

# Initiation à la Vérification

## Binary Decision Diagrams

# Set representations

---

The solution of the model-checking problem for CTL and *finite-state systems* can be expressed by operations on sets:

states satisfying (sub)formulae:  $\llbracket \psi \rrbracket$

subformulae combined by set operations:  $\cap, \cup, \dots$

e.g.,  $\llbracket \text{EX } \psi \rrbracket$  can be obtained by the operation

$$\text{pre}(\mathcal{S}) := \{ s \mid \exists t : s \rightarrow t \wedge t \in \mathcal{S} \}$$

**EG** and **EU** require fixed-point iterations on set equations

Likewise, computing the reachable states is expressible with set operations:

Start by setting  $X := I$ , the set of initial states;

Iterate  $X := X \cup \{ t \mid \exists s : s \rightarrow t \wedge s \in X \}$  until fixpoint.

# Set representations

---

How can such sets be represented:

explicit list:  $S = \{s_1, s_2, s_4, \dots\}$

**symbolic representation**: compact notation or data structure

Idea: Find a data structure that

can compress the representation of large state sets

permits efficient implementation of the operations needed for CTL

# Caveat

---

No (lossless) compression method can compress *all* sets (or work efficiently for all).

Binary decision diagrams (BDDs) usually work well for systems

- whose states can be represented as Boolean vectors
- with logical connections between them

In the following:  $S = \{0, 1\}^m$  for some  $m \geq 1$

Remark:

this can be made to work for all finite state sets  
(but may not be natural or adequate)

# Literature

---

Some pointers:

H.R. Andersen, *An Introduction to Binary Decision Diagrams*, Lecture notes,  
Department of Information Technology, IT University of Copenhagen

Available on WWW

Tools:

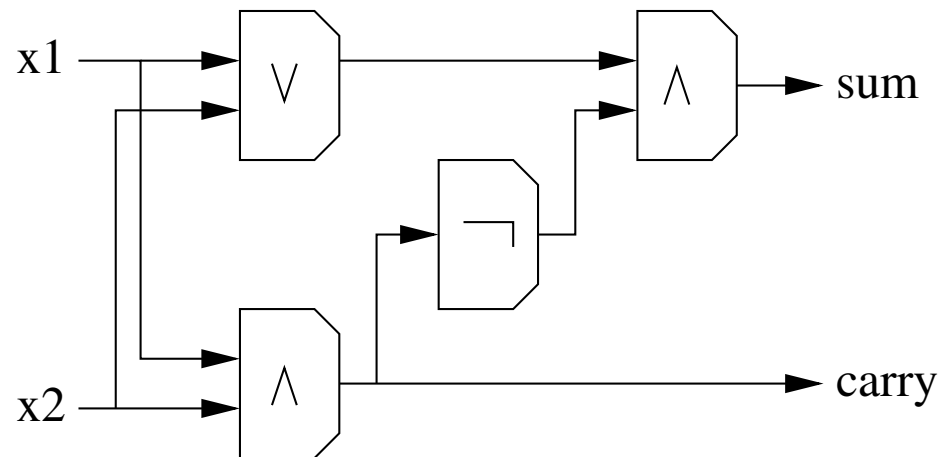
CUDD library

SMV (BDD-based model checker)

# Example: Circuit

---

Half-adder:



The circuit has got two inputs ( $x_1, x_2$ ) and two outputs ( $carry, sum$ ). Their admissible combinations can be denoted by Boolean 4-tuples, e.g.  $(1, 0, 0, 1)$  ( $x_1 = 1, x_2 = 0, carry = 0, sum = 1$ ) is a possible combination.

---

The admissible combinations in the Example correspond to the following formula of propositional logic:

$$F \equiv (\textit{carry} \leftrightarrow (x_1 \wedge x_2)) \wedge (\textit{sum} \leftrightarrow (x_1 \vee x_2) \wedge \neg \textit{carry})$$

In the following, we shall treat

**sets of states** (i.e. sets of Boolean vectors)

and **formulae of propositional logic**

simply as different representations of the same objects.

# Binary decision graphs

---

Let  $V$  be a set of variables (atomic propositions) and  $\prec$  a total order on  $V$ , e.g.

$$x_1 \prec x_2 \prec \textit{carry} \prec \textit{sum}$$

A **binary decision graph** (w.r.t.  $\prec$ ) is a directed, connected, acyclic graph with the following properties:

there is exactly one **root**, i.e. a node without incoming arcs;

there are at most two leaves, labelled by **0** or **1**;

all non-leaves are labelled with variables from  $V$ ;

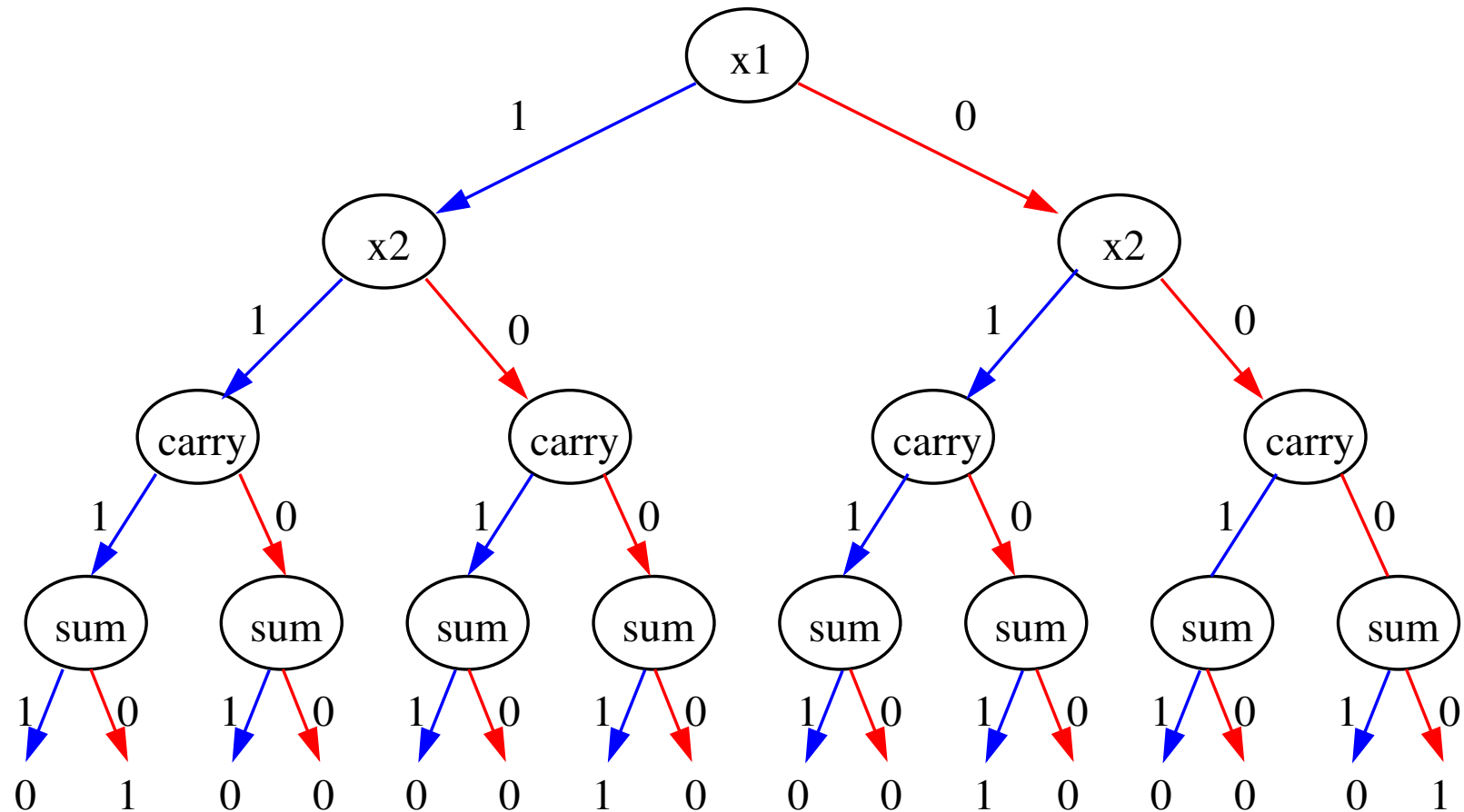
every non-leaf has two outgoing arcs labelled by **0** and **1**;

if there is an edge from an  $x$ -labelled node to a  $y$ -labelled node, then  $x \prec y$ .



