a closed semantic tree \( T \) for \( S \). Clauses of \( S \) can be associated with failure nodes
in \( T \). By Lemma 4.43, there is at least one inference node in \( T \). An application of
the resolution rule at that node adds the resolvent to the set, creating a failure node
by Lemma 4.47 and deleting two failure nodes, thus decreasing the number of failure
nodes. When the number of failure nodes has decreased to one, it must be the root
which is associated with the derivation of the empty clause by the resolution rule.

**Implementation**

We start with a program to convert a formula into CNF and then give a program to
perform resolution. The program for conversion to CNF performs three steps one after
another: elimination of operators other than negation, disjunction and conjunction,
reducing the scope of negations using De Morgan’s laws and double negation, and
finally distribution of disjunction over conjunction.

\[
\text{cnf}(A, A3) :-
\text{eliminate}(A, A1),
\text{demorgan}(A1, A2),
\text{distribute}(A2, A3).
\]

Elimination of operators is done by a recursive traversal of the formation tree. The
first three clauses eliminate \( \text{imp}, \text{eqv} \) and \( \text{neqv} \).

\[
\text{eliminate}(A \text{ eqv} B, (A1 \text{ and } B1) \text{ or } ((\text{neg } A1) \text{ and } (\text{neg } B1))) :-
\text{eliminate}(A, A1), \text{eliminate}(B, B1).
\]

\[
\text{eliminate}(A \text{ xor } B, (A1 \text{ and } (\text{neg } B1) \text{ or } ((\text{neg } A1) \text{ and } B1)) :-
\text{eliminate}(A, A1), \text{eliminate}(B, B1).
\]

\[
\text{eliminate}(A \text{ imp } B, (\text{neg } A1) \text{ or } B1) :-
\text{eliminate}(A, A1), \text{eliminate}(B, B1).
\]
The next four clauses simply traverse the tree for the other operators.

\[
\text{eliminate}(A \text{ or } B, A_1 \text{ or } B_1) :- \\
\quad \text{eliminate}(A, A_1), \text{eliminate}(B, B_1).
\]

\[
\text{eliminate}(A \text{ and } B, A_1 \text{ and } B_1) :- \\
\quad \text{eliminate}(A, A_1), \text{eliminate}(B, B_1).
\]

\[
\text{eliminate}(\neg A, \neg A_1) :- \\
\quad \text{eliminate}(A, A_1).
\]

\[
\text{eliminate}(A, A).
\]

The application of De Morgan’s laws is similar. Two clauses apply the laws for negations of conjunction and disjunction.

\[
\text{demorgan}(\neg (A \text{ and } B), A_1 \text{ or } B_1) :- \\
\quad \text{demorgan}(\neg A, A_1), \text{demorgan}(\neg B, B_1).
\]

\[
\text{demorgan}(\neg (A \text{ or } B), A_1 \text{ and } B_1) :- \\
\quad \text{demorgan}(\neg A, A_1), \text{demorgan}(\neg B, B_1).
\]

The next two clauses traverse the formula for non-negated formulas.

\[
\text{demorgan}(A \text{ and } B, A_1 \text{ and } B_1) :- \\
\quad \text{demorgan}(A, A_1), \text{demorgan}(B, B_1).
\]

\[
\text{demorgan}(A \text{ or } B, A_1 \text{ or } B_1) :- \\
\quad \text{demorgan}(A, A_1), \text{demorgan}(B, B_1).
\]

The final two clauses eliminate double negation and terminate the recursion at a literal.

\[
\text{demorgan}((\neg (\neg A)), A_1) :- \\
\quad \text{demorgan}(A, A_1).
\]

\[
\text{demorgan}(A, A).
\]

Distribution of disjunction over conjunction is more tricky, because one step of the distribution may produce additional structures that must be handled. For example, one step of distribution applied to \( p \lor ((q \land r) \land r) \) gives \( p \lor (q \land r) \land (p \lor r) \) and the distribution rule must be applied again.

\[
\text{distribute}(A \text{ or } (B \text{ and } C), ABC) :- !, \\
\quad \text{distribute}(A \text{ or } B, AB), \\
\quad \text{distribute}(A \text{ or } C, AC), \\
\quad \text{distribute}(AB \text{ and } AC, ABC).
\]

\[
\text{distribute}(((A \text{ and } B) \text{ or } C, ABC) :- !, \\
\quad \text{distribute}(A \text{ or } C, AC), \\
\quad \text{distribute}(B \text{ or } C, BC), \\
\quad \text{distribute}(AC \text{ and } BC, ABC).
\]
distribute(A or B, A1 or B1) :-
    distribute(A, A1),
    distribute(B, B1).

distribute(A and B, A1 and B1) :-
    distribute(A, A1),
    distribute(B, B1).

distribute(A, A).

To perform resolution we first convert a CNF formula to clausal form—a set of sets of literals—using a simple program not given here. The program also discards valid clauses like \{p, r, \neg p\}. The goal clause resolution(S) succeeds if the set of clauses S can be refuted, that is, if the empty clause can be produced by resolution. We start with two rules, one that fails if the set of clauses is empty which means that the formula is valid, and one that succeeds if the empty clause is contained in the set.

resolution([]) :- !, fail.
resolution(S) :- member([], S), !.

The other rule nondeterministically selects two clauses in the set, creates their resolvent, and recursively calls the predicate with the resolvent added to the set. The predicate will fail and backtrack in three cases:

- The two clauses cannot be resolved because they do not have clashing literals.
- The resolvent is a useless valid clause (a clause that clashes with itself).
- The resolvent already exists in the set.

resolution(S) :-
    member(C1, S),
    member(C2, S),
    clashing(C1, L1, C2, L2),
    delete(C1, L1, C1P),
    delete(C2, L2, C2P),
    union(C1P, C2P, C),
    \+ clashing(C, _, _, _),
    \+ member(C, S),
    resolution([C | S]).

