CS:4350 Logic in Computer Science

Binary Decision Diagrams

Cesare Tinelli

Spring 2021

Credits

These slides are largely based on slides originally developed by **Andrei Voronkov** at the University of Manchester. Adapted by permission.

Outline

Data Structures for Large Propositional Formulas

In some applications, large propositional formulas are reused repeatedly

Data Structures for Large Propositional Formulas

In some applications, large propositional formulas are reused repeatedly

For example, we may

- build a conjunction of several formulas
- negate a formula
- check if two formulas are equivalent
- …

Data Structures for Large Propositional Formulas

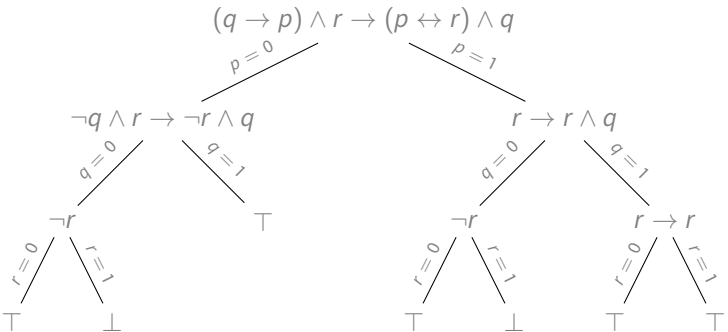
In some applications, large propositional formulas are reused repeatedly

We need data structures that

- provide a compact representation of formulas
  (or the Boolean functions they represent)
- facilitate Boolean operations on these formulas
  (e.g., building conjunctions of them);
- facilitate checking properties of these formulas
  (e.g., satisfiability, equivalence„ …)

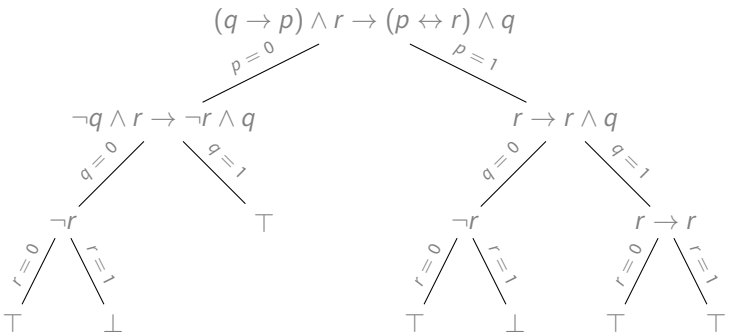
Splitting Tree

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



Splitting Tree

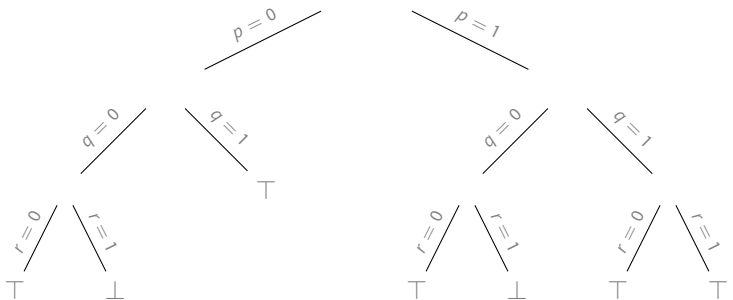
$$A = (q \to p) \wedge r \to (p \leftrightarrow r)$$



Let us ignore the concrete formulas in the tree

Splitting Tree

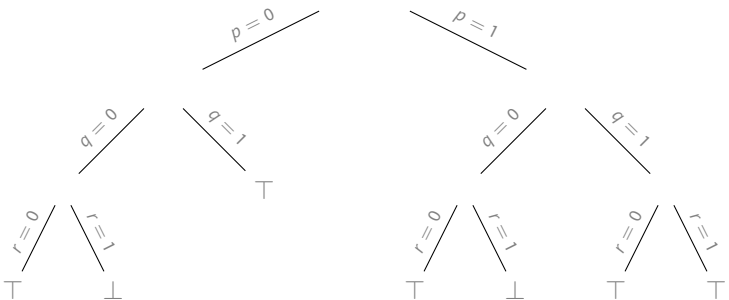
$$A = (q \to p) \wedge r \to (p \leftrightarrow r)$$



The semantics of formula A is preserved: the tree encodes all models of A

Splitting Tree

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



The semantics of formula A is preserved: the tree encodes all models of A

Any formula with the same tree has exactly the same models as A

Binary Decision Tree

$$\mathbb{B} = \{0, 1\}$$

Note: propositional formulas also represent Boolean functions

Example:

$$\begin{array}{lll} A_1 & = & p_1 \to p_2 \\ A_2 & = & p_2 \leftrightarrow p_3 \\ A_3 & = & p \wedge q \\ A_4 & = & (p_1 \to p_2) \wedge (p_2 \leftrightarrow p_3) \end{array} \qquad \begin{array}{l} f_1 : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ f_2 : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ f_3 : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ f_4 : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B} \end{array}$$

$$f_4(p_1, p_2, p_3) := \textit{if } p_1 \textit{ then } (\textit{if } p_2 \textit{ then } p_3 \textit{ else } 0) \textit{ else if } (p_2 = p_3) \textit{ then } 1 \textit{ else } 0$$

Exercise: Convince yourself that for any interpretation $\mathcal{I}$,

$$\mathcal{I} \models A_4 \text{ iff } f_4(\mathcal{I}(p_1), \mathcal{I}(p_2), \mathcal{I}(p_3)) = 1$$

Binary Decision Tree

$$\mathbb{B} = \{0, 1\}$$

Note: propositional formulas also represent Boolean functions

Example:

$$\begin{array}{lcll} A_1 & = & p_1 \to p_2 & f_1 : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ A_2 & = & p_2 \leftrightarrow p_3 & f_2 : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ A_3 & = & p \wedge q & f_3 : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ A_4 & = & (p_1 \to p_2) \wedge (p_2 \leftrightarrow p_3) & f_4 : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B} \end{array}$$

$$f_4(p_1, p_2, p_3) := \textit{if } p_1 \textit{ then } (\textit{if } p_2 \textit{ then } p_3 \textit{ else } 0) \textit{ else if } (p_2 = p_3) \textit{ then } 1 \textit{ else } 0$$

Exercise: Convince yourself that for any interpretation $\mathcal{I}$,

$$\mathcal{I} \models A_4 \text{ iff } f_4(\mathcal{I}(p_1), \mathcal{I}(p_2), \mathcal{I}(p_3)) = 1$$

Binary Decision Tree

$$\mathbb{B} = \{0, 1\}$$

Note: propositional formulas also represent Boolean functions

Example:

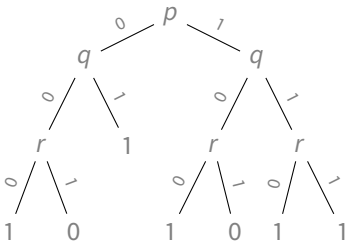
$$\begin{array}{lcll} A_1 & = & p_1 \to p_2 & f_1 : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ A_2 & = & p_2 \leftrightarrow p_3 & f_2 : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ A_3 & = & p \wedge q & f_3 : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \\ A_4 & = & (p_1 \to p_2) \wedge (p_2 \leftrightarrow p_3) & f_4 : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B} \end{array}$$

$$f_4(p_1, p_2, p_3) := \textit{ if } p_1 \textit{ then } (\textit{if } p_2 \textit{ then } p_3 \textit{ else } 0) \textit{ else if } (p_2 = p_3) \textit{ then } 1 \textit{ else } 0$$

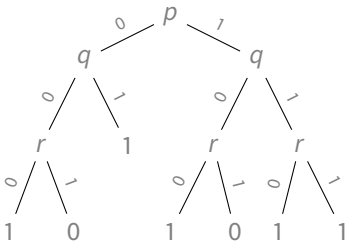
Exercise: Convince yourself that for any interpretation $\mathcal{I}$,

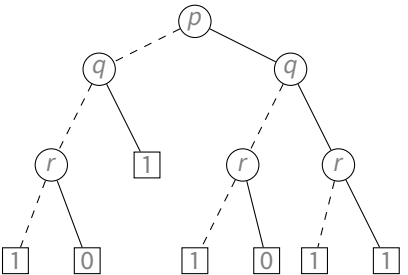
$$\mathcal{I} \models A_4 \text{ iff } f_4(\mathcal{I}(p_1), \mathcal{I}(p_2), \mathcal{I}(p_3)) = 1$$

Binary Decision Tree

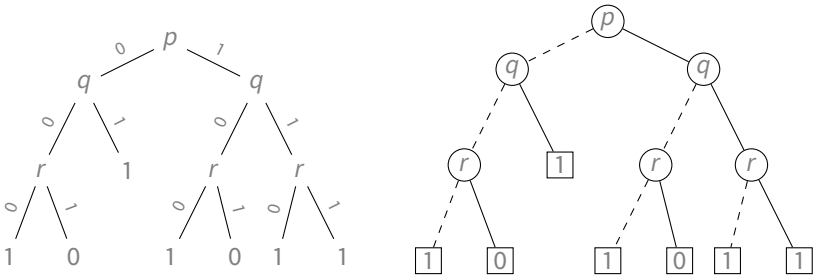


Binary Decision Tree



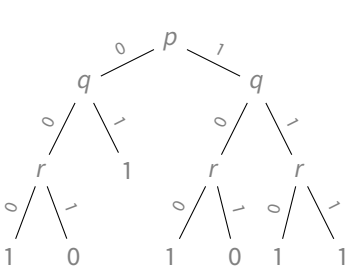


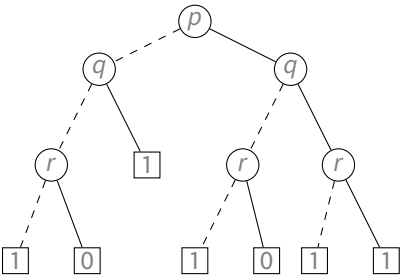
Binary Decision Tree



A circled node, e.g., $\textcircled{p}$, denotes the decision on the (input) variable in the node

Binary Decision Tree

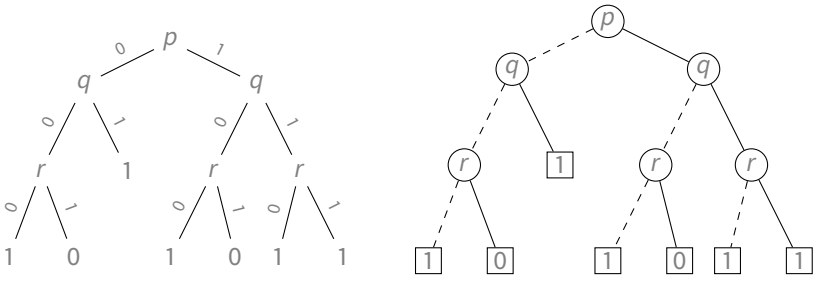




A circled node, e.g., (p), denotes the decision on the (input) variable in the node

Leaf nodes are squared, e.g., [1], and denote output values

Binary Decision Tree

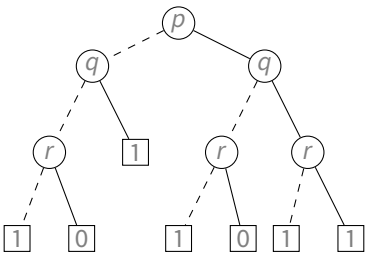


A circled node, e.g., (p), denotes the decision on the (input) variable in the node