# Example: Binary decision graph (here: a full tree)

---



Paths ending in **1** correspond to vectors whose entry in the truth table is 1.

# Binary decision diagrams

---

A **binary decision diagram** (BDD) is a binary decision graph with two additional properties:

no two subgraphs are isomorphic;

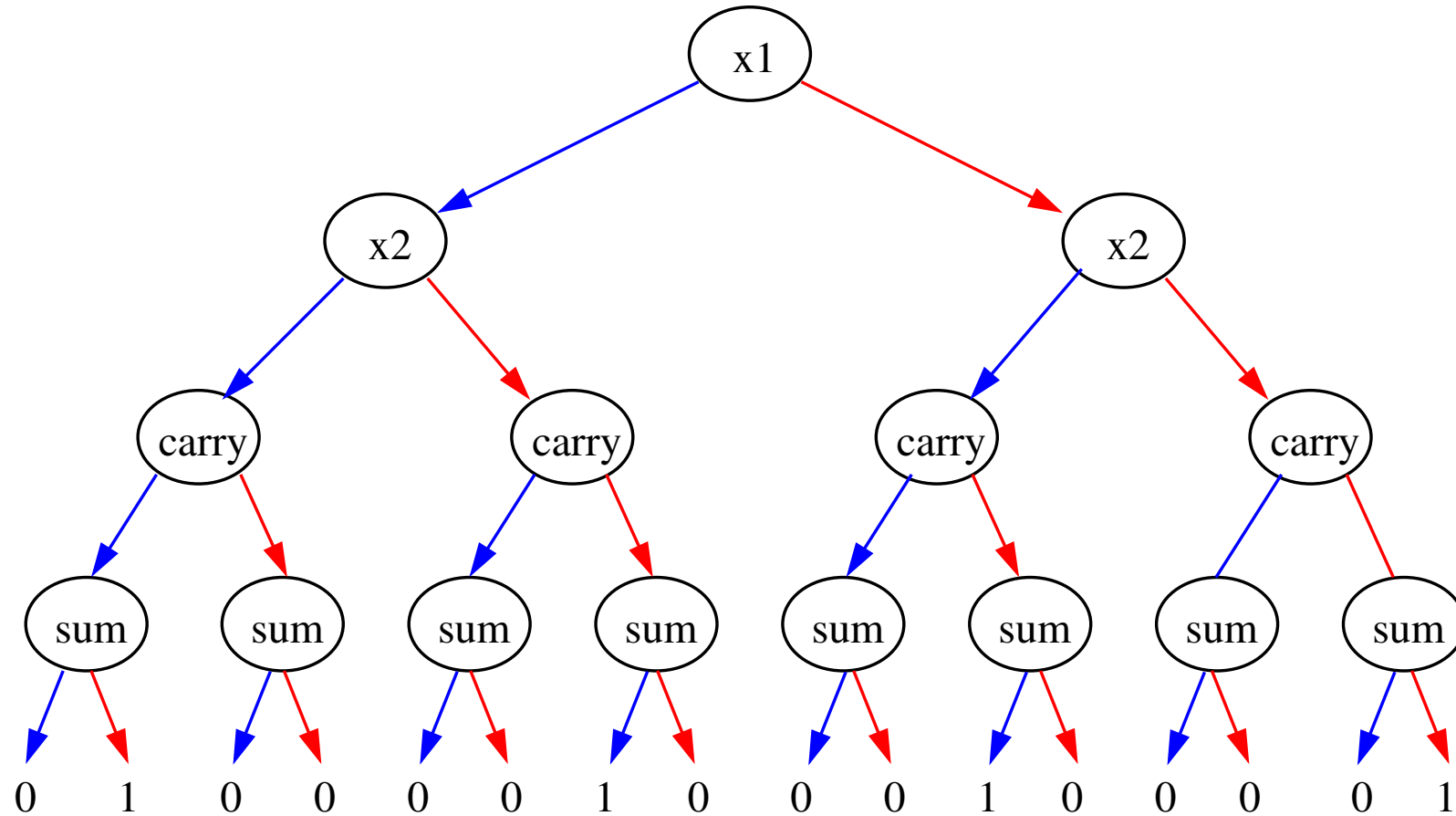
there are no *redundant* nodes, where both outgoing edges lead to the same target node.

Optionally, we omit the **0**-node and the edges leading there.

Remarks: On the following slides, the **blue** edges are meant to be labelled by 1, the **red** edges by 0.

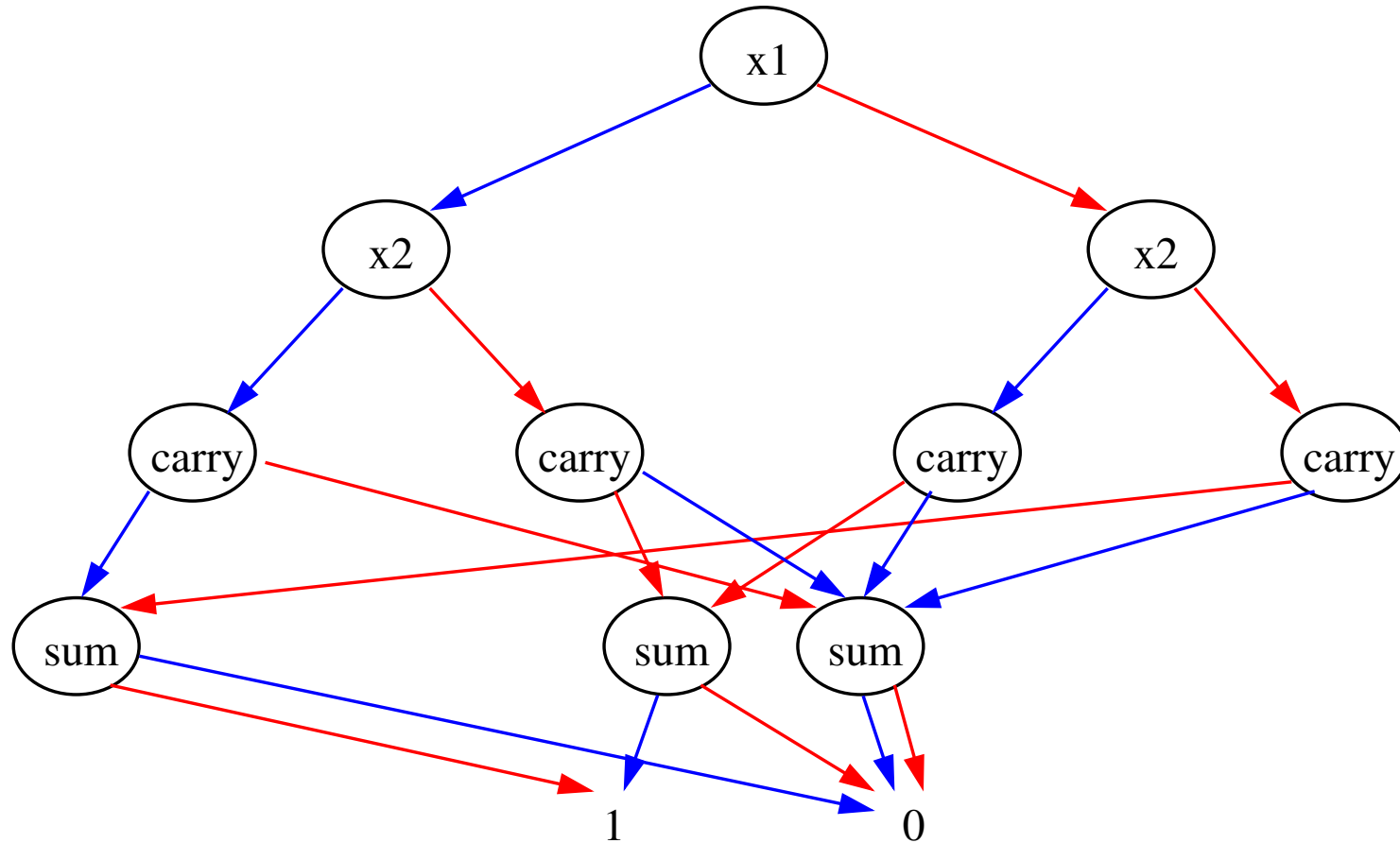
# Example: Eliminate isomorphic subgraphs (1/3)

