Binary Decision Diagrams

Literature

Some pointers:

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

URL: http://www.itu.dk/people/hra/bdd97-abstract.html

Tools:

DDcal ("BDD calculator")

URL: http://vlsi.colorado.edu/∼fabio/

Set representations

As we have seen, the solution of the model-checking problem for CTL can be expressed by operations on sets:

states satisfying some atomic proposition: $\mu(p)$ for $p \in AP$

states satisfying (sub)formulae: $[\![\psi]\!]_{\mathcal{K}}$

computation by set operations: pre, $\cap$, $\cup$, ...

How can such sets be represented:

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

symbolic representation: compact notation or data structure

Symbolic representation

There are many ways of representing sets symbolically. Some are commonly used in mathematics:

Intervals: $[1, 10]$ for $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

Characterizations: “odd numbers” for $\{1, 3, 5, \ldots\}$

Every symbolic representation is suitable for some sets and less so for others (for instance, intervals for odd numbers).

We are interested in a data structure suitable for representing sets of states in hardware systems, and where the necessary operations (pre, $\cap$ etc) can be implemented efficiently.

In the following, we assume that states can be represented as Boolean vectors

$$S = \{0, 1\}^m \qquad \text{for some } m \geq 1$$

Example:

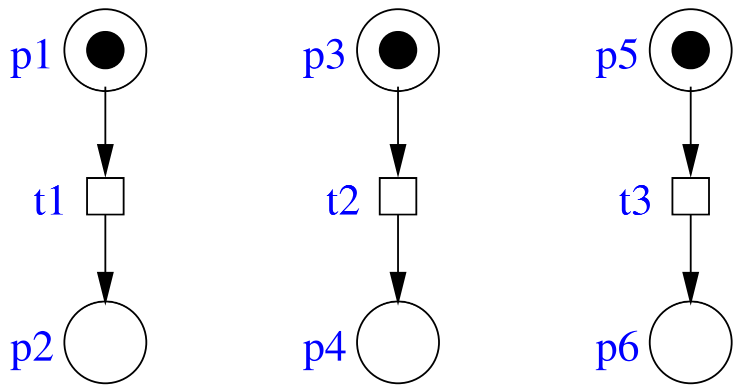
1-safe Petri nets (every place marked with at most one token)

circuits (all inputs and outputs are 0 or 1)

Remark: In general, the elements of *any* finite set can be represented by Boolean vectors if m is chosen large enough. (However, this may not be adequate for all sets.)

Example 1: Petri-net

Consider the following Petri net:

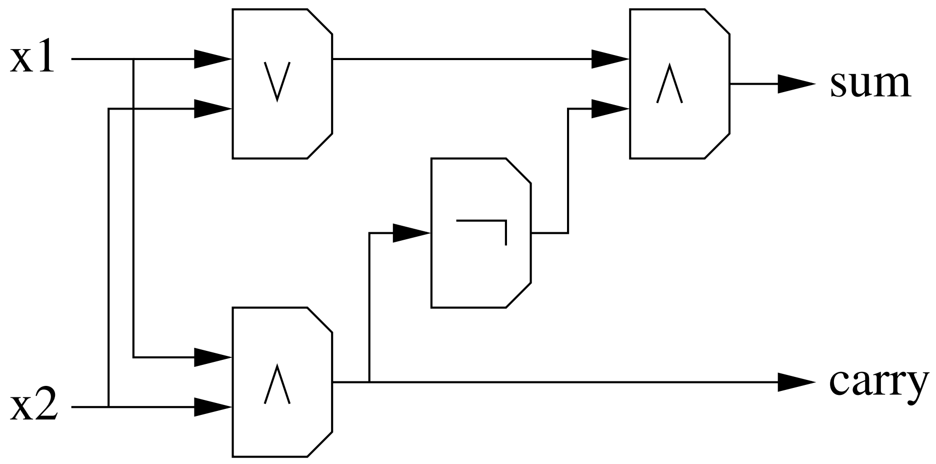


A state can be written as $(p_1, p_2, \ldots, p_6)$, where p_i, $1 \leq i \leq 6$ indicates whether there is a token on P_i.

Initial state$(1, 0, 1, 0, 1, 0)$;
other reachable states are, e.g., $(0, 1, 1, 0, 1, 0)$ or $(1, 0, 0, 1, 0, 1)$.

Example 2: 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.

Characteristic functions

Let $C \subseteq S = \{0,1\}^m$.
(i.e., a set of Boolean vectors.)

The set C is uniquely defined by its characteristic function $f_C \colon S \to \{0,1\}$ given by

$$f_C(s) := \begin{cases} 1 & \text{if } s \in C \\ 0 & \text{if } s \notin C \end{cases}$$

Remark: f_C is a Boolean function with m inputs and therefore corresponds to a formula F of propositional logic with m atomic propositions.

The characteristic function of the admissible combinations in Example 2 corresponds to the following formula of propositional logic:

$$F \equiv \Big(carry \leftrightarrow (x_1 \wedge x_2) \Big) \wedge \Big(sum \leftrightarrow (x_1 \vee x_2) \wedge \neg carry \Big)$$

In the following, we shall treat

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

characteristic functions

Formulae of propositional logic

simply as different representations of the same objects.

Representing formulae

Truth table:

x_1	x_2	*carry*	*sum*	F
0	0	0	0	1
0	0	0	1	0
		...		
0	1	0	1	1
		...		

A truth table is obviously *not* a compact representation.

However, we use it as a starting point for a graphical, more compact representation.

