Chapter 10

Binary Decision Diagrams

Contents

Imagine an application in which large propositional formulas are reused over and over again. For example, we can build other formulas from these formulas using connectives and check the new formulas for such properties as satisfiability or equivalence. To work with such applications efficiently, one needs data structures which

(1) give *a compact representation* of formulas, or the boolean functions represented by the formulas;

(2) facilitate *boolean operations* on formulas, for example, given representations of formulas $F_1, \ldots, F_n$, computing a representation of their conjunction $F_1 \wedge \ldots \wedge F_n$;

(3) facilitate *checking properties of formulas*, such as satisfiability or equivalence checking.

In this chapter we introduce binary decision diagrams (BDDs): a data structure which has many of these desired properties and is used in symbolic model checking algorithms discussed later in this book. There exists a close analogy between BDDs and the trees

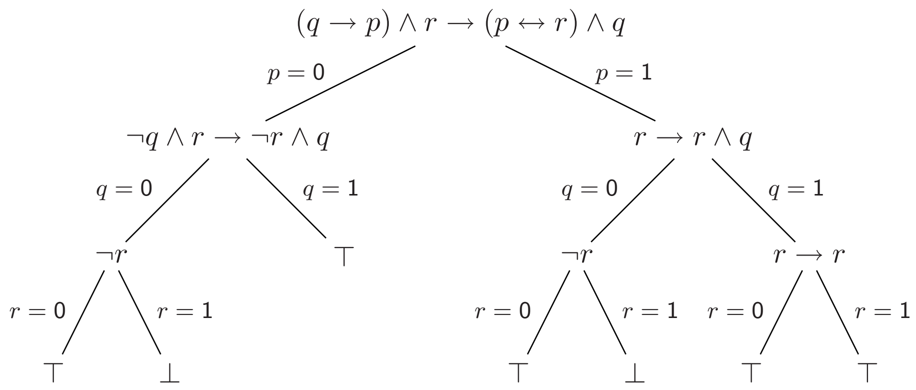


Figure 10.1: A splitting tree

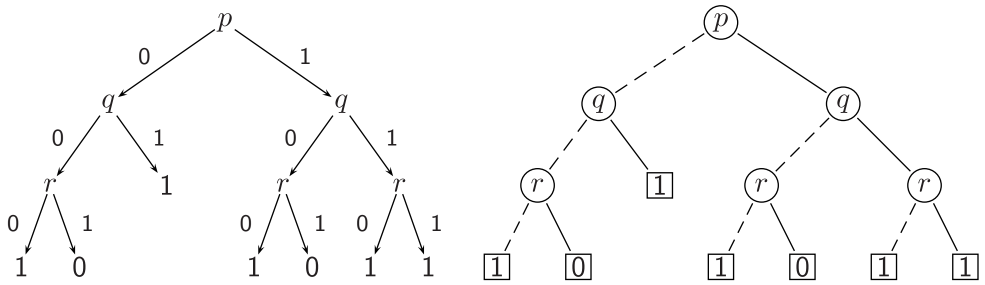


Figure 10.2: The corresponding binary decision tree, two representations

obtained by the splitting algorithm on propositional formulas. In fact, BDDs can be considered as a data structure for a compact encoding of these trees. Satisfiability checking for BDDs is trivial, but some boolean operations are difficult to implement. By imposing ordering conditions on BDDs we obtain ordered BDDs, or simply OBDDs: a special kind of BDDs that allows for boolean functions to be implemented efficiently.

10.1 Binary Decision Trees

Consider the tree obtained by applying the splitting algorithm to the formula $(q \to p) \wedge r \to (p \leftrightarrow r) \wedge q$ given in Figure 10.1. If we replace $\top$ and $\bot$ in the leaves of this tree by 1 and 0 respectively, replace all formulas in the internal nodes by the variable used for splitting at this node, and label the arcs by 0 and 1, we obtain the tree given in Figure 10.2.

This kind of tree is called a *binary decision tree*. In the sequel we will depict binary decision trees in a different more compact form, see the tree on the right of Figure 10.2.

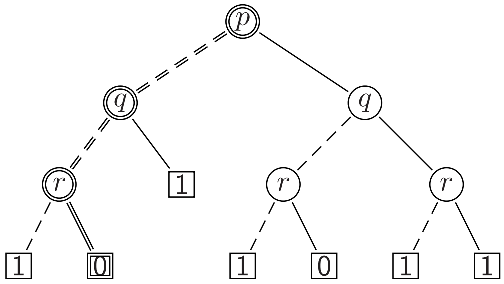


Figure 10.3: Evaluating a formula using its binary decision tree

We remove the labels $0, 1$ from the arcs. Instead, the arcs labeled by 0 are drawn using dashed lines, while those labeled by 1 using solid lines. Leaves labeled by 0 and 1 in D will always be denoted respectively by $\boxed{0}$ and $\boxed{1}$. Any internal node in a binary decision tree represents a "decision" or a "test" on a particular variable.

The binary decision tree of Figure 10.2 is obtained from the formula $(q \to p) \wedge r \to (p \leftrightarrow r) \wedge q$. We can obtain the same binary decision tree by applying splitting to some other formulas, for example $\neg\neg((q \to p) \wedge r \to (p \leftrightarrow r) \wedge q)$. Therefore, the same binary decision tree can represent different formulas (in fact, any binary decision tree represents an infinite number of formulas), so the information about the syntax of the formula is lost. Nonetheless, the binary decision tree contains the complete *semantical* information about the formula. For example, to evaluate this formula in the interpretation $\{p \mapsto 0, q \mapsto 0, r \mapsto 1\}$, we follow the path from the root of this tree corresponding to the decisions $p = 0$, $q = 0$, and $r = 1$, and read off the value 0 in the leaf, see Figure 10.3. This path is shown using double lines.

DEFINITION 10.1 (Binary Decision Tree) A *binary decision tree* is a tree d such that

(1) the internal nodes of d are labeled by variables;

(2) the leaves of d are labeled by 0 and 1;

(3) every internal node in d has exactly two children, the two arcs from d to the children are labeled by 0 (shown as a dashed line) and by 1 (shown as a solid line).

(4) nodes on every path in d have unique labels, i.e. every two different nodes on a single path are labeled by distinct variables. ❏

The last condition means the following: if a branch contains a test on an variable p, then this branch contains no further tests on p.

10.2 If-Then-Else Normal Form

In this section we formally define the correspondence between propositional formulas, or boolean functions, on one hand, and binary decision trees, on the other hand. To this end, let us introduce the following abbreviation. For all formulas F_1, F_2, F_3 we denote by $\mathit{if}\ F_1\ \mathit{then}\ F_2\ \mathit{else}\ F_3$ the formula $(F_1 \to F_2) \land (\neg F_1 \to F_3)$. Alternatively, we can consider $\mathit{if}\ \ldots\ \mathit{then}\ \ldots\ \mathit{else}\ \ldots$ as a new ternary connective, see Exercise 10.10. First, we show how one can convert an arbitrary propositional formula into a binary decision tree using formulas of special form built using $\mathit{if}\ \ldots\ \mathit{then}\ \ldots\ \mathit{else}\ \ldots$.