Using nondeterminism, it is trivial to check for clashing literals.

clashing(C1, L, C2, neg L) :-
    member(L, C1), member(neg L, C2), !.
clashing(C1, neg L, C2, L) :-
    member(neg L, C1), member(L, C2), !.
4.2 Binary decision diagrams (BDDs)

Suppose that you want to decide if two formulas $A_1$ and $A_2$ are logically equivalent. You could construct truth tables for both formulas and check if they are identical. Alternatively, you can check if $A_1 \leftrightarrow A_2$ is valid by constructing a semantic tableau for its negation, or by trying to refute the clausal form of its negation using resolution. All these methods can be inefficient if there are many atoms or if the formula is large.

In this section we describe the binary decision diagram (BDD), which is a data structure for propositional formulas. These data structures have the property that (under a certain condition) there is a unique structure for equivalent formulas. Algorithms for BDDs have been found to be surprisingly efficient in many cases. BDDs are extensively used in applications such as circuit design and program verification, where you want to prove that one formula—which describes the operation of a circuit or program—is equivalent to another formula—which specifies the behavior of the circuit or program.

In this section we present a few rather long-winded examples intended to help you understand the intuition behind the algorithms. The formal description of the algorithms and Prolog implementations are given in the next section.

Efficient truth tables

Consider the truth table for $p \lor (q \land r)$:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$r$</th>
<th>$p \lor (q \land r)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$T$</td>
<td>$F$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$F$</td>
<td>$F$</td>
<td>$T$</td>
</tr>
<tr>
<td>$F$</td>
<td>$T$</td>
<td>$T$</td>
<td>$T$</td>
</tr>
<tr>
<td>$F$</td>
<td>$T$</td>
<td>$F$</td>
<td>$T$</td>
</tr>
<tr>
<td>$F$</td>
<td>$F$</td>
<td>$T$</td>
<td>$F$</td>
</tr>
<tr>
<td>$F$</td>
<td>$F$</td>
<td>$F$</td>
<td>$F$</td>
</tr>
</tbody>
</table>

From the first two rows, we can see that when $p$ and $q$ are assigned $T$, the formula evaluates to $T$ regardless of the value of $r$. Similarly, for the second two rows. Thus the first four rows can be condensed into two rows:

<table>
<thead>
<tr>
<th>$p$</th>
<th>$q$</th>
<th>$r$</th>
<th>$p \lor (q \land r)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$T$</td>
<td>$*$</td>
<td>$T$</td>
</tr>
<tr>
<td>$T$</td>
<td>$F$</td>
<td>$*$</td>
<td>$T$</td>
</tr>
</tbody>
</table>
where * indicates that the value assigned to \( r \) is immaterial. In fact, we now see that the value assigned to \( q \) is immaterial, so these two rows collapse into one:

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( r )</th>
<th>( p \lor (q \land r) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>*</td>
<td>*</td>
<td>( T )</td>
</tr>
</tbody>
</table>

After collapsing the last two rows, the entire truth table can be expressed in four rows:

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( r )</th>
<th>( p \lor (q \land r) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>*</td>
<td>*</td>
<td>( T )</td>
</tr>
<tr>
<td>( F )</td>
<td>( T )</td>
<td>( F )</td>
<td>( T )</td>
</tr>
<tr>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>*</td>
<td>*</td>
<td>( F )</td>
</tr>
</tbody>
</table>

Let us try another example, this time for the formula \( p \oplus q \oplus r \). It is easy to compute the truth table for a formula whose only operator is \( \oplus \), since a row evaluates to true if and only if an odd number of atoms are assigned true.

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( r )</th>
<th>( p \oplus q \oplus r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( T )</td>
<td>( F )</td>
<td>( T )</td>
<td>( F )</td>
</tr>
<tr>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( F )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
</tbody>
</table>

Here, adjacent rows cannot be collapsed, but careful examination reveals that rows 5 and 6 show the same dependence on \( r \) as do rows 3 and 4. Rows 7 and 8 are similarly related to rows 1 and 2. Instead of explicitly writing the truth table entries for these rows, we can simply refer to the previous entries:

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( r )</th>
<th>( p \oplus q \oplus r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
<td>( T )</td>
</tr>
<tr>
<td>( T )</td>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( T )</td>
<td>( F )</td>
<td>( T )</td>
<td>( F )</td>
</tr>
<tr>
<td>( T )</td>
<td>( F )</td>
<td>( F )</td>
<td>( F )</td>
</tr>
<tr>
<td>( F )</td>
<td>( T )</td>
<td>*</td>
<td>(See rows 3 and 4.)</td>
</tr>
<tr>
<td>( F )</td>
<td>( F )</td>
<td>*</td>
<td>(See rows 1 and 2.)</td>
</tr>
</tbody>
</table>

We turn now to an alternate representation of the semantics of a propositional formula that is much more efficient than truth tables.
4.2 Binary decision diagrams (BDDs)

Reduction of BDDs

A binary decision diagram, like a truth table, is a representation of the value of a formula under all possible assignments to atomic propositions in the formula. We start with a graph that is simply a notational variant of a semantic tree: each node of the tree is labeled with an atom, and the solid and dotted edges leaving the node represent the assignment of true or false, respectively, to this atom. Along each branch, there is an edge for every atom in the formula, so there is a one-to-one correspondence between branches and assignments. The leaf of a branch is labeled with the value of the formula under its assignment.

Here is a BDD for \( p \lor (q \land r) \); check that the value of the formula for each assignment is the same as that given in the truth table above.

We can modify the structure of this tree to obtain a more concise representation without losing the ability to evaluate the formula under any assignment. The modifications are called reductions; when no more reductions can be made, the diagram is called a reduced BDD.

The first reduction is to coalesce all the leaves into just two, one for \( T \) and one for \( F \). The tree is now a dag (directed acyclic graph), where the direction of an edge is implicitly from a node to its child.

The second reduction is to remove nodes that are not needed to evaluate the formula. Once on the left-hand side of the diagram and twice on the right-hand side, the node
for \( r \) has both outgoing edges leading to the same node. This means that the partial assignment to \( p \) and \( q \) is sufficient to determine the value of the formula. In each of the three cases, the \( r \) node and its outgoing edges can be deleted and the incoming edge to the \( r \) node connected directly to the joint target node.