Binary decision graphs

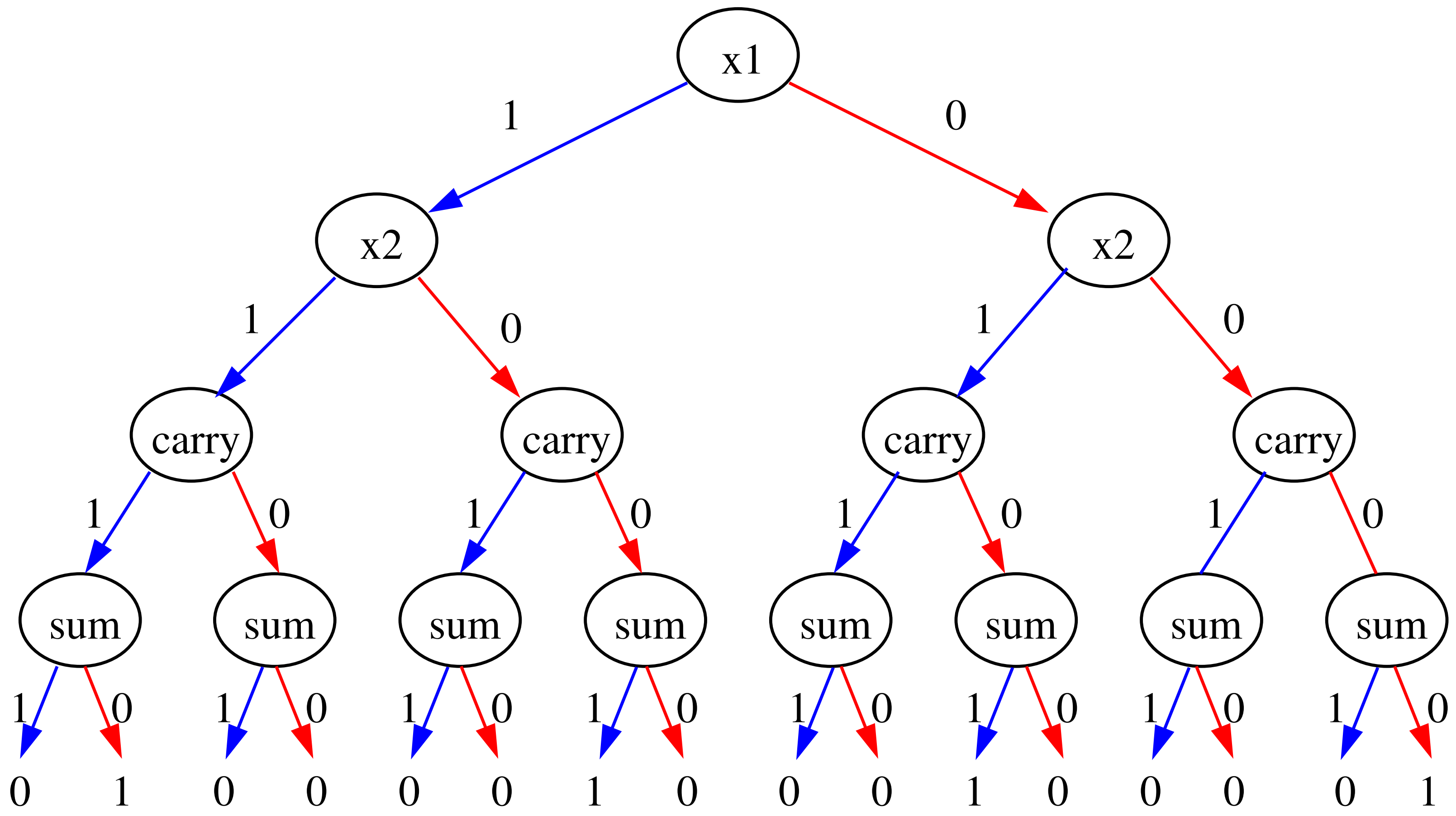
Let V be a set of variables (atomic propositions) and $<$ a total order on V, e.g.

$$x_1 < x_2 < carry < sum$$

A binary decision graph (w.r.t. $<$) 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 $\mathbf{0}$ or $\mathbf{1}$;
- all non-leaves are labelled with variabls 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 $x < y$.

Example 2: 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

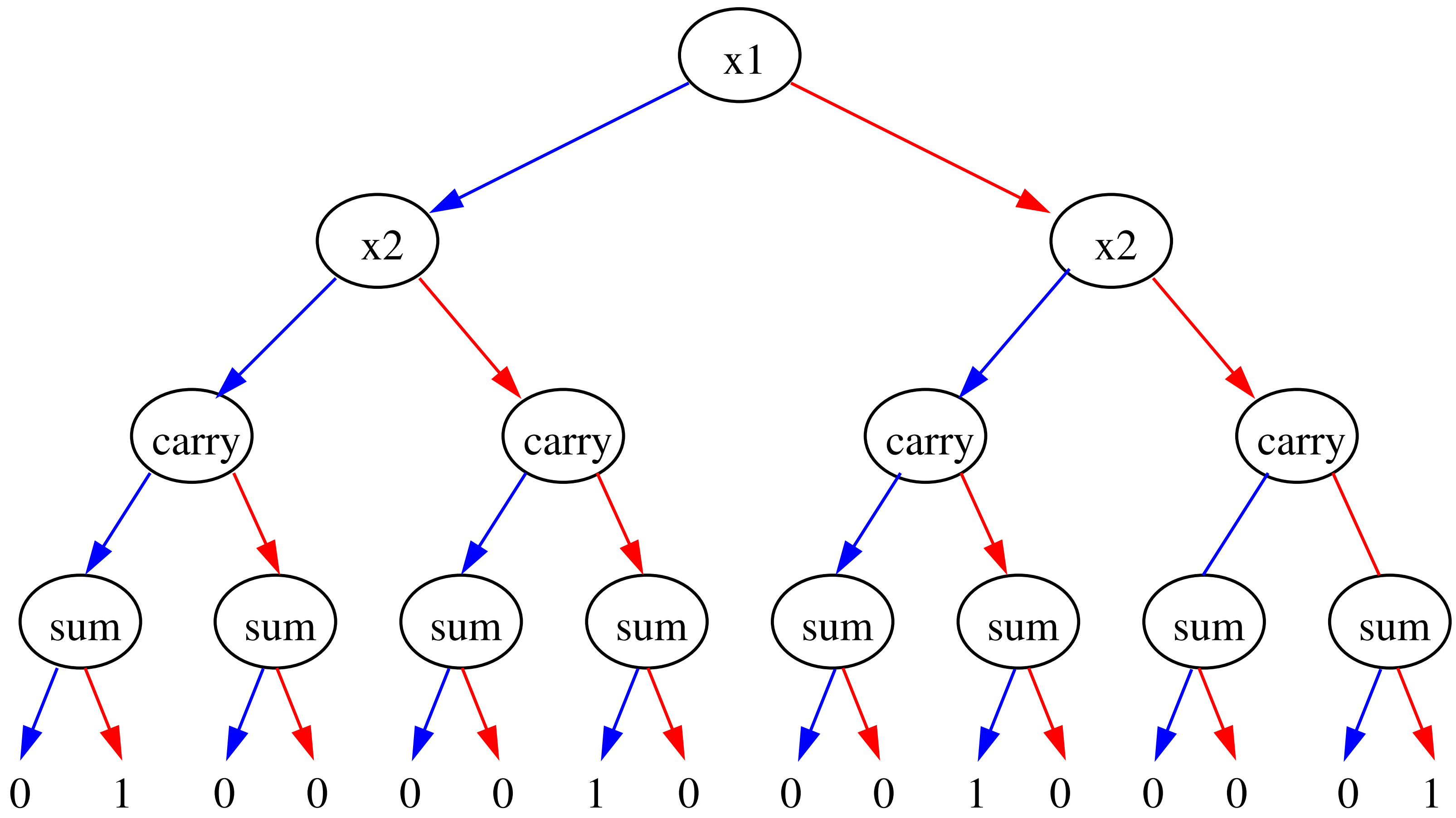
A binary decision diagram (BDD) is a binary decision graph with two additional properties:

- no two subgraphs are isomorphic;
- there are no *redundant* nodes, where both outgoing edges lead to the same target node.