DEFINITION 10.2 (If-Then-Else Normal Form) The notion of formula in *if-then-else normal form* is defined inductively as follows:

(1) $\top$ and $\bot$ are formulas in if-then-else normal form;

(2) if F_1, F_2 are formulas in if-then-else normal form not containing occurrences of a variable p, then $\mathit{if}\ p\ \mathit{then}\ F_1\ \mathit{else}\ F_2$ is in if-then-else normal form too.

A formula G is said to be an *if-then-else normal form* of a formula F if G is equivalent to F and G is in if-then-else normal form. ❏

For example, $\mathit{if}\ p\ \mathit{then}\ \bot\ \mathit{else}\ \top$ is an if-then-else normal form of the formula $\neg p$. The formula $\mathit{if}\ \top\ \mathit{then}\ \top\ \mathit{else}\ \bot$ is not in if-then-else normal, since the "if-part" of $\mathit{if}\ \ldots\ \mathit{then}\ \ldots\ \mathit{else}\ \ldots$ must be an atom. An if-then-else normal form is not unique, for example, both $\mathit{if}\ p\ \mathit{then}\ \top\ \mathit{else}\ \top$ and $\top$ are if-then-else normal forms of $\top$.

For every binary decision tree b and internal node n in it, denote by $neg(n)$ the subtree rooted at the dashed arc coming from n. Likewise, $pos(n)$ will denote the subtree rooted at the solid arc. If the variable at the node n is p, then $neg(n)$ is the tree corresponding to the decision $p = 0$, while $pos(n)$ is the tree corresponding to the decision $p = 1$.

DEFINITION 10.3 ($form(d)$) Let d be a binary decision tree. For every node n in b we define inductively a propositional formula F_n as follows.

(1) If n is $\boxed{0}$, then $F_n \stackrel{\text{def}}{=} \bot$; if n is $\boxed{1}$, then $F_n \stackrel{\text{def}}{=} \top$.

(2) If n is an internal node labeled by a variable p, then

$$F_n \stackrel{\text{def}}{=} \mathit{if}\ p\ \mathit{then}\ F_{pos(n)}\ \mathit{else}\ F_{neg(n)}.$$

We denote by $form(b)$ the formula F_r, where r is the root of b. We say that d is a *binary decision tree for a formula* F if F is equivalent to $form(d)$. ❏

$$\begin{array}{llllllllll}
\mathit{if} & p & \mathit{then} & \mathit{if} & q & \mathit{then} & \mathit{if} & r & \mathit{then} & \top \\
 & & & & & & & & \mathit{else} & \top \\
 & & & & & \mathit{else} & \mathit{if} & r & \mathit{then} & \bot \\
 & & & & & & & & \mathit{else} & \top \\
 & & \mathit{else} & \mathit{if} & q & \mathit{then} & \top & & & \\
 & & & & & \mathit{else} & \mathit{if} & r & \mathit{then} & \bot \\
 & & & & & & & & \mathit{else} & \top
\end{array}$$

Figure 10.4: Formula $\mathit{form}(b)$ for the decision tree of Figure 10.2

EXAMPLE 10.4 Consider the binary decision tree d of Figure 10.2 on page 134. The formula $\mathit{form}(d)$ for this decision tree is given in Figure 10.4. The binary decision tree d is the binary decision tree for $\mathit{form}(d)$, but also for every formula equivalent to it, for example for the formulas

$$\begin{array}{l}(q \to p) \land r \to (p \leftrightarrow r) \land q, \\ \neg q \to \neg r,\end{array}$$

or any formula other equivalent to this formula. ❏

Definition 10.3 and Example 10.3 show that, in a way, binary decision trees represent formulas in if-then-else normal form.

Let us show that every formula has an if-then-else normal form. Intuitively, it is obvious since the splitting algorithm applied to a formula F builds a binary decision tree for F, and this binary decision tree is an if-then-else normal form of F. However, let us a give a formal proof. First, we need a simple lemma.

LEMMA 10.5 *For every formula F and atom p the formulas $p \to F$ and $p \to F_p^\top$ are equivalent. Likewise, the formulas $\neg p \to F$ and $\neg p \to F_p^\bot$ are equivalent.*

PROOF. We prove only the first statement. Let I be any interpretation. If $I \not\models p$, then $I \models p \to F$ and $I \models p \to F_p^\top$, so $I \models (p \to F) \leftrightarrow (p \to F_p^\top)$. Suppose now $I \models p$. Then $I \models p \leftrightarrow \top$, so by Equivalent Replacement Lemma 3.8 we have $I \models (p \to F) \leftrightarrow (p \to F_p^\top)$ too. In both cases we have $I \models (p \to F) \leftrightarrow (p \to F_p^\top)$, so $p \to F$ is equivalent to $p \to F_p^\top$. ❏

COROLLARY 10.6 *For all formulas F, G and variable p the formula $\mathit{if}\ p\ \mathit{then}\ F\ \mathit{else}\ G$ is equivalent to $\mathit{if}\ p\ \mathit{then}\ F_p^\top\ \mathit{else}\ G_p^\bot$.*

PROOF. We know that $\mathit{if}\ p\ \mathit{then}\ F\ \mathit{else}\ G$ is an abbreviation for $(p \to F) \land (\neg p \to G)$. By Lemma 10.5 this formula is equivalent to $(p \to F_p^\top) \land (\neg p \to G_p^\bot)$, that is, $\mathit{if}\ p\ \mathit{then}\ F_p^\top\ \mathit{else}\ G_p^\bot$. ❏

procedure $bdt(F)$
input: propositional formula F
output: a binary decision tree
parameters: function $select\_variable$
begin
  F := $simplify(F)$
  if $F = \bot$ **then return** $\boxed{0}$
  if $F = \top$ **then return** $\boxed{1}$
  p := $select\_variable(F)$
  return p
    $bdt(F_p^{\bot})$ $bdt(F_p^{\top})$
end

Figure 10.5: The algorithm for building a binary decision tree

Note that the formulas $F_p^{\top}$ and $G_p^{\bot}$ have no occurrences of p, so this corollary can directly be used for building if-then-else normal forms.

THEOREM 10.7 *Every formula F has an if-then-else normal form.*

PROOF. By induction on the number of variables in F. When F has no occurrences of variables, either $F \equiv \top$, and then $\top$ is an if-then-else normal form of F; or $F \equiv \bot$, and then $\bot$ is an if-then-else normal form of F.

Suppose now that some variable p occurs in F. Evidently, F is equivalent to $if\ p\ then\ F\ else\ F$. By Corollary 10.6, F is equivalent to $if\ p\ then\ F_p^{\top}\ else\ F_p^{\bot}$. The formulas $F_p^{\top}$ and $F_p^{\bot}$ have a smaller number of variables than F, so by the induction hypothesis they have if-then-else normal forms F_1 and F_2. But then F is equivalent to $if\ p\ then\ F_1\ else\ F_2$. ❑

This proof shows that, in order to build a binary decision tree for a formula F containing at least one variable, we have to select a variable p in F, build binary decision trees b_1 for $F_p^{\bot}$ and b_2 for $F_p^{\top}$, and then build a binary tree having p at the root, b_1 as the left subtree, and b_2 as the right subtree. This process is, indeed, very similar to the splitting algorithm used for checking satisfiability of propositional formulas and can be summarized as follows.