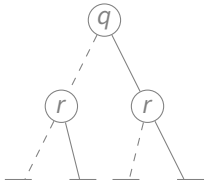
Leaf nodes are squared, e.g., [1], and denote output values

Solid lines correspond to value 1 and dashed lines to value 0 for the variable

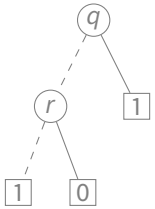
Nodes as “if _ then _ else” tests



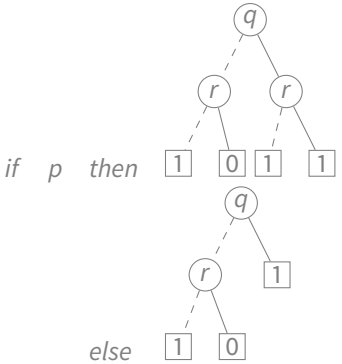
if p then

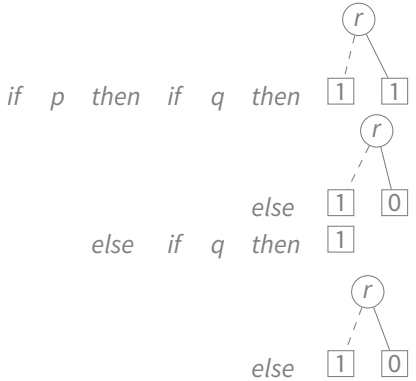


else



Nodes as “if-then-else” tests





Tests correspond to “if-then-else”

if	*p*	*then*	*if*	*q*	*then*	*if*	*r*	*then*	1
								else	1
					else	*if*	*r*	*then*	1
								else	0
		else	*if*	*q*	*then*	1			
					else	*if*	*r*	*then*	1
								else	0

Note:

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

Tests correspond to “if-then-else”

$$
\begin{array}{llllllllll}
\mathit{if} & p & \mathit{then} & \mathit{if} & q & \mathit{then} & \mathit{if} & r & \mathit{then} & 1 \\
 & & & & & & & & \mathit{else} & 1 \\
 & & & & & \mathit{else} & \mathit{if} & r & \mathit{then} & 1 \\
 & & & & & & & & \mathit{else} & 0 \\
 & & \mathit{else} & \mathit{if} & q & \mathit{then} & 1 & & & \\
 & & & & & \mathit{else} & \mathit{if} & r & \mathit{then} & 1 \\
 & & & & & & & & \mathit{else} & 0
\end{array}
$$

Note:

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

If-Then-Else Normal Form

Any formula can be converted to an equivalent one in *If-Then-Else Normal Form*:

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

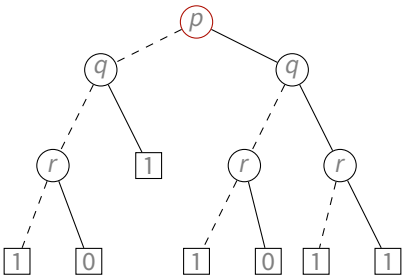
Evaluating the Formula

We can evaluate a formula on in interpretation $\mathcal{I}$ if we know its binary decision tree

Evaluating the Formula

We can evaluate a formula on in interpretation $\mathcal{I}$ if we know its binary decision tree

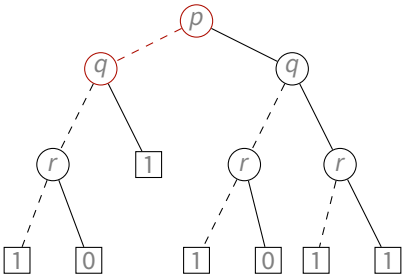
Example $\mathcal{I} = \{ p \mapsto 0, q \mapsto 0, r \mapsto 1 \}$



Evaluating the Formula

We can evaluate a formula on in interpretation $\mathcal{I}$ if we know its binary decision tree

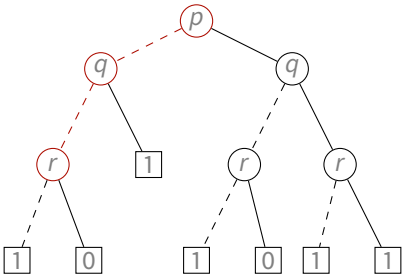
Example $\mathcal{I} = \{ p \mapsto 0, q \mapsto 0, r \mapsto 1 \}$



Evaluating the Formula

We can evaluate a formula on in interpretation $\mathcal{I}$ if we know its binary decision tree

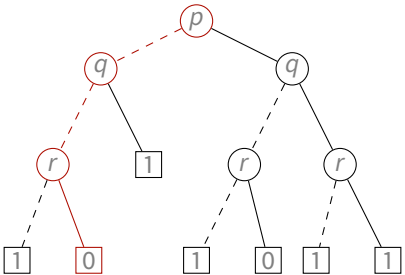
Example $\mathcal{I} = \{ p \mapsto 0, q \mapsto 0, r \mapsto 1 \}$



Evaluating the Formula

We can evaluate a formula on in interpretation $\mathcal{I}$ if we know its binary decision tree

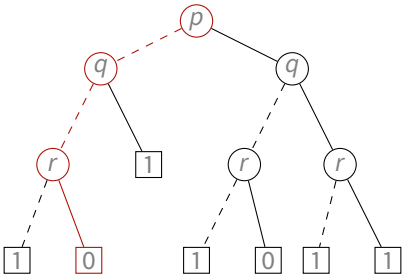
Example $\mathcal{I} = \{ p \mapsto 0, q \mapsto 0, r \mapsto 1 \}$



Evaluating the Formula

We can evaluate a formula on in interpretation $\mathcal{I}$ if we know its binary decision tree

Example $\mathcal{I} = \{ p \mapsto 0, q \mapsto 0, r \mapsto 1 \}$



Any formula with this decision tree is false in this interpretation

Properties

Properties of binary decision trees (n = number of vars, s = tree size):

Properties

Properties of binary decision trees (n = number of vars, s = tree size):

- Size s is exponential in n in the worst case

Properties

Properties of binary decision trees (n = number of vars, s = tree size):

- Size s is exponential in n in the worst case

One needs data structures that

- facilitate checking properties of formulas, e.g., satisfiability or equivalence

Properties

Properties of binary decision trees (n = number of vars, s = tree size):

- Size s is exponential in n in the worst case
- Checking truth in an interpretation can be done in time linear in n

One needs data structures that

- facilitate checking properties of formulas, e.g., satisfiability or equivalence

Properties

Properties of binary decision trees (n = number of vars, s = tree size):

- Size s is exponential in n in the worst case
- Checking truth in an interpretation can be done in time linear in n
- Satisfiability/validity checking can be done in time linear in s

One needs data structures that

- facilitate checking properties of formulas, e.g., satisfiability or equivalence

Properties

Properties of binary decision trees (n = number of vars, s = tree size):

- Size s is exponential in n in the worst case
- Checking truth in an interpretation can be done in time linear in n
- Satisfiability/validity checking can be done in time linear in s

One needs data structures that

- facilitate checking properties of formulas, e.g., satisfiability or equivalence

Properties

Properties of binary decision trees (n = number of vars, s = tree size):

- Size s is exponential in n in the worst case
- Checking truth in an interpretation can be done in time linear in n
- Satisfiability/validity checking can be done in time linear in s
- Equivalence checking is very hard

One needs data structures that

- facilitate checking properties of formulas, e.g., satisfiability or equivalence

Properties

Properties of binary decision trees (n = number of vars, s = tree size):

- Size s is exponential in n in the worst case
- Checking truth in an interpretation can be done in time linear in n
- Satisfiability/validity checking can be done in time linear in s
- Equivalence checking is very hard

One needs data structures that

- facilitate checking properties of formulas, e.g., satisfiability or equivalence
- facilitate boolean operations on formulas, e.g., conjunctions

Properties

Properties of binary decision trees (n = number of vars, s = tree size):

- Size s is exponential in n in the worst case
- Checking truth in an interpretation can be done in time linear in n
- Satisfiability/validity checking can be done in time linear in s
- Equivalence checking is very hard
- Some boolean operations, ($\wedge$) are hard to implement

One needs data structures that

- facilitate checking properties of formulas, e.g., satisfiability or equivalence
- facilitate boolean operations on formulas, e.g., conjunctions

Properties

Properties of binary decision trees (n = number of vars, s = tree size):

- Size s is exponential in n in the worst case
- Checking truth in an interpretation can be done in time linear in n
- Satisfiability/validity checking can be done in time linear in s
- Equivalence checking is very hard
- Some boolean operations, ($\wedge$) are hard to implement

One needs data structures that

- facilitate checking properties of formulas, e.g., satisfiability or equivalence
- facilitate boolean operations on formulas, e.g., conjunctions
- provide a compact representation of formulas, or the Boolean functions they represent

Properties

Properties of binary decision trees (n = number of vars, s = tree size):

- Size s is exponential in n in the worst case
- Checking truth in an interpretation can be done in time linear in n
- Satisfiability/validity checking can be done in time linear in s
- Equivalence checking is very hard
- Some boolean operations, ($\wedge$) are hard to implement

One needs data structures that

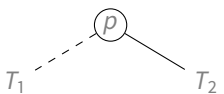
- facilitate checking properties of formulas, e.g., satisfiability or equivalence
- facilitate boolean operations on formulas, e.g., conjunctions
- provide a compact representation of formulas, or the Boolean functions they represent

Are binary decision trees compact?

Algorithm for Building Binary Decision Trees

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

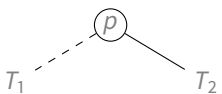
- $simplify(A)$ as in the splitting procedure
- $tree(T_1, p, T_2)$ builds the tree:



Algorithm for Building Binary Decision Trees

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

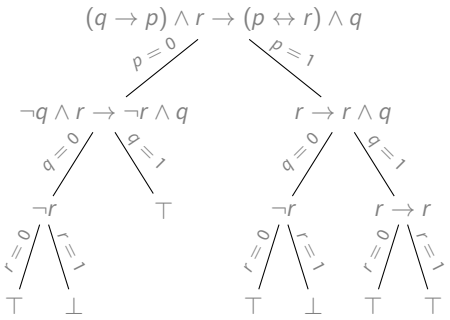
- $simplify(A)$ as in the splitting procedure
- $tree(T_1, p, T_2)$ builds the tree:



Note resemblance to the splitting procedure!