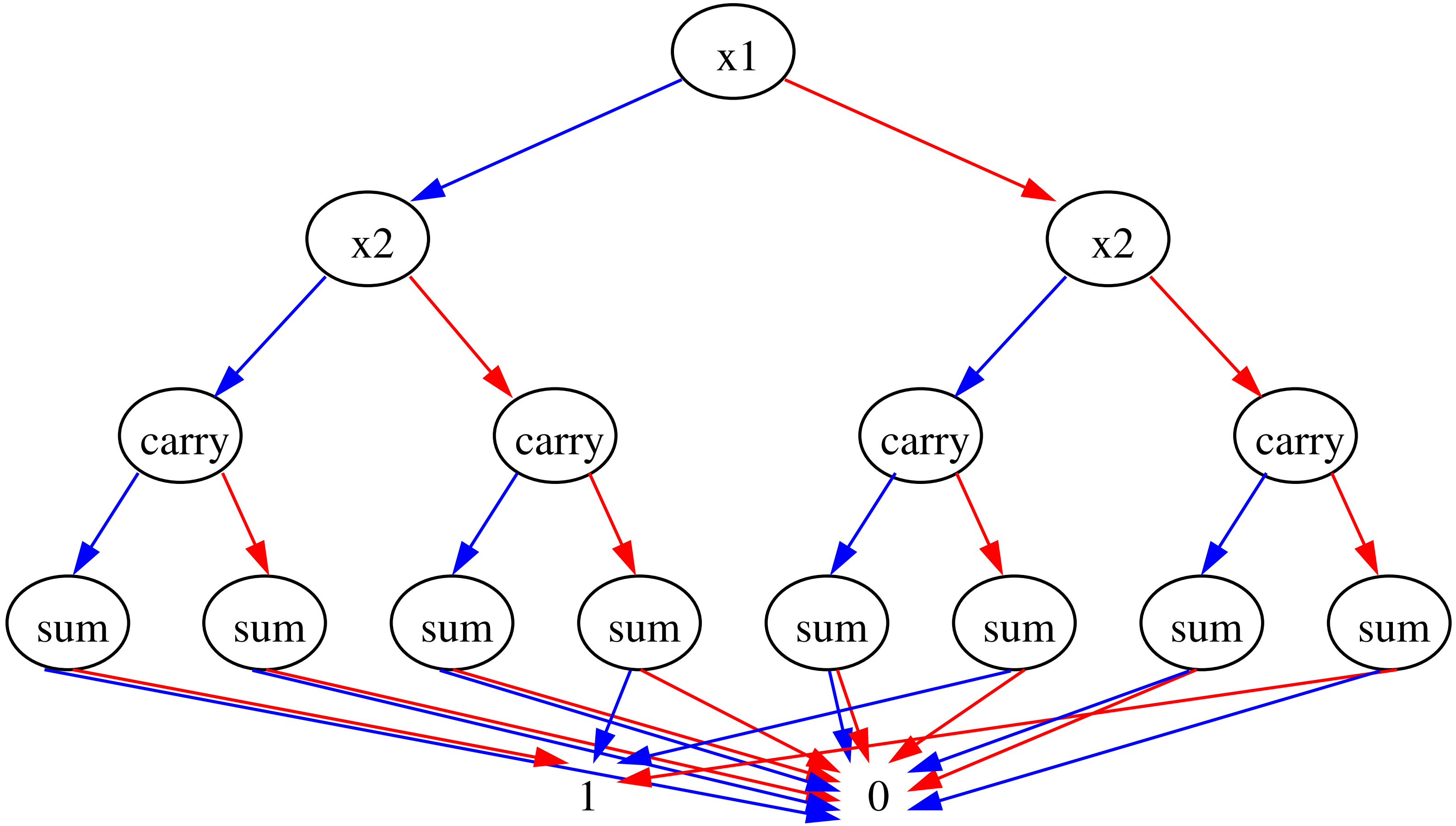
We also allow to omit the $\mathbf{0}$-node and the edges leading there.

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

Example 2: Eliminate isomorphic subgraphs (1/4)

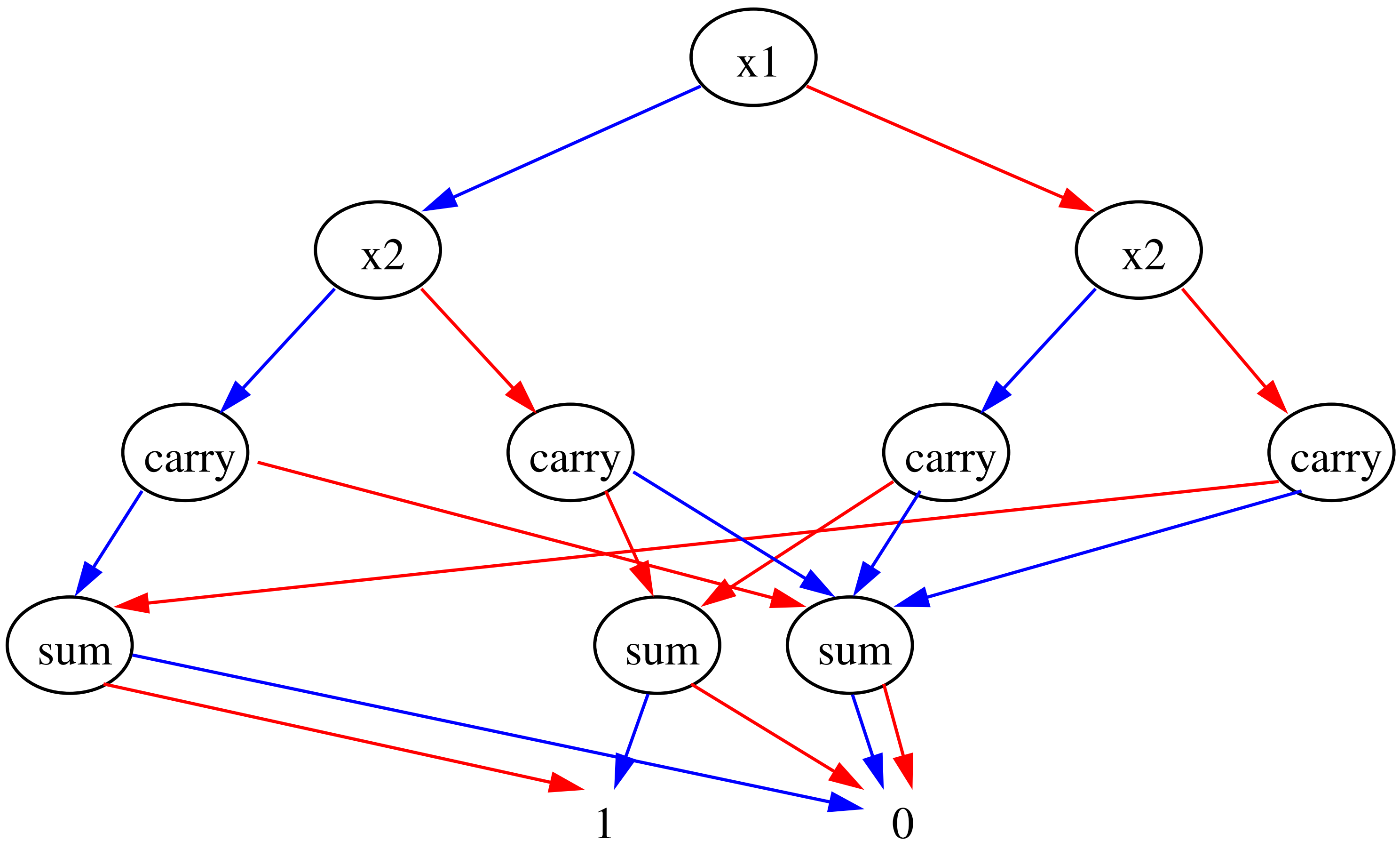


Example 2: Eliminate isomorphic subgraphs (2/4)