ALGORITHM 10.8 (Binary Decision Tree Construction) An algorithm for building a binary decision tree from a propositional formula F is given in Figure 10.5. It is parametrized by a function $select\_variable$ returning a variable occurring in F. The function $simplify$ simplifies formulas using the rewrite rules of Figure 4.5. ❑

Let us note some properties of binary decision trees.

(1) In general, the size of a binary decision tree for a formula F is exponential in the size of F.

(2) Satisfiability and validity checking on binary decision trees can be done in time linear in the size of the tree.

Indeed, given a binary decision tree for a formula F, we can verify if F is satisfiable by simply inspecting all the leaf nodes of the tree: F is satisfiable if and only if at least one of them is $\boxed{1}$. Likewise, F is valid if and only if all leaves in F are $\boxed{1}$.

On a negative side, we have the following.

(1) It is unclear how to implement equivalence checking efficiently.

(2) It is unclear how to implement some boolean operations, for example, conjunction.

In addition, binary decision trees are not especially compact.

10.3 Binary Decision Diagrams

Binary decision trees can be turned into more compact data structures by eliminating two obvious kinds of redundancies. To illustrate them, consider the rightmost subtree rooted at the node r ("test" on r) in Figure 10.2 on page 134. The value of the formula is 1 independently of these tests. Therefore, these tests can eliminated, so that the whole subtree is replaced by 1. This pruning operation is called *elimination of redundant tests*.

Another kind of redundancy is due to repeated occurrences of the same subtrees. We can merge them into one subtree, thus obtaining a dag instead of a tree. For example, in Figure 10.2 there are two identical subtrees rooted at r. This operation is called *merging isomorphic subdags*. By eliminating redundant tests and merging isomorphic subdags we obtain a data structure known as a *binary decision diagram*.

DEFINITION 10.9 (BDD) A *binary decision diagram*, or simply *BDD* is a rooted dag d such that d satisfies the properties of Definition 10.1 on page 135 plus the following two properties:

(5) for every node n, its left and right subdags are distinct;

(6) every pair of subdags of d rooted at two different nodes n_1, n_2 are non-isomorphic. ❏

These two conditions formalize the properties that d contains no redundant tests and that the isomorphic subdags of d are merged.

Given a binary decision diagram d, the formula $form(d)$ is defined exactly as in Definition 10.3 for decision trees. We also say that a BDD d is *for a formula F* if F is equivalent to $form(d)$.

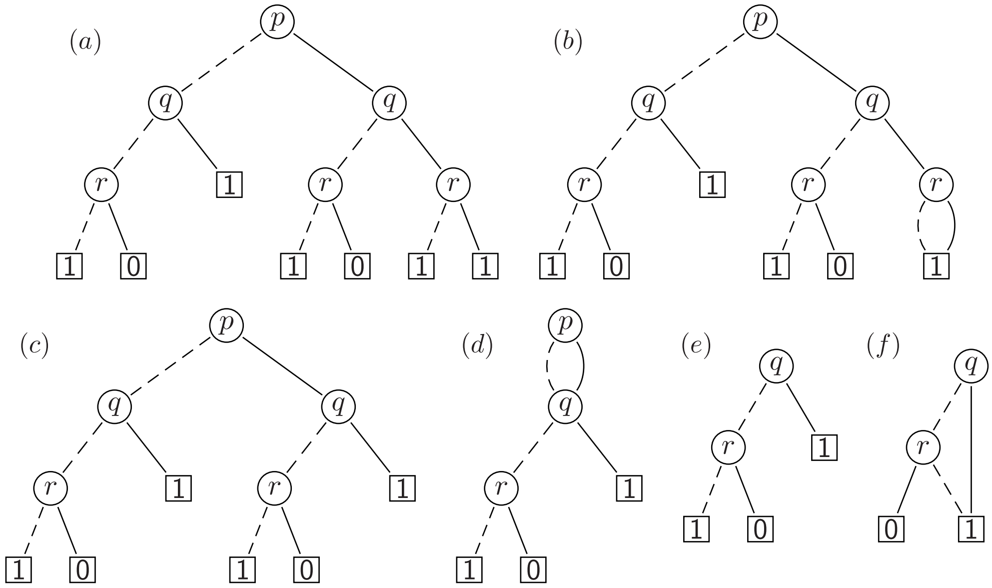


Figure 10.6: Transformation of a binary decision tree into a BDD

There exists a straightforward algorithm for building a BDD from a binary decision tree. In order to satisfy conditions 5–6 of Definition 10.9 we can apply transformations to eliminate redundancies corresponding to these conditions. These transformations are:

(1) *elimination of redundant tests*: if there exists a node n such that $neg(n)$ and $pos(n)$ are the same dag, then remove this node, i.e., replace the subdag rooted at n by $neg(n)$;

(2) *merging isomorphic subdags*: if the subdags rooted at two different nodes n_1 and n_2 are isomorphic, then merge them into one, i.e., remove the subdag rooted at n_2 and replace all arcs to n_2 by arcs to n_1.

EXAMPLE 10.10 Transformation of a binary decision tree into a BDD is illustrated in Figure 10.6. In this figure, dag (b) is obtained from (a) by merging isomorphic subdags rooted at 1. Dag (c) is obtained from (b) by removing a redundant test on r. Dag (d) is obtained from (c) by merging isomorphic subdags rooted at q. Dag (e) is obtained from (d) by removing a redundant test on p. Finally, dag (f) is obtained from (e) by merging isomorphic subdags rooted at 1. ❏

By applying elimination of redundant tests and merging isomorphic subdags to a binary decision tree in a bottom-up fashion, one can build a BDD from a binary decision tree. One can argue that finding isomorphic subdags is a hard problem. One can show that it is enough to find isomorphic subdags of a very special form. However, we will not do this, since we defined binary decision trees only for an illustration. One can build a BDD directly from a propositional formula, as we will show in the next section.

Let us note some properties of BDDs.

(1) In general, the size of a BDD for a formula A is exponential in the size of A.

(2) Satisfiable and validity checking on binary decision trees can be done in *constant time*.

Indeed, it is not hard to argue that the formula A is unsatisfiable if and only if b consists of a single node $\boxed{0}$. Likewise, A is valid if and only if b consists of a single node $\boxed{1}$.

Thus, BDDs have some advantages over binary decision trees. However, some problems still remain, namely

(1) it is unclear how to implement equivalence checking efficiently;

(2) it is unclear how to implement some boolean operations, for example, conjunction.

10.4 Ordered BDDs

The shape and size of a BDD depend on the order of tests in it. Different orders can result in a drastic increase or decrease in size. When we build a BDD, the order of selecting variables on different branches of a tree may be different. In this section we study ordered BDDs in which the order is the same on all branches. When the order is fixed in advance, there is a unique ordered BDD corresponding to this order. If we have several boolean functions represented by ordered BDDs one can build, using a relatively simple algorithm, new ordered BDDs representing various combinations of these boolean functions, for example, their conjunction.