Example

Splitting Procedure



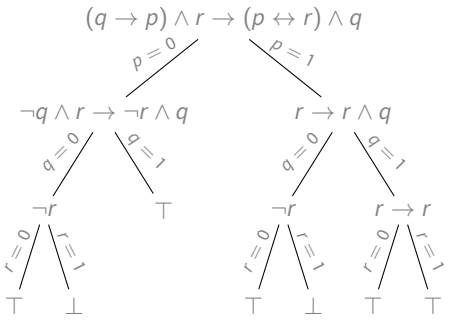
Explored search tree (conceptual)

BDT Procedure

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

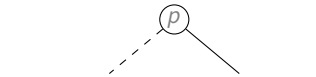
Example

Splitting Procedure



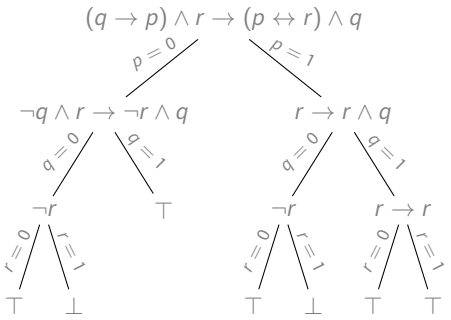
Explored search tree (conceptual)

BDT Procedure



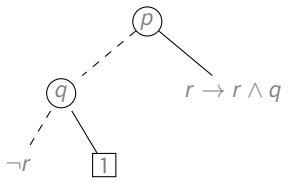
Example

Splitting Procedure



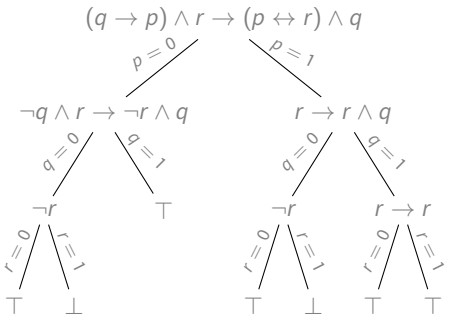
Explored search tree (conceptual)

BDT Procedure



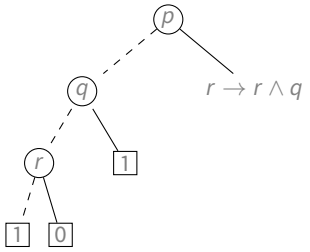
Example

Splitting Procedure



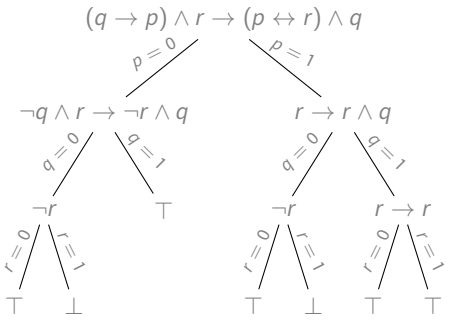
Explored search tree (conceptual)

BDT Procedure



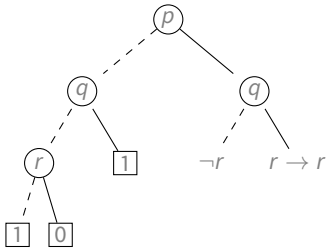
Example

Splitting Procedure



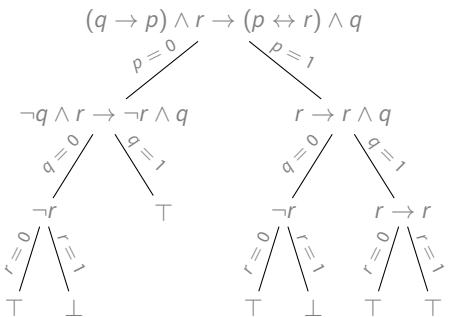
Explored search tree (conceptual)

BDT Procedure



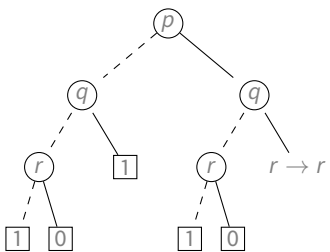
Example

Splitting Procedure



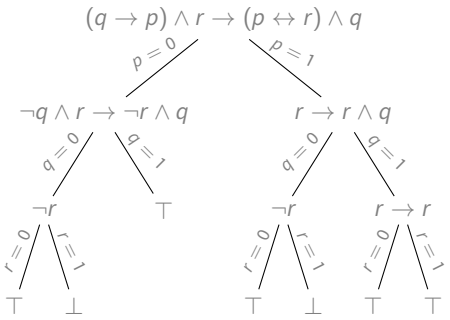
Explored search tree (conceptual)

BDT Procedure



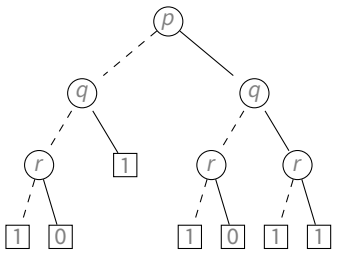
Example

Splitting Procedure



Explored search tree (conceptual)

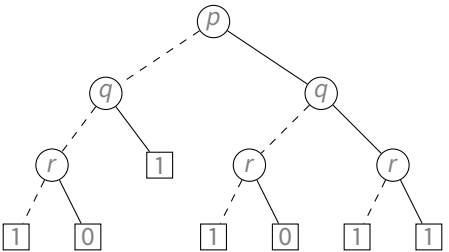
BDT Procedure



Returned decision tree (actual data structure)

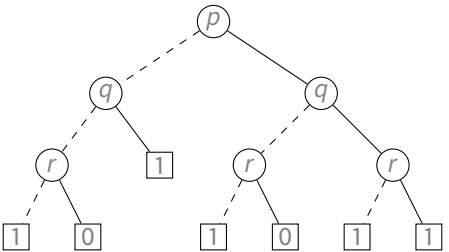
Redundant Tests

Are binary decision trees compact?



Redundant Tests

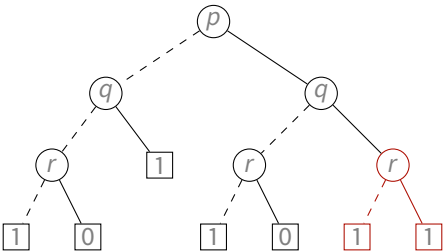
Are binary decision trees compact? No



Redundant Tests

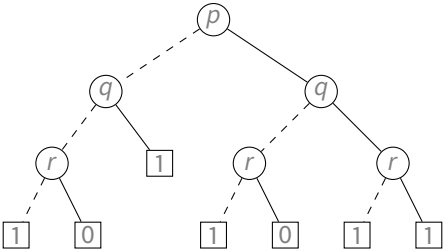
Are binary decision trees compact? No

They may contain redundant tests (nodes):



Isomorphic Subtrees

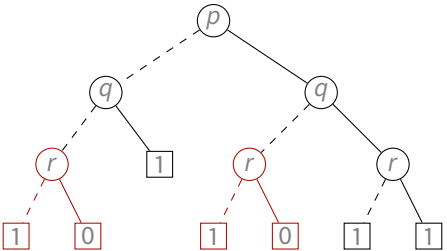
Are binary decision trees compact? No



Isomorphic Subtrees

Are binary decision trees compact? No

They may contain isomorphic subtrees:



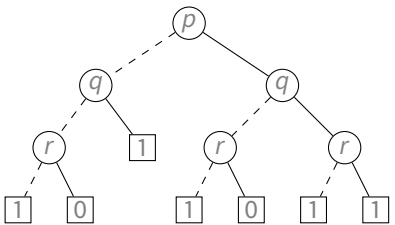
Binary Decision Diagrams

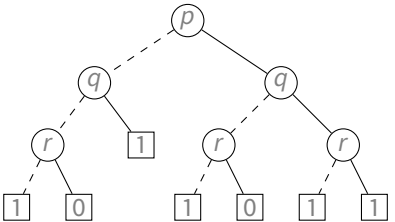
A *binary decision diagram*, or *BDD*, is a directed acyclic graph (built like a BDT but) containing

- no redundant nodes
- no isomorphic subgraphs

From BDTs to BDDs

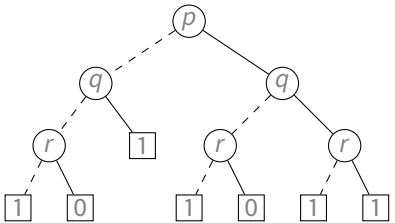
Binary Decision Tree $\Rightarrow$ Binary Decision Diagram