---



## Example: Eliminate isomorphic subgraphs (2/3)

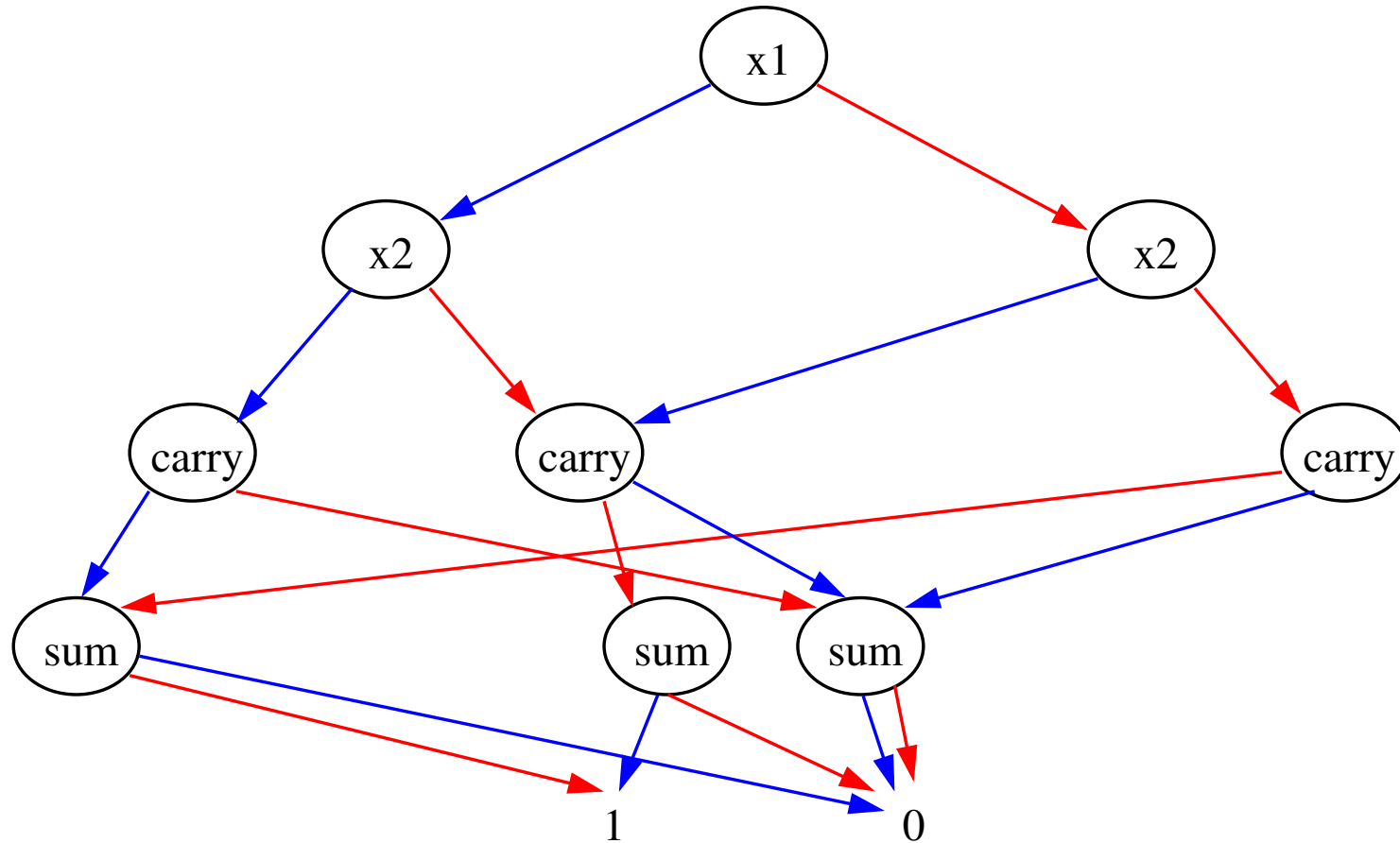
---



Merged the isomorphic *sum*-nodes (and the leaves).

## Example: Eliminate isomorphic subgraphs (3/3)

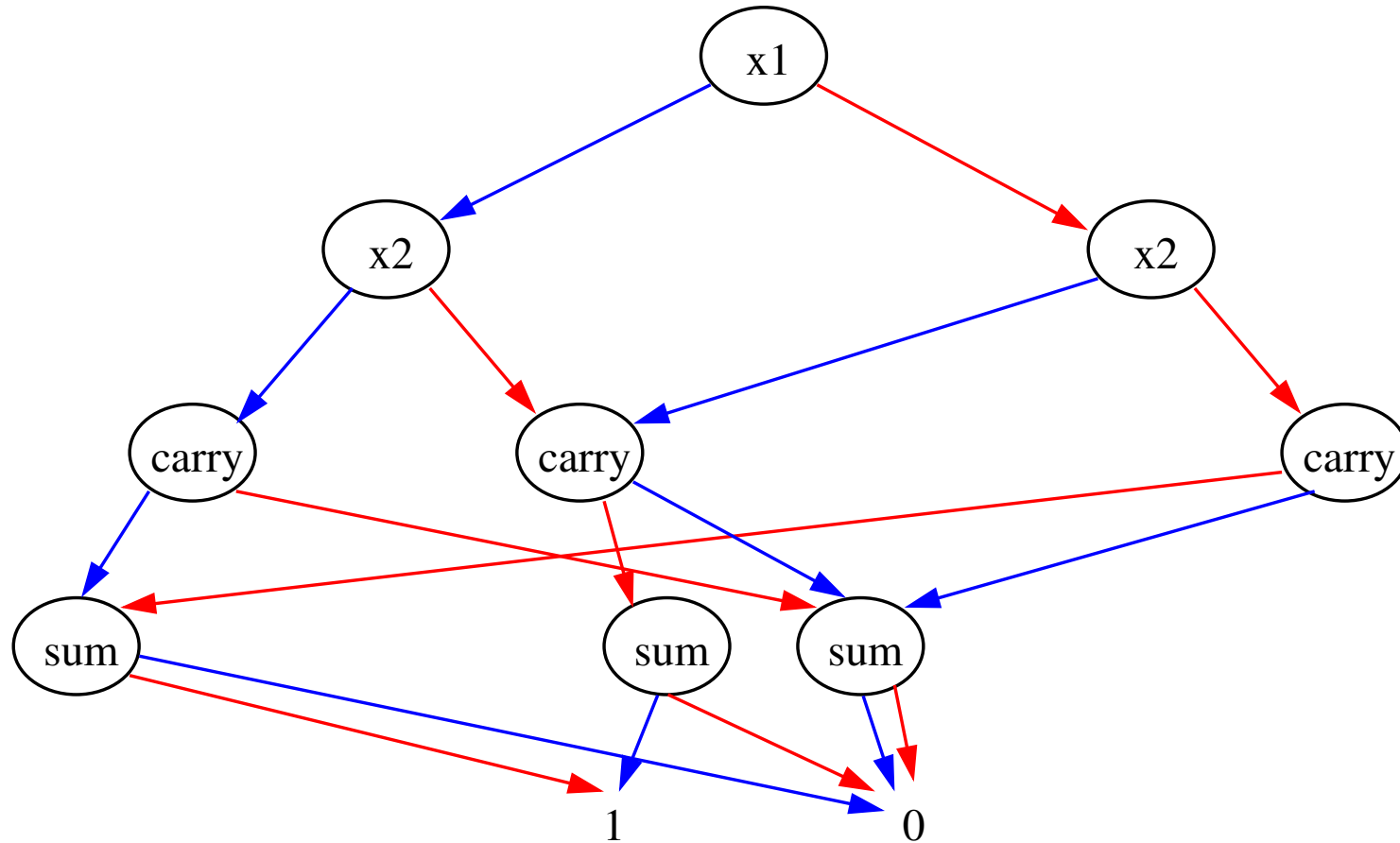
---



Two carry nodes can be merged, but no others → done.