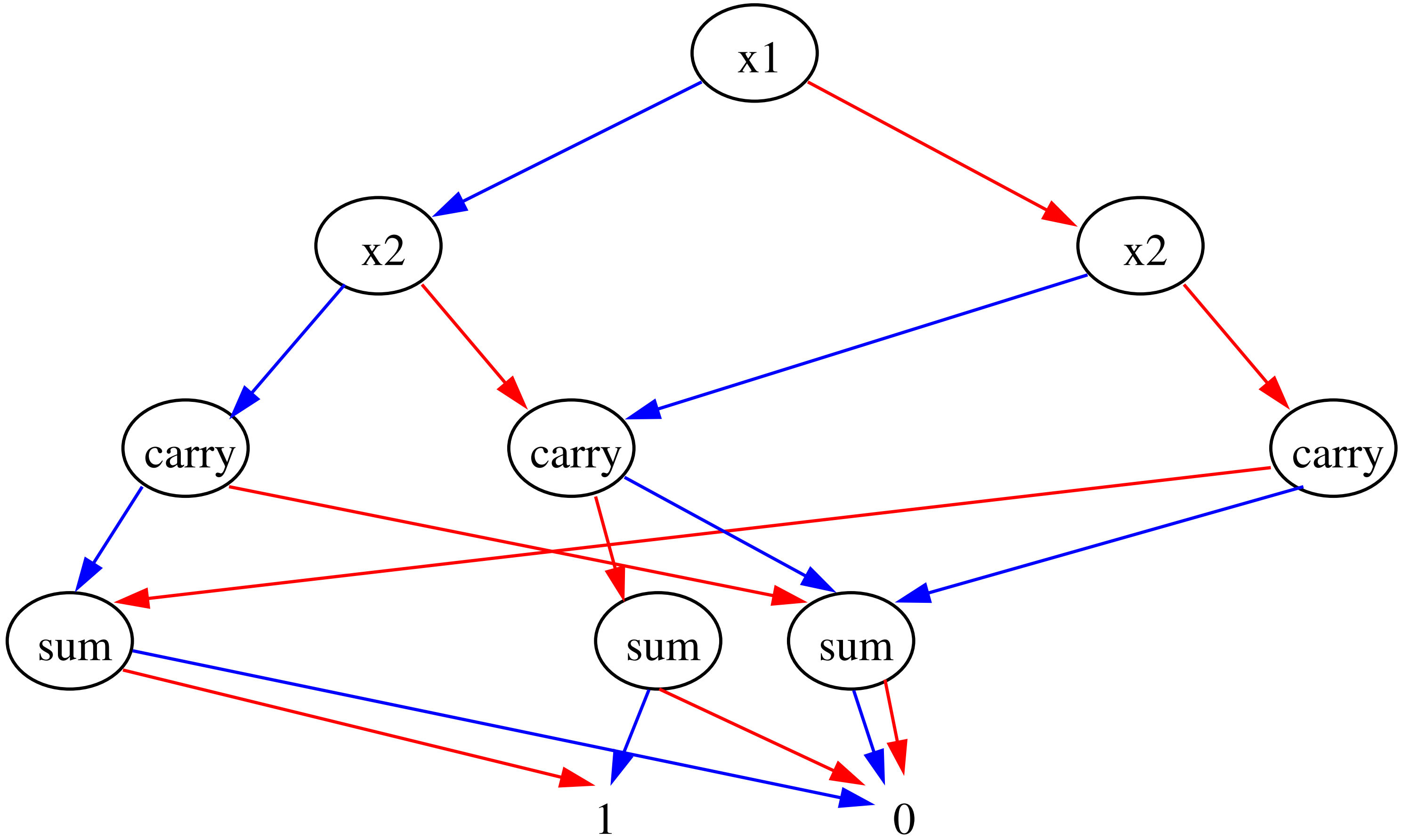
0-nodes and 1-nodes merged, respectively.

Example 2: Eliminate isomorphic subgraphs (3/4)



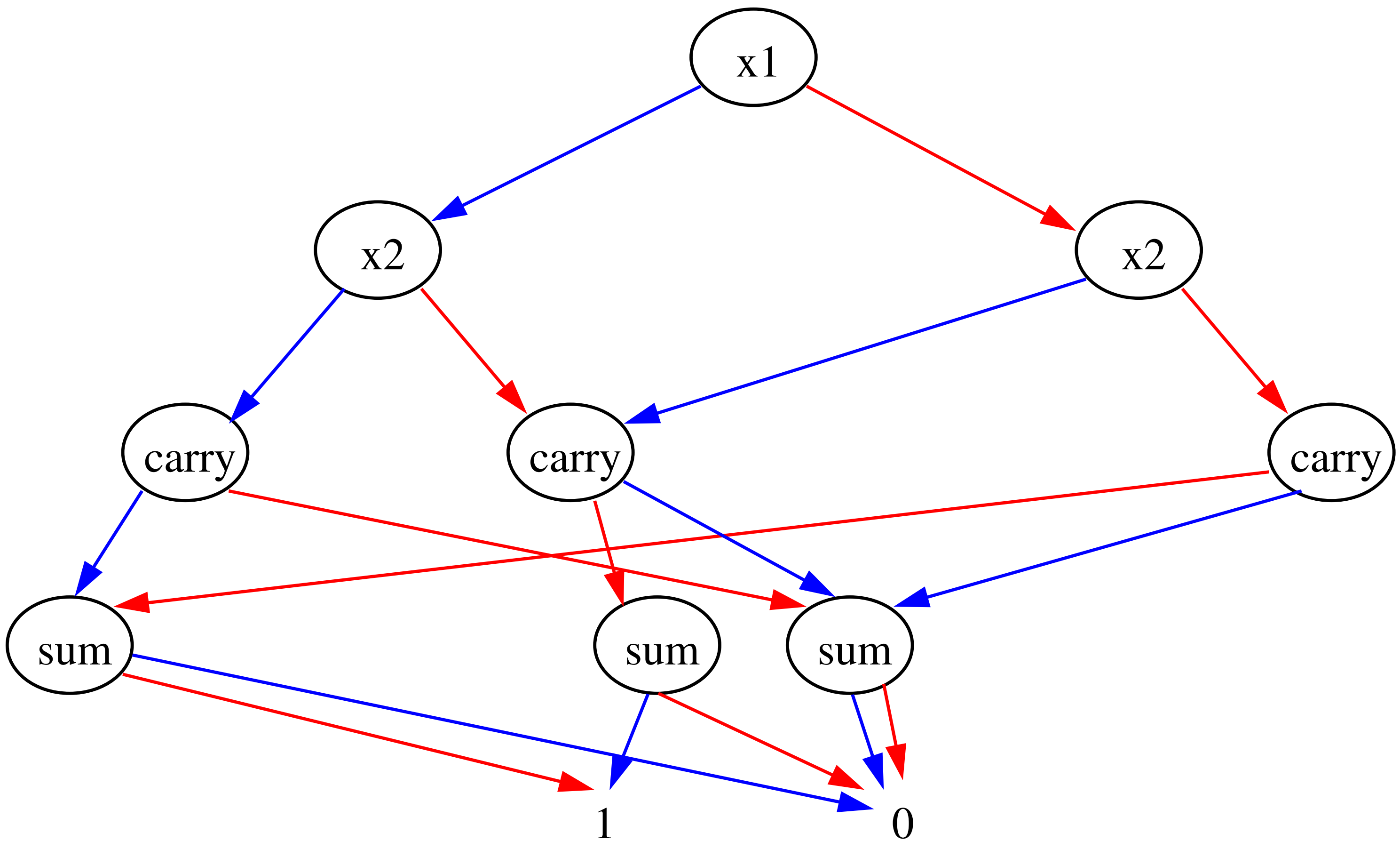
Merged the isomorphic *sum*-nodes.