Now, the right-hand node for \( q \) becomes redundant and can be eliminated using the same type of reduction.

No more reductions can be made on this diagram which is now reduced. Evaluation of the formula for a given truth assignment can be efficiently computed because many nodes have been deleted. For example, when evaluating \( p \lor (q \land r) \), if the assignment begins by assigning \( T \) to \( p \), we immediately reach the \( T \) leaf.

Let us now consider the formula \( p \oplus q \oplus r \). Here is the dag after the leaves of the tree have been coalesced by the first type of reduction.
The second type of reduction (removing nodes with both outgoing edges pointing to a single node) is not applicable, but examination of the BDD reveals that the two outermost nodes for \( r \) have the same structure as do the two innermost nodes, because their true and false edges point to the same subgraphs, in this case the leaves. So the true (solid) edge from the right-hand node for \( q \) can be made to point to the leftmost node for \( r \), and the false (dotted) edge from that node can be made to point to the second \( r \) node from the left. Deleting the two \( r \) nodes on the right and their outgoing edges gives the following reduced BDD.

![Reduced BDD Diagram](image)

We invite you to check that these BDDs represent the same evaluations of the formulas as do the truth tables.

**Definition of BDDs**

**Definition 4.48** A *Binary Decision Diagram (BDD)* for a formula \( A \) is a rooted directed binary acyclic graph. Each nonterminal is labeled with an atom and each leaf is labeled with a truth value. No atom appears more than once in a path from the root to an edge. One outgoing edge is the false edge and is denoted by a dotted line; the other outgoing edge is the true edge and is denoted by a solid line. With each path from the root to a leaf is associated an assignment to the atoms of \( A \). Assign \( F \) to the atom if the false edge is taken from the node labeled by the atom, and assign \( T \) if the true edge is taken. The leaf gives the value of \( A \) under this assignment. The path need not include all atoms in \( A \), but it must include assignments to enough atoms to enable the value of \( A \) to be computed.

**Example 4.49** There are four paths in the reduced BDD for \( p \lor (q \land r) \) on page 84. Written as literals to denote the assignments, they are (from left to right):

\[
(\neg p, \neg q), (\neg p, q, \neg r), (\neg p, q, r), (p). \]

The first two paths represent all the assignments that give the value \( F \) to the formula, and the second two represent assignments that give \( T \).
**The apply operation**

It hardly seems worthwhile to create a BDD if we start from the full binary tree whose size is about the same as the size of the truth table. The power of BDDs comes from the ability to perform propositional operations directly on the two reduced BDDs. That is, there is an algorithm \textbf{Apply} that constructs the BDD for $A_1 \text{ op } A_2$ directly and efficiently from the reduced BDDs for $A_1$ and $A_2$. The algorithm is also used to construct an initial BDD for an arbitrary formula by building it up from the BDDs for atoms. The BDD for the atom $p$ is simply:

\begin{center}
\begin{tikzpicture}[scale=0.8]
    \node (p) at (0,0) {$p$};
    \node (true) at (2,0) {$\top$};
    \node (false) at (-2,0) {$\bot$};
    \draw (p) -- (true);
    \draw (p) -- (false);
\end{tikzpicture}
\end{center}

We now construct the BDD for $(p \oplus q) \oplus (p \oplus r)$ from the BDDs for $p \oplus q$ and $p \oplus r$, by applying the operator $\oplus$ to the pair of BDDs. In the following diagram, we have drawn the two BDDs with the operator $\oplus$ between them.

\begin{center}
\begin{tikzpicture}[scale=0.8]
    \node (p) at (0,0) {$p$};
    \node (true) at (2,0) {$\top$};
    \node (false) at (-2,0) {$\bot$};
    \draw (p) -- (true);
    \draw (p) -- (false);
    \node (q) at (0,-2) {$q$};
    \node (false) at (-2,-2) {$\bot$};
    \node (true) at (2,-2) {$\top$};
    \draw (q) -- (true);
    \draw (q) -- (false);
    \node (q') at (0,-3) {$\bot$};
    \node (true) at (2,-3) {$\top$};
    \node (false) at (-2,-3) {$\bot$};
    \draw (q') -- (true);
    \draw (q') -- (false);
    \node (p) at (0,-4) {$p$};
    \node (true) at (2,-4) {$\top$};
    \node (false) at (-2,-4) {$\bot$};
    \draw (p) -- (true);
    \draw (p) -- (false);
\end{tikzpicture}
\end{center}

Since $(p \oplus q) \oplus (p \oplus r) \equiv q \oplus r$, the result of the construction should be the BDD for $q \oplus r$.

The algorithm uses structural induction on the pair of BDDs. Given a BDD, the nodes joined to the root are themselves BDDs of formulas obtained by substituting $T$ and $F$ into the formula. The two BDDs above can be considered to have the following structure:

\begin{center}
\begin{tikzpicture}[scale=0.8]
    \node (p) at (0,0) {$p$};
    \node (true) at (2,0) {$\top$};
    \node (false) at (-2,0) {$\bot$};
    \draw (p) -- (true);
    \draw (p) -- (false);
    \node (q) at (0,-2) {$q$};
    \node (false) at (-2,-2) {$\bot$};
    \node (true) at (2,-2) {$\top$};
    \draw (q) -- (true);
    \draw (q) -- (false);
    \node (q') at (0,-3) {$\bot$};
    \node (true) at (2,-3) {$\top$};
    \node (false) at (-2,-3) {$\bot$};
    \draw (q') -- (true);
    \draw (q') -- (false);
    \node (p) at (0,-4) {$p$};
    \node (true) at (2,-4) {$\top$};
    \node (false) at (-2,-4) {$\bot$};
    \draw (p) -- (true);
    \draw (p) -- (false);
\end{tikzpicture}
\end{center}
If \( p \) is assigned \( F \) in both \( p \oplus q \) and \( p \oplus r \), it will be assigned \( F \) in \( (p \oplus q) \oplus (p \oplus r) \). So to compute \( (p \oplus q) \oplus (p \oplus r) \) for this assignment, we need to compute \( (F \oplus q) \oplus (F \oplus r) \), which simplifies to \( q \oplus r \). Similarly, if \( p \) is assigned \( T \), we need to compute \( (T \oplus q) \oplus (T \oplus r) \), which simplifies to \( \neg q \oplus \neg r \). Graphically, this can be represented by recursing on the BDDs. The following diagram shows the pair of BDDs obtained by taking the right-hand edges that assign \( T \) to \( p \).

