Compact representation of Boolean functions and formulas:

BDTs/BDDs/OBDDs

Data Structures for Large Propositional Formulas

Suppose that large propositional formulas are reused over and over again. For example, we may

- Build a conjunction of several formulas;
- Negate a formula;
- Check if two formulas are equivalent . . .

One needs data structures which

- give compact representation of formulas, or the boolean functions represented by the formulas;
- facilitate boolean operations on this formulas, for example, taking conjunction of several formulas;
- facilitate checking properties of formulas, such as satisfiability or equivalence checking.

Data Structures for Large Propositional Formulas

Suppose that large propositional formulas are reused over and over again. For example, we may

- Build a conjunction of several formulas;
- Negate a formula;
- Check if two formulas are equivalent ...

One needs data structures which

- give compact representation of formulas, or the boolean functions represented by the formulas;
- facilitate boolean operations on this formulas, for example, taking conjunction of several formulas;
- facilitate checking properties of formulas, such as satisfiability or equivalence checking.

Data Structures for Large Propositional Formulas

Suppose that large propositional formulas are reused over and over again. For example, we may

- Build a conjunction of several formulas;
- Negate a formula;
- Check if two formulas are equivalent . . .

One needs data structures which

- give compact representation of formulas, or the boolean functions represented by the formulas;
- facilitate boolean operations on this formulas, for example, taking conjunction of several formulas;
- facilitate checking properties of formulas, such as satisfiability or equivalence checking.

Data Structures for Large Propositional Formulas

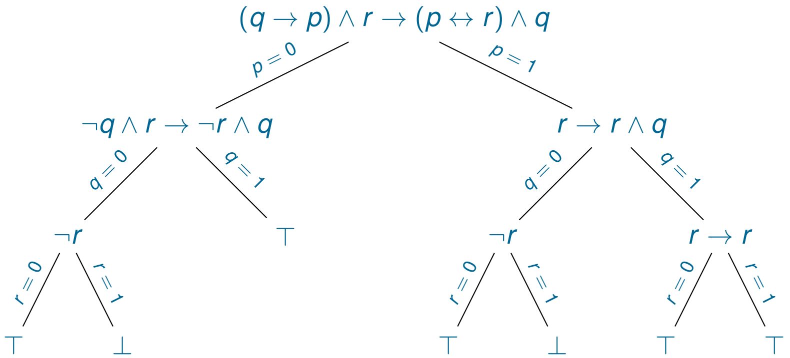
Suppose that large propositional formulas are reused over and over again. For example, we may

- Build a conjunction of several formulas;
- Negate a formula;
- Check if two formulas are equivalent ...

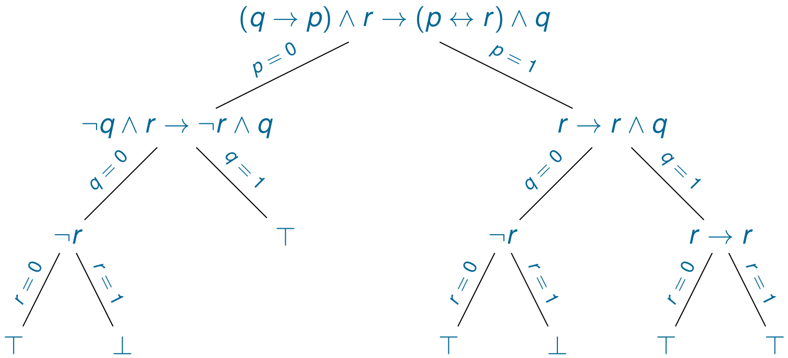
One needs data structures which

- give compact representation of formulas, or the boolean functions represented by the formulas;
- facilitate boolean operations on this formulas, for example, taking conjunction of several formulas;
- facilitate checking properties of formulas, such as satisfiability or equivalence checking.

Splitting Tree

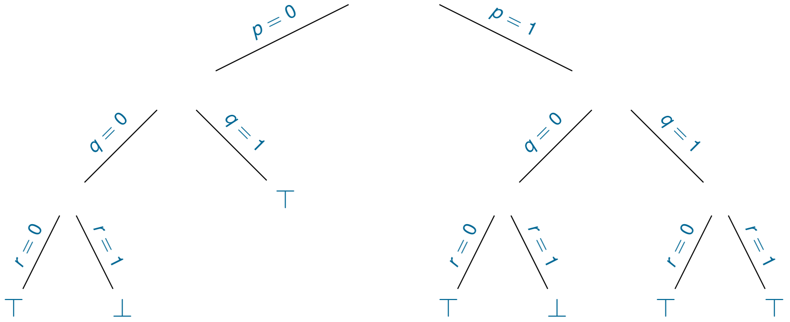


Splitting Tree

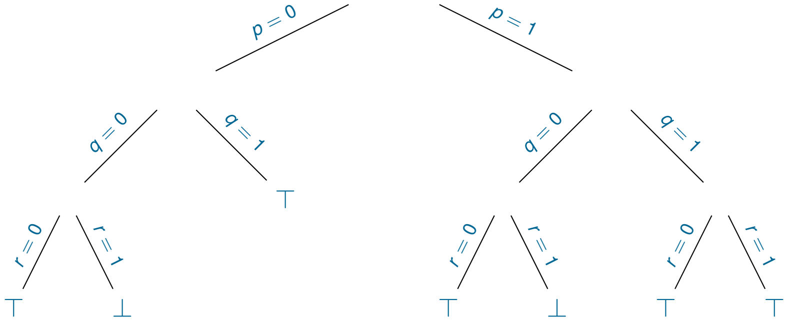


Let us “forget” about formulas in the tree

Splitting Tree

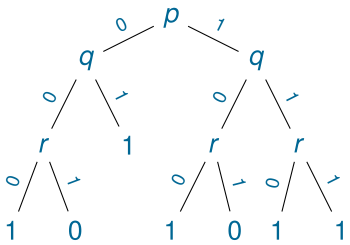


Splitting Tree

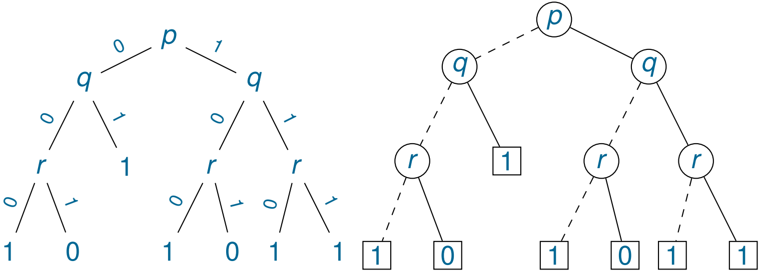


We do not see the syntax of the formula but the semantics is preserved: the tree encodes all models of the formula. Any formula having the same tree with “forgotten” formulas has the same models.

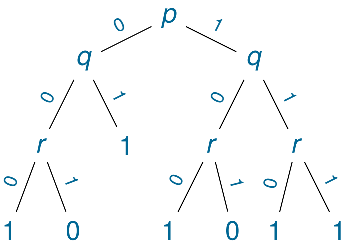
Binary Decision Tree

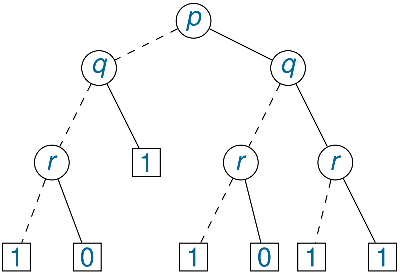


Binary Decision Tree



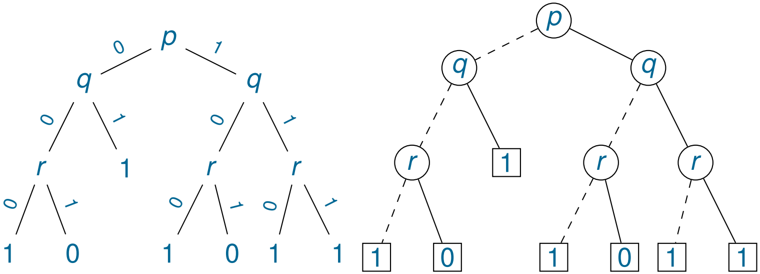
Binary Decision Tree





A circled node, such as (p), denotes the decision on the variable of this node.

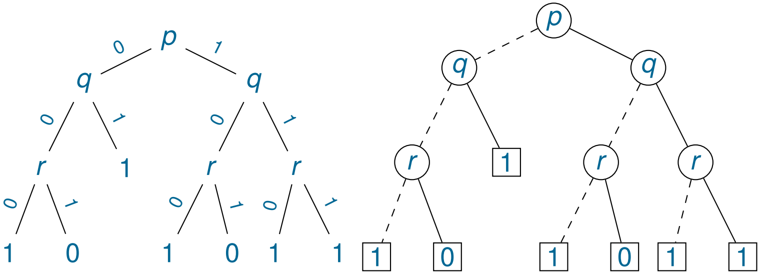
Binary Decision Tree



A circled node, such as p, denotes the decision on the variable of this node.

Terminal nodes are squared, for example $\boxed{1}$.