Example 2: Eliminate isomorphic subgraphs (4/4)



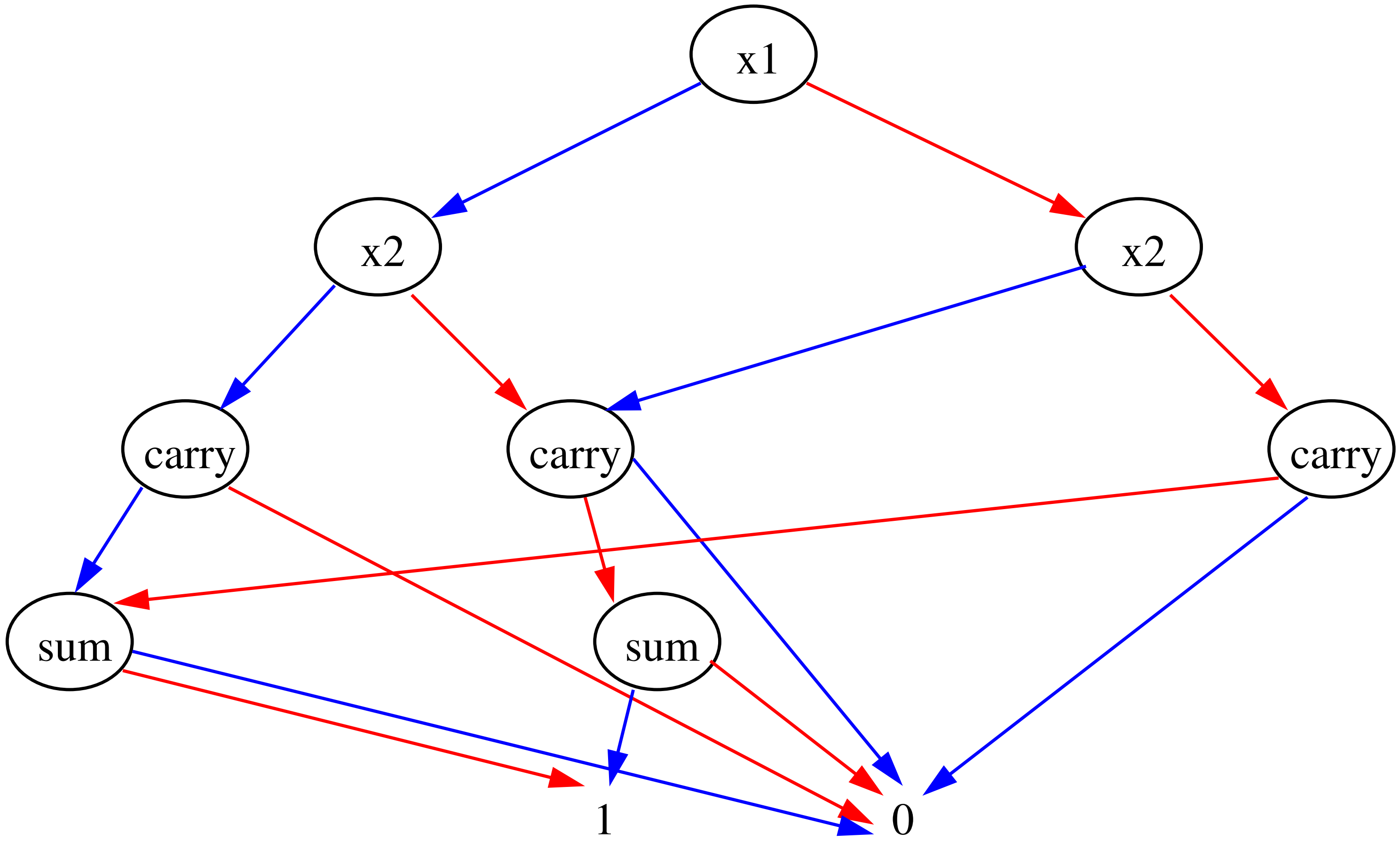
No isomorphic subgraphs are left → we are done.

Example 2: Remove redundant nodes (1/2)