The recursion can be continued by assigning \( T \) and then \( F \) to \( q \). Since the right-hand formula \( \neg r \) does not depend on the assignment to \( q \), it does not split into two recursive subcases as does the left-hand formula \( \neg q \). Instead, the algorithm must be applied recursively for each sub-BDD of \( \neg q \) together with the entire BDD for \( \neg r \). The following diagram shows the computation that must be done if the right-hand (false) branch of the BDD for \( \neg q \) is taken.

The left-hand BDD has reached the base case of the recursion. Recursing now on the BDD for \( \neg r \) also gives base cases, one for the left-hand (true) branch:

and one for the right-hand (false) branch:
On these base cases, the computation of the resultant BDD is immediate, as shown. Returning up the recursive evaluation sequence, these two results can be combined to give the result for the non-leaf BDD.

\[
\begin{align*}
\text{BDD for } \neg F & \quad \text{BDD for } \neg r \\
\top & \quad \oplus \\
\bot & \quad \top \\
\end{align*}
\]

\[
\begin{align*}
\text{BDD for } \neg F \oplus \neg r & \equiv \\
\top & \quad \top \\
\bot & \quad \bot
\end{align*}
\]

The BDD produced by continuing the algorithm to completion is:

![Diagram of BDD]

and when it is reduced the expected answer is obtained:

\[
\begin{align*}
\text{BDD for } q \oplus r & \\
\top & \quad \top \\
\bot & \quad \bot
\end{align*}
\]

Many optimizations can be made to improve the efficiency of the algorithm as described in the section on implementation.

### 4.3 Algorithms on BDDs

The algorithm **Reduce** constructs an canonical BDD if the atoms on each path have compatible orderings.

**Definition 4.50** The ordered sequence of atoms labeling the nodes on a path is called an *ordering* of the atoms. A set of orderings \( \{O_1, \ldots, O_n\} \) is *compatible* iff there are no atoms \( p, p' \) such that \( p \) comes before \( p' \) in \( O_i \) and \( p' \) comes before \( p \) in \( O_j, i \neq j \).
Definition 4.51 An Ordered Binary Decision Diagram (OBDD) is a BDD such that the set of orderings of the atoms of all paths is compatible.

Example 4.52 The orderings of atoms along the four paths in the reduced BDD for \( p \lor (q \land r) \) on page 84 are \((p, q), (p, q, r), (p, q, r), (p)\), so the BDD is an OBDD.

The algorithm Reduce

Algorithm 4.53 (Reduce)

Input: An ordered binary decision diagram \( bdd \).
Output: An ordered binary decision diagram in reduced form.

If \( bdd \) has more than two distinct leaves (one labeled \( T \) and one labeled \( F \)), remove duplicate leaves and direct all edges that point to leaves to the single remaining leaf for each truth value. Then perform the following steps as long as possible:

1. If both the false and the true edges of a node labeled \( v_i \) point to the same node labeled \( v_j \), delete this node for \( v_i \) and direct \( v_i \)'s incoming edges to \( v_j \).

2. If two nodes labeled \( v_i \) are the roots of identical sub-BDDs, delete one sub-BDD and direct its incoming edges to the other node.

When the algorithm terminates (as it must), the BDD is said to be reduced.

Theorem 4.54 (Bryant) The algorithm Reduce is correct. It returns a reduced OBDD that is equivalent to the OBDD in the input, in the sense that they give the same value to each assignment.

For a given ordering of atoms, the reduced OBDDs for logically equivalent formulas are structurally identical.

The proofs of the theorems in this section can be found in Bryant (1986).

The algorithm Reduce and Theorem 4.54 immediately provide a set of trivial algorithms for properties of propositional formulas:

- A formula is satisfiable iff \( T \) appears in its reduced OBDD.
- A formula is valid iff its OBDD is simply \( T \).
- A formula is unsatisfiable iff its OBDD is simply \( F \).
- \( A \equiv B \) if their OBDD's are identical.
The usefulness of OBDDs depends of course on the efficiency of the algorithm Reduce (and others that we will describe), which in turn depends on the size of reduced OBDDs. It turns out that in many useful cases the size is quite small. However, the size of the reduced OBDD for a formula depends on the ordering.

**Theorem 4.55 (Bryant)** The OBDD for the formula \( (p_1 \land p_2) \lor \ldots \lor (p_{2n-1} \land p_{2n}) \) has \( 2n + 2 \) nodes under the ordering \( p_1, \ldots, p_{2n} \), and \( 2^{n+1} \) nodes under the ordering \( p_1, p_{n+1}, p_2, p_{n+2}, \ldots, p_n, p_{2n} \).

Fortunately, you can generally use heuristics to choose an efficient ordering. Unfortunately, not all formulas have orderings that lead to small OBDDs.

**Theorem 4.56 (Bryant)** There is a formula \( A \) with \( n \) atoms such that the reduced OBDD for any ordering of the atoms has at least \( 2^{cn} \) nodes for some \( c > 0 \).

The theorem is constructive in that Bryant gave a specific formula and bound \( c \). Theorem 4.56 is not surprising for reasons to be discussed in Section 4.4.

**The algorithm Apply**

The following algorithm returns an OBDD for \( A_1 \) op \( A_2 \) constructed from the OBDDs for the formulas \( A_1 \) and \( A_2 \).

**Algorithm 4.57 (Apply)**

**Input**: A Boolean operator \( \text{op} \), an OBDD \( \text{bdd}_1 \) for formula \( A_1 \), and an OBDD \( \text{bdd}_2 \) for formula \( A_2 \), such that the union of the set of orderings of atoms in the two OBDDs is compatible.