Binary Decision Tree

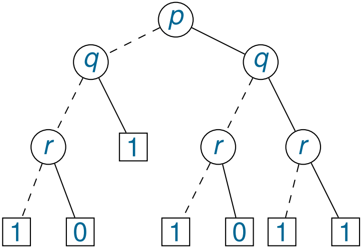


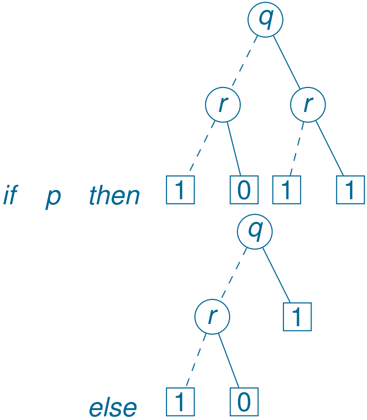
A circled node, such as p, denotes the decision on the variable of this node.

Terminal nodes are squared, for example $\boxed{1}$.

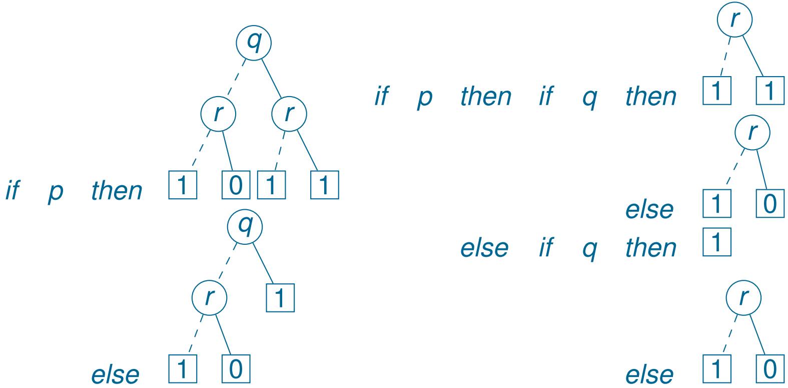
Solid lines correspond to choices when the variable is true, dashed lines to the choices when the variable is false.

Tests correspond to “if-then-else”





Tests correspond to “if-then-else”



Tests correspond to "if-then-else"

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

$$\textit{if } A \textit{ then } B \textit{ else } C \equiv (A \to B) \land (\neg A \to C).$$

Tests correspond to "if-then-else"

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

$$\textit{if } A \textit{ then } B \textit{ else } C \equiv (A \to B) \wedge (\neg A \to C).$$

If-Then-Else Normal Form

If-Then-Else Normal Form.

- The only connectives are if-then-else, $\top$ and $\bot$;
- All formulas A in *if A then B else C* are atomic.

Theorem. Every propositional formula can be transformed into an equivalent If-Then-Else normal form.

If-Then-Else Normal Form

If-Then-Else Normal Form.

- The only connectives are if-then-else, $\top$ and $\perp$;
- All formulas A in *if* A *then* B *else* C are atomic.

Theorem. Every propositional formula can be transformed into an equivalent If-Then-Else normal form.

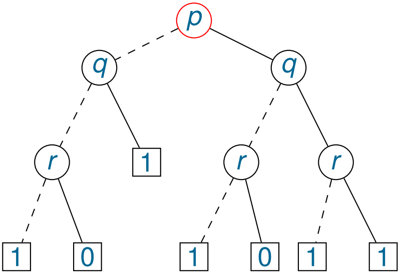
Evaluating the Formula

We can evaluate a formula on an interpretation I if we know the binary decision tree of this formula.

Evaluating the Formula

We can evaluate a formula on an interpretation I if we know the binary decision tree of this formula.

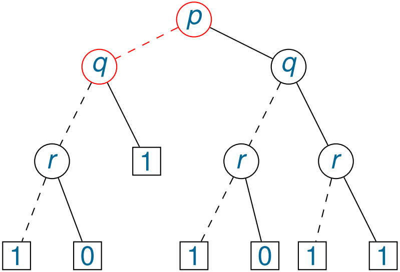
Consider, for example the interpretation $\{p \mapsto 0, q \mapsto 0, r \mapsto 1\}$ and the following tree.



Evaluating the Formula

We can evaluate a formula on an interpretation I if we know the binary decision tree of this formula.

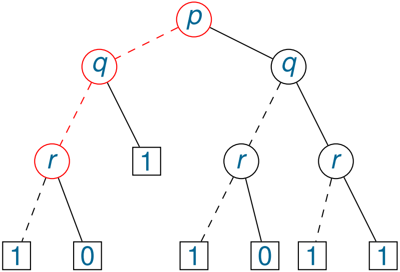
Consider, for example the interpretation $\{p \mapsto 0, q \mapsto 0, r \mapsto 1\}$ and the following tree.



Evaluating the Formula

We can evaluate a formula on an interpretation I if we know the binary decision tree of this formula.

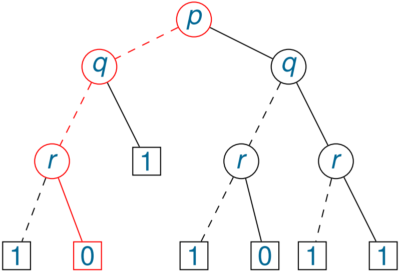
Consider, for example the interpretation $\{p \mapsto 0, q \mapsto 0, r \mapsto 1\}$ and the following tree.



Evaluating the Formula

We can evaluate a formula on an interpretation I if we know the binary decision tree of this formula.

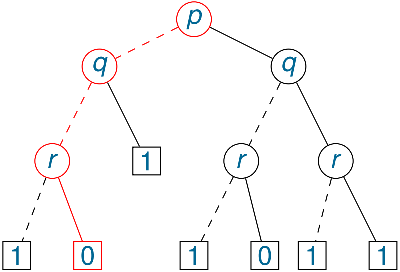
Consider, for example the interpretation $\{p \mapsto 0, q \mapsto 0, r \mapsto 1\}$ and the following tree.



Evaluating the Formula

We can evaluate a formula on an interpretation I if we know the binary decision tree of this formula.

Consider, for example the interpretation $\{p \mapsto 0, q \mapsto 0, r \mapsto 1\}$ and the following tree.



So any formula with this decision tree is false in this interpretation.

Properties

Properties of binary decision trees:

Properties

Properties of binary decision trees:

- Checking truth in an interpretation can be done in linear time in the number of variables.

Properties

Properties of binary decision trees:

- Checking truth in an interpretation can be done in linear time in the number of variables.
- Satisfiability checking can be done in time linear in the size of the tree;

Properties

Properties of binary decision trees:

- Checking truth in an interpretation can be done in linear time in the number of variables.
- Satisfiability checking can be done in time linear in the size of the tree;
- Validity checking can be done in time linear in the size of the tree;

Properties

Properties of binary decision trees:

- Checking truth in an interpretation can be done in linear time in the number of variables.
- Satisfiability checking can be done in time linear in the size of the tree;
- Validity checking can be done in time linear in the size of the tree;
- Equivalence checking is hard.

Properties

Properties of binary decision trees:

- Checking truth in an interpretation can be done in linear time in the number of variables.
- Satisfiability checking can be done in time linear in the size of the tree;
- Validity checking can be done in time linear in the size of the tree;
- Equivalence checking is hard.
- Some boolean operations, e.g., conjunction, are hard to implement.

Properties

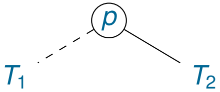
Properties of binary decision trees:

- Checking truth in an interpretation can be done in linear time in the number of variables.
- Satisfiability checking can be done in time linear in the size of the tree;
- Validity checking can be done in time linear in the size of the tree;
- Equivalence checking is hard.
- Some boolean operations, e.g., conjunction, are hard to implement.
- The size of the tree is in the worst case exponential in the number of variables.

Algorithm for Building Binary Decision Trees

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

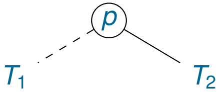
Here $tree(T_1, p, T_2)$ builds the tree



Algorithm for Building Binary Decision Trees

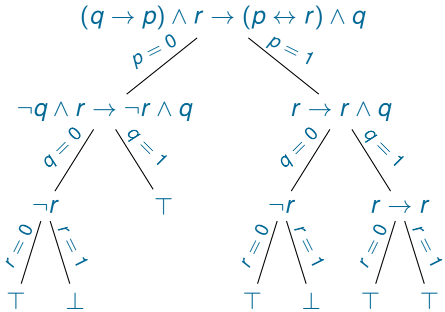
procedure $bdt(A)$
input: propositional formula A
output: a binary decision tree
parameters: function $select\_variable$
begin
  $A \ := \ simplify(A)$
  if $A = \bot$ **then** **return** $\boxed{0}$
  if $A = \top$ **then** **return** $\boxed{1}$
  $p \ := \ select\_variable(A)$
  return $tree(bdt(A_p^{\bot}),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

Here $tree(T_1, p, T_2)$ builds the tree



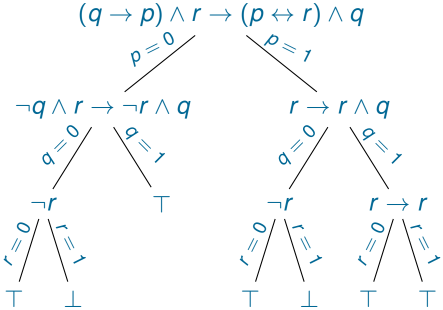
Note resemblance to the splitting algorithm!