DEFINITION 10.11 (OBDD) Let $>$ be a linear order on variables and d be a BDD. We say that d *respects* $>$, if for every node n_1 labeled by a variable p_1 and its child n_2 labeled by a variable p_2 we have $p_1 > p_2$. A BDD is called *ordered*, or simply an *OBDD*, if it respects some order. We call an *OBDD for a formula F and order $>$* any OBDD for F which respects $>$. ❏

An example of a BDD which is not ordered in given in Figure 10.7. This BDD contains a node q having r as a child, and also a node r having q as a child, but there is no order $>$ such that $q > r$ and $r > q$.

Our next task is to show that for every propositional formula F and a linear order $>$ on its variables, there exists a unique OBDD for F and $>$.

LEMMA 10.12 *Let p be a variable and F_1, F_2, G_1, G_2 be formulas not containing p. Then $(\textit{if } p \textit{ then } F_1 \textit{ else } F_2) \equiv (\textit{if } p \textit{ then } G_1 \textit{ else } G_2)$ if and only if $F_1 \equiv G_1$ and $F_2 \equiv G_2$.*

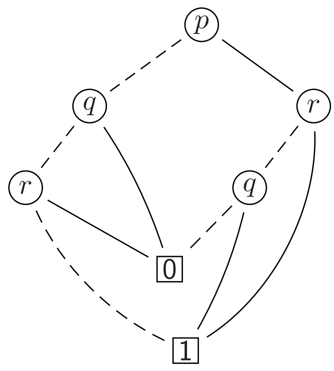


Figure 10.7: A BDD which is not ordered

PROOF. ($\Rightarrow$) Suppose $\mathit{if}\ p\ \mathit{then}\ F_1\ \mathit{else}\ F_2 \equiv \mathit{if}\ p\ \mathit{then}\ G_1\ \mathit{else}\ G_2$. We will prove $F_1 \equiv G_1$ (the proof of $F_2 \equiv G_2$ is analogous). We have to show that F_1 and G_1 have the same models. Take an arbitrary model I of F_1. Define I' by

$$I'(q) \stackrel{\text{def}}{=} \begin{cases} I(q), & \text{if } p \neq q; \\ 1, & \text{if } p = q. \end{cases}$$

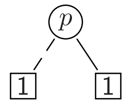
Since p does not occur in F_1 and I' agrees with I on all variables different from p, we have $I' \models F_1$. Since $I' \models p$ and $I' \models F_1$, we also have $I' \models \mathit{if}\ p\ \mathit{then}\ F_1\ \mathit{else}\ F_2$. This and $(\mathit{if}\ p\ \mathit{then}\ F_1\ \mathit{else}\ F_2) \equiv (\mathit{if}\ p\ \mathit{then}\ G_1\ \mathit{else}\ G_2)$ implies $I' \models \mathit{if}\ p\ \mathit{then}\ G_1\ \mathit{else}\ G_2$. From this and $I' \models p$ it follows that $I' \models G_1$. Since p does not occur in G_1, we have $I \models G_1$. So we proved that every model of F_1 is a model of G_1. The proof that every model of G_1 is a model of F_1 is symmetric.

($\Leftarrow$) Suppose that $F_1 \equiv G_1$ and $F_2 \equiv G_2$. Then by Equivalent Replacement Theorem 3.9 we have $(\mathit{if}\ p\ \mathit{then}\ F_1\ \mathit{else}\ F_2) \equiv (\mathit{if}\ p\ \mathit{then}\ G_1\ \mathit{else}\ G_2)$. ❑

This lemma implies that, when the order on variables is fixed, every formula has a unique OBDD. *In the rest of this chapter we assume that $>$ is a fixed order on variables.* Instead of saying "OBDD for F and $>$" we will simply say *"OBDD for F"*.

THEOREM 10.13 (Canonicity of OBDDs) *Let d_1, d_2 be two OBBDs for a formula F. Then d_1 is isomorphic to d_2.*

PROOF. By induction on the number of all variables occurring in F. When the number is 0, that is F have no variables, then either $F \equiv \top$ or $F \equiv \bot$. We consider only the first case, the second one is similar. We claim that $\boxed{1}$ is the only OBBD for F. Suppose, by contradiction, that there exists another OBDD d for F. Evidently, d cannot contain $\boxed{0}$. But then d must contain a node of the form



and hence contain a redundant test. Contradiction.

Assume now that F contains at least one variable. Let p be the maximal (w.r.t. $>$) variable occurring in F. Evidently, $F \equiv (\mathit{if}\ p\ \mathit{then}\ F\ \mathit{else}\ F)$. By Lemma 10.5 we obtain $F \equiv (\mathit{if}\ p\ \mathit{then}\ F_p^{\top}\ \mathit{else}\ F_p^{\perp})$. Consider two cases:

(Case: $F_p^{\top} \equiv F_p^{\perp}$.) In this case From $F \equiv (\mathit{if}\ p\ \mathit{then}\ F_p^{\top}\ \mathit{else}\ F_p^{\perp})$ it follows that $F \equiv F_p^{\top}$, so F has the same OBDDs as $F_p^{\top}$. But by the induction hypothesis $F_p^{\top}$ has a unique OBDD.

(Case: $F_p^{\top} \not\equiv F_p^{\perp}$.) We claim that in this case every OBDD d for F has p in the root. Denote $\mathit{form}(d)$ by G, then $F \equiv G$. Since the root of d does not have p, then G does not contain p. Since $F \equiv G$, we also have $(\mathit{if}\ p\ \mathit{then}\ F\ \mathit{else}\ F) \equiv (\mathit{if}\ p\ \mathit{then}\ G\ \mathit{else}\ G)$. By Lemma 10.5 this implies $(\mathit{if}\ p\ \mathit{then}\ F_p^{\top}\ \mathit{else}\ F_p^{\perp}) \equiv (\mathit{if}\ p\ \mathit{then}\ G\ \mathit{else}\ G)$. By Lemma 10.12 we have $F_p^{\top} \equiv G$ and $F_p^{\perp} \equiv G$, so $F_p^{\top} \equiv F_p^{\perp}$, which contradicts to the case assumption. This implies that every OBDD for F has p in the root. Take any two OBDDs for F. They have the form

Then $F \equiv (\mathit{if}\ p\ \mathit{then}\ \mathit{form}(d_2)\ \mathit{else}\ \mathit{form}(d_1))$ and $F \equiv (\mathit{if}\ p\ \mathit{then}\ \mathit{form}(t_2)\ \mathit{else}\ \mathit{form}(t_1))$. By Lemma 10.12 this implies $\mathit{form}(d_2) \equiv \mathit{form}(t_2)$ and $\mathit{form}(d_1) \equiv \mathit{form}(t_1)$. The induction hypothesis yields that d_2 is isomorphic to t_2 and d_1 is isomorphic to t_1. But then the two OBDDs for F are isomorphic too. ❑

Note that equivalent formulas have the same BDDs, so this theorem implies that OBBDs give a canonical representation of formulas up to equivalence (or of boolean functions computed by these formulas). This means that two formulas are equivalent if and only if they have isomorphic OBDDs.