## Example: Remove redundant nodes (1/2)

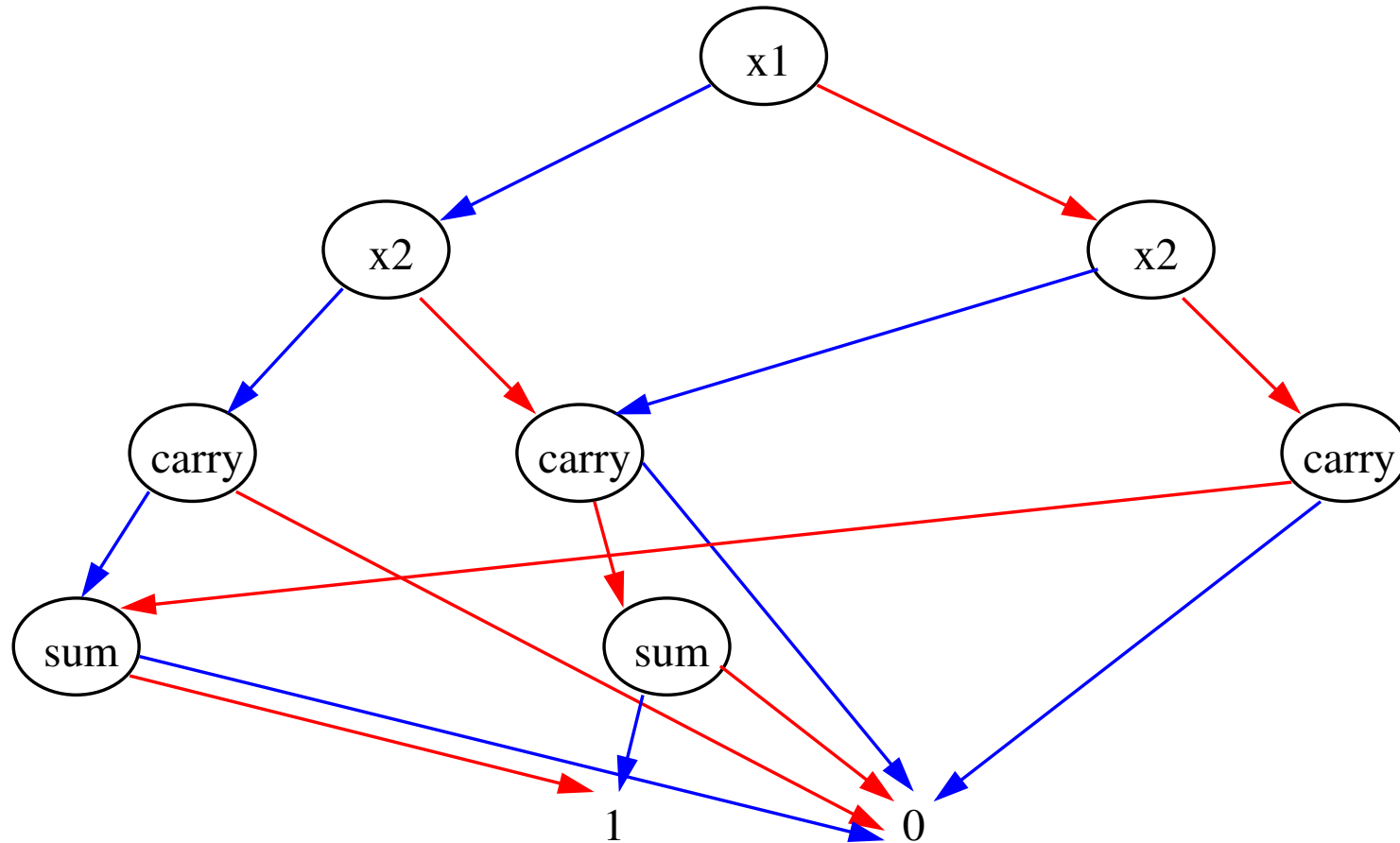
---



Both edges of the right *sum*-node point to 0.

## Example: Remove redundant nodes (2/2)

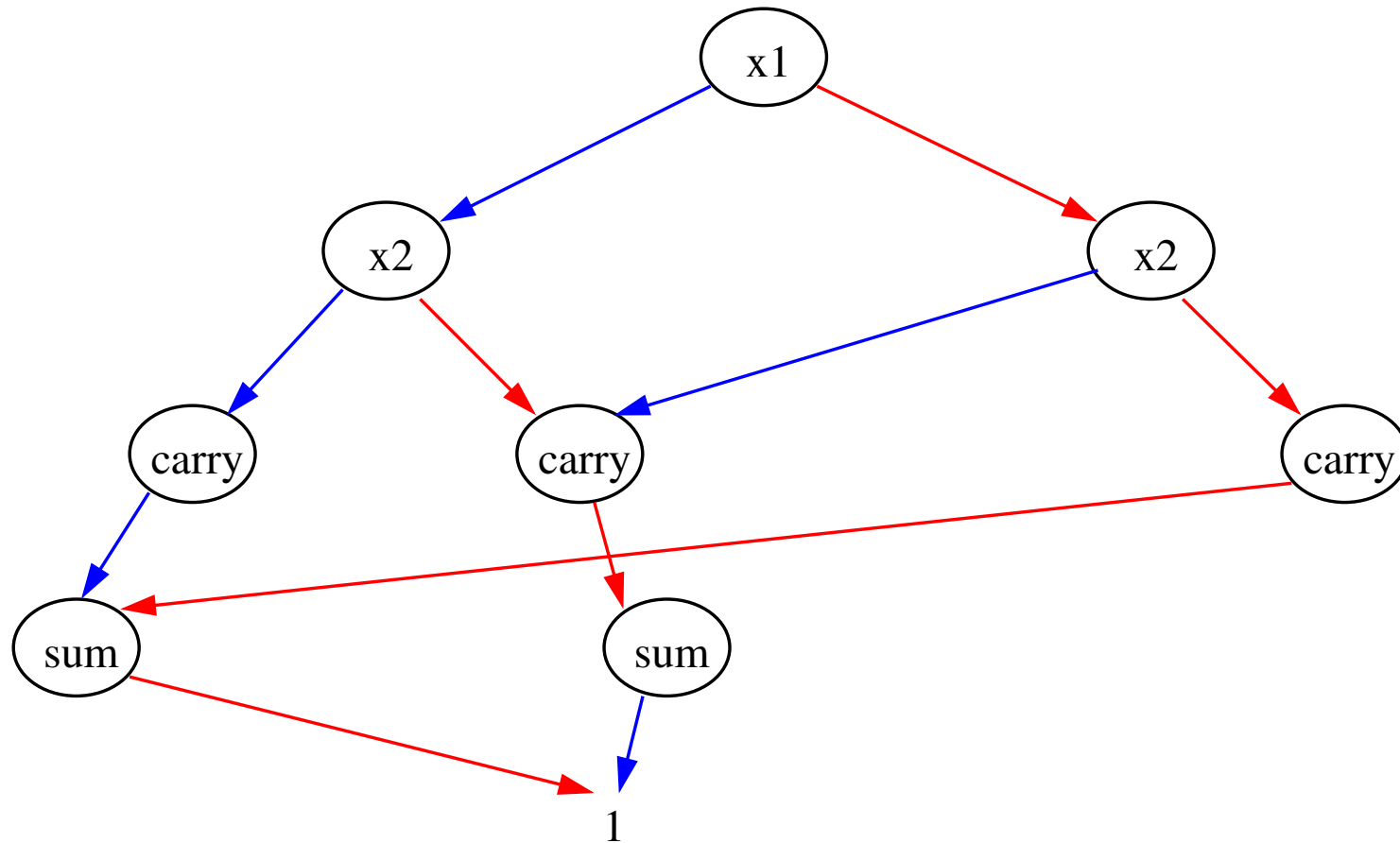
---



No more redundant nodes → we are done.