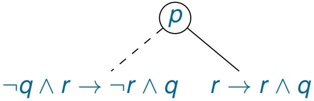
Example



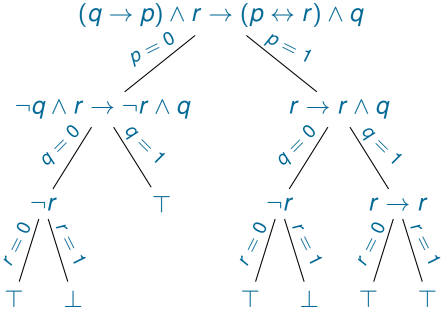
$(q \to p) \land r \to (p \leftrightarrow r) \land q$

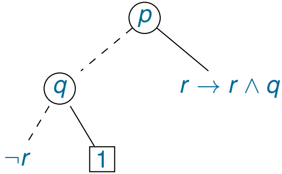
Example


$(q \to p) \wedge r \to (p \leftrightarrow r) \wedge q$
$p = 0$
$p = 1$
$\neg q \wedge r \to \neg r \wedge q$
$r \to r \wedge q$
$q = 0$
$q = 1$
$q = 0$
$q = 1$
$\neg r$
$\top$
$\neg r$
$r \to r$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$\top$
$\bot$
$\top$
$\bot$
$\top$
$\top$

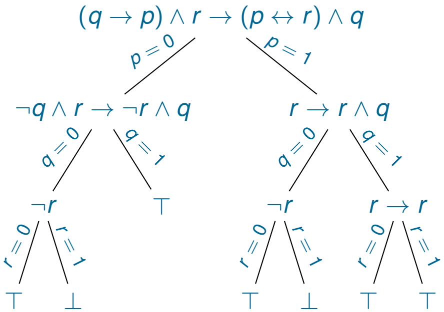

p
$\neg q \wedge r \to \neg r \wedge q$
$r \to r \wedge q$

Example

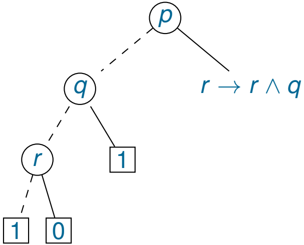

$(q \to p) \wedge r \to (p \leftrightarrow r) \wedge q$
$p = 0$
$p = 1$
$\neg q \wedge r \to \neg r \wedge q$
$r \to r \wedge q$
$q = 0$
$q = 1$
$q = 0$
$q = 1$
$\neg r$
$\top$
$\neg r$
$r \to r$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$\top$
$\bot$
$\top$
$\bot$
$\top$
$\top$


p
q
$r \to r \wedge q$
$\neg r$
1

Example

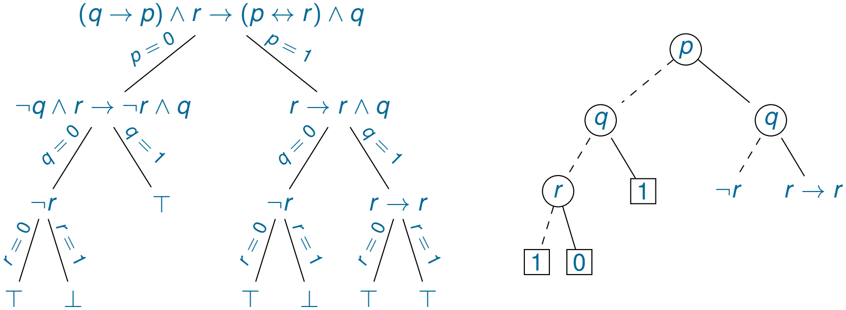


$(q \to p) \land r \to (p \leftrightarrow r) \land q$
$p = 0$
$p = 1$
$\neg q \land r \to \neg r \land q$
$r \to r \land q$
$q = 0$
$q = 1$
$q = 0$
$q = 1$
$\neg r$
$\top$
$\neg r$
$r \to r$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$\top$
$\bot$
$\top$
$\bot$
$\top$
$\top$



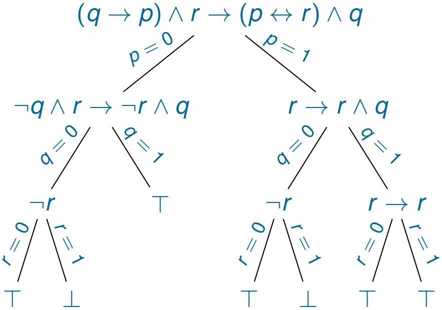
p
q
$r \to r \land q$
r
1
1
0

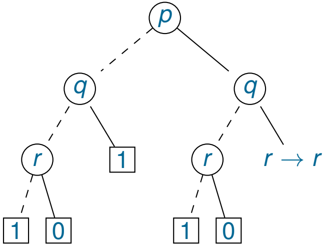
Example



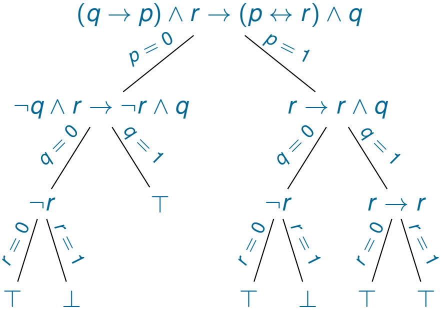
$(q \to p) \land r \to (p \leftrightarrow r) \land q$
$p = 0$
$p = 1$
$\neg q \land r \to \neg r \land q$
$r \to r \land q$
$q = 0$
$q = 1$
$q = 0$
$q = 1$
$\neg r$
$\top$
$\neg r$
$r \to r$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$\top$
$\bot$
$\top$
$\bot$
$\top$
$\top$
p
q
q
r
1
$\neg r$
$r \to r$
1
0

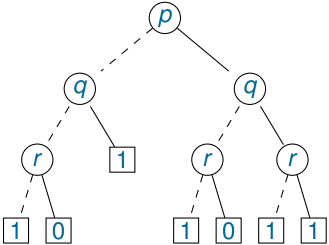
Example


$(q \to p) \wedge r \to (p \leftrightarrow r) \wedge q$
$p = 0$
$p = 1$
$\neg q \wedge r \to \neg r \wedge q$
$r \to r \wedge q$
$q = 0$
$q = 1$
$q = 0$
$q = 1$
$\neg r$
$\top$
$\neg r$
$r \to r$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$\top$
$\bot$
$\top$
$\bot$
$\top$
$\top$


p
q
q
r
1
r
$r \to r$
1
0
1
0

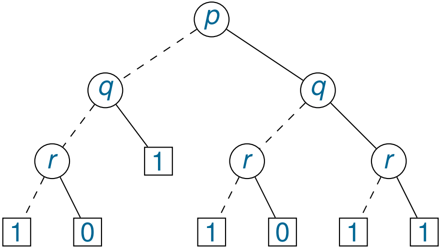
Example


$(q \to p) \land r \to (p \leftrightarrow r) \land q$
$p = 0$
$p = 1$
$\neg q \land r \to \neg r \land q$
$r \to r \land q$
$q = 0$
$q = 1$
$q = 0$
$q = 1$
$\neg r$
$\top$
$\neg r$
$r \to r$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$r = 0$
$r = 1$
$\top$
$\bot$
$\top$
$\bot$
$\top$
$\top$


p
q
q
r
1
r
r
1
0
1
0
1
1

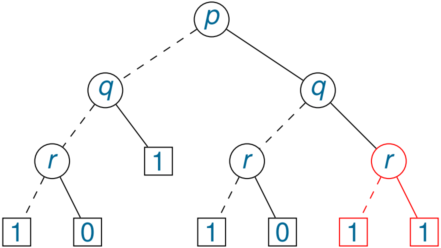
Are BDTs compact?

Are binary decision trees compact?



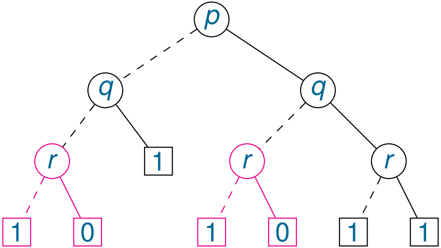
Are BDTs compact?

Are binary decision trees compact?
No: they may contain redundant tests:



Are BDTs compact?

Are binary decision trees compact?
No: they may contain redundant tests:
No: they may contain isomorphic subtrees:



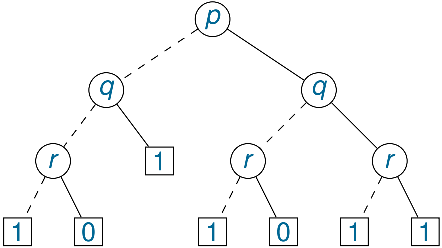
Binary Decision Diagrams (BDDs)

Binary Decision Diagrams

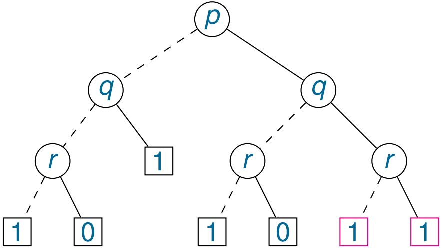
A binary decision diagrams or simply a BDD is a dag (directed acyclic graph) built like a binary decision tree but containing

- no redundant tests; and
- no isomorphic subtrees.