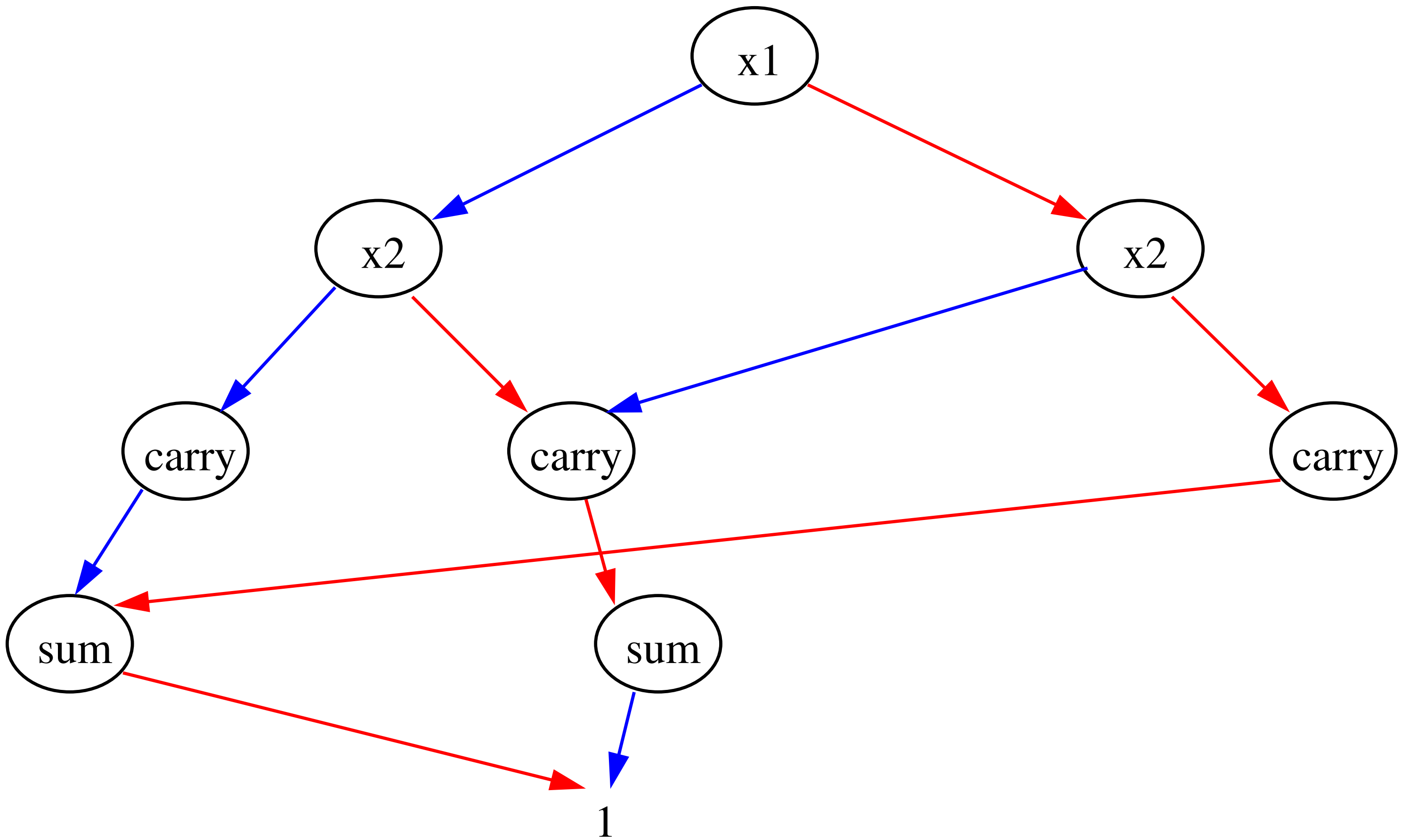
Both edges of the right *sum*-node point to 0.

Example 2: Remove redundant nodes (2/2)



No more redundant nodes → we are done.

Example 2: Omit 0-node



Optionally, we can remove the 0-node and edges leading to it, which makes the representation clearer.

Preview

In the following, we shall investigate operations on BDDs that are needed for CTL model checking.

Construction of a BDD (from a PL formula)

Equivalence check

Intersection, complement, union

Relations, computing predecessors

Propositional logic with constants

In the following, we will consider formulae of propositional logic (PL), extended with the constants $\mathbf{0}$ and $\mathbf{1}$, where:

$\mathbf{0}$ is an unsatisfiable formula;

$\mathbf{1}$ is a tautology.

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 us introduce a new, ternary PL operator. We shall call it *ite* (if-then-else).

Note: *ite* does not extend the expressiveness of PL, it is simply a convenient shorthand notation.

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

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

The set of INF formulae (if-then-else normal form) is inductively defined as follows:

$\mathbf{0}$ and $\mathbf{1}$ 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:

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

Proof: In the following, G denotes the right-hand side of the equivalence above. Let ν be a valuation s.t. $\nu \models F$. Either $\nu(x) = 1$, then ν 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 ν is a model of $F[x/1]$. Then, however, ν 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 equivalence above multiple times.)

Construction of BDDs

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

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

Otherwise, let x be the smallest variable (w.r.t. $<$) 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 isomorpic; if yes, then F is represented by B_0. Otherwise we merge all isomorphic subtrees in B_0 and B_1.

Example: BDD construction

Consider again the formula from Example 2:

$$F \equiv \Big(carry \leftrightarrow (x_1 \wedge x_2)\Big) \wedge \Big(sum \leftrightarrow (x_1 \vee x_2) \wedge \neg carry\Big)$$

We have, e.g.:

$$F[x_1/0] \equiv \neg carry \wedge (sum \leftrightarrow x_2)$$

$$F[x_1/1] \equiv (carry \leftrightarrow x_2) \wedge (sum \leftrightarrow \neg carry)$$

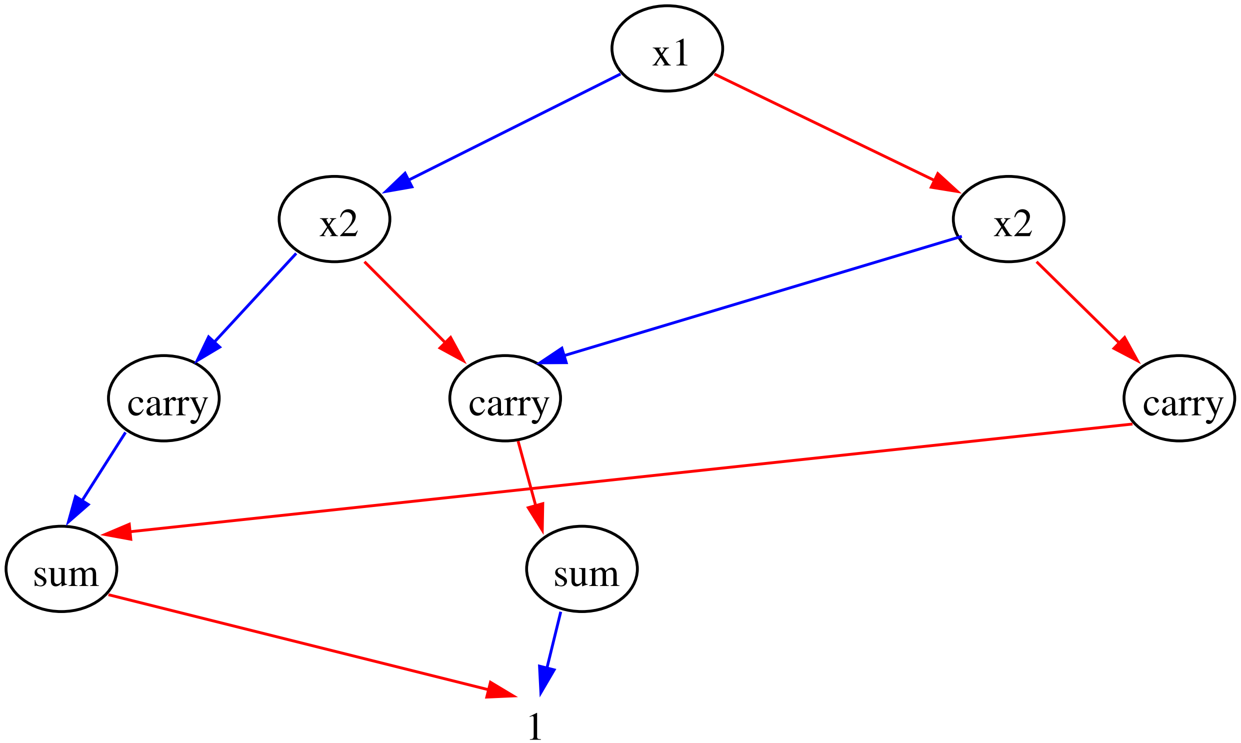
$$F[x_1/0][x_2/0] \equiv \neg carry \wedge \neg sum$$

$$F[x_1/0][x_2/1] \equiv F[x_1/1][x_2/0] \equiv \neg carry \wedge sum$$

$$F[x_1/1][x_2/1] \equiv carry \wedge sum$$

Example: BDD construction

By applying the construction, we obtain the same BDD as before:



Remark: Obviously, we can also obtain an INF formula from each BDD.