## Example: Omit 0-node

---



For a clearer graphical representation, one can remove the 0-node  
→ missing edges implicitly lead to 0



# Preview

---

We shall study operations on BDDs needed for CTL model checking.

Construction of a BDD (from a PL formula)

Equivalence check

Intersection, complement, union

Relations, computing predecessors

We consider formulae of propositional logic (PL), extended with **0** and **1**.

# Substitution

---

Let  $F$  and  $G$  be formulae of PL (possibly with constants), and let  $x$  be an atomic proposition.

$F[x/G]$  denotes the PL formula obtained by replacing each occurrence of  $x$  in  $F$  by  $G$ .

In particular, we will consider formulae of the form  $F[x/0]$  and  $F[x/1]$ .

**Example:** Let  $F = x \wedge y$ . Then  $F[x/1] = 1 \wedge y \equiv y$  and  $F[x/0] = 0 \wedge y \equiv 0$ .

# If-then-else

---

Let *ite* (if-then-else) be a new *ternary* PL operator.

Let  $F, G, H$  be PL formulae. We define

$$ite(F, G, H) := (F \wedge G) \vee (\neg F \wedge H).$$

**INF formulae** (if-then-else normal form) are inductively defined as follows:

**0** and **1** are INF formulae;

if  $x$  is an atomic proposition and  $G, H$  are INF formulae, then  $ite(x, G, H)$  is an INF formula.

# Shannon partitioning

---

Let  $F$  be a PL formula and  $x$  an atomic proposition. We have:

$$F \equiv \text{ite}(x, F[x/1], F[x/0])$$

**Proof:** In the following,  $G$  denotes the right-hand side of the equivalence above. Let  $\nu$  be a valuation s.t.  $\nu \models F$ . Either  $\nu(x) = 1$ , then  $\nu$  is also a model of  $F[x/1]$  and of  $x$  and therefore also of  $G$ . The case  $\nu(x) = 0$  is analogous. For the other direction, suppose  $\nu \models G$ . Then either  $\nu(x) = 1$  and the “rest” of  $\nu$  is a model of  $F[x/1]$ . Then, however,  $\nu$  will be a model for any formula in which some of the ones in  $F[x/1]$  are replaced by  $x$ , in particular also for  $F$ . The case  $\nu(x) = 0$  is again analogous.

**Remark:**  $G$  is called the **Shannon partitioning** of  $F$ .

**Corollary:** Every PL formula is equivalent to an INF formula.

**Proof:** apply the above partitioning multiple times.

# Construction of BDDs

---

We can now solve our first BDD-related problem: Given a PL formula  $F$  and some ordering of variables  $\prec$ , construct a BDD w.r.t.  $\prec$  that represents  $F$ .

If  $F$  does not contain any atomic propositions at all, then either  $F \equiv 0$  or  $F \equiv 1$ , and the corresponding BDD is simply the corresponding leaf node.

Otherwise, let  $x$  be the smallest variable (w.r.t.  $\prec$ ) occurring in  $F$ . Construct BDDs  $B_0$  and  $B_1$  for  $F[x/1]$  and  $F[x/0]$ , respectively (these formulae have one variable less than  $F$ ).

Because of the Shannon partitioning,  $F$  is representable by a binary decision *graph* whose root is labelled by  $x$  and whose subtrees are  $B_0$  and  $B_1$ . To obtain a BDD, we check whether  $B_0$  and  $B_1$  are isomorphic; if yes, then  $F$  is represented by  $B_0$ . Otherwise we merge all isomorphic subtrees in  $B_0$  and  $B_1$ .

# BDDs are unique

---

Given a PL formula  $F$  and a variable ordering  $\prec$ , there is (up to isomorphism) exactly one BDD that respects  $\prec$  and represents  $F$ .

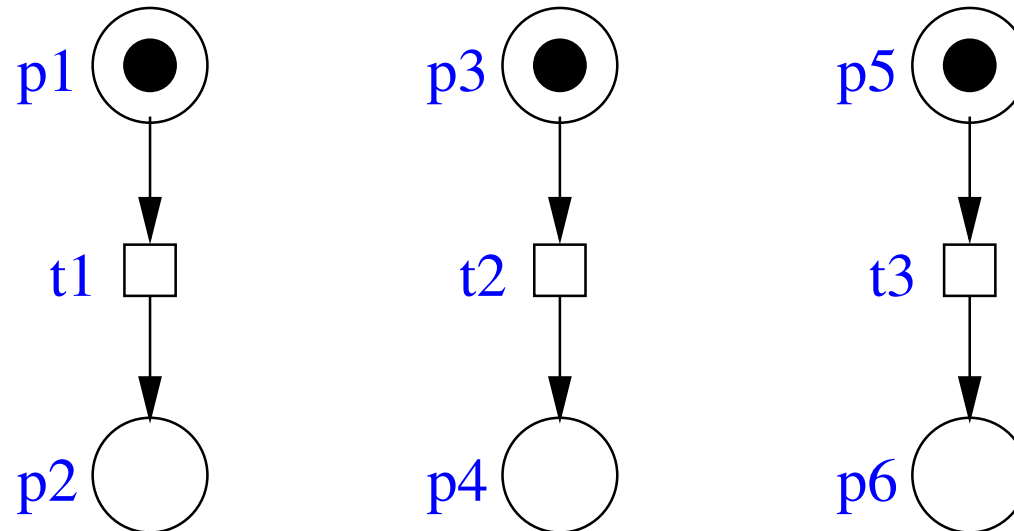
**Proof:** (sketch) by induction on the number of variables, start with 0 (constant functions), then use Shannon partitioning.

Remark: Different orderings still lead to different BDDs.  
(possibly with vastly different sizes!)

# Example: Variable orderings

---

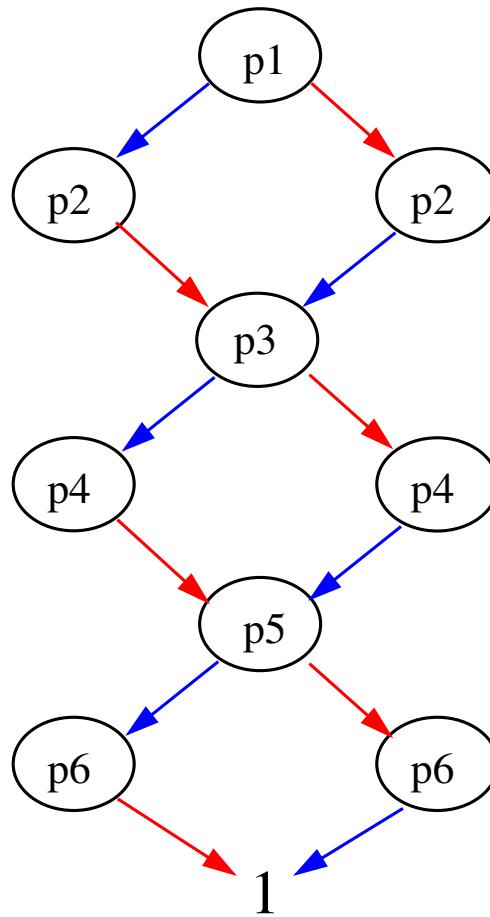
Recall Example 1 (the Petri net), and let us construct a BDD representing the reachable markings:



Remark:  $P_1$  is marked iff  $P_2$  is not, etc.

---

The corresponding BDD for the ordering  $p_1 \prec p_2 \prec p_3 \prec p_4 \prec p_5 \prec p_6$ :





---

Remarks:

If we increase the number of components from 3 to  $n$  (for some  $n \geq 0$ ), the size of the corresponding BDD will be linear in  $n$ .