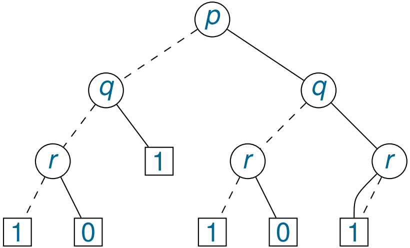
Transforming Binary Decision Tree into a BDD



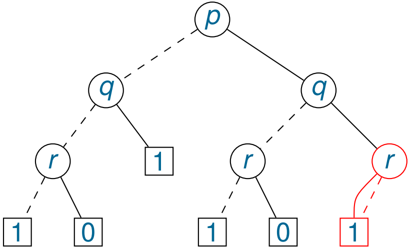
Transforming Binary Decision Tree into a BDD



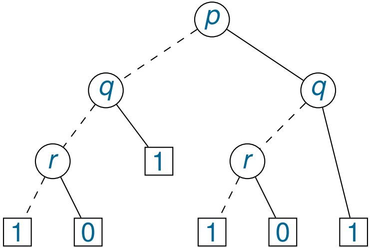
Transforming Binary Decision Tree into a BDD



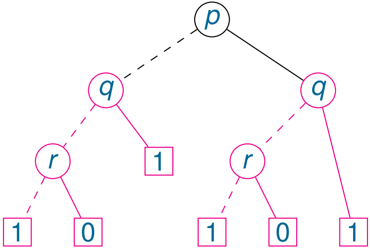
Transforming Binary Decision Tree into a BDD



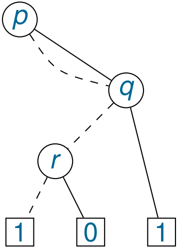
Transforming Binary Decision Tree into a BDD



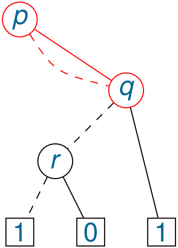
Transforming Binary Decision Tree into a BDD



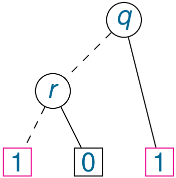
Transforming Binary Decision Tree into a BDD



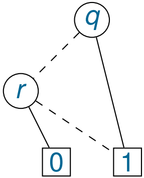
Transforming Binary Decision Tree into a BDD



Transforming Binary Decision Tree into a BDD



Transforming Binary Decision Tree into a BDD



Properties

What is the complexity of satisfiability, validity and equivalence checking for BDDs?

- Satisfiability checking can be done in constant time;
- Validity checking can be done in constant time;
- Equivalence checking
- Some boolean operations, e.g., conjunction

Properties

What is the complexity of satisfiability, validity and equivalence checking for BDDs?

- Satisfiability checking can be done in constant time;
- Validity checking can be done in constant time;
- Equivalence checking
- Some boolean operations, e.g., conjunction

Properties

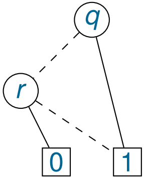
What is the complexity of satisfiability, validity and equivalence checking for BDDs?

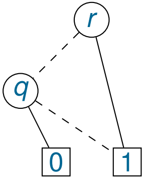
- Satisfiability checking can be done in constant time;
- Validity checking can be done in constant time;
- Equivalence checking
- Some boolean operations, e.g., conjunction

Properties

What is the complexity of satisfiability, validity and equivalence checking for BDDs?

- Satisfiability checking can be done in constant time;
- Validity checking can be done in constant time;
- Equivalence checking
- Some boolean operations, e.g., conjunction

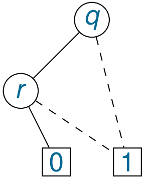


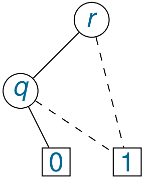


Properties

What is the complexity of satisfiability, validity and equivalence checking for BDDs?

- Satisfiability checking can be done in constant time;
- Validity checking can be done in constant time;
- Equivalence checking
- Some boolean operations, e.g., conjunction

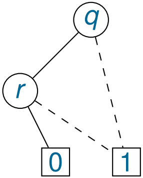


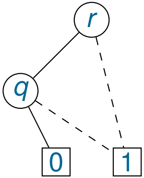


Properties

What is the complexity of satisfiability, validity and equivalence checking for BDDs?

- Satisfiability checking can be done in constant time;
- Validity checking can be done in constant time;
- Equivalence checking is hard.
- Some boolean operations, e.g., conjunction

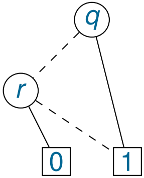


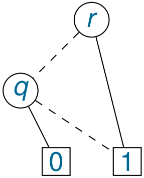


Properties

What is the complexity of satisfiability, validity and equivalence checking for BDDs?

- Satisfiability checking can be done in constant time;
- Validity checking can be done in constant time;
- Equivalence checking is hard.
- Some boolean operations, e.g., conjunction, are hard to implement.



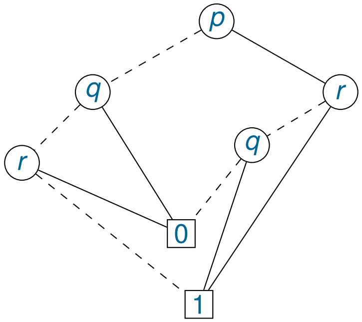


OBDDs

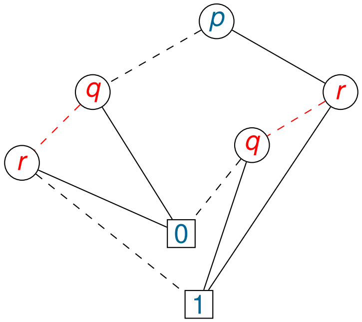
Ordered Binary Decision Diagrams (OBDDs)

Donald Knuth regarding OBDDs:
"one of the only really fundamental data structures that came out in the last twenty-five years"

BDDs

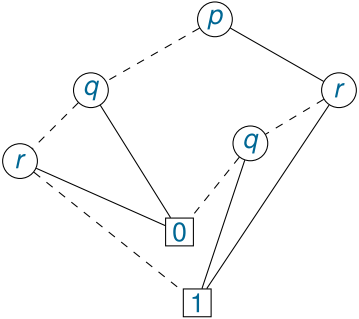


BDDs



Problem: on different branches of the dag tests are made in a different order.

BDDs



Problem: on different branches of the dag tests are made in a different order.

OBDDs:

- introduce an order $>$ on variables;
- make tests in this order.

OBDDs Properties

- Every Bolean function has a unique representation as OBDD!
  In other words OBDDs are canonical for Boolean functions.

- Validity checking can be done in constant time;
- Satisfiability checking can be done in constant time;

OBDDs Properties

- Every Bolean function has a unique representation as OBDD!
  In other words OBDDs are canonical for Boolean functions.
- Equivalence checking can be done in constant time;
- Validity checking can be done in constant time;
- Satisfiability checking can be done in constant time;

OBDDs Properties

- Every Bolean function has a unique representation as OBDD!
  In other words OBDDs are canonical for Boolean functions.
- Equivalence checking can be done in constant time;
- Validity checking can be done in constant time;
- Satisfiability checking can be done in constant time;

OBDDs Properties

- Every Bolean function has a unique representation as OBDD!
  In other words OBDDs are canonical for Boolean functions.
- Equivalence checking can be done in constant time;
- Validity checking can be done in constant time;
- Satisfiability checking can be done in constant time;

OBDDs Properties

- Every Bolean function has a unique representation as OBDD!
  In other words OBDDs are canonical for Boolean functions.
- Equivalence checking can be done in constant time;
- Validity checking can be done in constant time;
- Satisfiability checking can be done in constant time;
- Boolean operations e.g., conjunction, are easy to implement.

Integrating a node in a dag

All OBDD algorithms will use the same procedure for integrating a node in a dag.

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_2 *else* F_1

Integrating a node in a dag

All OBDD algorithms will use the same procedure for integrating a node in a dag.

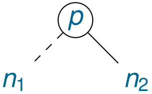
procedure *integrate*(n_1, p, n_2, D)
parameters: global dag D
input: nodes n_1, n_2 in D representing formulas F_1, F_2, variable p
output: node n in (modified) D representing *if* p *then* F_2 *else* F_1
begin
  if $n_1 = n_2$ **then** **return** n_1;

end

Integrating a node in a dag

All OBDD algorithms will use the same procedure for integrating a node in a dag.

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_2 *else* F_1
begin
  if $n_1 = n_2$ **then** **return** n_1;
  if D contains a node n having the form



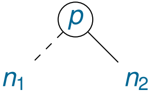
   then **return** n;

end

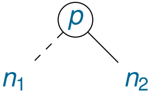
Integrating a node in a dag

All OBDD algorithms will use the same procedure for integrating a node in a dag.

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_2 else F_1*
begin
  if $n_1 = n_2$ **then** **return** n_1;
  if D contains a node n having the form



   then **return** n;
  else add to D a new node n of the form



  return n
end

Building OBDDs

procedure $obdd(F)$
input: propositional formula F
parameters: order on variables $p_1 > p_2 \ldots > p_k$
global dag D respecting using this order
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

Building OBDDs

procedure $obdd(F)$
input: propositional formula F
parameters: order on variables $p_1 > p_2 \ldots > p_k$
global dag D respecting using this order
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

- This procedure puts together the algorithms for building decision trees and the algorithm for eliminating redundancies.

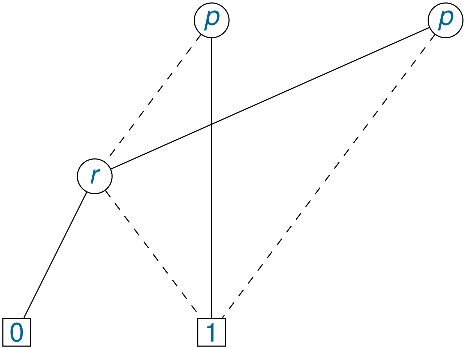
Building OBDDs

procedure $obdd(F)$
input: propositional formula F
parameters: order on variables $p_1 > p_2 \ldots > p_k$
global dag D respecting using this order
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

- This procedure puts together the algorithms for building decision trees and the algorithm for eliminating redundancies.
- Redundancy elimination is done by the procedure *integrate*.

