

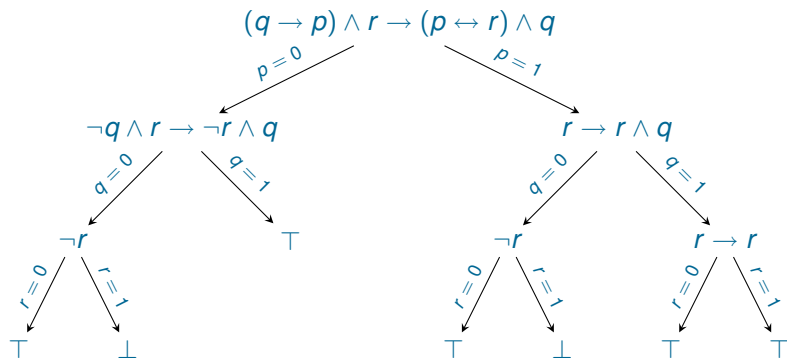
Algorithm for Building Binary Decision Trees

```
procedure bdt(A)  
input: propositional formula A  
output: a binary decision tree  
parameters: function select_atom  
begin  
  A := simplify(A)  
  if A =  $\perp$  then return 0  
  if A =  $\top$  then return 1  
  p := select_atom(A)  
  return tree(bdt( $A_p^\perp$ ), p, bdt( $A_p^\top$ ))  
end
```

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



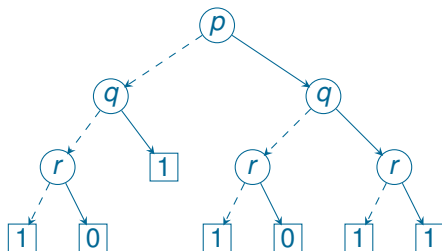
Example



Redundant Tests

Are binary decision trees **compact**?

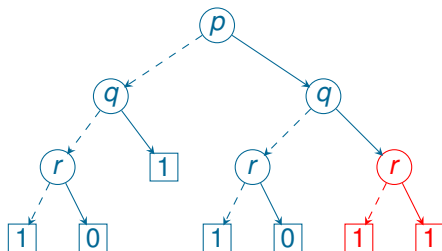
No: they may contain **redundant tests**:



Redundant Tests

Are binary decision trees **compact**?

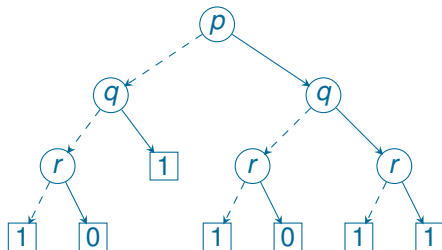
No: they may contain **redundant tests**:



Isomorphic Subtrees

Are binary decision trees **compact**?

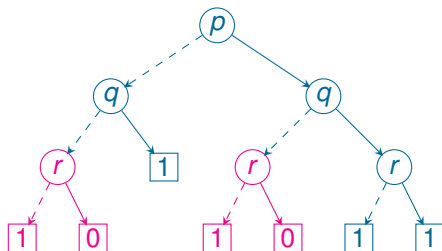
No: they may contain **isomorphic subtrees**:



Isomorphic Subtrees

Are binary decision trees **compact**?

No: they may contain **isomorphic subtrees**:

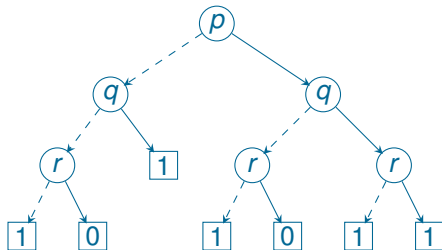


Binary Decision Diagrams

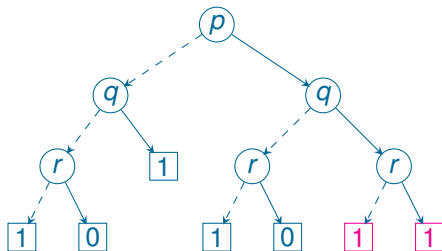
A **binary decision diagrams** or simply a **BDD** is a dag built like a binary decision tree but containing

- ▶ **no redundant tests**; and
- ▶ **no isomorphic subtrees**.