In other words, a BDD of size  $n$  can represent  $2^n$  (or even more) valuations.

However, the size of a BDD strongly depends on the ordering!

Example: Repeat the previous construction for the ordering

$$p_1 \prec p_3 \prec p_5 \prec p_2 \prec p_4 \prec p_6.$$

# Equivalence test

---

To implement CTL model checking, we need a test for equivalence between BDDs (e.g., to check the termination of a fixed-point computation).

**Problem:** Given BDDs  $B$  and  $C$  (w.r.t. the same ordering) do  $B$  and  $C$  represent equivalent formulae?

**Solution:** Test whether  $B$  and  $C$  are isomorphic.

**Special cases:**

Unsatisfiability test: Check if the BDD consists just of the  $0$  leaf.

Tautology test: Check if the BDD consists just of the  $1$  leaf.

# Implementing BDDs with hash tables

---

Suppose we want to write an application in which we need to manipulate multiple BDDs.

Efficient BDDs implementations exploit the uniqueness property by storing all BDD nodes in a hash table. (Recall that each node is in fact the root of some BDD.)

Initially, the hash table has only two unique entries, the leaves **0** and **1**.

Every other node is uniquely identified by the triple  $(x, B_0, B_1)$ , where  $x$  is the atomic proposition labelling that node and  $B_0, B_1$  are the subtrees of that node, represented by pointers to their respective roots.

---

Usually, one implements a function  $mk(x, B_0, B_1)$  that checks whether the hash table already contains such a node; if yes, then the pointer to that node is returned, otherwise a new node is created.

Each BDD is then simply represented by a pointer to its root.

A multitude of BDDs is then stored as a “forest” (a DAG with multiple roots).

Problem: garbage collection (by reference counting)

# Equivalence test II

---

Let us reconsider the equivalence-checking problem.

(Given two BDDs  $B$  and  $C$ , do  $B$  and  $C$  represent equivalent formulae?)

If  $B$  and  $C$  are stored in hash tables (as described previously), then  $B$  and  $C$  are representable by pointers to their roots.

Due to the uniqueness property, one then simply has to check whether the pointers are the same (a **constant-time** procedure).

# Logical operations I: Complement

---

Let  $F$  be a PL formula and  $B$  a BDD representing  $F$ .

**Problem:** Compute a BDD for  $\neg F$ .

**Solution:** Exchange the two leaves of  $B$ .

(Caution: This is not quite so simple with the hash-table implementation.)

# Logical operations II: Intersection/Conjunction

---

Let  $F, G$  be PL formulae and  $B, C$  the corresponding BDDs (with the same ordering).

**Problem:** Compute a BDD for  $F \wedge G$  from  $B$  and  $C$ .

We have the following equivalence:

$$F \wedge G \equiv \text{ite}(x, (F \wedge G)[x/1], (F \wedge G)[x/0]) \equiv \text{ite}(x, F[x/1] \wedge G[x/1], F[x/0] \wedge G[x/0])$$

If  $x$  is the smallest variable occurring in either  $F$  or  $G$ , then

$F[x/1], F[x/0], G[x/1], G[x/0]$  are either the children of the roots of  $B$  and  $C$  (or the roots themselves).

---

We construct a BDD for conjunction according to the following, recursive strategy:

If  $B$  and  $C$  are equal, then return  $B$ .

If either  $B$  or  $C$  are the  $0$  leaf, then return  $0$ .

If either  $B$  or  $C$  are the  $1$  leaf, then return the other BDD.

Otherwise, compare the two variables labelling the roots of  $B$  and  $C$ , and let  $x$  be the smaller among the two (or the one labelling both).

If the root of  $B$  is labelled by  $x$ , then let  $B_1, B_0$  be the subtrees of  $B$ ; otherwise, let  $B_1, B_0 := B$ . We define  $C_1, C_0$  analogously.

Apply the strategy recursively to the pairs  $B_1, C_1$  and  $B_0, C_0$ , yielding BDDs  $E$  and  $F$ . If  $E = F$ , return  $E$ , otherwise  $mk(x, E, F)$ .



## Example: Intersection

---

Let  $x \prec y \prec z \prec u \prec w$ . Compute BDDs for

$$F_1 := (w \vee x \vee z) \wedge (x \rightarrow (u \wedge w))$$

$$F_2 := (y \leftrightarrow z) \rightarrow w$$

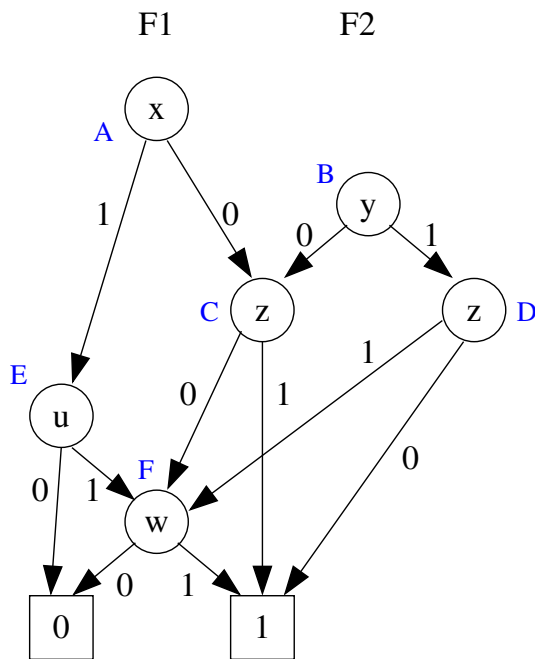
# Example: Intersection

---

Let  $x \prec y \prec z \prec u \prec w$ . Compute BDDs for

$$F_1 := (w \vee x \vee z) \wedge (x \rightarrow (u \wedge w))$$

$$F_2 := (y \leftrightarrow z) \rightarrow w$$



## Example: Intersection

---

Let us compute the intersection  $F_1 \wedge F_2$  from their BDDs.

Each box indicates one recursive computation.

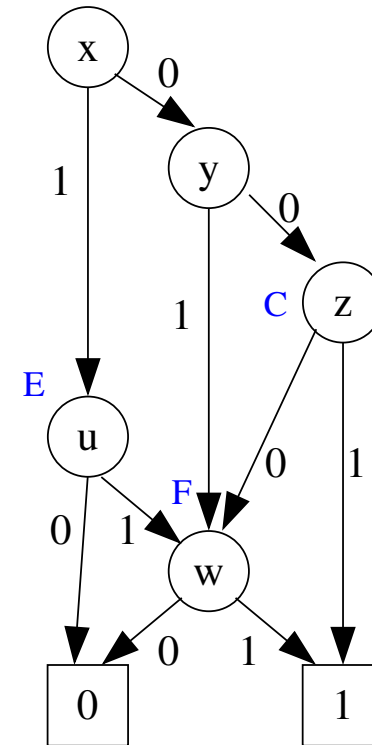
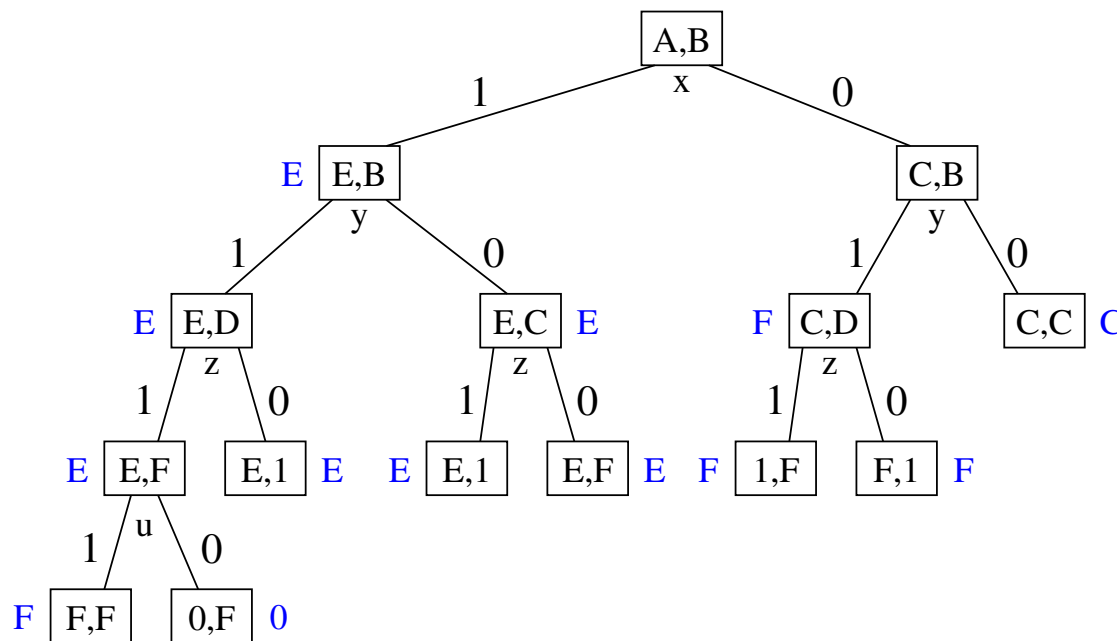
We denote the result next to each box.