**Output**: An OBDD for the formula \( A_1 \) op \( A_2 \).

- If \( \text{bdd}_1 \) and \( \text{bdd}_2 \) are both leaves labeled \( v_1 \) and \( v_2 \), respectively, return the leaf labeled by \( v_1 \) op \( v_2 \).
- If the roots of \( \text{bdd}_1 \) and \( \text{bdd}_2 \) are both labeled by the same atom \( p \), then return the BDD whose root is labeled by \( p \) and whose left (right) sub-BDD is obtained by recursively performing this algorithm on the left (right) sub-BDDs of \( \text{bdd}_1 \) and \( \text{bdd}_2 \).
- If the root of \( \text{bdd}_1 \) is labeled by an atom \( p_1 \), and the root of \( \text{bdd}_2 \) is labeled by some other atom \( p_2 \) such that \( p_1 < p_2 \) in the ordering, then return the BDD whose root is labeled by \( p_1 \) and whose left (right) sub-BDD is obtained by recursively performing this algorithm with \( \text{bdd}_1 \) and the left (right) sub-BDD of \( \text{bdd}_2 \). This construction is also performed if \( \text{bdd}_2 \) is a leaf, but \( \text{bdd}_1 \) is not.
• Otherwise, we have a symmetrical case to the previous one. The BDD returned has its root labeled by $p_2$ and its left (right) sub-BDD obtained by recursively performing this algorithm on the left (right) sub-BDD of $bddd_1$ and $bddd_2$.

\textit{Restriction and quantification*}

\textbf{Definition 4.58} The \textit{restriction} operation takes a formula $A$, an atom $p$ and a value $v = T$ or $v = F$, and returns the formula obtained by substituting $v$ for $p$ and partially evaluating $A$. Notation: $A|_{p=v}$.

\textbf{Example 4.59} Let $A = p \lor (q \land r)$.

\[A|_{r=T} = p \lor (q \land T) = p \lor q\]
\[A|_{r=F} = p \lor (q \land F) = p \lor F = p.\]

The algorithm \textbf{Reduce} is justified by appealing to the following theorem which expresses the application of an operator in terms of its application to restrictions.

\textbf{Theorem 4.60 (Shannon expansion)}

\[A_1 \text{ op } A_2 \equiv (p \land (A_1|_{p=T} \text{ op } A_2|_{p=T})) \lor (\neg p \land (A_1|_{p=F} \text{ op } A_2|_{p=F})).\]

\textbf{Proof:} Exercise. Restriction is very easy to implement on OBDDs.

\textbf{Algorithm 4.61 (Restrict)}

\textbf{Input}: An ordered binary decision diagram $bdd$ for a formula $A$, an atom $p$ occurring in $A$ and a value $v$.

\textbf{Output}: An ordered binary decision diagram for $A|_{p=v}$.

The restriction is obtained by a recursive traverse of the OBDD.

1. If the root of $bdd$ is a leaf, return the leaf.
2. If the root of $bdd$ is labeled with $p$, then return the false or true sub-BDD, according to the value of $v$.
3. Otherwise (the root is labeled with $q \neq p$), apply the algorithm to the left and right sub-BDDs, and return the tree whose root is $q$ and whose left and right sub-BDDs are those returned by the recursive calls.
Propositional Calculus: Resolution and BDDs

Example 4.62 The OBDD of $A = p \lor (q \land r)$ is shown in (a) below. (b) is $A|_{r=T}$ and (c) is $A|_{r=F}$. Note that the restriction may not be reduced; the reduction of the OBDD in (c) is shown in (d).

(a) $\begin{array}{c}
\text{p} \\
\text{F} \\
\text{q} \\
\text{T} \\
\text{r} \\
\text{F} \\
\end{array}$
(b) $\begin{array}{c}
\text{p} \\
\text{F} \\
\text{q} \\
\text{T} \\
\text{r} \\
\text{F} \\
\end{array}$
(c) $\begin{array}{c}
\text{p} \\
\text{F} \\
\text{q} \\
\text{T} \\
\text{r} \\
\text{T} \\
\end{array}$
(d) $\begin{array}{c}
\text{p} \\
\text{F} \\
\text{q} \\
\text{T} \\
\text{r} \\
\text{T} \\
\end{array}$

Compare the OBDDs in (b) and (d) with the formulas in Example 4.59.

Is $A = p \lor (q \land r)$ true for some value of $r$ or for all values of $r$? Of course we could construct a truth table and check, but it is much more efficient to use BDDs.

Definition 4.63 Let $A$ be a formula and $p$ an atom. The existential quantification of $A$ is the formula denoted $\exists pA$ that is true iff for some assignment to $p$, $A$ is true. The universal quantification of $A$ is the formula denoted $\forall pA$ that is true iff for all assignments to $p$, $A$ is true.

Theorem 4.64 $\exists pA = A|_{p=F} \lor A|_{p=T}$ and $\forall pA = A|_{p=F} \land A|_{p=T}$.

Proof: Exercise.

Quantification is easily computed using OBDDs:

$\exists pA = \text{apply}(\text{restrict}(A, p, F), \text{or}, \text{restrict}(A, p, T))$.
$\forall pA = \text{apply}(\text{restrict}(A, p, F), \text{and}, \text{restrict}(A, p, T))$.

Example 4.65

$\exists r (p \lor (q \land r)) = p \lor (p \lor q) = p \lor q$
$\forall r (p \lor (q \land r)) = p \land (p \lor q) = p$.

We leave it as an exercise to perform these computations using OBDDs.

Implementation

Atoms are represented by integers: think of $N$ as standing for the atom $p_N$. BDDs are represented by the predicate $\text{bdd}(N, \text{False}, \text{True})$, where $N$ is the atom labeling the
4.3 Algorithms on BDDs

root, False is the sub-BDD when \( N \) is assigned \( F \) and True is the sub-BDD when \( N \) is assigned \( T \).

The algorithm **Reduce** does a recursive traversal of the BDD: calling \( \text{Reduce}(B, BR) \) with a BDD \( B \) returns the reduced BDD in \( BR \). A cache of reduced BDDs is maintained as a dynamic database, so that it is easy to check if a BDD already exists as required by the second type of reduction in the algorithm. \( \text{retraceall} \) should be called before executing \( \text{reduce} \) in order to initialize the database.