BDDs are unique

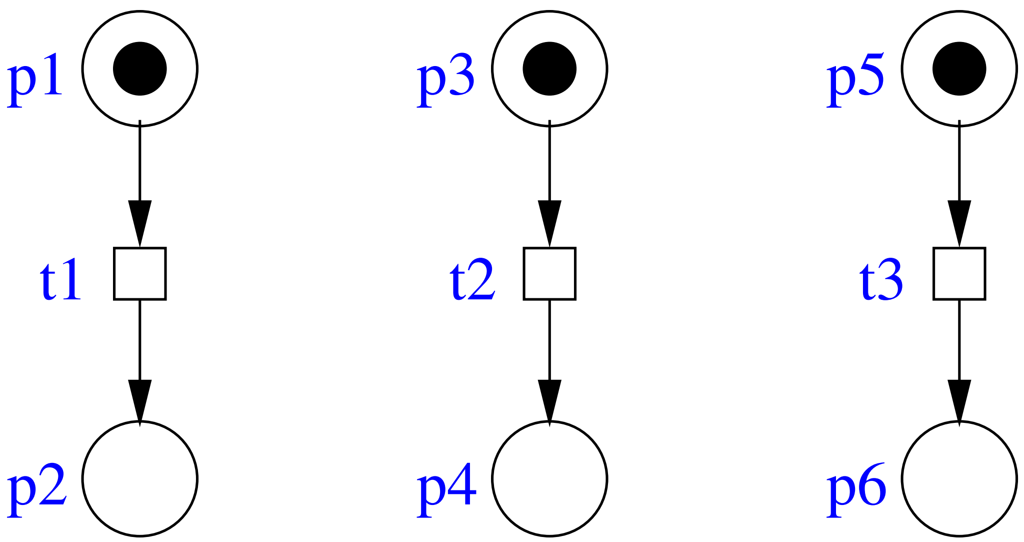
Remark: The result of the previously given construction is *unique* (up to isomorphism).

In other words, given F and $<$, there is (up to isomorphism) exactly one BDD that respects $<$ and represents F.

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

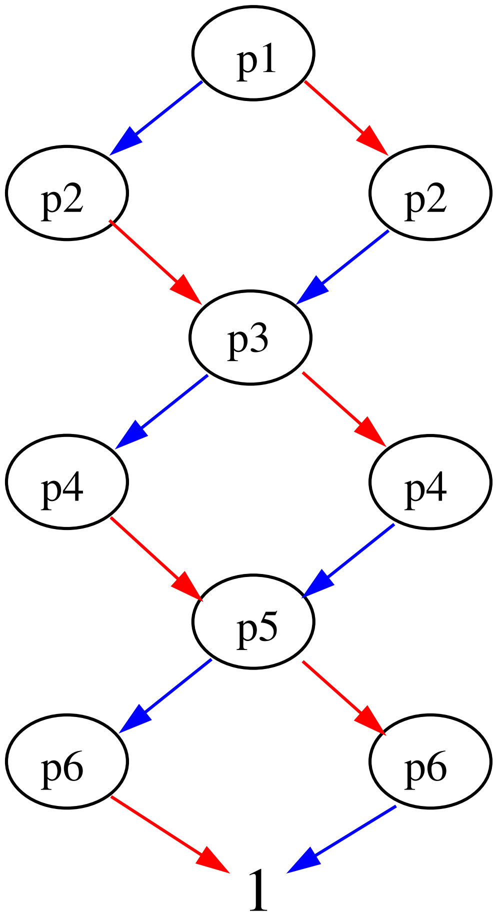
Example: Variable orderings

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 $p_1 < p_2 < p_3 < p_4 < p_5 < p_6$:



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

$$p_1 < p_3 < p_5 < p_2 < p_4 < p_6.$$

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 $\mathbf{0}$ leaf.

Tautology test: Check if the BDD consists just of the $\mathbf{1}$ 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.)

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

Initially, the hash table has only two unique entries, the leaves $\mathbf{0}$ and $\mathbf{1}$.

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.

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: Disjunction/union

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.

We have the following equivalence:

$$F \vee G \equiv ite(x, (F \vee G)[x/1], (F \vee G)[x/0]) \equiv ite(x, F[x/1] \vee G[x/1], F[x/0] \vee G[x/0])$$

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 disjunction according to the following, recursive strategy:

If B and C are equal, then return B.

If either B or C are the $\mathbf{1}$ leaf, then return $\mathbf{1}$.

If either B or C are the $\mathbf{0}$ 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 $mk(x, E, F)$.

Logical operations III: Intersection

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.

Solution: Analogous to union, with the rules for $\mathbf{1}$ and $\mathbf{0}$ 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

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

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

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

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

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

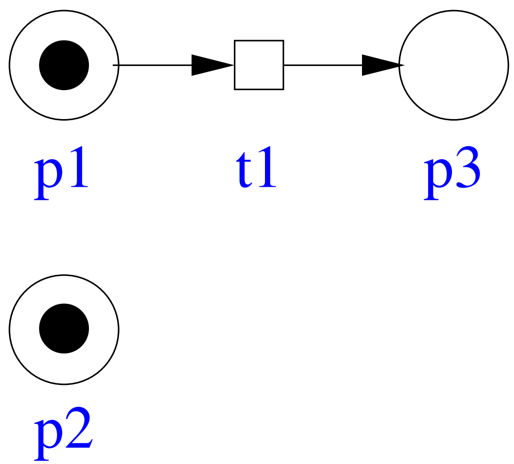
Thus, we can rewrite $pre(M)$ as follows:

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

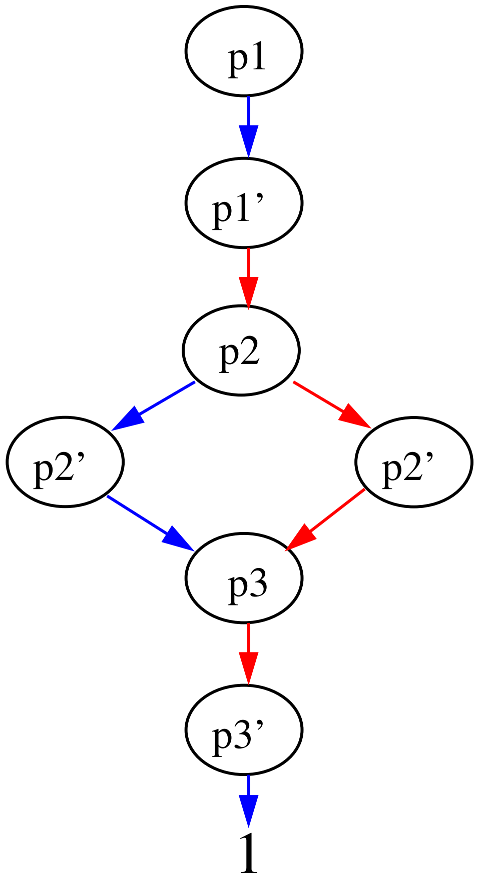
Then, pre reduces to the operations intersection and existential abstraction.