# Example: Intersection

Let us compute the intersection  $F_1 \wedge F_2$  from their BDDs.

Each box indicates one recursive computation.

We denote the result next to each box.



# Logical operations III: Union/Disjunction

---

Let  $F, G$  be PL formulae and  $B, C$  the corresponding BDDs (with the same ordering).

**Problem:** Compute a BDD for  $F \vee G$  from  $B$  and  $C$ .

**Solution:** Analogous to conjunction, with the rules for **1** and **0** leaves adapted accordingly.

**Complexity:** With dynamic programming:  $\mathcal{O}(|B| \cdot |C|)$  (every pair of nodes at most once).

# Computing predecessors

---

In the following, we derive a strategy for computing the set

$$pre(M) = \{s \mid \exists s' : (s, s') \in \rightarrow \wedge s' \in M\}.$$

Note that the relation  $\rightarrow$  is a subset of  $S \times S$  whereas  $M \subseteq S$ .

We represent  $M$  by a BDD with variables  $y_1, \dots, y_m$ .

$\rightarrow$  will be represented by a BDD with variables  $x_1, \dots, x_m$  and  $y_1, \dots, y_m$  (states “before” and “after”).

---

Remark: Every BDD for  $M$  is at the same time a BDD for  $S \times M$ !

Thus, we can rewrite  $pre(M)$  as follows:

$$\{s \mid \exists s' : (s, s') \in \rightarrow \cap (S \times M)\}$$

Then,  $pre$  reduces to the operations intersection and existential abstraction.

# Existential abstraction

---

Existential abstraction w.r.t. an atomic proposition  $x$  is defined as follows:

$$\exists x : F \equiv F[x/0] \vee F[x/1]$$

I.e.,  $\exists x : F$  is true for those valuations that can be extended with a value for  $x$  in such a way that they become models for  $F$ .

**Example:** Let  $F \equiv (x_1 \wedge x_2) \vee x_3$ . Then

$$\exists x_1 : F \equiv F[x_1/0] \vee F[x_1/1] \equiv (x_3) \vee (x_2 \vee x_3) \equiv x_2 \vee x_3$$

By extension, we can consider existential abstraction over sets of atomic propositions (abstract from each of them in turn).



# BDDs with complement arcs

---

Shared BDDs make negation a costly operation.

→ use **BDD with complement arcs** (CBDD)

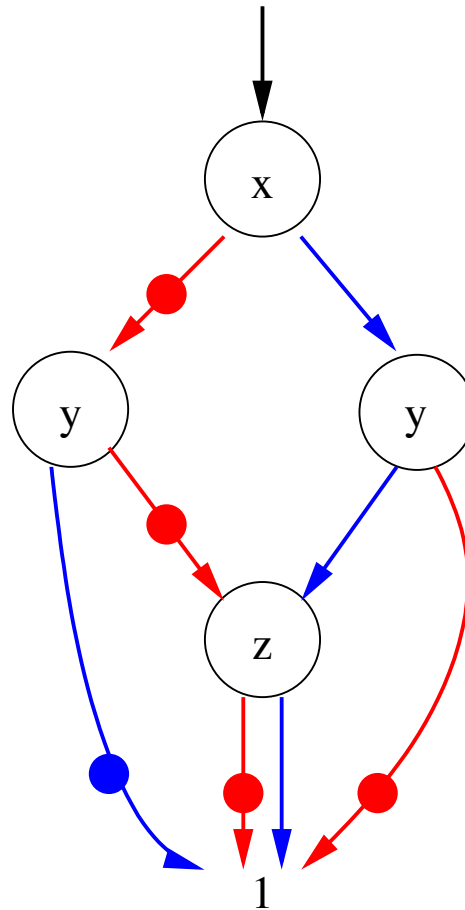
CBDD: Additional bit on every edge:

if bit true, negates meaning of its target node

Representation: if the bit is set, we put a filled circle onto the edge.

# CBDD: Example

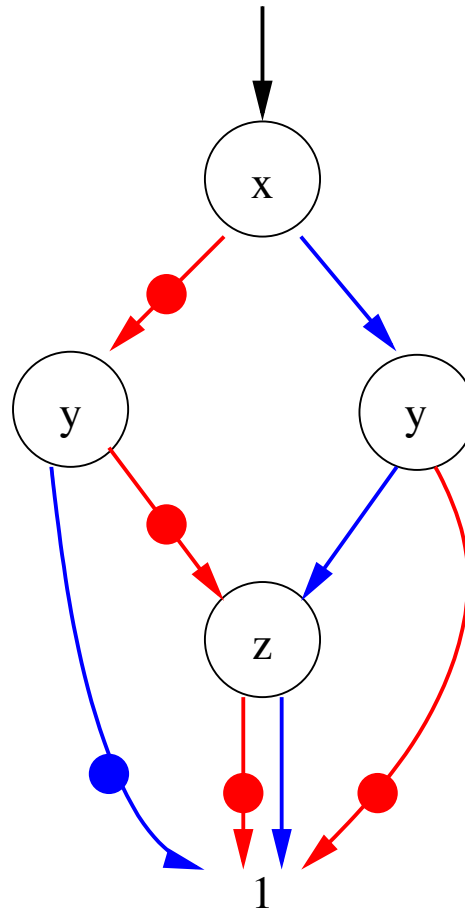
---



The red arc leaving the **z**-labelled node has its negation bit set, it therefore effectively leads to **0**.

# CBDD: Example

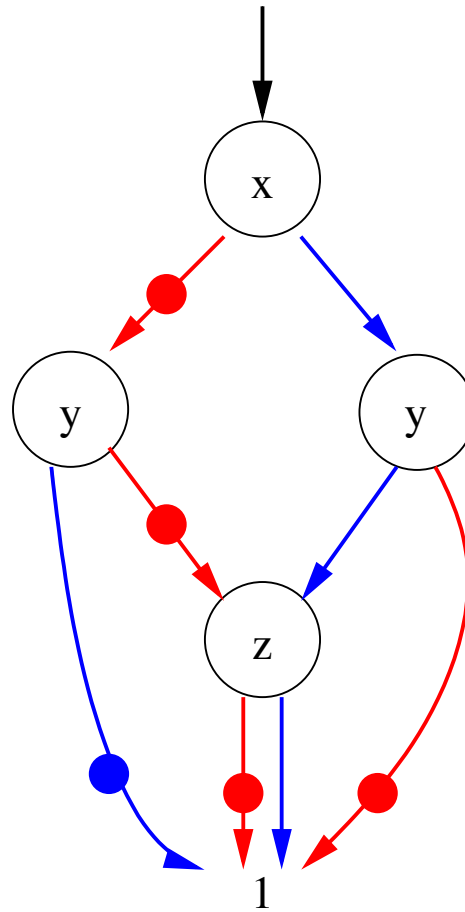
---



For this reason, the **z**-labelled node is *not redundant*.  
The **0**-leaf can be omitted altogether.