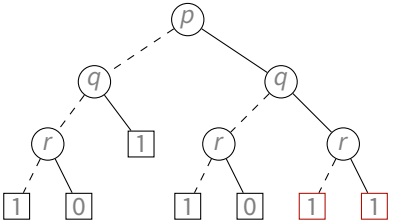




From BDTs to BDDs

Binary Decision Tree ⇒ Binary Decision Diagram

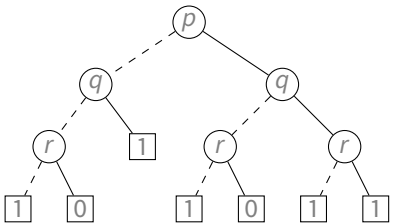


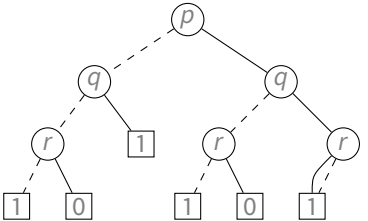


1. Merge isomorphic subgraphs
2. Eliminate redundant node

From BDTs to BDDs

Binary Decision Tree ⇒ Binary Decision Diagram

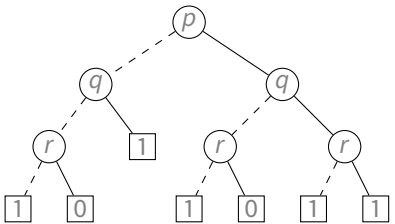


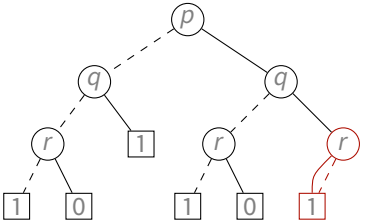


1. Merge isomorphic subgraphs
2. Eliminate redundant node

From BDTs to BDDs

Binary Decision Tree ⇒ Binary Decision Diagram

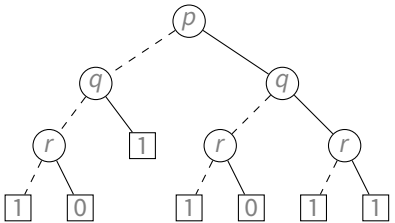


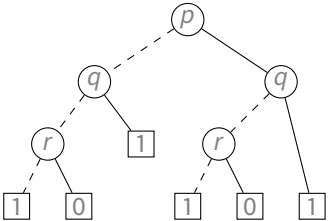


1. Merge isomorphic subgraphs
2. Eliminate redundant node

From BDTs to BDDs

Binary Decision Tree ⇒ Binary Decision Diagram

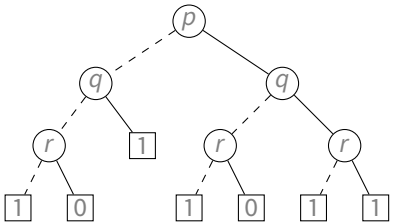


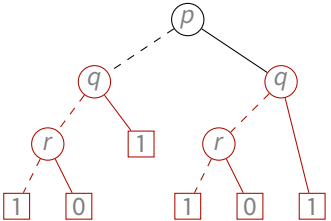


1. Merge isomorphic subgraphs
2. Eliminate redundant node

From BDTs to BDDs

Binary Decision Tree ⇒ Binary Decision Diagram

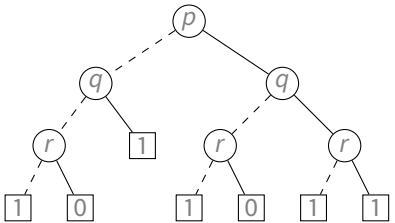


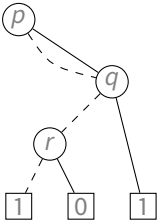


1. Merge isomorphic subgraphs
2. Eliminate redundant node

From BDTs to BDDs

Binary Decision Tree ⇒ Binary Decision Diagram

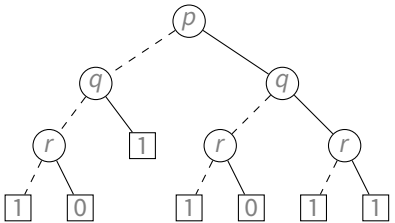


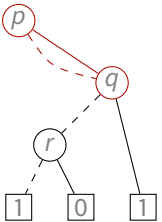


1. Merge isomorphic subgraphs
2. Eliminate redundant node

From BDTs to BDDs

Binary Decision Tree ⇒ Binary Decision Diagram

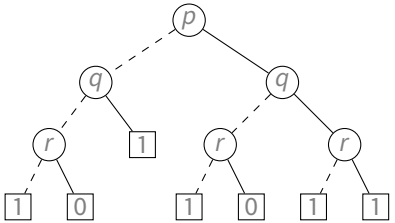


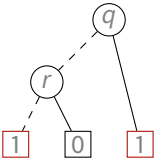


1. Merge isomorphic subgraphs
2. Eliminate redundant node

From BDTs to BDDs

Binary Decision Tree ⇒ Binary Decision Diagram

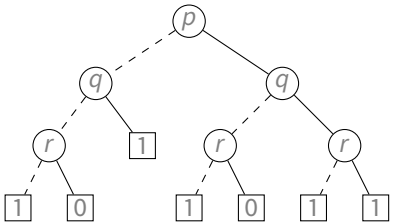


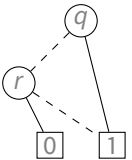


1. Merge isomorphic subgraphs
2. Eliminate redundant node

From BDTs to BDDs

Binary Decision Tree $\Rightarrow$ Binary Decision Diagram

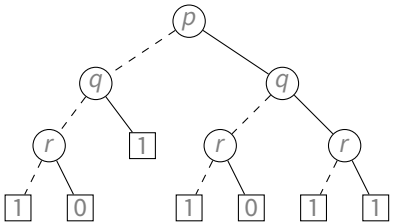


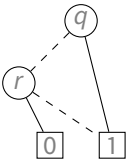


1. Merge isomorphic subgraphs
2. Eliminate redundant node

From BDTs to BDDs

Binary Decision Tree $\Rightarrow$ Binary Decision Diagram

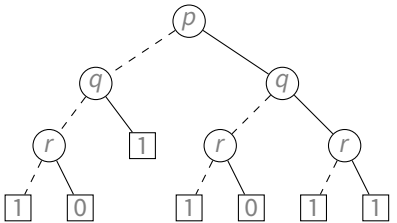


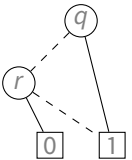


The original diagram and the *reduced* one represent the same Boolean function

From BDTs to BDDs

Binary Decision Tree $\Rightarrow$ Binary Decision Diagram





The original diagram and the *reduced* one represent the same Boolean function

Compact formula for that function: $(\neg q \wedge \neg r) \vee q$

Even more compact formula: $\neg r \vee q$

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 ($\wedge$)

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 ($\wedge$)

Properties

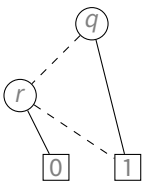
What is the complexity of satisfiability, validity and equivalence checking for BDDs?

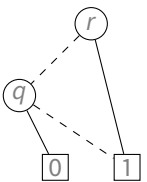
- Satisfiability checking can be done in constant time
- Validity checking can be done in constant time
- Equivalence checking
- Some Boolean operations ($\wedge$)

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 ($\wedge$)

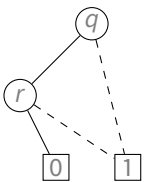


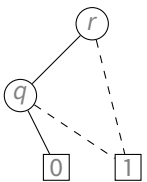


Properties

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 ($\wedge$)

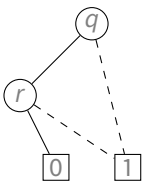


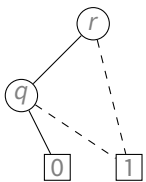


Properties

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 still very hard (exponential in the number of vars)
- Some Boolean operations ($\wedge$)

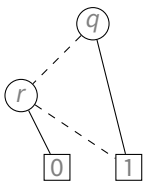


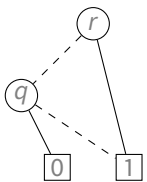


Properties

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 still very hard (exponential in the number of vars)
- Some Boolean operations ($\wedge$)

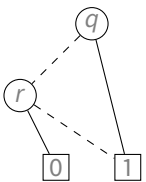


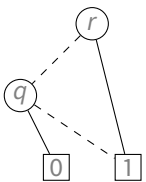


Properties

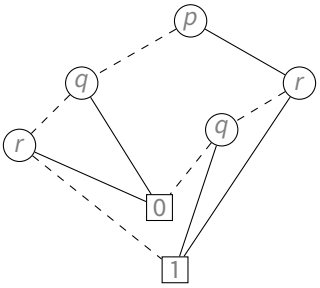
What is the complexity of satisfiability, validity and equivalence checking for BDDs?

- Satisfiability checking can be done in constant time
- Validity checking can be done in constant time
- Equivalence checking is still very hard (exponential in the number of vars)
- Some Boolean operations ($\wedge$) are still hard to implement

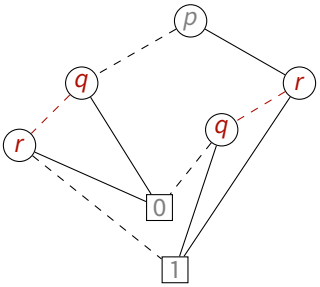




Ordered BDDs

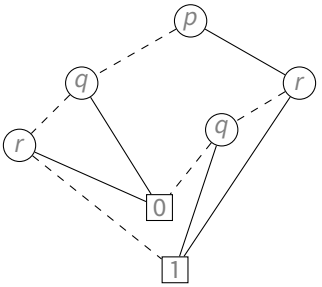


Ordered BDDs



Problem: variables are checked in a different order on different branches

Ordered BDDs

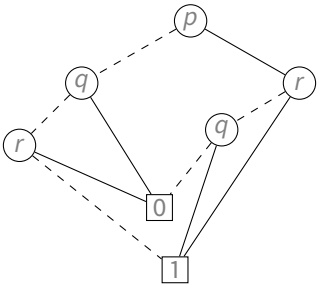


Problem: variables are checked in a different order on different branches

Idea:

- introduce an order $>$ on variables
- perform tests in this order in each branch

Ordered BDDs



Problem: variables are checked in a different order on different branches

Idea:

- introduce an order $>$ on variables
- perform tests in this order in each branch

We then we obtain ordered binary decision diagrams, or OBDDs

OBDDs Properties

- Satisfiability checking in constant time
- Validity checking in constant time

OBDDs Properties

- Satisfiability checking in constant time
- Validity checking in constant time
- Equivalence checking in constant time

OBDDs Properties

- Satisfiability checking in constant time
- Validity checking in constant time
- Equivalence checking in constant time
- Boolean operations ($\wedge$) easy to implement

Integrating a node in a dag

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

Integrating a node in a dag

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

Integrating a node in a dag

procedure *integrate*(n_1, p, n_2)
parameters: global dag D
input: variable p, nodes n_1, n_2 in D representing formulas F_1, F_2
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

end

Integrating a node in a dag

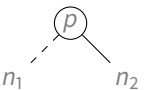
procedure *integrate*(n_1, p, n_2)
parameters: global dag D
input: variable p, nodes n_1, n_2 in D representing formulas F_1, F_2
output: node n in (modified) D representing *if* p *then* F_1 *else* F_2
begin
if $n_1 = n_2$ **then return** n_1
if D contains a node n having the form

then return n

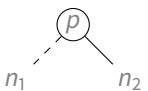
end

Integrating a node in a dag

procedure *integrate*(n_1, p, n_2)
parameters: global dag D
input: variable p, nodes n_1, n_2 in D representing formulas F_1, F_2
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
else 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
  $F := simplify(F)$ // usual simplifications with rewrite rules
  if $F = \bot$ **then return** $\boxed{0}$
  if $F = \top$ **then return** $\boxed{1}$
  $p := max\_variable(F)$ // var of F highest in variable ordering
  $n_1 := obdd(F_p^{\bot})$
  $n_2 := obdd(F_p^{\top})$
  return $integrate(n_1, p, n_2)$
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
  $F := simplify(F)$ // usual simplifications with rewrite rules
  if $F = \bot$ **then return** $\boxed{0}$
  if $F = \top$ **then return** $\boxed{1}$
  $p := max\_variable(F)$ // var of F highest in variable ordering
  $n_1 := obdd(F_p^{\bot})$
  $n_2 := obdd(F_p^{\top})$
  return $integrate(n_1, p, n_2)$
end

- *obdd* puts together the algorithms for building BDTs and for eliminating redundancies

Building OBDDs

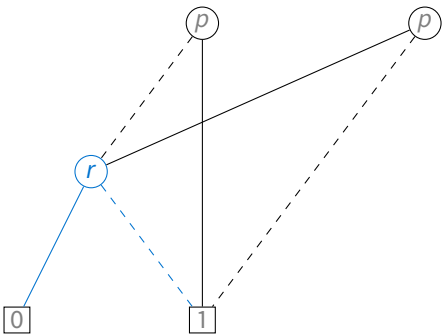
procedure *obdd*(F)
input: propositional formula F
parameters: global dag D
output: a node n in (modified) D which represents F
begin
  F := *simplify*(F) // usual simplifications with rewrite rules
  if $F = \bot$ **then return** $\boxed{0}$
  if $F = \top$ **then return** $\boxed{1}$
  p := *max_variable*(F) // var of F highest in variable ordering
  n_1 := *obdd*$(F_p^{\bot})$
  n_2 := *obdd*$(F_p^{\top})$
  return *integrate*(n_1, p, n_2)
end

- *obdd* puts together the algorithms for building BDTs and for eliminating redundancies
- Redundancy elimination is performed by *integrate*