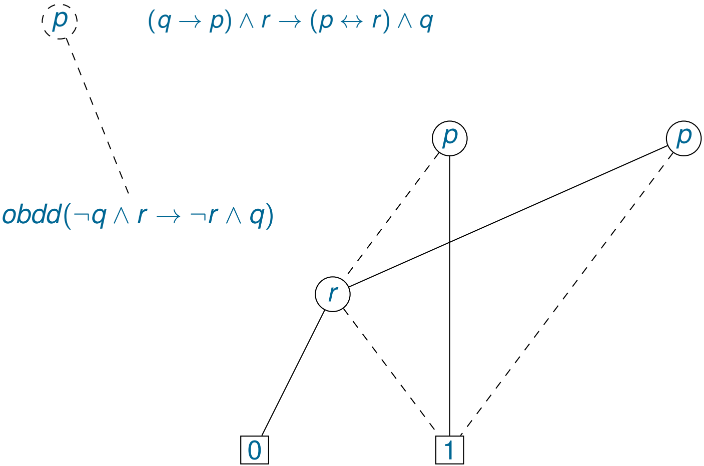
Example: Building OBDD: $p > q > r$

$obdd((q \rightarrow p) \wedge r \rightarrow (p \leftrightarrow r) \wedge q)$

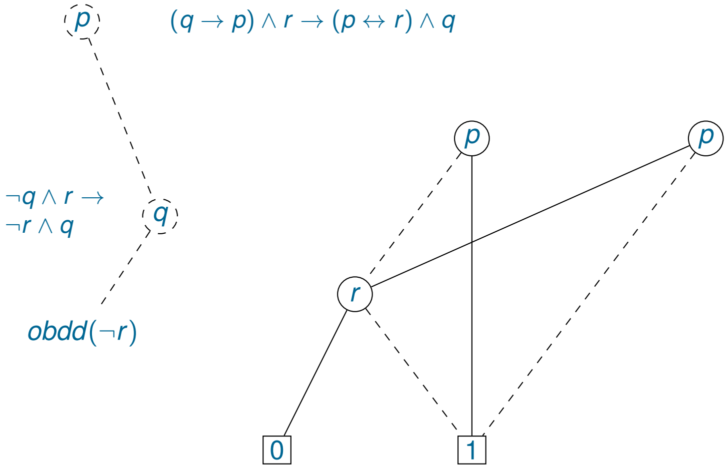


Example: Building OBDD: $p > q > r$

$(q \to p) \wedge r \to (p \leftrightarrow r) \wedge q$

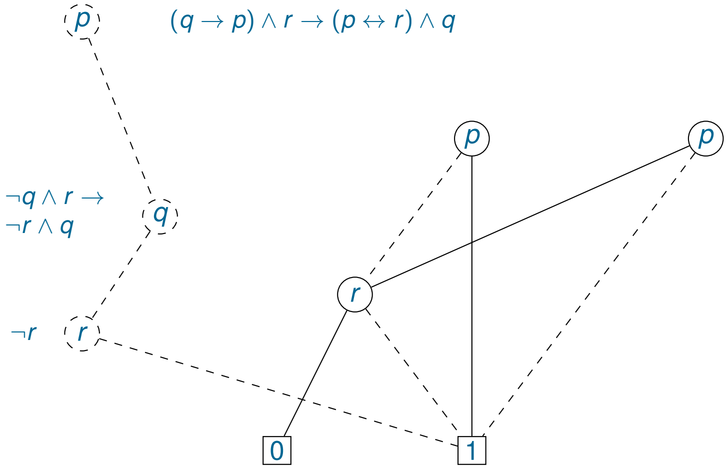


Example: Building OBDD: $p > q > r$

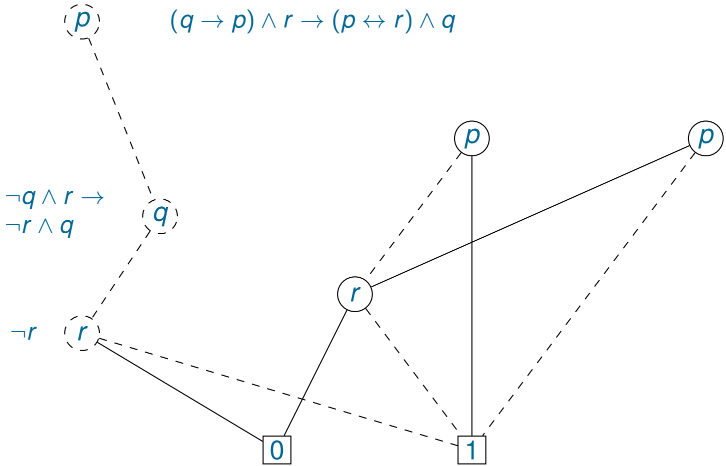


Example: Building OBDD: $p > q > r$

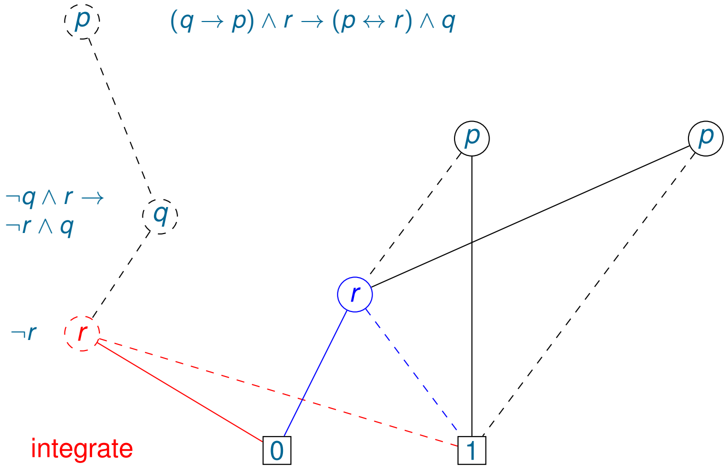
$(q \to p) \land r \to (p \leftrightarrow r) \land q$