The first clause checks if the current BDD is in the cache; if so, unification is used to return the existing BDD.

\[
\text{reduce}(B, B) :- B, !.
\]

Next we check if the BDD is a leaf; if so, it is placed into the cache and returned.

\[
\text{reduce}(B, B) :- B = \text{bdd}(\text{leaf}, _, _), !, \text{assert}(B).
\]

The third clause recurses on the sub-BDDs, but before returning it calls \( \text{remove} \) to perform the first type of reduction.

\[
\text{reduce}(\text{bdd}(N, \text{False}, \text{True}), \text{NewNode}) :-
\text{reduce}(\text{False}, \text{NewFalse}),
\text{reduce}(\text{True}, \text{NewTrue}),
\text{remove}(\text{bdd}(N, \text{NewFalse}, \text{NewTrue}), \text{NewNode}).
\]

If the two edges from this node \( N \) point to the same subBDD, \( N \) must be removed and one copy of the subBDD returned instead.

\[
\text{remove}(\text{bdd}(_, \text{SubBDD}, \text{SubBDD}), \text{SubBDD}) :- !.
\]

Otherwise, the new BDD formed by \( N \) and the subBDDs returned by the recursion is returned unchanged, but stored in the cache for future use.

\[
\text{remove}(B, B) :- \text{assert}(B).
\]

The algorithm **Apply** requires a simultaneous recursive traversal of two BDDs. The base case is if both BDDs are leaves. In this case, simply apply the operator to the values in the leaves.

\[
\text{apply}(\text{bdd}(\text{leaf}, \text{Val1}, _), \text{Opr}, \text{bdd}(\text{leaf}, \text{Val2}, _),
\text{bdd}(\text{leaf}, \text{ValResult}, x)) :- !,
\text{opr}(\text{Opr}, \text{Val1}, \text{Val2}, \text{ValResult}).
\]

If the same atom is at the root of both BDDs, a simultaneous recursion is done and the resultant BDD constructed from the BDDs that are returned.
apply(bdd(N, False1, True1), Opr, bdd(N, False2, True2),
    bdd(N, FalseResult, TrueResult)) :- !,
apply(False1, Opr, False2, FalseResult),
apply(True1, Opr, True2, TrueResult).

The difficult part of the implementation is when one of the BDDs has a atom at the
root and the other is a leaf, or when the roots are labeled with different atoms. The first
clause is taken if the right-hand sub-BDD is a leaf or has a higher-numbered atom; in
this case, the sub-BDDs of the left-hand node are applied to the entire right-hand node
Node2. The two cases can be treated together, using the ; operator in Prolog which
succeeds if either of its operand does.

apply(bdd(N1, False1, True1), Opr, Node2,
    bdd(N1, FalseResult, TrueResult)) :-
Node2 = bdd(N2, _, _),
(N2 = leaf ; (N1 \= leaf, N1 < N2)), !,
apply(False1, Opr, Node2, FalseResult),
apply(True1, Opr, Node2, TrueResult).

(The check that N1 is not a leaf is not needed by the algorithm; it just ensures that
we don’t try to evaluate leaf<N2 which is illegal in Prolog.) The second clause is
symmetrical if the left-hand node is a leaf or has a higher-numbered atom.

apply(Node1, Opr, bdd(N2, False2, True2),
    bdd(N2, FalseResult, TrueResult)) :-
apply(Node1, Opr, False2, FalseResult),
apply(Node1, Opr, True2, TrueResult).

The source archive extends the implementation to include optimizations that are es-

sential for practical use of the algorithm:

- Create a cache bdd_pair(B1, B2, B) of pairs of BDDs and the result of
  applying the operator to the pair. As in the algorithm for reduce, first check if
  the result is in the database before performing the traversal.

- If one of the BDDs is a leaf, check if its value is a controlling operand
  for the operator. A value is controlling if the result of the operation does not depend
  on the other operand. T is controlling for \( \lor \) (as shown in the diagram), F is
  controlling for \( \land \), and F is controlling for the left operand of \( \rightarrow \).

<table>
<thead>
<tr>
<th>BDD for ( T )</th>
<th>BDD for formula ( A )</th>
<th>BDD for ( T )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T )</td>
<td>( \lor )</td>
<td>( \equiv )</td>
</tr>
<tr>
<td></td>
<td>( \circ )</td>
<td>( T )</td>
</tr>
</tbody>
</table>
In the presence of a controlling value, the other sub-BDD can be ignored, and the BDD for the controlling value returned immediately.

- Integrate `reduce` with `apply` instead of first creating an unreduced BDD.

4.4 Complexity*

Let us review some of the definitions in algorithmic complexity.

An algorithm is deterministic if its computation (and hence its result) is fully determined by its input. A deterministic algorithm is correct iff the (single) result is correct.

**Example 4.66** The method of truth tables is a deterministic algorithm for deciding both satisfiability and validity in the propositional calculus: the algorithm constructs the truth table for a formula and checks if \( T \) appears in some or all of the rows of the table.

An algorithm is nondeterministic if it is not deterministic, that is, if there may be more than one computation for a given input. A nondeterministic algorithm is correct iff the result of some computation is correct.

**Example 4.67** The construction of a semantic tableau is a nondeterministic algorithm because at any stage of the construction, we can choose a leaf to expand, and further, choose a formula in the label of the leaf to which a rule will be applied. If fact, every computation (construction) produces a correct answer, so the algorithm is not characteristic of a nondeterministic algorithm.

**Example 4.68** Here is a nondeterministic algorithm for deciding the satisfiability of a formula \( A \).

Choose an interpretation \( v \) for \( A \). If \( v(A) = T \), then \( A \) is satisfiable.

If \( A \) is satisfiable, for some computation (choice of \( v \)), the result is correct. Of course, other choices may not give the correct answer, but that does not affect the correctness of the nondeterministic algorithm.