Example

Let us consider the following Petri net with just one transition:



The BDD F_{t_1} describes the effect of t_1, where p_1, p_2, p_3 describe the state *before* and p'_1, p'_2, p'_3 the state *after* firing t_1.



Existential abstraction

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

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

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.

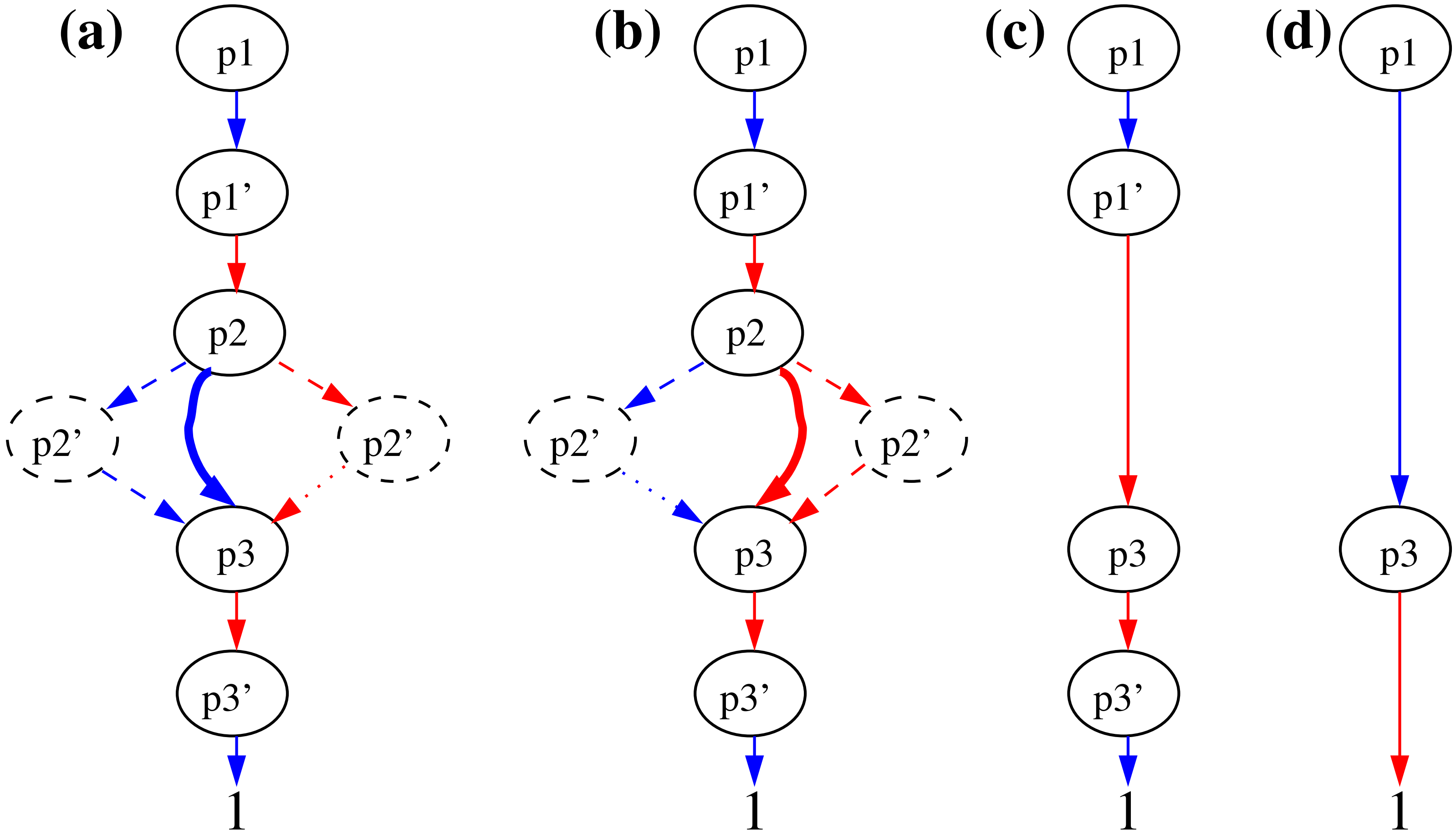
Example: Let $F \equiv (x_1 \wedge x_2) \vee x_3$. Then

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

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

Example: Existential abstraction

(a) $F_{t_1}[p'_2/1]$;   (b) $F_{t_1}[p'_2/0]$;   (c) $\exists p'_2 : F_{t_1}$;   (d) $\exists p'_1, p'_2, p'_3 : F_{t_1}$



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 $M := M \cap pre(C)$.

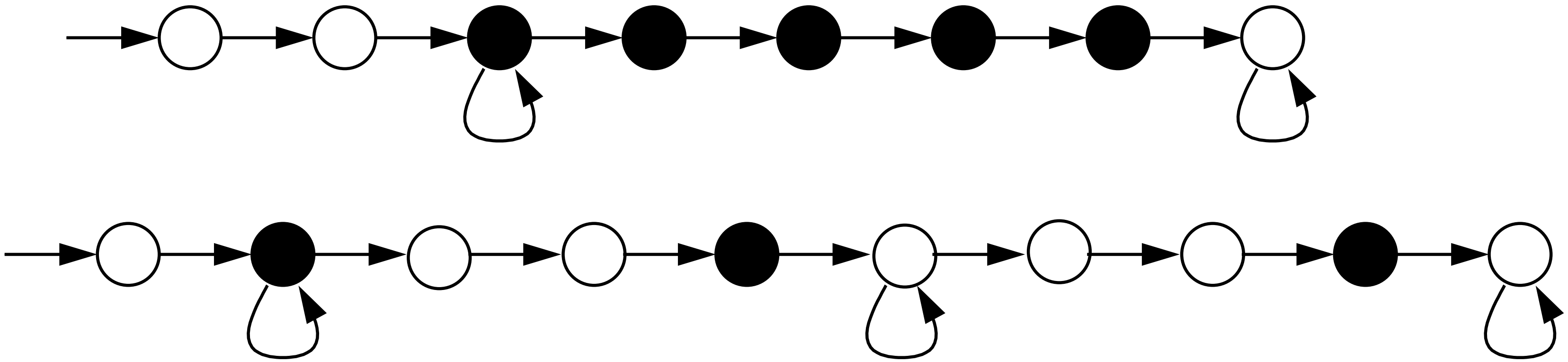
5. If M has changed, then go to step 2, otherwise stop.

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.