# CBDD: Example

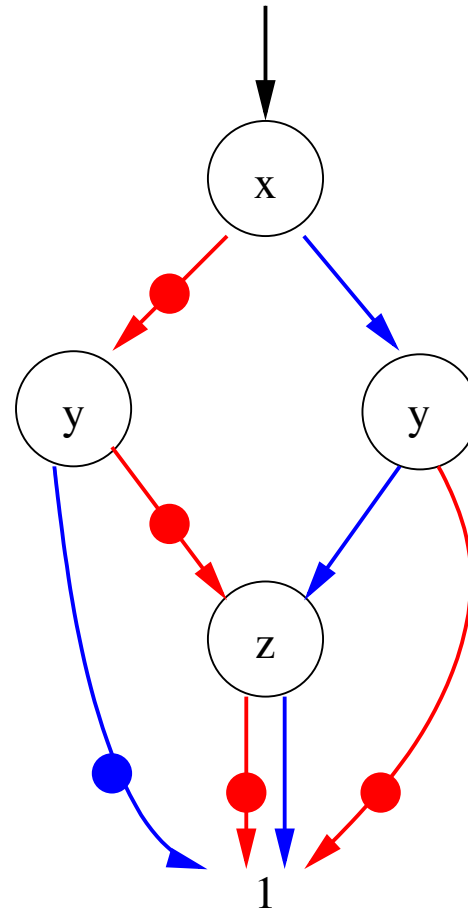
---



The left  $y$ -labelled node represents the formula  $\neg y \wedge \neg z$ .

# CBDD: Example

---



The pointer to the root is also equipped with a negation bit (false in this case).

# Remarks

---

A valuation  $\nu$  is a model of the formula represented by a CBDD iff the number of negations on the path corresponding to  $\nu$  is *even* (including the pointer to the root).

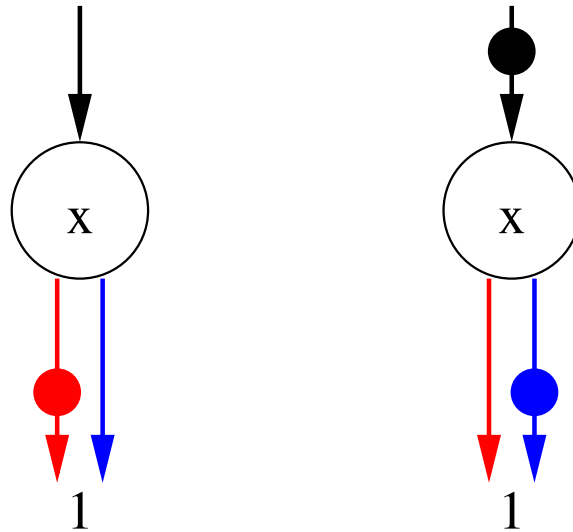
Negation with CBDDs: trivial, invert the negation bit of the pointer to the root (constant-time operation).

Implementation: coded into the least significant bit of the pointer

**Problem:** CBDDs (as presented until now) are not unique!

# CBDDs are not (yet) unique

---



Both of the CBDDs shown above represent the formula  $x$ .

# Canonical form

---

To ensure uniqueness, one can additionally prohibit negation bits on 0-labelled edges.

For this, we exploit the following equivalence:

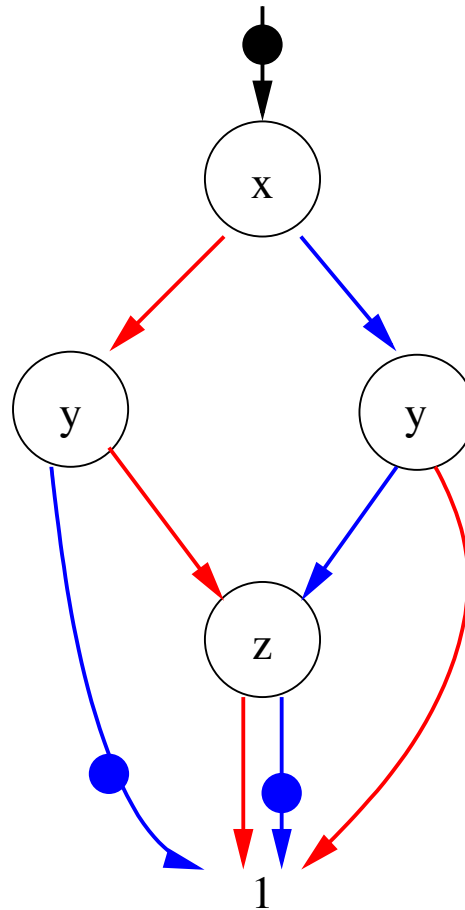
$$ite(x, F, \neg G) \equiv \neg ite(x, \neg F, G)$$

Given any CBDD, one can eliminate negated 0-labelled edges by inverting all the negation bits on those edges that are incident with its source node (starting at the leaves, finishing with the root).



# Canonical form

---



The CBDD shown above represents the same formula as before and does not have any negated 0-labelled edges.

# LTL with BDDs

---

**Question:** Can one implement also LTL model checking using BDDs?

**Answer:** Yes and no (worst-case: quadratical, but works ok in practice).

**Problems:** BDD not compatible with depth-first search, combination with partial-order reduction difficult.

# Symbolic algorithms for LTL

---

**Idea:** Find non-trivial SCCs with an accepting state, then search backwards for an initial state.

**Algorithms:** Emerson-Lei (EL), OWCTY

# Emerson-Lei (1986)

---

1. Assign to  $M$  the set of all states.
2. Let  $B := M \cap F$ .
3. Compute the set  $C$  of states that can reach elements of  $B$ .
4. Let  $M := M \cap \text{pre}(C)$ .
5. If  $M$  has changed, then go to step 2, otherwise stop.

# One-Way-Catch-Them-Young

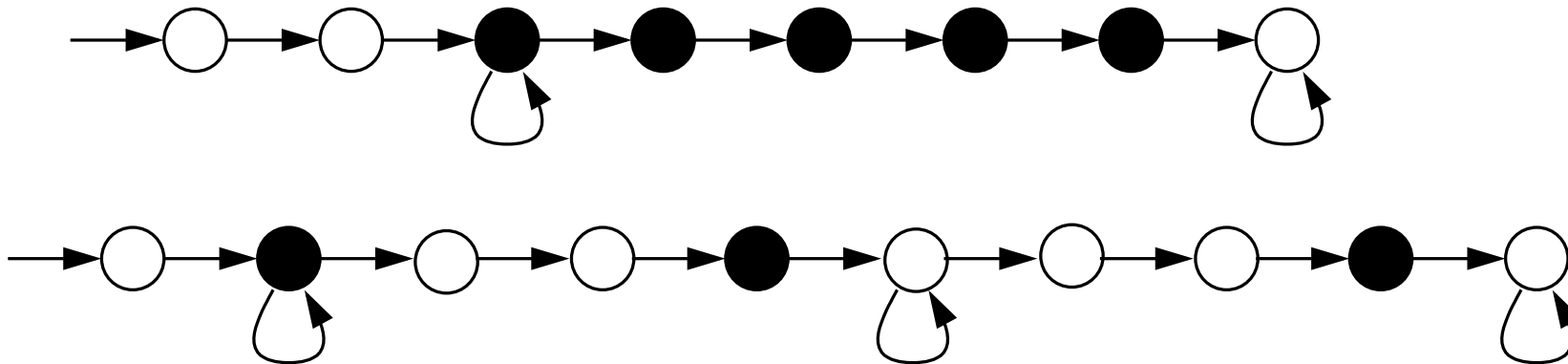
---

(Hardin et al 1997, Fisler et al 2001)

Like EL, but after step 4 repeat  $M := M \cap pre(M)$  until  $M$  does not change any more.

# EL and OWCTY

---



In the upper case, OWCTY is superior, in the lower case EL is.

In practice, OWCTY appears to work better.