The proof of this theorem gives us an algorithm for finding the OBDD for a formula F. Before defining this algorithm, let us make a few comments about algorithms on OBDDs in general.

First, all algorithms working on OBDDs build OBDDs bottom-up. This makes it easier to share isomorphic subdags and discover redundant tests. This will be clear when we define a procedure for integrating a node in a dag.

Second, when an OBDD-based algorithm works with several OBDDs simultaneously, these OBDDs are integrated in a *global dag* D which stores several OBDDs. This dag may not be an OBDD, because it is not necessarily rooted. In this global dag, all isomorphic subdags are shared, i.e., every subdag has a *unique occurrence* in D. This assumption facilitates integration of new dags in D. In particular, there exists only one occurrence of $\boxed{0}$ and only one occurrence of $\boxed{1}$ in the global dag. For example, suppose that we are dealing with two formulas $p \vee q$ and $p \rightarrow q$. Then the global dag will contain OBDDs for both formulas, see Figure 10.8. If n is a node in the global dag, then the dag rooted at n is an OBBD d. If d is an OBDD for a formula F, we say that n *represents* d. For example, in the global dag of Figure 10.8 the node rooted at q represents the formula $\mathit{if}\ q\ \mathit{then}\ \top\ \mathit{else}\ \perp$ and also any formula equivalent to it, for instance, q. We assume that the global dag always contains both $\boxed{0}$ and $\boxed{1}$.

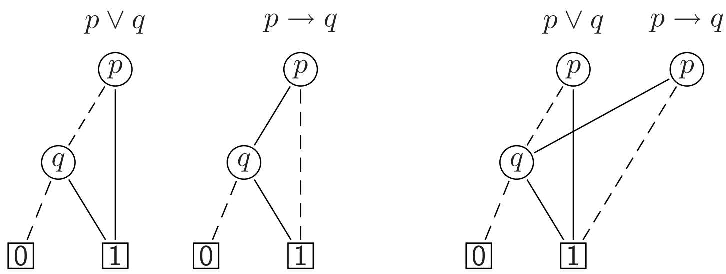


Figure 10.8: Two OBDDs and a global dag containing both of them

procedure $integrate(n_1, p, n_2, D)$
parameters: global dag D
input: nodes n_1, n_2 in D representing formulas F_1, F_2, variable p
output: node n in (modified) D representing *if* p *then* F_1 *else* F_2
begin
  if $n_1 = n_2$ **then** **return** n_1;
  if D contains a node n having the form p
  n_1 n_2
    then **return** n;
  add to D a new node n of the form p;
  n_1 n_2
  return n
end

Figure 10.9: An algorithm for integrating a node into a dag

Let us introduce a helpful definition. Let n be a node in the global dag D. We say that n *represents a formula* F *in* D if the subdag of D rooted at n is a BDD for F.

Let us define a procedure $integrate$ which integrates a new OBDD in the global dag D. This procedure will be used in all OBDD algorithms. It is given in Figure 10.9. The procedure $integrate$ first tries to check whether the OBDD to be built is already included in the dag, and returns the root of this OBDD if it is. Otherwise, it builds a new node. Note that this procedure can be implemented by building a map containing entries $(p, n_1, n_2) \mapsto n$ such that the dag D contains a node n labeled p and having n_1 and n_2 as the children. If this map is implemented using a hash table, then $integrate$ works in constant time.

ALGORITHM 10.14 (OBDD construction) The algorithm for building the OBDD for a a propositional formula F is given in Figure 10.10. The function $max\_variable(F)$ returns the maximal w.r.t. $>$ variable occurring in F. The function $simplify$ simplifies the formula

procedure $obdd(F)$
input: propositional formula F
parameters: global dag D
output: a node n in (modified) D which represents F
begin
  F := $simplify(F)$
  if $F = \bot$ **then** **return** $\boxed{0}$
  if $F = \top$ **then** **return** $\boxed{1}$
  p := $max\_variable(F)$
  n_1 := $obdd(F_p^\bot)$
  n_2 := $obdd(F_p^\top)$
  return $integrate(n_1, p, n_2, D)$
end

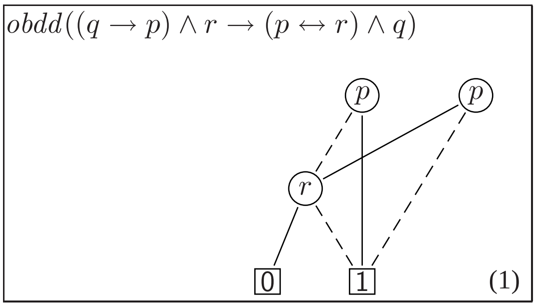
Figure 10.10: An algorithm for building an OBDD

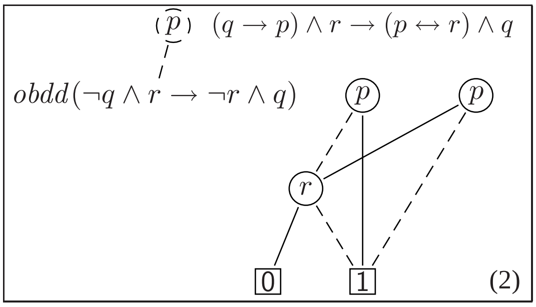
using the rewrite rules of Figure 4.5. ❏

EXAMPLE 10.15 We illustrate this algorithm on the input formula $(q \to p) \wedge r \to (p \leftrightarrow r) \wedge q$ by giving a sequence of snapshots showing the global dag and intermediate steps of the computation. Each snapshot is put in a box. For convenience, snapshots are enumerated, their numbers are shown in the lower right corner.

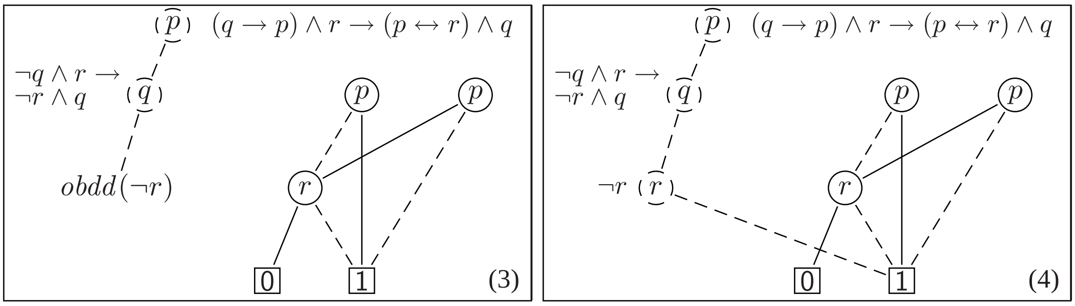
We assume the initial global dag shown in snapshot (1) below. Calls of the algorithm on a formula F are denoted by $obdd(F)$. To save space, we simplify F as soon as it contains occurrences of $\bot$ or $\top$. If $obdd$ calls $max\_variable(p)$, then we put p in a dashed circle, see snapshot (2). This dashed circle can become a part of the dag, when $integrate$ will be called on results of the corresponding recursive calls.