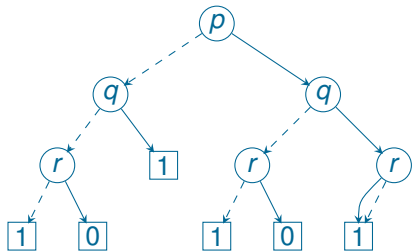
Transforming Binary Decision Tree into a BDD



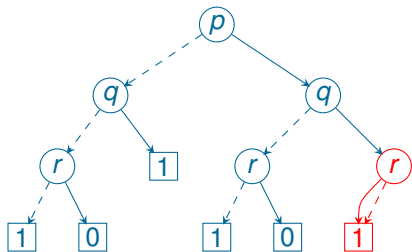
Transforming Binary Decision Tree into a BDD



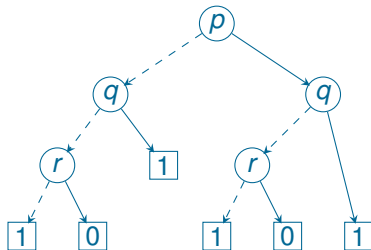
Transforming Binary Decision Tree into a BDD



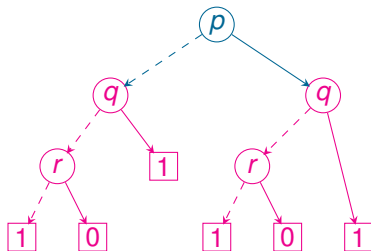
Transforming Binary Decision Tree into a BDD



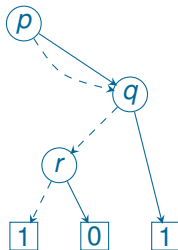
Transforming Binary Decision Tree into a BDD



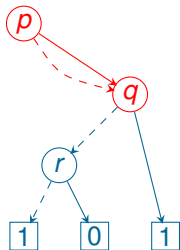
Transforming Binary Decision Tree into a BDD



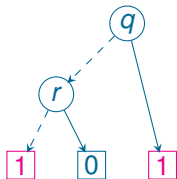
Transforming Binary Decision Tree into a BDD



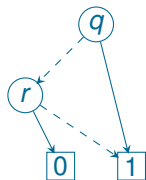
Transforming Binary Decision Tree into a BDD



Transforming Binary Decision Tree into a BDD



Transforming Binary Decision Tree into a BDD



Properties

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

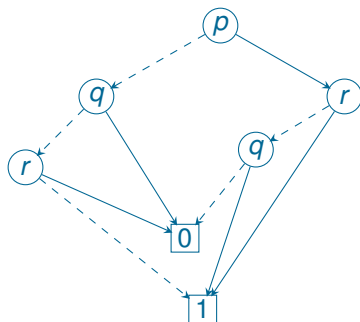
Properties

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

Properties

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

OBDDs



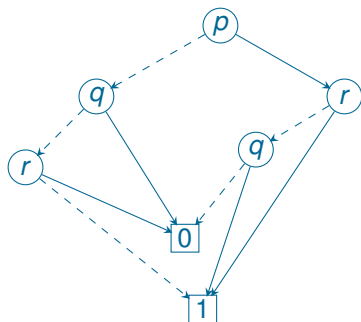
Problem: on different branches on the dag tests are made in a different order.

Idea:

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

Then we obtain **ordered binary decision diagrams** or **OBDDs**.

OBDDs



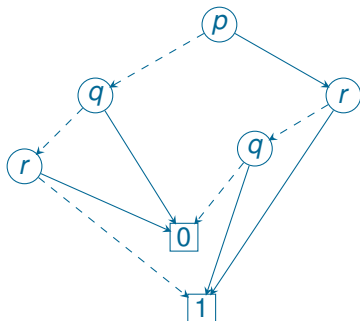
Problem: on different branches on the dag tests are made in a different order.

Idea:

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

Then we obtain **ordered binary decision diagrams** or **OBDDs**.

OBDDs



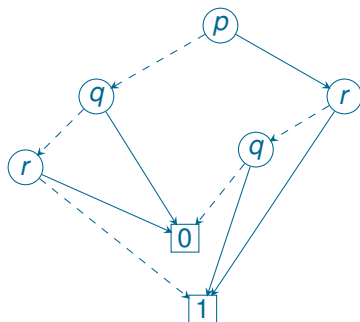
Problem: on different branches on the dag tests are made in a different order.