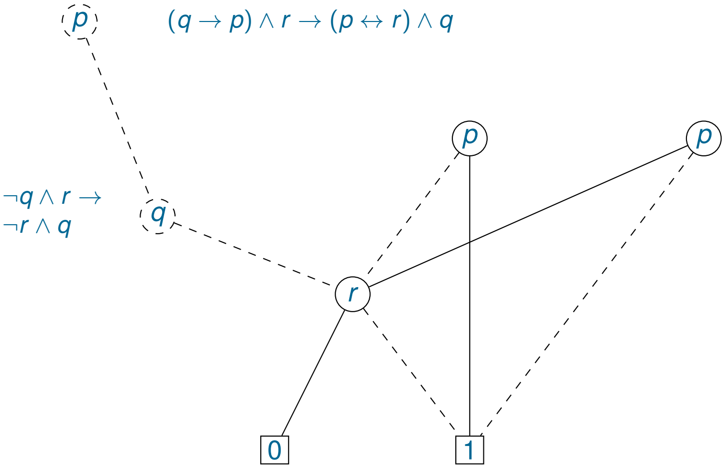
Example: Building OBDD: $p > q > r$



Example: Building OBDD: $p > q > r$

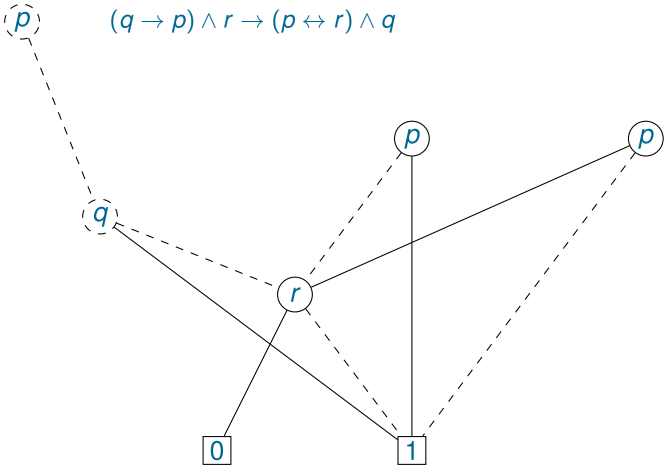


Example: Building OBDD: $p > q > r$



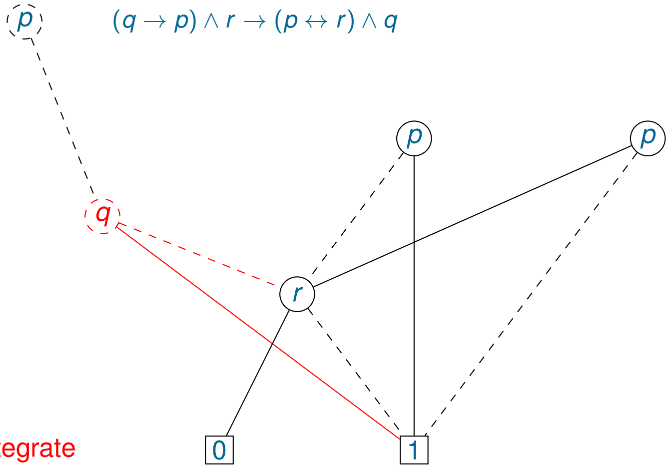
Example: Building OBDD: $p > q > r$

$(q \to p) \land r \to (p \leftrightarrow r) \land q$



Example: Building OBDD: $p > q > r$

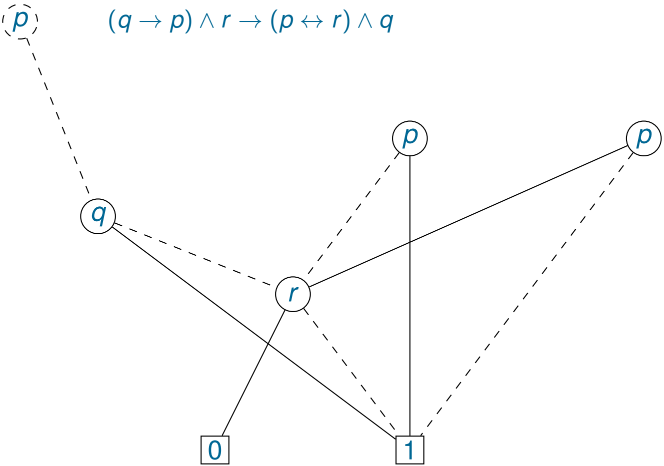
$(q \to p) \wedge r \to (p \leftrightarrow r) \wedge q$



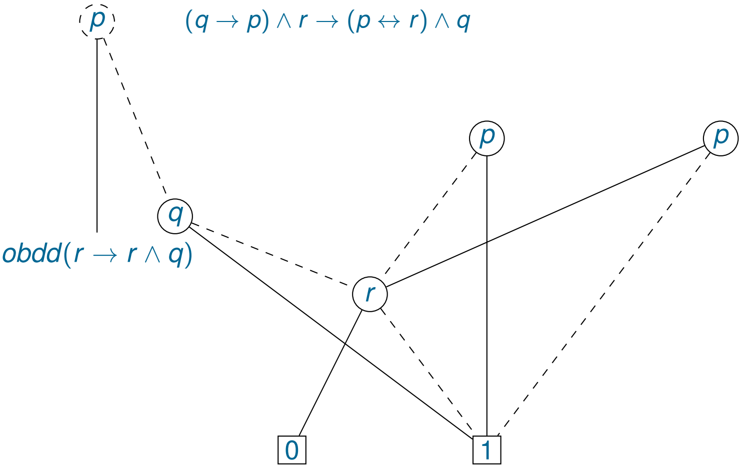
integrate

Example: Building OBDD: $p > q > r$

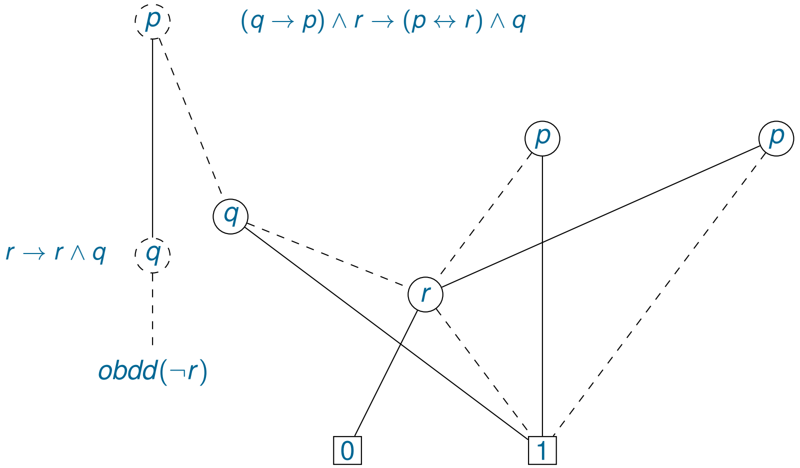
$(q \to p) \land r \to (p \leftrightarrow r) \land q$