An algorithm is said to run in polynomial time if its running time can be bounded from above by a polynomial in \( n \), where \( n \) is the size of the input. An algorithm is said to run in exponential time if its running time can be bounded from below by \( 2^{cn} \) for some positive \( c \).

The truth-table method is not an efficient algorithm for satisfiability or validity in the propositional calculus because we construct \( 2^n \) rows, where \( n \) is the number of variables. For a formula whose size is polynomial in the number of variables, the
complexity of the method will be exponential. The method of semantic tableaux and the resolution procedure are usually more efficient than truth tables; nevertheless, there are families of formulas for which these methods have exponential complexity.

The nondeterministic algorithm in Example 4.68 is a polynomial algorithm for satisfiability: evaluation of the truth value of a propositional formula under any particular interpretation is very efficient. Searching for a satisfying interpretation has been replaced by nondeterministic choice of the correct answer.

This shows that the problem of satisfiability in the propositional calculus is in the class $NP$ of problems solvable by a Nondeterministic algorithm in Polynomial time. It is not known if satisfiability is in the class $P$ of problems solvable in Polynomial time by a deterministic algorithm. Though the clairvoyance of the nondeterministic algorithm seems to be a powerful tool for defining efficient algorithms, no one has been able to prove that there exists a problem in $NP$ which is not in $P$. This is called the $P=NP$-problem.

The evidence is overwhelming that $P \neq NP$. In 1971, S. A. Cook proved that satisfiability is an $NP$-complete problem, that is, if satisfiability is in $P$, then every problem in $NP$ is in $P$. Since then, hundreds of problems have been shown to be $NP$-complete. The discovery of a deterministic polynomial-time algorithm for any of these problems, including satisfiability, implies the existence of such an algorithm for all of them! Many of these problems are famous and have been studied for years by researchers seeking efficient algorithms; thus it is highly unlikely that $P=NP$ and hence it is highly unlikely that there is a polynomial algorithm for satisfiability.

Consider now the complementary problem, unsatisfiability or validity. Unsatisfiability is ostensibly a much more difficult problem than satisfiability, because to prove unsatisfiability, we have to show that there is no satisfying interpretation. Unsatisfiability is in the class $co-NP$ of problems whose complement (here satisfiability) is in $NP$. It can be shown that $co-NP=NP$ if and only if unsatisfiability is in $NP$. It is not known if there is a nondeterministic polynomial decision procedure for unsatisfiability, and hence it is also not known if $co-NP=NP$ or not.

**Hard examples for resolution**

The complexity of specific algorithms has been extensively studied. It can be shown that methods of truth tables, semantic tableaux and resolution are all of exponential complexity by exhibiting families of formulas on which the algorithms are inefficient. Here we give an example of this area of research by defining a family of sets of clauses for which resolution is exponential.

Let $G$ be an undirected graph. Label the nodes with 0 or 1 and the edges with distinct atoms. The following graph will be used as an example throughout this section.
4.4 Complexity

Definition 4.69 The parity of a number is 0 if it is even and 1 if it is odd. \( \Pi(C) \), the \textit{parity} of a clause \( C \), is the parity of the number of complemented literals in \( C \). \( \Pi(v) \), the \textit{parity} of an interpretation \( v \), is the parity of the number of atoms assigned \( T \) in \( v \).

With each graph we associate a set of clauses.

Definition 4.70 Let \( n \) be a node of \( G \), let \( b_n \) be the label (0 or 1) of \( n \) and let \( P(n) = \{ p_1, \ldots, p_k \} \) be the set of atoms labeling edges incident with \( n \). \( C(n) \), the \textit{set of clauses associated with} \( n \), is the set of clauses whose literals are all the atoms in \( P(n) \), some of which are negated so that \( \Pi(C) \neq b_n \). \( v_n \) is an \textit{interpretation associated with} \( n \) if \( v_n \) assigns truth values to exactly the literals in \( C(n) \).

Example 4.71 The sets of clauses associated with the four nodes of the graph are (clockwise from the upper-left corner):

- \{ \bar{p}q, \bar{p} \bar{q} \}
- \{ \bar{p}rs, \bar{p}\bar{r}s, \bar{p}r\bar{s} \}
- \{ \bar{q}rt, \bar{q}\bar{r}t, q\bar{r}t, \bar{q}\bar{r}\bar{t} \}

Checking some of the clauses against the definition:

\[ \Pi(\bar{p}rs) = 0 \neq 1 = b_n. \]
\[ \Pi(\bar{q}rt) = 1 \neq 0 = b_n. \]

Lemma 4.72 \( v_n \) satisfies all clauses in \( C(n) \) if and only if \( \Pi(v_n) = b_n \).

Proof: Suppose \( \Pi(v_n) \neq b_n \). Consider the clause \( C \) defined by:

\[ l_i = p_i \text{ if } v(p_i) = F \quad \text{ and } \quad l_i = \bar{p}_i \text{ if } v(p_i) = T. \]

Then

\[ \Pi(C) = \text{ (by definition) } \]
\[ \text{parity of negated atoms of } C = \text{ (by construction) } \]
\[ \text{parity of literals assigned } T = \text{ (by definition) } \]
\[ \Pi(v_n) \neq b_n. \text{ (by assumption) } \]
so \( C \in C(n) \). But \( v_n(C) = F \) since \( v_n \) assigns \( F \) to each literal \( l_i \in C \), therefore \( v_n \) does not satisfy all clauses in \( C(n) \).

We leave the proof of the converse as an exercise.

**Example 4.73** Consider an assignment to the atoms adjacent to the upper right node \( n \) defined by \( v(p) = v(r) = v(s) = T \). For this interpretation, \( \Pi(v) = 1 = b_n \) and it is easy to see that \( v(C) = T \) for all clauses in \( C(n) \). Conversely, \( v(p) = v(r) = v(s) = F \) is an interpretation such that \( \Pi(v) = 0 \neq b_n \) and \( v(prs) = F \) so \( v \) does not satisfy all clauses in \( C(n) \).

**Lemma 4.74** If \( \sum_{n \in G} b_n = 1 \) where \( \sum \) is modulo two sum, then \( C(G) = \bigcup_{n \in G} C(n) \) is unsatisfiable.