We assume the order $p > q > r$, so the procedure first makes a decision on p, see snapshot (2). After substitution $\bot$ for p and simplification we obtain the formula $\neg q \wedge r \to \neg r \wedge q$.

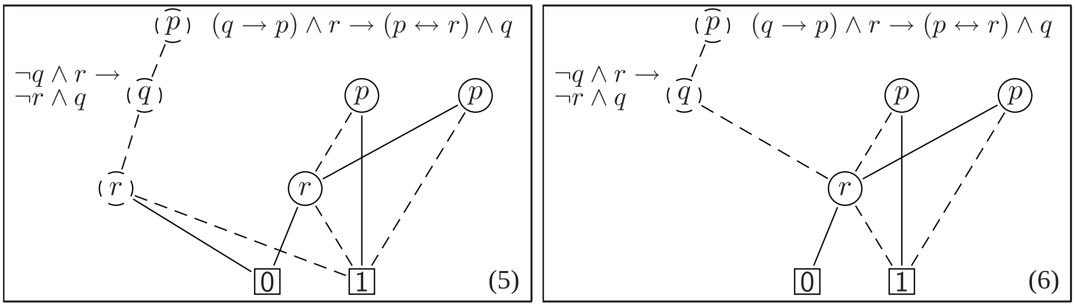




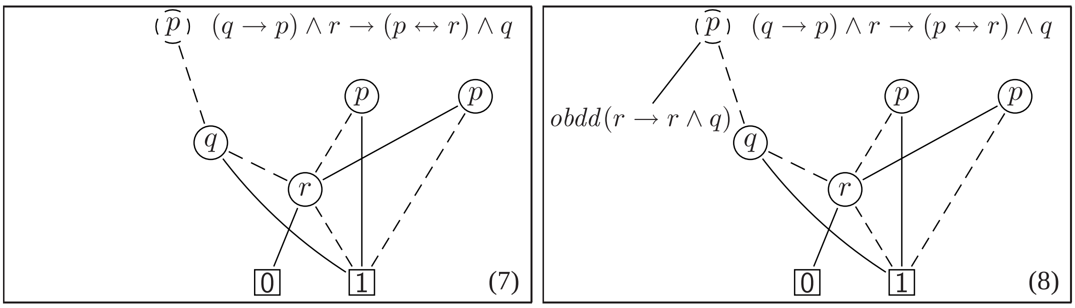
In snapshots (3) and (4) the procedure makes two more recursive calls, making decisions on q and r. When $r = 0$, the formula $\neg r$ is simplified to $\top$, so the procedure returns $\boxed{1}$.



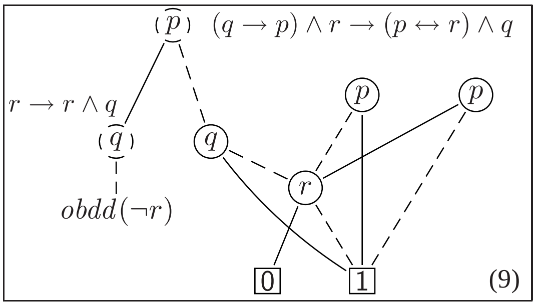
In snapshot (5) both branches of the algorithm return nodes in D, so the algorithm calls $integrate$ The result is shown in snapshot (6): the integration returned an existing node.

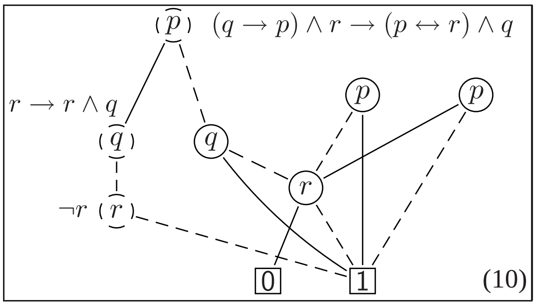


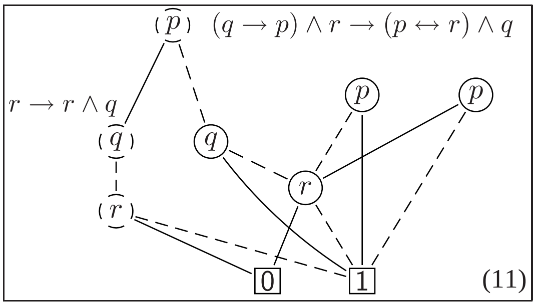
In snapshot (7) $integrate$ has been applied again, this time resulting in a new node q added to the global dag.

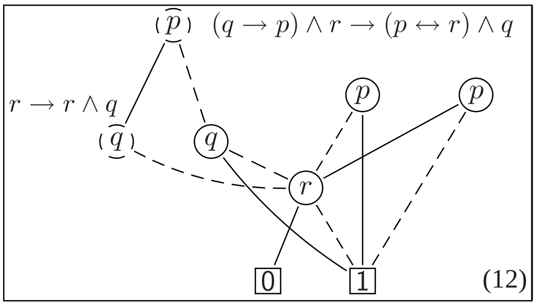


In snapshot (9) $obdd$ is again called on the formula $\neg r$, so the next few snapshots are analogous to those starting with snapshot (3).

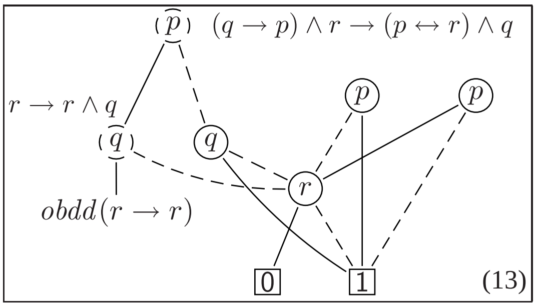


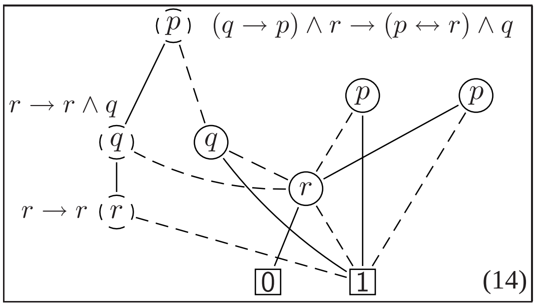




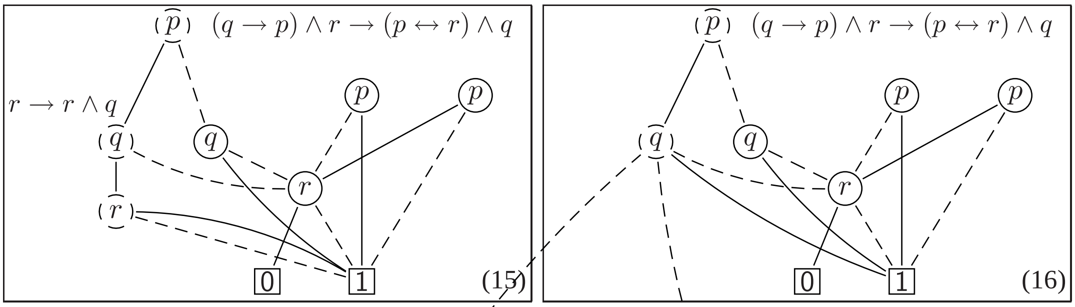


In snapshot (13) and (14) the formula $r \to r \wedge \top$ is simplified into $r \to r$.