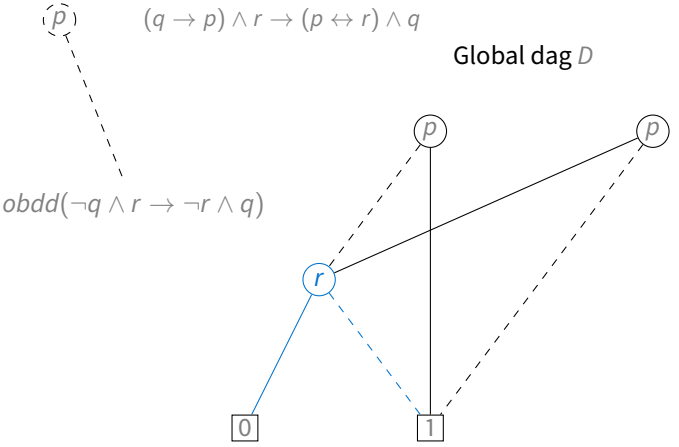
Building OBDDs, Example

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

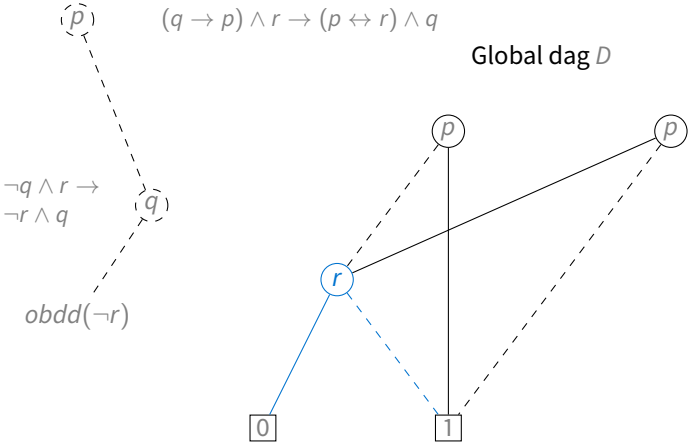
Global dag D



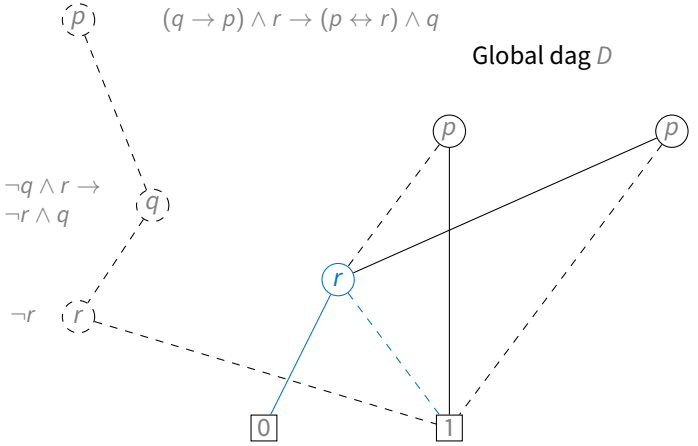
Building OBDDs, Example



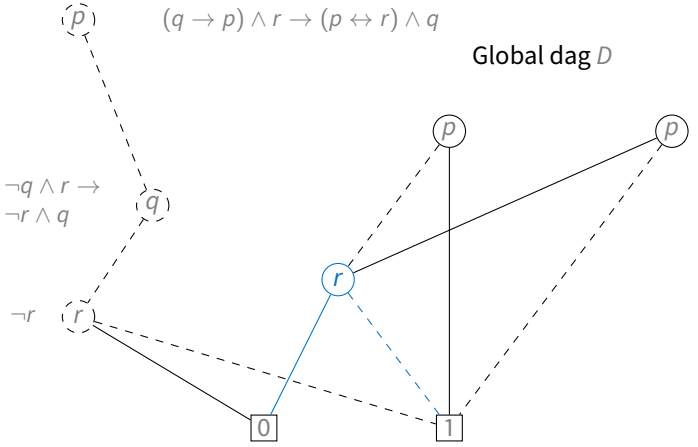
Building OBDDs, Example



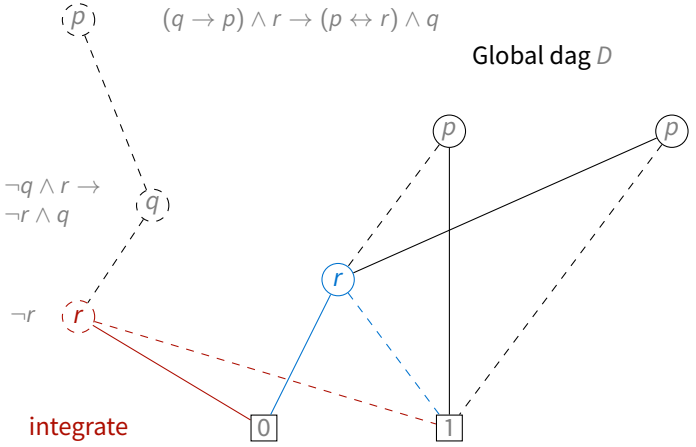
Building OBDDs, Example



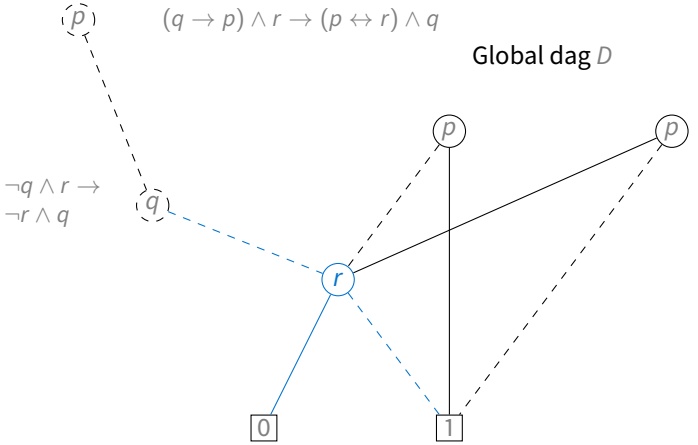
Building OBDDs, Example



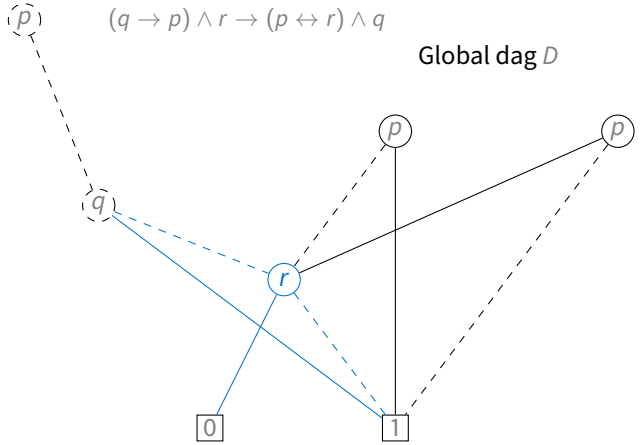
Building OBDDs, Example



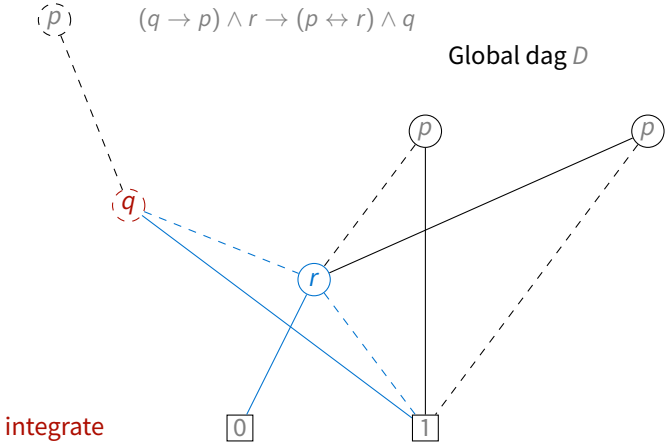
Building OBDDs, Example



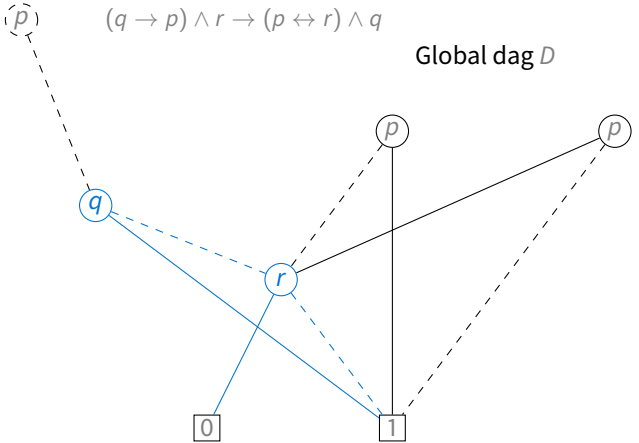
Building OBDDs, Example



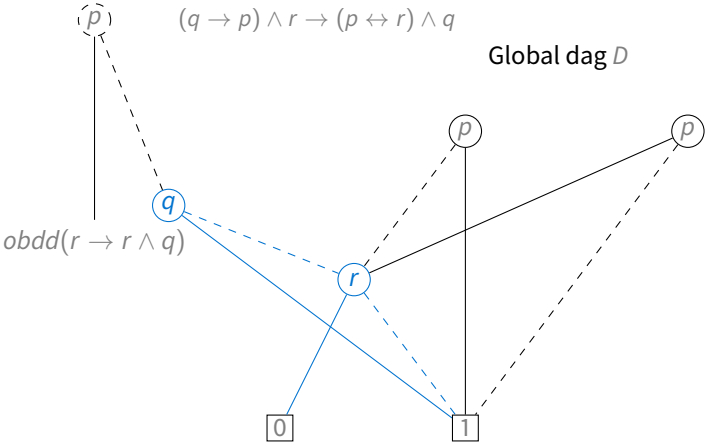
Building OBDDs, Example



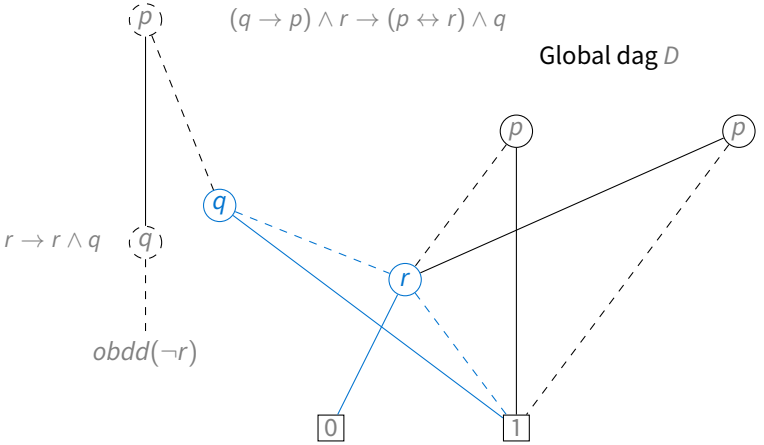
Building OBDDs, Example



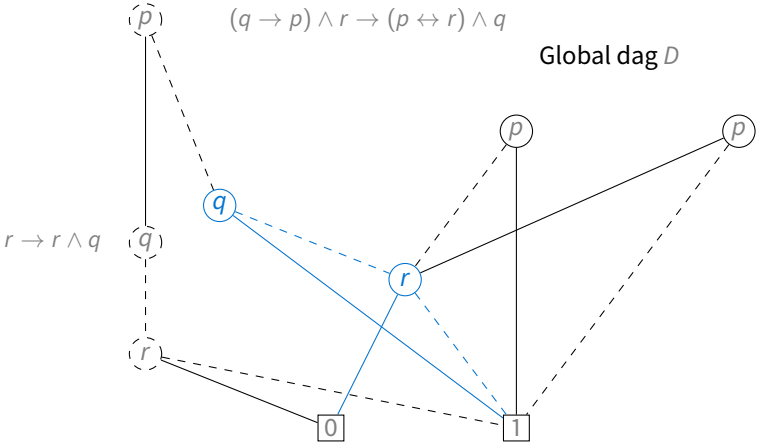
Building OBDDs, Example



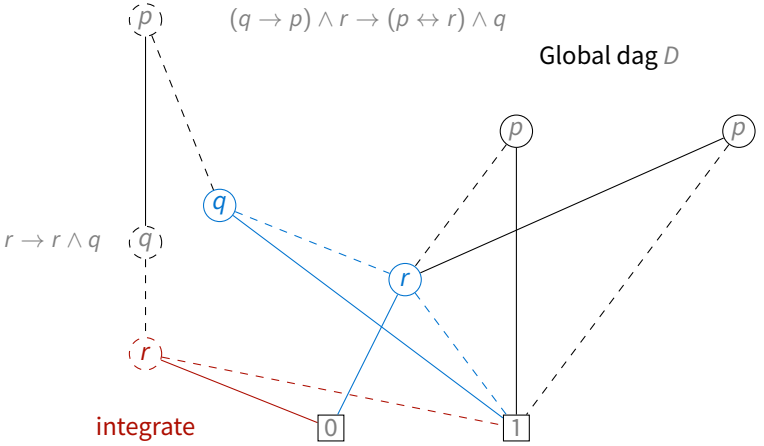
Building OBDDs, Example



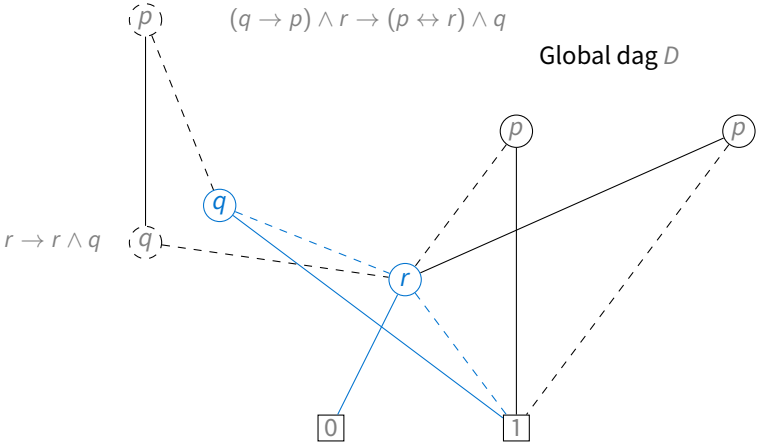
Building OBDDs, Example



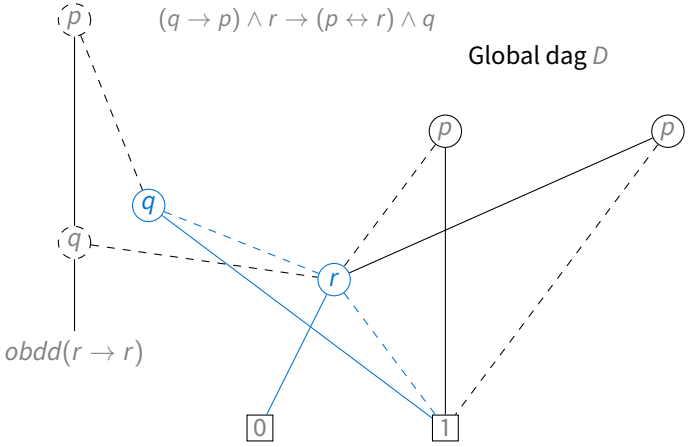
Building OBDDs, Example



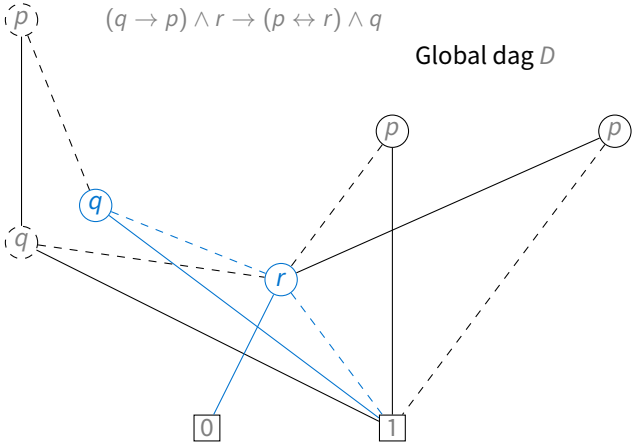
Building OBDDs, Example



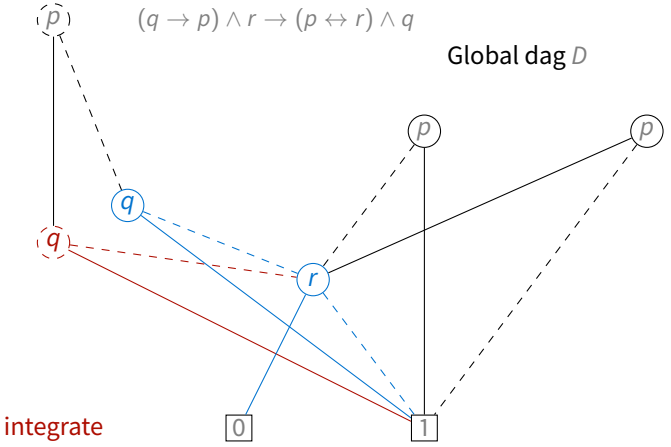
Building OBDDs, Example



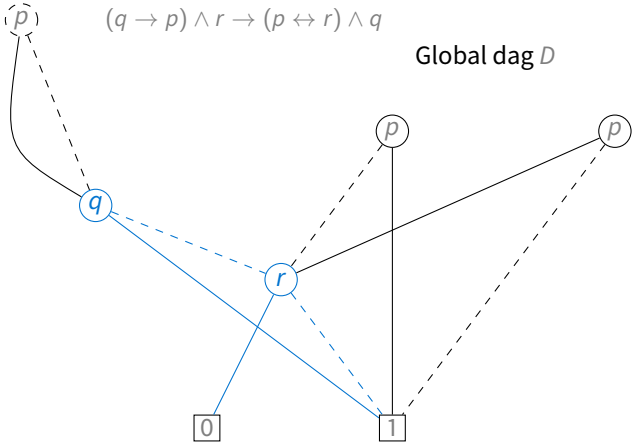
Building OBDDs, Example



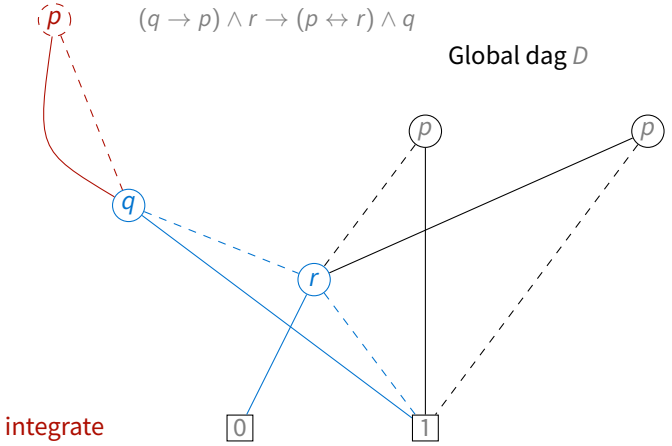
Building OBDDs, Example



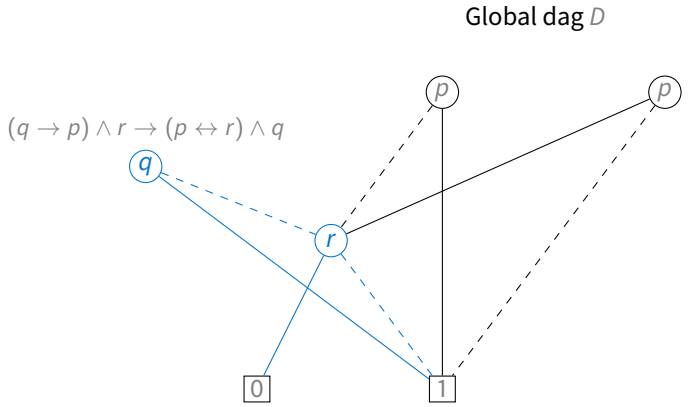
Building OBDDs, Example



Building OBDDs, Example

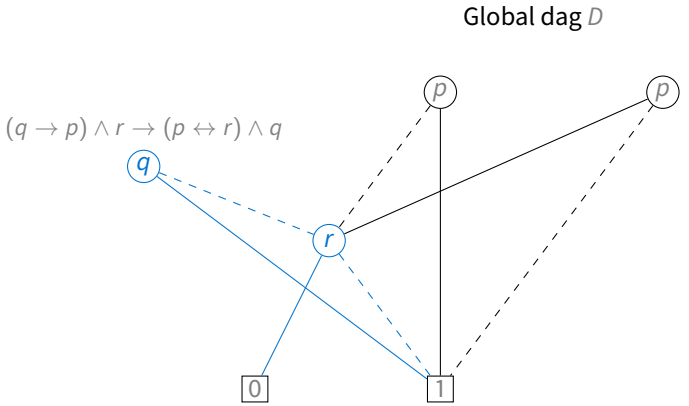


Building OBDDs, Example



We return the new node rooted at q

Building OBDDs, Example



We return the new node rooted at q

Note: The application of this procedure modified the global dag

Algorithms on OBDDs

Let $f(x_1, \ldots, x_n) \stackrel{\text{def}}{=} x_1 \vee \ldots \vee x_n$

Let $D_1, \ldots, D_n$ be OBDDs representing formulas $F_1, \ldots, F_n$, respectively

How do we compute the OBDD representing $f(F_1, \ldots, F_n)$?

Algorithms on OBDDs

Let $f(x_1, \ldots, x_n) \stackrel{\text{def}}{=} x_1 \vee \ldots \vee x_n$

Let $D_1, \ldots, D_n$ be OBDDs representing formulas $F_1, \ldots, F_n$, respectively

How do we compute the OBDD representing $f(F_1, \ldots, F_n)$?

Algorithms on OBDDs

Let $f(x_1, \ldots, x_n) \stackrel{\text{def}}{=} x_1 \vee \ldots \vee x_n$

Let $D_1, \ldots, D_n$ be OBDDs representing formulas $F_1, \ldots, F_n$, respectively

How do we compute the OBDD representing $f(F_1, \ldots, F_n)$?