**Proof:** By the previous lemma, if \( v \) is an arbitrary model for \( C(G) \), then \( \Pi(v_n) = b_n \) where \( v_n \) is \( v \) restricted to the literals incident with \( n \), so \( \sum_{n \in G} \Pi(v_n) = \sum_{n \in G} b_n \). Suppose that an atom labels an edge whose endpoints are \( n' \) and \( n'' \). Then each assignment to a atom appears twice in the sum, once for \( v(n') \) and once for \( v(n'') \). Thus the number of literals assigned \( T \) in \( \sum_{n \in G} \Pi(v_n) \) is even and \( 0 = \sum_{n \in G} \Pi(v_n) = \sum_{n \in G} b_n \). Therefore, if \( \sum_{n \in G} b_n = 1 \) there cannot be a model for \( C(G) \) so the set of clauses is unsatisfiable.

**Example 4.75** Let \( n' \) be the upper left corner and \( n'' \) the upper right corner and \( v \) be such that \( v(p) = T \). Then \( v(p) = T \) is counted twice and its contribution to the total parity is 0.

There are unsatisfiable sets of clauses associated with arbitrarily large graphs. If the graphs have just a few edges incident with each node—such as a grid with at most four edges per node—the size of the set of clauses will be small. With \( N \) nodes and four edges per node, there will be at most \( 8N \) clauses of four literals each. An appropriate set of clauses was defined in 1968 by G. S. Tseitin, but not until 1987 was the following theorem finally proved:

**Theorem 4.76 (Haken and Urquhart)** For arbitrarily large \( N \), there is a graph and a set of associated clauses of size about \( N \), such that the number of distinct clauses created in any resolution refutation is greater than \( 2^{\mathcal{O}} \) for some fixed \( c > 0 \).

The proof is extremely difficult and beyond the scope of this textbook. The difficulty stems from the choice in the resolution procedure—at every step, any two clashing clauses can be chosen. A combinatorial argument is used to show that no sequence of choices gives a non-exponential refutation.

Ironically, the formulas obtained by Tseitin’s clauses are problematic only when represented in clausal form. They can be expressed as sets of equivalences, and there is
a simple and efficient algorithm for checking the validity of such formulas. Therefore, the theorem shows only that resolution is an exponential algorithm. It does not preclude the (unlikely) existence of a polynomial algorithm for satisfiability.

4.5 Exercises

1. A formula is in disjunctive normal form (DNF) iff it is a disjunction of conjunctions of literals. Show that every formula is equivalent to one in DNF.

2. A formula $A$ is in complete DNF iff it is in DNF and each propositional letter in $A$ appears in a literal in each conjunction. For example, $(p \land q) \lor (\bar{p} \land q)$ is in complete DNF. Show that every formula is equivalent to one in complete DNF.

3. Write a program to transform a formula into an equivalent formula in complete DNF.

4. Simplify the following sets of literals, that is, for each set $S$ find a simpler set $S'$, such that $S \approx S'$.
   
   $\{pq, qr, rs, \bar{p}\bar{s}\}$,
   $\{pqr, \bar{q}, \bar{p}\bar{r}, qs, \bar{p}\bar{s}\}$,
   $\{pqrs, \bar{q}rs, \bar{p}rs, qs, \bar{p}\bar{s}\}$,
   $\{pq, qrs, \bar{p}qrs, \bar{r}, q\}$.

5. Given the set of clauses $\{\bar{p}qr, pr, qr, \bar{r}\}$ construct two refutations: one by resolving the literals in the order $\{p, q, r\}$ and the other in the order $\{r, q, p\}$.

6. Transform the set of formulas
   
   \[
   \begin{align*}
   p & \rightarrow ((q \lor r) \land \neg(q \land r)), \\
   p & \rightarrow ((s \lor t) \land \neg(s \land t)), \\
   s & \rightarrow q, \\
   \neg r & \rightarrow t, \\
   t & \rightarrow s
   \end{align*}
   \]

   into clausal form and refute using resolution.

7. * The half-adder of Example 1.1 implements the pair of formulas:
   
   $s \leftrightarrow \neg(b1 \land b2) \land (b1 \lor b2)$,  
   $c \leftrightarrow b1 \land b2$.

   Transform the formulas to a set of clauses. Show that the addition of the unit clauses $\{b1, b2, \bar{s}, \bar{c}\}$ gives an unsatisfiable set while the addition of $\{b1, b2, \bar{s}, c\}$ gives a satisfiable set. Explain what this means in terms of the behavior of the circuit.

8. Prove that adding a unit clause on a new atom to a set of clauses and adding its complement to clauses in the set preserves satisfiability (the converse direction of Lemma 4.11).
9. Prove that if the set of clauses labeling the leaves of a resolution tree is satisfiable then the clause at the root is satisfiable (Theorem 4.28).

10. Prove the Shannon Expansion (Theorem 4.60) and the formula for propositional quantification (Theorem 4.64).

11. Show that $\exists r (p \lor (q \land r)) = p \lor q$ and $\forall r (p \lor (q \land r)) = p$ using BDDs (Example 4.65).

12. Implement the optimizations of the BDD algorithms discussed as the end of Section 4.3.

13. * Construct a resolution refutation for the set of clauses associated with the graph on page 97.

14. * Construct the set of Tseitin clauses corresponding to a labeled complete graph on five vertices and give a resolution refutation of the set.

15. * Show that if $\Pi(v_n) = b_n$, then $v_n$ satisfies all clauses in $C(n)$ (the converse direction of Lemma 4.72).

16. * Let $\{q_1, \ldots, q_n\}$ be literals on distinct atoms. Show that $q_1 \leftrightarrow \cdots \leftrightarrow q_n$ is satisfiable if $\{p \leftrightarrow q_1, \ldots, p \leftrightarrow q_n\}$ is satisfiable, where $p$ is a new atom. Construct an efficient decision procedure for sets of formulas whose only operators are $\neg$, $\leftrightarrow$ and $\oplus$. 
Mathematical Logic for Computer Science
Ben-Ari, M.
2001, XIV, 304 p., Softcover