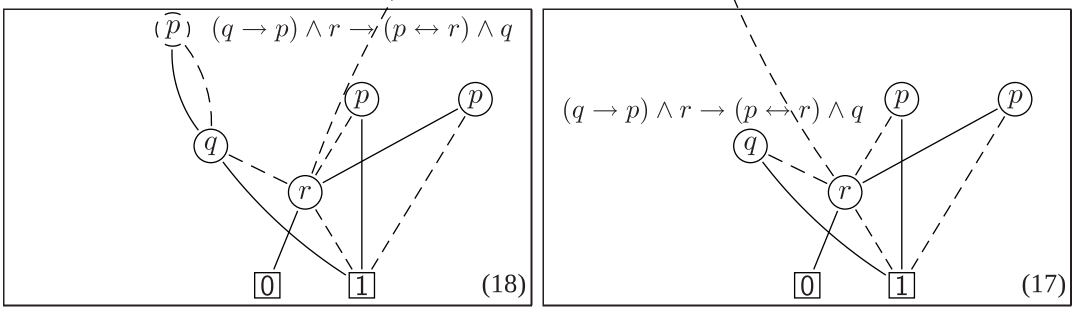




Snapshot (15) contains a redundant test on r, so $integrate$ simply returns the node $\boxed{1}$, thus eliminating the redundant test. In snapshot (16) we have to apply $integrate$. It simply returns the existing node q.



The last application of $integrate$ in snapshot (17) discovers a redundant test. The resulting global dag, including the node q returned by the procedure, is given in snapshot (18).



❏

It is not hard to argue that the following theorem holds.

THEOREM 10.16 *$obdd(F)$ returns a node n that represents F in D. The dag rooted at n is the OBDD for F.* ❏

Suppose that we have a global dag storing several OBDDs. Further, suppose that the dag contains nodes n_1 representing a formula F_1 and n_2 representing a formula F_2. If F_1 and F_2 are equivalent, then the OBDD rooted at n_1 is isomorphic to the OBDD rooted at n_2. Since in the global dag isomorphic OBDDs are shared, n_1 coincides with n_2. Likewise, if n_1 and n_2 are different nodes, then F_1 is not equivalent to F_2. This means that, if we use the global dag, *equivalence checking on OBDDs can be done in constant time:* checking equivalence amounts to checking equality of two pointers.

10.5 Composing OBDDs

OBDDs are also convenient for composing boolean functions. In this section we discuss algorithms for composing propositional formulas represented by OBDDs.

Let us first formalize what it means to compose boolean functions in terms of propositional formulas which represent these boolean functions. Suppose that we have m boolean functions of the same variables $p_1, \ldots, p_n$. Denote by $\bar{p}$ the vector $p_1, \ldots, p_n$. Suppose that the functions are $g_1(\bar{p}), \ldots, g_m(\bar{p})$. We have a function $f(q_1, \ldots, q_m)$ which represents the composition. For example $f(q_1, \ldots, q_m)$ can be the conjunction $q_1 \wedge \ldots \wedge q_m$. Then their composition using f is defined as the boolean function $f(q_1(\bar{p}), \ldots, g_m(\bar{p}))$ of the variables $\bar{p}$. When the boolean functions are represented by formulas, the situation in quite similar. There is a formula $F(q_1, \ldots, q_m)$ of variables $q_1, \ldots, q_m$ which represents the composition function f, and formulas $F_1, \ldots, F_m$ of variables $\bar{p}$ representing the functions $g_1, \ldots, g_m$. The composition is represented by the formula $F(F_1, \ldots, F_m)$, i.e., the formula obtained from $F(q_1, \ldots, q_m)$ by substituting the formulas $F_1, \ldots, F_m$ for the variables $q_1, \ldots, q_m$.

Assuming that the formulas $F_1, \ldots, F_m$ are represented by OBDDs, how can we build an OBDD for $F(F_1, \ldots, F_m)$? To do this, one can use an interesting property of the if-then-else formulas expressed by the following lemma.

LEMMA 10.17 *We have*

$$\begin{array}{ll} F(& \mathit{if}\ p\ \mathit{then}\ F_1\ \mathit{else}\ B_1, \\ & \cdots, \\ & \mathit{if}\ p\ \mathit{then}\ F_n\ \mathit{else}\ B_n\ ) \equiv \\ \multicolumn{2}{l}{\mathit{if}\ p\ \mathit{then}\ F(F_1, \ldots, F_n)\ \mathit{else}\ F(B_1, \ldots, B_n).} \end{array}$$ ❑

This lemma gives us a way for defining a generic algorithm to compose OBDDs.

ALGORITHM 10.18 (Composition of OBDDs) An algorithm for composing OBDDs is given in Figure 10.11. In this algorithm D is a global dag which satisfies all conditions 1–6 of the definition of BDDs and which contains OBDDs $d_1, \ldots, d_m$ as subdags. The function $\mathit{simplify}$ simplifies the formula using the rewrite rules of Figure 4.5, and the function $\mathit{max\_variable}(n_1, \ldots, n_m)$ returns the maximal variable occurring in the OBDDs rooted at $n_1, \ldots, n_m$. ❑

THEOREM 10.19 *Suppose that $F_1, \ldots, F_m$ are formulas and $n_1, \ldots, n_m$ are nodes representing these formulas in D. Then $\mathit{compose}(F, q_1, \ldots, q_m, n_1, \ldots, n_m)$ returns a node that represents $F(F_1, \ldots, F_m)$.* ❑

10.6 Composition Algorithms for Standard Boolean Functions

For some standard boolean functions the composition algorithm can be simplified. For an illustration, suppose that $F(q_1, \ldots, q_m)$ is $q_1 \vee \ldots \vee q_m$, i.e., we would like to derive a special algorithm for computing an OBDD representing a disjunction of formulas. Then, if any q_i is $\perp$, we have to compute the OBDD for $q_1 \vee \ldots \vee q_{i-1} \vee q_{i+1} \vee \ldots \vee q_m$. If any q_i is $\top$, then the result is always the node $\boxed{1}$. In addition, a disjunction of a single formula q_1 is simply q_1. These considerations result in an algorithm given in Figure 10.12, which can be considered as a specialization of the general composition algorithm. In a similar way one can derive an algorithm for computing conjunction, see Exercise 10.9.

```
procedure compose(F, q_1, ..., q_m, n_1, ..., n_m)
parameters: global dag D
input: propositional formula F(q_1, ..., q_m) of variables q_1, ..., q_m
       nodes n_1, ..., n_m representing formulas F_1, ..., F_m in D
output: a node n representing F(F_1, ..., F_m) in (modified) D
begin
  F := simplify(F)
  if F = ⊥ then return [0]
  if F = ⊤ then return [1]
  if some n_i is [0] then
    return compose(F^⊥_{q_i}, q_1, ..., q_{i-1}, q_{i+1}, ..., q_m, n_1, ..., n_{i-1}, n_{i+1}, ..., n_m)
  if some n_i is [1] then
    return compose(F^⊤_{q_i}, q_1, ..., q_{i-1}, q_{i+1}, ..., q_m, n_1, ..., n_{i-1}, n_{i+1}, ..., n_m)
  p := max_variable(n_1, ..., n_m)
  forall i = 1...m
    if n_i is labeled by p
      then (l_i, r_i) := (neg(n_i), pos(n_i))
      else (l_i, r_i) := (n_i, n_i)
  k_1 := compose(F, q_1, ..., q_m, l_1, ..., l_m)
  k_2 := compose(F, q_1, ..., q_m, r_1, ..., r_m)
  return integrate(k_1, p, k_2, D)
end
```

Figure 10.11: An algorithm for composing OBDDs

10.7 Variations on OBDDs

Negation, equations etc.