- We fix the same variable ordering for all OBDDs
- We assume isomorphic subdags are shared across different OBDDs
- We use one fundamental property of *if _ then _ else _*

Algorithms on OBDDs

Let $f(x_1, \ldots, x_n) \stackrel{\text{def}}{=} x_1 \vee \ldots \vee x_n$

Let $D_1, \ldots, D_n$ be OBDDs representing formulas $F_1, \ldots, F_n$, respectively

How do we compute the OBDD representing $f(F_1, \ldots, F_n)$?

- We fix the same variable ordering for all OBDDs
- We assume isomorphic subdags are shared across different OBDDs
- We use one fundamental property of *if _ then _ else _*

Algorithms on OBDDs

Let $f(x_1, \ldots, x_n) \stackrel{\text{def}}{=} x_1 \vee \ldots \vee x_n$

Let $D_1, \ldots, D_n$ be OBDDs representing formulas $F_1, \ldots, F_n$, respectively

How do we compute the OBDD representing $f(F_1, \ldots, F_n)$?

- We fix the same variable ordering for all OBDDs
- We assume isomorphic subdags are shared across different OBDDs
- We use one fundamental property of *if _ then _ else _*

Exercise in Compiler Optimization

- Consider the expression in Java

  ```
  ((x > 0) ? y1 : y2) + ((x > 0) ? z1 : z2)
  ```

- Can we simplify it?
- Suppose `x > 0` evaluates to `true`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y1` and
  `((x > 0) ? z1 : z2)` evaluates to `z1`, so the sum evaluates to `y1 + z1`
- Suppose `x > 0` evaluates to `false`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y2` and
  `((x > 0) ? z1 : z2)` evaluates to `z2`, so the sum evaluates to `y2 + z2`
- To simplify the expression, we could use the following property:
  $(E \ ? \ E_1 \ : \ E_2) \ + \ (E \ ? \ F_1 \ : \ F_2) \ = \ E \ ? \ (E_1 \ + \ F_1) \ : \ (F_2 \ + \ F_2)$

Exercise in Compiler Optimization

- Consider the expression in Java (C, C++, Perl, ...)

```
((x > 0) ? y1 : y2) + ((x > 0) ? z1 : z2)
```

- Can we simplify it?
- Suppose `x > 0` evaluates to `true`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y1` and
  `((x > 0) ? z1 : z2)` evaluates to `z1`, so the sum evaluates to `y1 + z1`
- Suppose `x > 0` evaluates to `false`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y2` and
  `((x > 0) ? z1 : z2)` evaluates to `z2`, so the sum evaluates to `y2 + z2`
- To simplify the expression, we could use the following property:
  $(E\ ?\ E_1\ :\ E_2)\ +\ (E\ ?\ F_1\ :\ F_2)\ =\ E\ ?\ (E_1\ +\ F_1)\ :\ (F_2\ +\ F_2)$