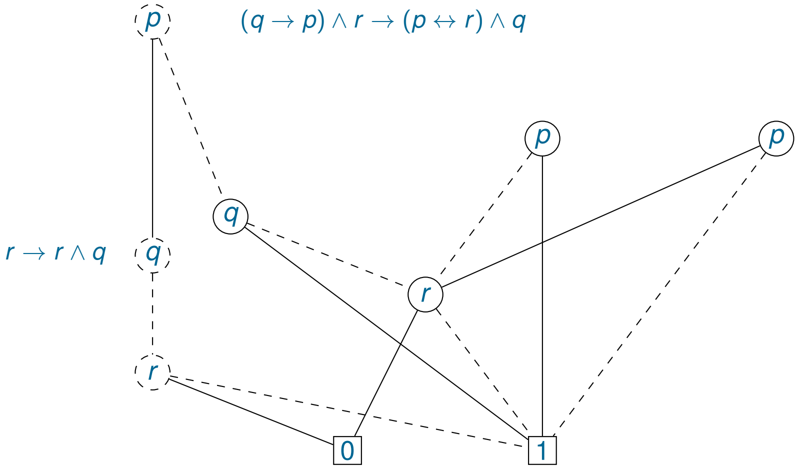
Example: Building OBDD: $p > q > r$



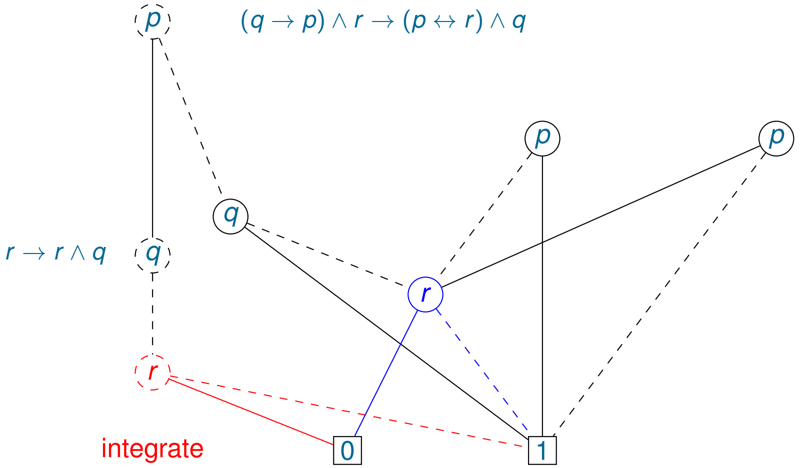
Example: Building OBDD: $p > q > r$



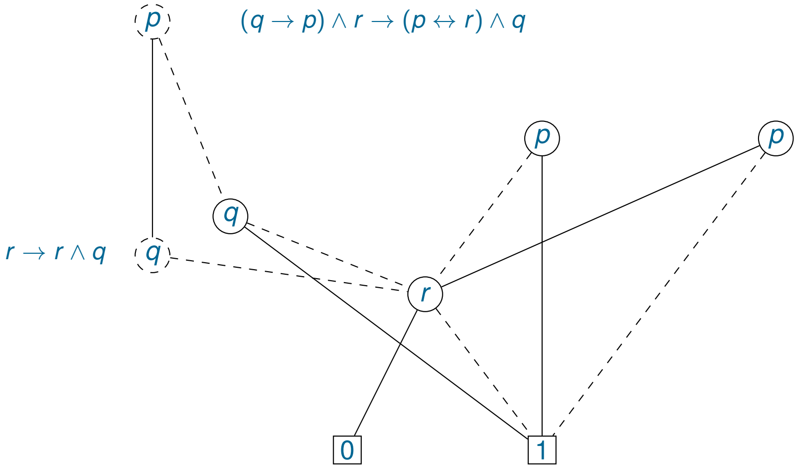
Example: Building OBDD: $p > q > r$



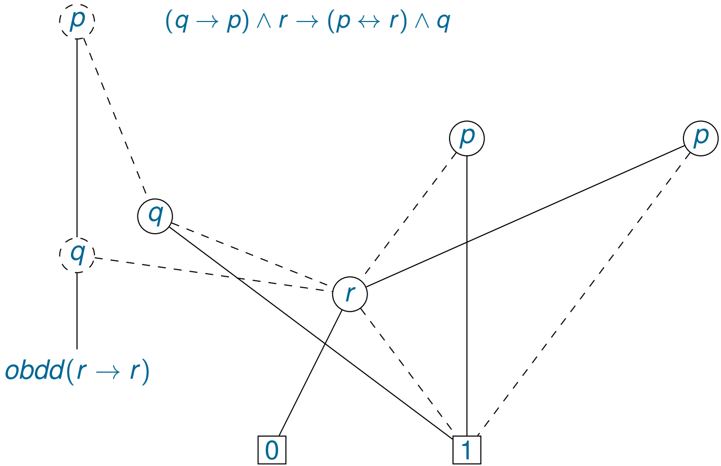
Example: Building OBDD: $p > q > r$



Example: Building OBDD: $p > q > r$

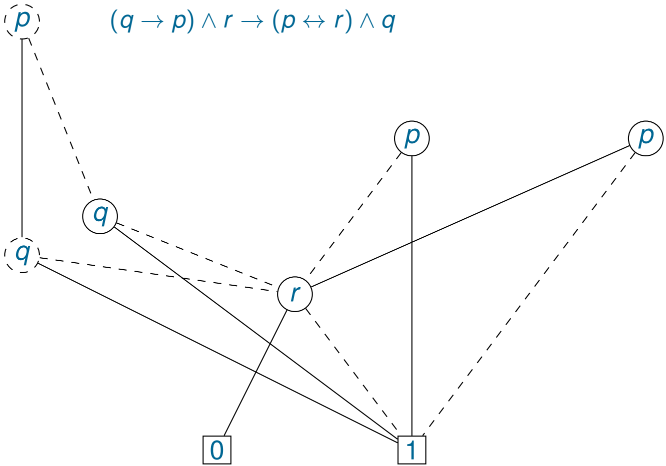


Example: Building OBDD: $p > q > r$



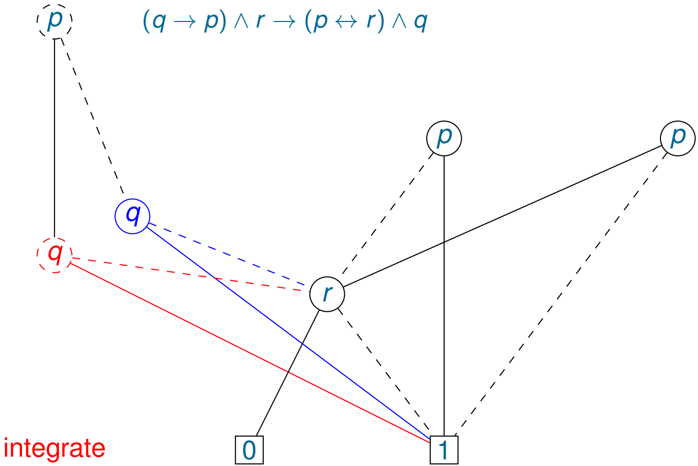
Example: Building OBDD: $p > q > r$

$(q \to p) \land r \to (p \leftrightarrow r) \land q$



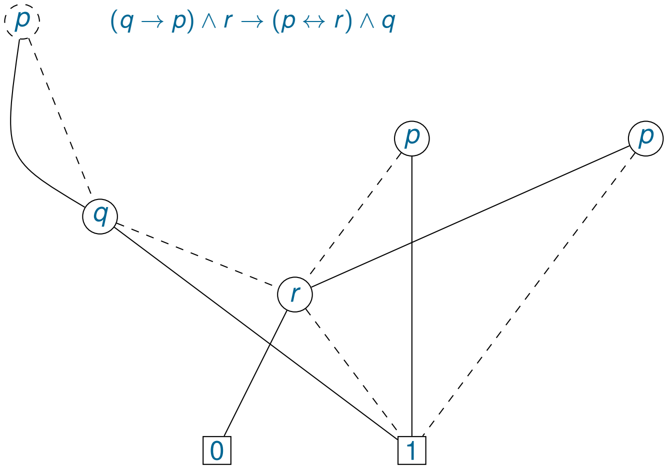
Example: Building OBDD: $p > q > r$

$(q \to p) \land r \to (p \leftrightarrow r) \land q$

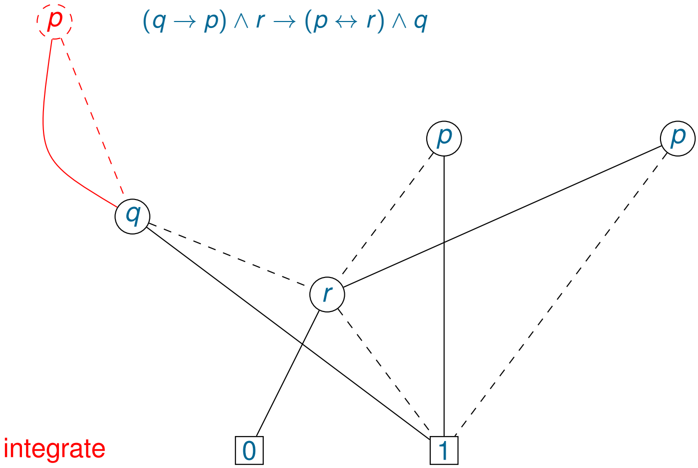


Example: Building OBDD: $p > q > r$

$(q \to p) \land r \to (p \leftrightarrow r) \land q$

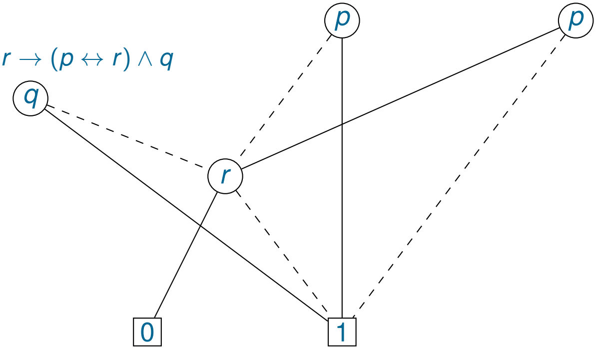


Example: Building OBDD: $p > q > r$



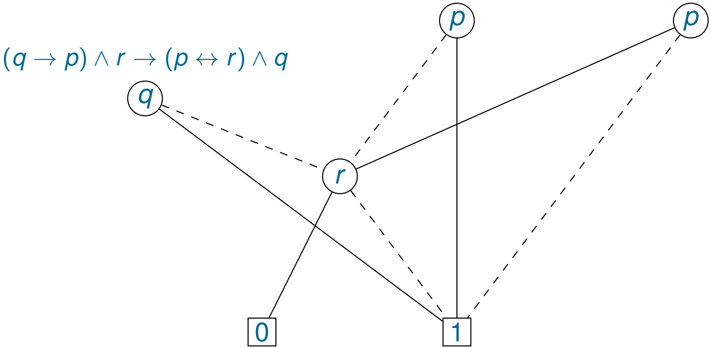
Example: Building OBDD: $p > q > r$

$(q \to p) \land r \to (p \leftrightarrow r) \land q$



We return the new node rooted at q.

Example: Building OBDD: $p > q > r$



We return the new node rooted at q.

Note that the application of this procedure modified the global dag.

Applying Boolean functions to OBDDs

Applying Boolean functions OBDDs

- Suppose we have a boolean function f, for example, disjunction $f(x_1, \ldots, x_n) \stackrel{\text{def}}{=} x_1 \vee \ldots \vee x_n$.

.

Applying Boolean functions OBDDs

- Suppose we have a boolean function f, for example, disjunction $f(x_1, \ldots, x_n) \stackrel{\text{def}}{=} x_1 \vee \ldots \vee x_n$.
- Further, we have OBDDs $D_1, \ldots, D_n$ representing some formulas $F_1, \ldots, F_n$.

.

Applying Boolean functions OBDDs

- Suppose we have a boolean function f, for example, disjunction $f(x_1, \ldots, x_n) \stackrel{\mathrm{def}}{=} x_1 \vee \ldots \vee x_n$.
- Further, we have OBDDs $D_1, \ldots, D_n$ representing some formulas $F_1, \ldots, F_n$.
- We want to compute the OBDD representing $f(F_1, \ldots, F_n)$.

.

Applying Boolean functions OBDDs

- Suppose we have a boolean function f, for example, disjunction $f(x_1, \ldots, x_n) \stackrel{\text{def}}{=} x_1 \vee \ldots \vee x_n$.
- Further, we have OBDDs $D_1, \ldots, D_n$ representing some formulas $F_1, \ldots, F_n$.
- We want to compute the OBDD representing $f(F_1, \ldots, F_n)$.

- We assume a global dag that contains all OBDDs so that all isomorphic subdags are shared.

.

Applying Boolean functions OBDDs

- Suppose we have a boolean function f, for example, disjunction $f(x_1, \ldots, x_n) \stackrel{\text{def}}{=} x_1 \vee \ldots \vee x_n$.
- Further, we have OBDDs $D_1, \ldots, D_n$ representing some formulas $F_1, \ldots, F_n$.
- We want to compute the OBDD representing $f(F_1, \ldots, F_n)$.

- We assume a global dag that contains all OBDDs so that all isomorphic subdags are shared.
- Use the following fundamental property of if-then-else: if-then-else commutes with functions.

Fundamental property of if-then-else

If-then-else commutes with any function.

Fundamental property of if-then-else

If-then-else commutes with any function.

$$
\begin{array}{l}
f(\ \ \textit{if } p \textit{ then } l_1 \textit{ else } r_1, \\
\qquad \ldots, \\
\qquad \textit{if } p \textit{ then } l_n \textit{ else } r_n\ ) = \\
\textit{if } p \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)
\end{array}
$$

Fundamental property of if-then-else

If-then-else commutes with any function.

$$
\begin{aligned}
& f(\ \ \textit{if } p \textit{ then } l_1 \textit{ else } r_1, \\
& \qquad \ldots, \\
& \qquad \textit{if } p \textit{ then } l_n \textit{ else } r_n\ ) = \\
& \textit{if } p \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)
\end{aligned}
$$

Proof? By case analysis on p.

Fundamental property of if-then-else

If-then-else commutes with any function.

$$
\begin{array}{l}
f(\ \ \textit{if } p \textit{ then } l_1 \textit{ else } r_1, \\
\qquad \ldots, \\
\qquad \textit{if } p \textit{ then } l_n \textit{ else } r_n\ ) = \\
\textit{if } p \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)
\end{array}
$$

Proof? By case analysis on p.