Exercises

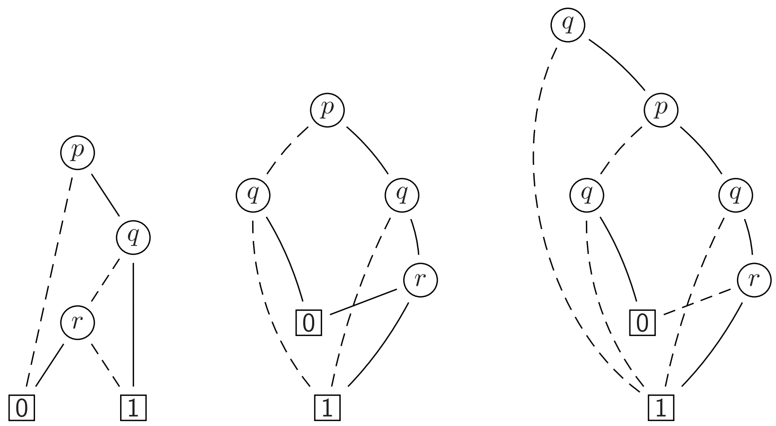
EXERCISE 10.1 Compute the OBDD for each of the following formulas and the order $p_3 > p_2 > p_1$:

$$p_1 \vee p_2 \rightarrow p_2 \wedge p_1;$$
$$\neg(p_1 \wedge p_2) \rightarrow (p_1 \vee p_3).$$ ❏

EXERCISE 10.2 Which of the following dags are OBDDs for $\neg q \wedge (\neg p \vee r)$?

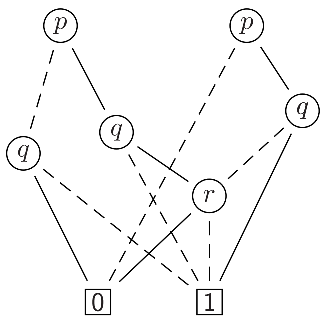
procedure $disjunction(n_1, \ldots, n_m)$
parameters: global dag D
input: nodes $n_1, \ldots, n_m$ representing $F_1, \ldots, F_m$ in D
output: a node n representing $F_1 \vee \ldots \vee F_m$ in (modified) D
begin
  if some n_i is $\boxed{1}$ **then** **return** $\boxed{1}$
  if $m = 1$ **then** **return** n_1
  if some n_i is $\boxed{0}$ **then**
    return $disjunction(n_1, \ldots, n_{i-1}, n_{i+1}, \ldots, n_m)$
  p := $max\_variable(n_1, \ldots, n_m)$
  forall $i = 1 \ldots m$
    if n_i is labeled by p
      then (l_i, r_i) := $(neg(n_i), pos(n_i))$
      else (l_i, r_i) := (n_i, n_i)
  k_1 := $disjunction(l_1, \ldots, l_m)$
  k_2 := $disjunction(r_1, \ldots, r_m)$
  return $integrate(k_1, p, k_2, D)$
end

Figure 10.12: An algorithm for computing a disjunction of OBDDs



Explain your answer. ❏

EXERCISE 10.3 Consider the following global dag D.



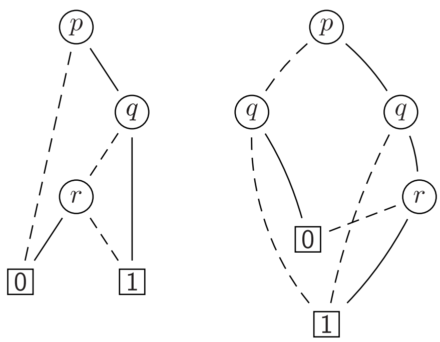
It has two different subdags d_1, d_2 rooted at p. Let d_1, d_2 represent formulas F_1, F_2, respectively. Draw the global dag D after the OBDD for $F_1 \wedge F_2$ has been integrated into it. ❏

EXERCISE 10.4 A propositional formula $F(p_1, \ldots, p_n)$ of variables $p_1, \ldots, p_n$ is called a *parity check formula* if its models are exactly those that assign 1 to an even number of variables among $p_1, \ldots, p_n$. Draw an OBDD d for a parity check formula of n variables. How many nodes does d contain? ❏

EXERCISE 10.5 A propositional formula F of variables $p_1, \ldots, p_n$ is true in an interpretation I if and only if exactly one atom from $p_1, \ldots, p_n$ is true in I. Draw the OBDD for F and the order $p_1 > p_2 > \ldots$. ❏

EXERCISE 10.6 A propositional formula F of variables $p_1, \ldots, p_n$ is true in an interpretation I if and only if at most one atom from $p_1, \ldots, p_n$ is false in I. Draw the OBDD for F and the order $p_1 > p_2 > \ldots$. ❏

EXERCISE 10.7 The formulas F and G have the following OBDDs:



Find the OBDD for the formulas $F \wedge G$, $F \vee G$, and $F \rightarrow G$ using the Composition Algorithm for OBDDs. ❏

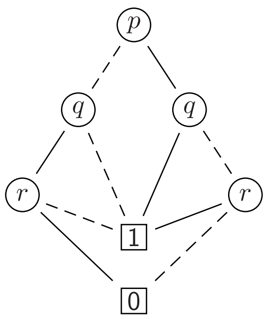
EXERCISE 10.8 Design an algorithm which counts the number of different models of a formula F of n variables, given an OBDD which represents F. ❏

EXERCISE 10.9 Specialize OBDD Composition Algorithm 10.18 for the special case of computing the conjunction of OBBDs. ❑

EXERCISE 10.10 Suppose that *if* ... *then* ... *else* ... is added to the set of connectives of propositional logic. Define tableau rules for this connective. ❑

EXERCISE 10.11 Since satisfiability of BDDs can be checked in constant time, one can check satisfiability of a formula F using the following algorithm: (1) build a BDD for F; (2) check (in constant time) that this BDD if not of the form $\boxed{0}$. Explain why using this algorithm is not such a good idea. ❑

EXERCISE 10.12 A formula F has the following OBDD.



Find all models of F. ❑