Exercise in Compiler Optimization

- Consider the expression in Java (C, C++, Perl, ...)

  ```
  ((x > 0) ? y1 : y2) + ((x > 0) ? z1 : z2)
  ```

- Can we simplify it?
- Suppose `x > 0` evaluates to `true`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y1` and
  `((x > 0) ? z1 : z2)` evaluates to `z1`, so the sum evaluates to `y1 + z1`
- Suppose `x > 0` evaluates to `false`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y2` and
  `((x > 0) ? z1 : z2)` evaluates to `z2`, so the sum evaluates to `y2 + z2`
- To simplify the expression, we could use the following property:
  $(E \ ? \ E_1 \ : \ E_2) \ + \ (E \ ? \ F_1 \ : \ F_2) \ = \ E \ ? \ (E_1 \ + \ F_1) \ : \ (F_2 \ + \ F_2)$

Exercise in Compiler Optimization

- Consider the expression in Java (C, C++, Perl, . . .)

  ```
  ((x > 0) ? y1 : y2) + ((x > 0) ? z1 : z2)
  ```

- Can we simplify it?
- Suppose `x > 0` evaluates to `true`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y1` and
  `((x > 0) ? z1 : z2)` evaluates to `z1`, so the sum evaluates to `y1 + z1`
- Suppose `x > 0` evaluates to `false`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y2` and
  `((x > 0) ? z1 : z2)` evaluates to `z2`, so the sum evaluates to `y2 + z2`
- To simplify the expression, we could use the following property:

  $(E\ ?\ F_1\ :\ F_2) + (E\ ?\ F_1\ :\ F_2) \quad = \quad E\ ?\ (F_1 + F_1)\ :\ (F_2 + F_2)$

Exercise in Compiler Optimization

- Consider the expression in Java (C, C++, Perl, ...)

  ```
  ((x > 0) ? y1 : y2) + ((x > 0) ? z1 : z2)
  ```

- Can we simplify it?
- Suppose `x > 0` evaluates to `true`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y1` and
  `((x > 0) ? z1 : z2)` evaluates to `z1`, so the sum evaluates to `y1 + z1`
- Suppose `x > 0` evaluates to `false`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y2` and
  `((x > 0) ? z1 : z2)` evaluates to `z2`, so the sum evaluates to `y2 + z2`
- To simplify the expression, we could use the following property:

  $$(E\ ?\ E_1\ :\ E_2)\ +\ (E\ ?\ F_1\ :\ F_2)\ \ =\ \ E\ ?\ (E_1\ +\ F_1)\ :\ (E_2\ +\ F_2)$$

Exercise in Compiler Optimization

- Consider the expression in Java (C, C++, Perl, ...)

  ```
  ((x > 0) ? y1 : y2) + ((x > 0) ? z1 : z2)
  ```

- Can we simplify it?
- Suppose `x > 0` evaluates to `true`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y1` and
  `((x > 0) ? z1 : z2)` evaluates to `z1`, so the sum evaluates to `y1 + z1`
- Suppose `x > 0` evaluates to `false`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y2` and
  `((x > 0) ? z1 : z2)` evaluates to `z2`, so the sum evaluates to `y2 + z2`
- To simplify the expression, we could use the following property:

  $(E \texttt{ ? } E_1 \texttt{ : } E_2) \texttt{ + } (E \texttt{ ? } F_1 \texttt{ : } F_2) \quad = \quad E \texttt{ ? } (E_1 \texttt{ + } F_1) \texttt{ : } (F_2 \texttt{ + } F_2)$

Exercise in Compiler Optimization

- Consider the expression in Java (C, C++, Perl, ...)

  ```
  ((x > 0) ? y1 : y2) + ((x > 0) ? z1 : z2)
  ```

- Can we simplify it?
- Suppose `x > 0` evaluates to `true`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y1` and
  `((x > 0) ? z1 : z2)` evaluates to `z1`, so the sum evaluates to `y1 + z1`
- Suppose `x > 0` evaluates to `false`. Then,
  `((x > 0) ? y1 : y2)` evaluates to `y2` and
  `((x > 0) ? z1 : z2)` evaluates to `z2`, so the sum evaluates to `y2 + z2`
- To simplify the expression, we could use the following property:

  $(E \texttt{ ? } E_1 \texttt{ : } E_2) + (E \texttt{ ? } F_1 \texttt{ : } F_2) \;=\; E \texttt{ ? } (E_1 + F_1) \texttt{ : } (F_2 + F_2)$

  That is, $(E \texttt{ ? } \_ \texttt{ : } \_)$ commutes with $+$

Fundamental property of if-then-else

In fact, for any predicate P,
if P *then* _ *else* _ commutes with any function f:

$$f(\textit{if } P \textit{ then } l_1 \textit{ else } r_1, \ldots, \textit{if } P \textit{ then } l_n \textit{ else } r_n) = \textit{if } P \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)$$

Fundamental property of if-then-else

In fact, for any predicate P,

if P *then* _ *else* _ commutes with any function f:

$$f(\textit{if } P \textit{ then } l_1 \textit{ else } r_1, \ldots, \textit{if } P \textit{ then } l_n \textit{ else } r_n) = \textit{if } P \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)$$

1. Apply f to the subdags corresponding to $p = 0$, obtaining a dag D_0
2. Apply f to the subdags corresponding to $p = 1$, obtaining a dag D_1

3. Build and return the dag

Fundamental property of if-then-else

In fact, for any predicate P,
if P *then* _ *else* _ commutes with any function f:

$$f(\textit{if } P \textit{ then } l_1 \textit{ else } r_1, \ldots, \textit{if } P \textit{ then } l_n \textit{ else } r_n) = \textit{if } P \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)$$

(Proof? By case analysis on P)

1. Apply f to the subdags corresponding to $p = 0$, obtaining a dag D_0
2. Apply f to the subdags corresponding to $p = 1$, obtaining a dag D_1

3. Build and return the dag

Fundamental property of if-then-else

In fact, for any predicate P,
if P *then* _ *else* _ commutes with any function f:

$$f(\textit{if } P \textit{ then } l_1 \textit{ else } r_1, \ldots, \textit{if } P \textit{ then } l_n \textit{ else } r_n) = \textit{if } P \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)$$

Hence, to apply f to n OBDDs rooted at variable p,

1. Apply f to the subdags corresponding to $p = 0$, obtaining a dag D_0
2. Apply f to the subdags corresponding to $p = 1$, obtaining a dag D_1
3. Build and return the dag

Fundamental property of if-then-else

In fact, for any predicate P,
if P *then* _ *else* _ commutes with any function f:

$$f(\textit{if } P \textit{ then } l_1 \textit{ else } r_1, \ldots, \textit{if } P \textit{ then } l_n \textit{ else } r_n) = \textit{if } P \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)$$

Hence, to apply f to n OBDDs rooted at variable p,

1. Apply f to the subdags corresponding to $p = 0$, obtaining a dag D_0
2. Apply f to the subdags corresponding to $p = 1$, obtaining a dag D_1
3. Build and return the dag

Fundamental property of if-then-else

In fact, for any predicate P,
if P *then* _ *else* _ commutes with any function f:

$$f(\textit{if } P \textit{ then } l_1 \textit{ else } r_1, \ldots, \textit{if } P \textit{ then } l_n \textit{ else } r_n) = \textit{if } P \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)$$

Hence, to apply f to n OBDDs rooted at variable p,

1. Apply f to the subdags corresponding to $p = 0$, obtaining a dag D_0
2. Apply f to the subdags corresponding to $p = 1$, obtaining a dag D_1
3. Build and return the dag

Fundamental property of if-then-else

In fact, for any predicate P,
if P *then* _ *else* _ commutes with any function f:

$$f(\textit{if } P \textit{ then } l_1 \textit{ else } r_1, \ldots, \textit{if } P \textit{ then } l_n \textit{ else } r_n) = \textit{if } P \textit{ then } f(l_1, \ldots, l_n) \textit{ else } f(r_1, \ldots, r_n)$$

Hence, to apply f to n OBDDs rooted at variable p,

1. Apply f to the subdags corresponding to $p = 0$, obtaining a dag D_0
2. Apply f to the subdags corresponding to $p = 1$, obtaining a dag D_1
3. Build and return the dag 

Negation

$$\neg(\mathit{if}\ p\ \mathit{then}\ L\ \mathit{else}\ R) \equiv \mathit{if}\ p\ \mathit{then}\ \neg L\ \mathit{else}\ \neg R$$

Negation

$$\neg(\mathit{if}\ p\ \mathit{then}\ L\ \mathit{else}\ R) \equiv \mathit{if}\ p\ \mathit{then}\ \neg L\ \mathit{else}\ \neg R$$

procedure *negation*(n)
parameters: global dag D
input: node n representing formula F in D
output: a node n' representing $\neg F$ in (modified) D
begin
 if n is $\boxed{1}$ **then return** $\boxed{0}$
 if n is $\boxed{0}$ **then return** $\boxed{1}$
 $p := \mathit{max\_variable}(n)$
 $(l, r) := (\mathit{neg}(n), \mathit{pos}(n))$
 $l' := \mathit{negation}(l)$
 $r' := \mathit{negation}(r)$
 return $\mathit{integrate}(l', p, r')$
end

Negation

$$\neg(\mathit{if}\ p\ \mathit{then}\ L\ \mathit{else}\ R) \equiv \mathit{if}\ p\ \mathit{then}\ \neg L\ \mathit{else}\ \neg R$$

procedure *negation*(n)
parameters: global dag D
input: node n representing formula F in D
output: a node n' representing $\neg F$ in (modified) D
begin
 if n is $\boxed{1}$ **then return** $\boxed{0}$
 if n is $\boxed{0}$ **then return** $\boxed{1}$
 $p := \mathit{max\_variable}(n)$
 $(l, r) := (\mathit{neg}(n), \mathit{pos}(n))$ // negative and positive subdiagram of n
 $l' := \mathit{negation}(l)$
 $r' := \mathit{negation}(r)$
 return $\mathit{integrate}(l', p, r')$
end

Disjunction

$$(\mathit{if}\ p\ \mathit{then}\ L_1\ \mathit{else}\ R_1) \vee (\mathit{if}\ p\ \mathit{then}\ L_1\ \mathit{else}\ R_1) \equiv \mathit{if}\ p\ \mathit{then}\ L_1 \vee L_2\ \mathit{else}\ R_1 \vee R_2$$

Disjunction

$$(\mathit{if}\ p\ \mathit{then}\ L_1\ \mathit{else}\ R_1) \vee (\mathit{if}\ p\ \mathit{then}\ L_1\ \mathit{else}\ R_1) \equiv \mathit{if}\ p\ \mathit{then}\ L_1 \vee L_2\ \mathit{else}\ R_1 \vee R_2$$

procedure *disjunction*$(n_1, \ldots, n_m)$
parameters: global dag D
input: 1 or more nodes $n_1, \ldots, n_m$ representing $F_1, \ldots, F_m$ in D
output: a node n representing $F_1 \vee \cdots \vee F_m$ in (modified) D
begin
 if $m = 1$ **then return** n_1
 if some n_i is $\boxed{1}$ **then return** $\boxed{1}$
 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) // (*)
 l := *disjunction*$(l_1, \ldots, l_m)$
 r := *disjunction*$(r_1, \ldots, r_m)$
 return *integrate*(l, p, r)
end