This is what this formula teaches us: to apply f to n OBDDs rooted at p,

Fundamental property of if-then-else

If-then-else commutes with any function.

$$
\begin{array}{l}
f(\ \textit{if } p \textit{ then } l_1 \textit{ else } r_1, \\
\quad \ldots, \\
\quad \textit{if } p \textit{ then } l_n \textit{ else } r_n\ ) = \\
\textit{if } p \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)
\end{array}
$$

Proof? By case analysis on p.

This is what this formula teaches us: to apply f to n OBDDs rooted at p,

- apply f to the subdags corresponding to $p = 0$ (the subdags under the dashed edge), obtaining a dag D_0;

Fundamental property of if-then-else

If-then-else commutes with any function.

$$\begin{aligned} & f(\ \textit{if } p \textit{ then } l_1 \textit{ else } r_1, \\ & \quad \ldots, \\ & \quad \textit{if } p \textit{ then } l_n \textit{ else } r_n\ ) = \\ & \textit{if } p \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n) \end{aligned}$$

Proof? By case analysis on p.

This is what this formula teaches us: to apply f to n OBDDs rooted at p,

- apply f to the subdags corresponding to $p = 0$ (the subdags under the dashed edge), obtaining a dag D_0;
- apply f to the subtrees corresponding to $p = 1$ (the subdags under the solid edge), obtaining a dag D_1;

Fundamental property of if-then-else

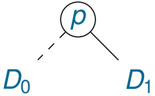
If-then-else commutes with any function.

$$
\begin{aligned}
& f(\ \textit{if } p \textit{ then } l_1 \textit{ else } r_1, \\
& \quad \ldots, \\
& \quad \textit{if } p \textit{ then } l_n \textit{ else } r_n\ ) = \\
& \textit{if } p \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)
\end{aligned}
$$

Proof? By case analysis on p.

This is what this formula teaches us: to apply f to n OBDDs rooted at p,

- apply f to the subdags corresponding to $p = 0$ (the subdags under the dashed edge), obtaining a dag D_0;
- apply f to the subtrees corresponding to $p = 1$ (the subdags under the solid edge), obtaining a dag D_1;
- Build and return the dag of the form

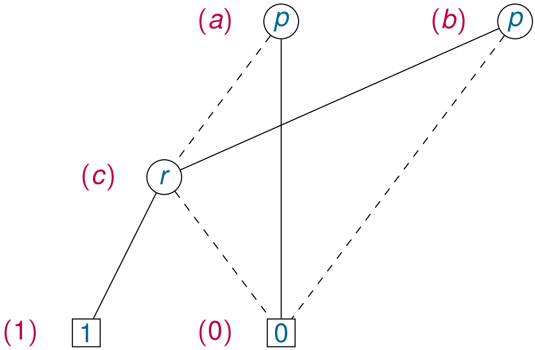


Disjunction

```
procedure disjunction(n_1, ..., n_m)
parameters: global dag D
input: nodes n_1, ..., n_m representing F_1, ..., F_m in D
output: a node n representing F_1 ∨ ... ∨ F_m in (modified) D
begin
  if some n_i is [1] then return [1]
  if m = 0 then return [0]
  if m = 1 then return n_1
  if some n_i is [0] then
    return disjunction(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 labelled 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, ..., l_m)
  k_2 := disjunction(r_1, ..., r_m)
  return integrate(k_1, p, k_2, D)
end
```

Example: Disjunction $p > r$

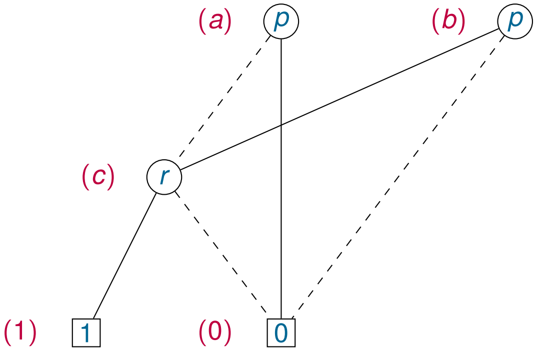
(a) represents $(\neg p \wedge r)$; (b) represents $(p \wedge r)$



Example: Disjunction $p > r$

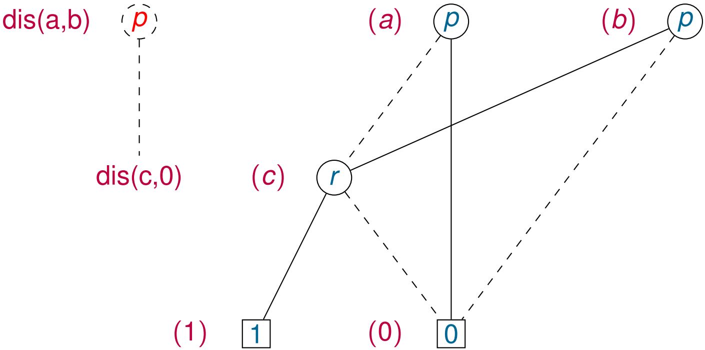
(a) represents $(\neg p \wedge r)$; (b) represents $(p \wedge r)$

dis(a,b)



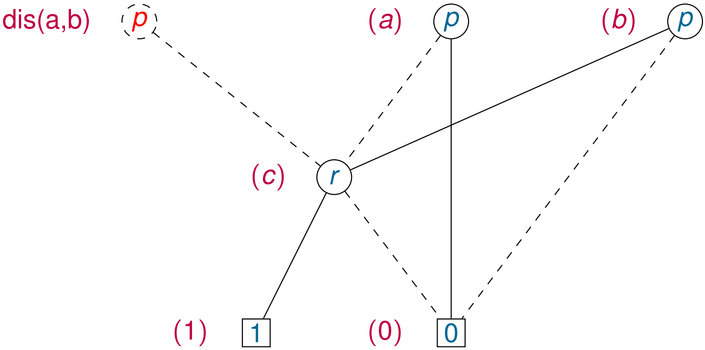
Example: Disjunction $p > r$

(a) represents $(\neg p \wedge r)$; (b) represents $(p \wedge r)$



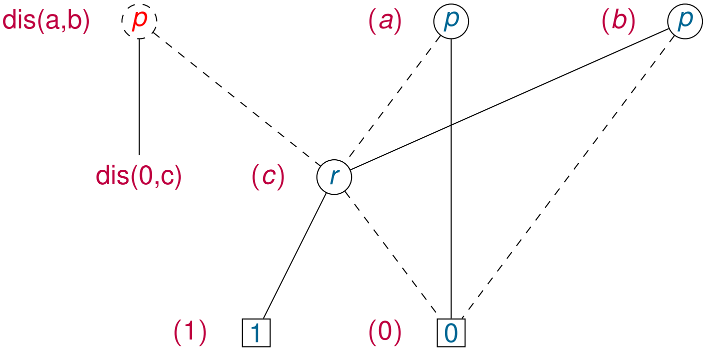
Example: Disjunction $p > r$

(a) represents $(\neg p \wedge r)$; (b) represents $(p \wedge r)$



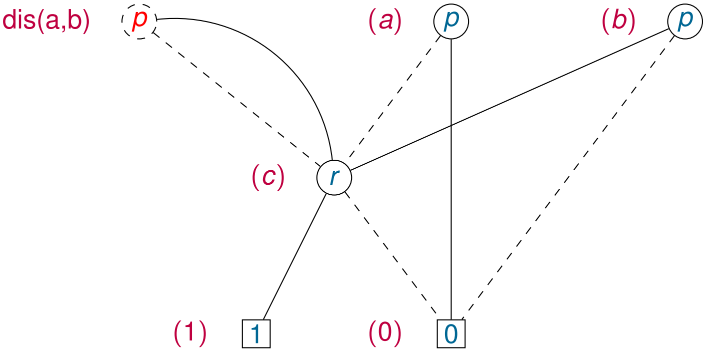
Example: Disjunction $p > r$

(a) represents $(\neg p \land r)$; (b) represents $(p \land r)$



Example: Disjunction $p > r$

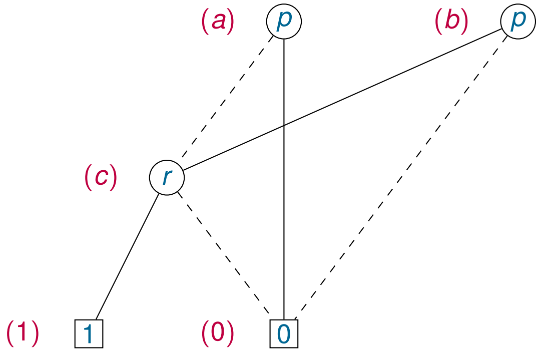
(a) represents $(\neg p \wedge r)$; (b) represents $(p \wedge r)$



Example: Disjunction $p > r$

(a) represents $(\neg p \wedge r)$; (b) represents $(p \wedge r)$

dis(a,b) = c



Disjunction

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 $m = 0$ **then** **return** $\boxed{0}$
  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 labelled 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

Disjunction

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 $m = 0$ **then** **return** $\boxed{0}$
  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 labelled 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

$F \vee \top \equiv \top$
$F \vee \bot \equiv F$

Conjunction

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

$F \wedge \bot \equiv \bot$
$F \wedge \top \equiv F$

Summary

Binary decision trees, BDDs, OBDDs.

OBDDs:

- canonical representation of Boolean functions;
- can transform any formula to equivalent OBDD representation. But in some cases the size of the OBDD can grow exponentially.
- constant time checks for main logical problems: satisfiability, validity, equivalence;
- Boolean operations e.g. conjunction, disjunction, etc. are easy to implement.