Idea:

- ▶ introduce an **order** $>$ on atoms;
- ▶ make **tests** in this order.

Then we obtain **ordered binary decision diagrams** or **OBDDs**.

OBDDs



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

Idea:

- ▶ introduce an **order** $>$ on atoms;
- ▶ make **tests** in this order.

Then we obtain **ordered binary decision diagrams** or **OBDDs**.

Properties

- ▶ Satisfiability checking can be done in constant time;
- ▶ Validity checking can be done in constant time;
- ▶ Equivalence checking can be done in constant time;
- ▶ Boolean operations e.g., conjunction, are easy to implement.

Properties

- ▶ Satisfiability checking can be done in constant time;
- ▶ Validity checking can be done in constant time;
- ▶ Equivalence checking can be done in constant time;
- ▶ Boolean operations e.g., conjunction, are easy to implement.

Properties

- ▶ Satisfiability checking can be done in constant time;
- ▶ Validity checking can be done in constant time;
- ▶ Equivalence checking can be done in constant time;
- ▶ Boolean operations e.g., conjunction, are easy to implement.

Algorithms on OBDDs

Suppose we have to compute an OBDD that represents a boolean function $f(b_1, \dots, b_n)$, for example $b_1 \vee \dots \vee b_n$ and we have already built OBDDs for b_1, \dots, b_n .

- ▶ We assume a **global dag** that contains all OBDDs so that all isomorphic subdags are shared.
- ▶ Use the property

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

Algorithms on OBDDs

Suppose we have to compute an OBDD that represents a boolean function $f(b_1, \dots, b_n)$, for example $b_1 \vee \dots \vee b_n$ and we have already built OBDDs for b_1, \dots, b_n .

- ▶ We assume a **global dag** that contains all OBDDs so that all isomorphic subdags are **shared**.
- ▶ Use the property

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

Algorithms on OBDDs

Suppose we have to compute an OBDD that represents a boolean function $f(b_1, \dots, b_n)$, for example $b_1 \vee \dots \vee b_n$ and we have already built OBDDs for b_1, \dots, b_n .

- ▶ We assume a **global dag** that contains all OBDDs so that all isomorphic subdags are **shared**.
- ▶ Use the property

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

Integrating a node in a dag

procedure *integrate*(n_1, p, n_2, D)

parameters: global dag D

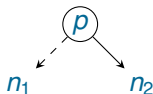
input: nodes n_1, n_2 in D representing formulas F_1, F_2 , variable p

output: node n in (modified) D representing *if* p *then* F_1 *else* F_2

begin

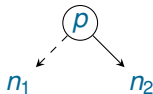
if $n_1 = n_2$ **then return** n_1 ;

if D contains a node n having the form



then return n ;

add to D a new node n of the form



return n

end

Building OBDDs

procedure *obdd*(F)

input: propositional formula F

parameters: global dag D

output: a node n in (modified) D which represents F

begin

$F := \text{simplify}(F)$

if $F = \perp$ **then return** $\boxed{0}$

if $F = \top$ **then return** $\boxed{1}$

$p := \text{max_atom}(F)$

$n_1 := \text{obdd}(F_p^\perp)$

$n_2 := \text{obdd}(F_p^\top)$

return $\text{integrate}(n_1, p, n_2, D)$

end

Disjunction

procedure *disjunction*(n_1, \dots, n_m)

parameters: global dag D

input: nodes n_1, \dots, n_m representing F_1, \dots, F_m in D

output: a node n representing $F_1 \vee \dots \vee F_m$ in (modified) D

begin

if some n_i is 1 **then return** 1

if $m = 1$ **then return** n_1

if some n_i is 0 **then**

return *disjunction*($n_1, \dots, n_{i-1}, n_{i+1}, \dots, n_m$)

$p := \text{max_atom}(n_1, \dots, n_m)$

forall $i = 1 \dots m$

if n_i is labelled by p

then $(l_i, r_i) := (\text{neg}(n_i), \text{pos}(n_i))$

else $(l_i, r_i) := (n_i, n_i)$

$k_1 := \text{disjunction}(l_1, \dots, l_m)$

$k_2 := \text{disjunction}(r_1, \dots, r_m)$

return *integrate*(k_1, p, k_2, D)

end