Disjunction

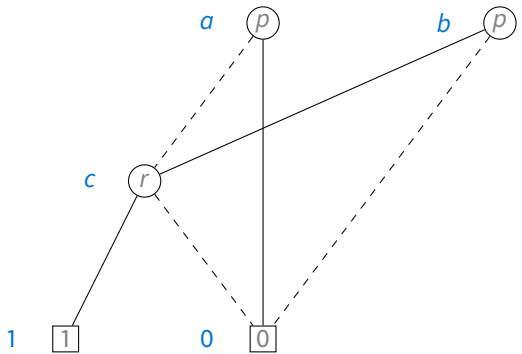
$$(\mathit{if}\ p\ \mathit{then}\ L_1\ \mathit{else}\ R_1) \vee (\mathit{if}\ p\ \mathit{then}\ L_1\ \mathit{else}\ R_1) \equiv \mathit{if}\ p\ \mathit{then}\ L_1 \vee L_2\ \mathit{else}\ R_1 \vee R_2$$

procedure *disjunction*$(n_1, \ldots, n_m)$
parameters: global dag D
input: 1 or more nodes $n_1, \ldots, n_m$ representing $F_1, \ldots, F_m$ in D
output: a node n representing $F_1 \vee \cdots \vee F_m$ in (modified) D
begin
 if $m = 1$ **then return** n_1
 if some n_i is $\boxed{1}$ **then return** $\boxed{1}$
 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) // (*)
 l := *disjunction*$(l_1, \ldots, l_m)$
 r := *disjunction*$(r_1, \ldots, r_m)$
 return *integrate*(l, p, r)
end

(*) Consider fictitious redundant node k_i with $n_i = neg(k_i) = pos(k_i)$

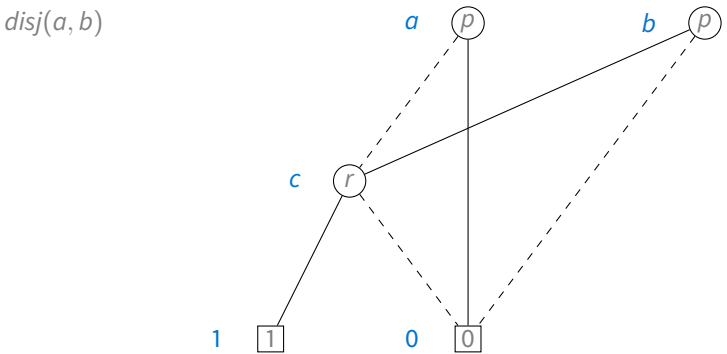
Example: Disjunction

Computing $(\neg p \wedge r) \vee (p \wedge r)$ where a represents $\neg p \wedge r$ and b represents $p \wedge r$:



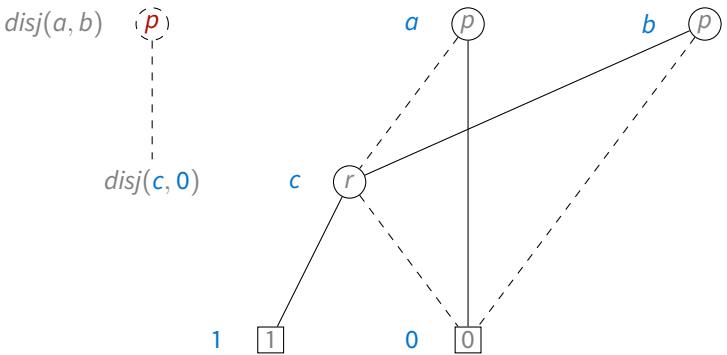
Example: Disjunction

Computing $(\neg p \wedge r) \vee (p \wedge r)$ where a represents $\neg p \wedge r$ and b represents $p \wedge r$:



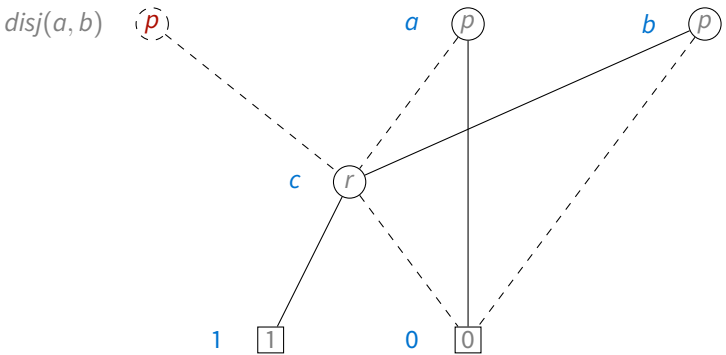
Example: Disjunction

Computing $(\neg p \wedge r) \vee (p \wedge r)$ where a represents $\neg p \wedge r$ and b represents $p \wedge r$:



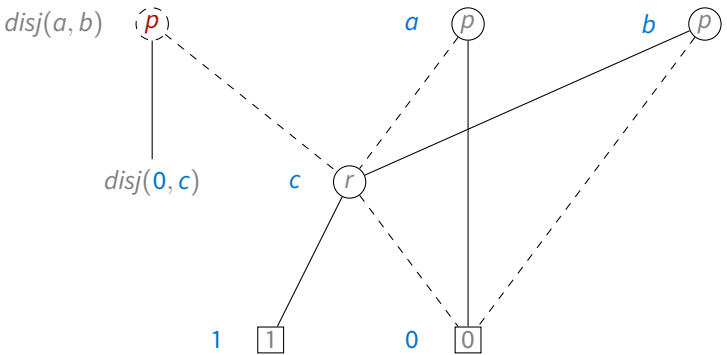
Example: Disjunction

Computing $(\neg p \wedge r) \vee (p \wedge r)$ where a represents $\neg p \wedge r$ and b represents $p \wedge r$:



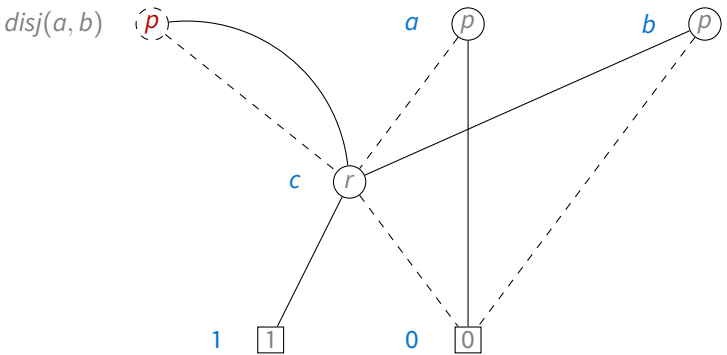
Example: Disjunction

Computing $(\neg p \wedge r) \vee (p \wedge r)$ where a represents $\neg p \wedge r$ and b represents $p \wedge r$:



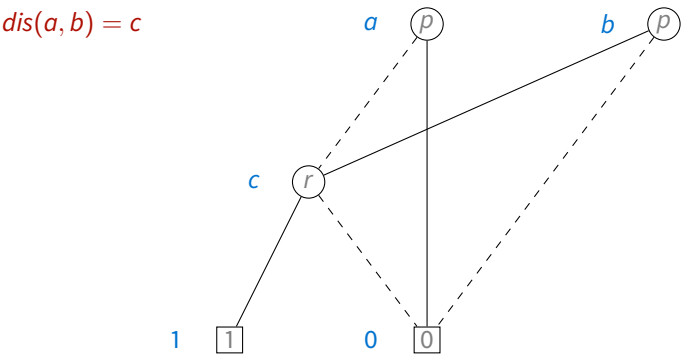
Example: Disjunction

Computing $(\neg p \wedge r) \vee (p \wedge r)$ where a represents $\neg p \wedge r$ and b represents $p \wedge r$:



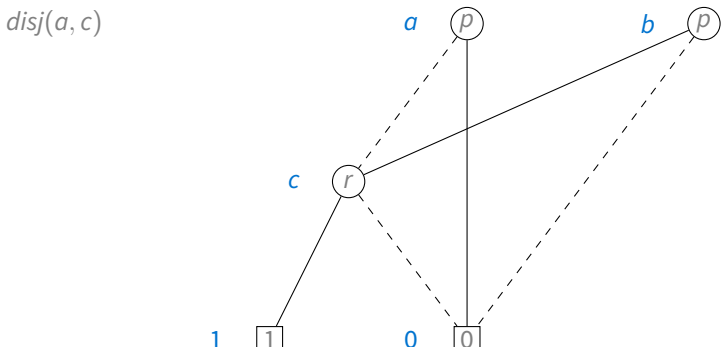
Example: Disjunction

Computing $(\neg p \wedge r) \vee (p \wedge r)$ where a represents $\neg p \wedge r$ and b represents $p \wedge r$:



Exercise

Compute $(\neg p \wedge r) \vee r$ where a represents $\neg p \wedge r$ and c represents r:



Disjunction (recall)

procedure *disjunction*$(n_1, \ldots, n_m)$
parameters: global dag D
input: 1 or more nodes $n_1, \ldots, n_m$ representing $F_1, \ldots, F_m$ in D
output: a node n representing $F_1 \vee \cdots \vee F_m$ in (modified) D
begin
 if $m = 1$ **then return** n_1
 if some n_i is $\boxed{1}$ **then return** $\boxed{1}$
 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)
 l := *disjunction*$(l_1, \ldots, l_m)$
 r := *disjunction*$(r_1, \ldots, r_m)$
 return *integrate*(l, p, r)
end

Disjunction (recall)

procedure *disjunction*$(n_1, \ldots, n_m)$
parameters: global dag D
input: 1 or more nodes $n_1, \ldots, n_m$ representing $F_1, \ldots, F_m$ in D
output: a node n representing $F_1 \vee \cdots \vee F_m$ in (modified) D
begin
 if $m = 1$ **then return** n_1
 if some n_i is $\boxed{1}$ **then return** $\boxed{1}$
 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)
 l := *disjunction*$(l_1, \ldots, l_m)$
 r := *disjunction*$(r_1, \ldots, r_m)$
 return *integrate*(l, p, r)
end

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

Conjunction

procedure $conjunction(n_1, \ldots, n_m)$
parameters: global dag D
input: 1 or more nodes $n_1, \ldots, n_m$ representing $F_1, \ldots, F_m$ in D
output: a node n representing $F_1 \wedge \cdots \wedge F_m$ in (modified) D
begin
 if $m = 1$ **then return** n_1
 if some n_i is $\boxed{0}$ **then return** $\boxed{0}$
 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)
 l := $conjunction(l_1, \ldots, l_m)$
 r := $conjunction(r_1, \ldots, r_m)$
 return $integrate(l, p, r)$
end

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

Other connectives

procedure *implication*(n_1, n_2)
parameters: global dag D
input: nodes n_1, n_2 representing formulas F_1, F_2 in D
output: a node n representing $F_1 \to F_2$ in (modified) D
begin
 return *disjunction*(*negation*$(n_1), n_2)$
end

procedure *bi_implication*(n_1, n_2)
parameters: global dag D
input: nodes n_1, n_2 representing formulas F_1, F_2 in D
output: a node n representing $F_1 \leftrightarrow F_2$ in (modified) D
begin
 return *conjunction*(*implication*(n_1, n_2), *implication*$(n_2, n_1))$
end

Other connectives

procedure *implication*(n_1, n_2)
parameters: global dag D
input: nodes n_1, n_2 representing formulas F_1, F_2 in D
output: a node n representing $F_1 \rightarrow F_2$ in (modified) D
begin
 return *disjunction*(*negation*$(n_1), n_2)$
end

procedure *bi_implication*(n_1, n_2)
parameters: global dag D
input: nodes n_1, n_2 representing formulas F_1, F_2 in D
output: a node n representing $F_1 \leftrightarrow F_2$ in (modified) D
begin
 return *conjunction*(*implication*(n_1, n_2), *implication*$(n_2, n_1))$
end