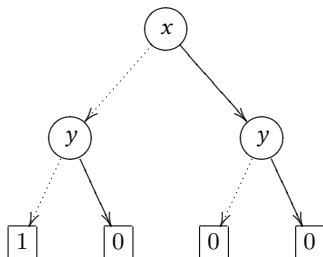# Binary decision trees

Tree for the boolean function $f(x, y) \doteq \neg x \wedge \neg y$

Note on notation:

- 0, 1 for $\perp$ (False), $\top$ (True)
- Often also have: $+, \cdot, ^-$ for $\vee, \wedge, \neg$

To compute value:

1. Start at root
2. Take dashed line if value of var at current node is 0
3. Take solid line if value of var at current node is 1
4. Function value is value at terminal node reached

Binary decision diagram

Similar to Binary Decision Trees, except that nodes can have multiple in-edges.

A **binary decision diagram** (BDD) is a finite DAG (Directed Acyclic Graph) with:

- a unique initial node;
- all non-terminals labelled with a boolean variable;
- all terminals labelled with 0 or 1;
- all edges are labelled 0 (dashed) or 1 (solid);
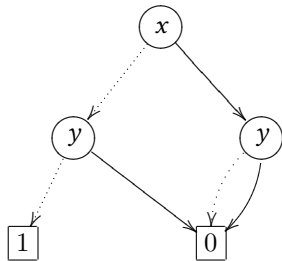- each non-terminal has exactly: one out-edge labelled 0, and one out-edge labelled 1.

We will use BDDs with two extra properties:

1. *Reduced* – eliminate redundancy
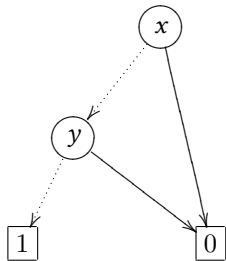2. *Ordered* – canonical ordering of the boolean variables
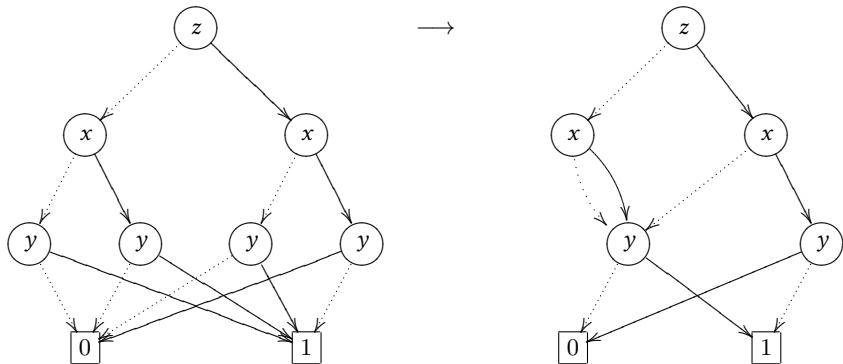
Reducing BDDs I
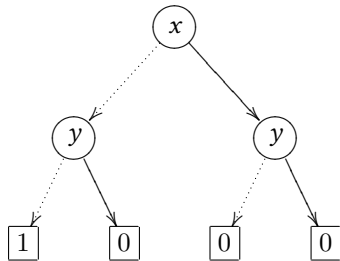
remove
duplicate
terminals
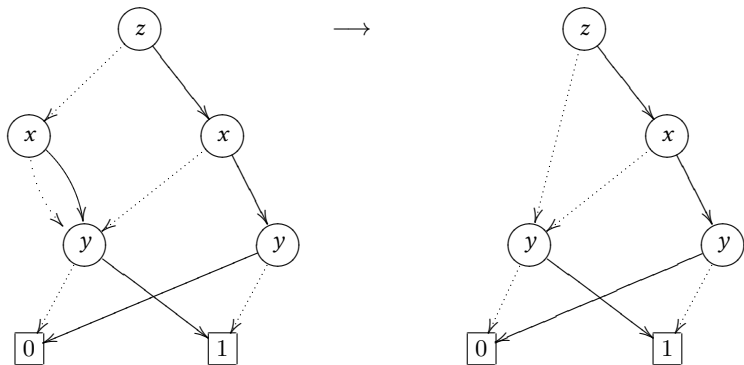$\longrightarrow$

remove
redundant
test $\longrightarrow$

Reducing BDDs II

Removing duplicate non-terminals:

# Reducing BDDs III

Removing redundant test:

Reduction Operations

1. **Removal of duplicate terminals.** If a BDD contains more than one terminal 0-node, then redirect all edges which point to such a 0-node to just one of them. Do the same with terminal nodes labelled 1.

2. **Removal of redundant tests.** If both outgoing edges of a node n point to the same node m, then remove node n, sending all its incoming edges to m.

3. **Removal of duplicate non-terminals.** If two distinct nodes n and m in the BDD are the roots of structurally identical subBDDs, then eliminate one of them and redirect all its incoming edges to the other one.
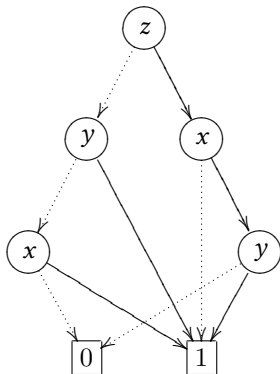
All of these operations preserve the BDD-ness of the DAG.

A BDD is **reduced** if it has been simplified as much as possible using these reduction operations.

Generality of BDDs
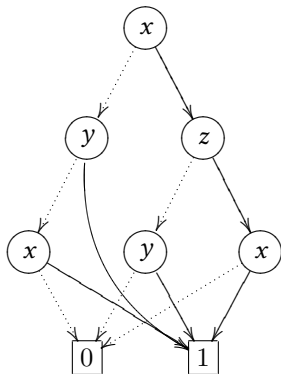
A variable might occur more
than once on a path

Ordering of variables on paths is not fixed

Ordered BDDs

- ▶ Let $[x_1, ..., x_n]$ be an ordered list of variables without duplicates;
- ▶ A BDD B **has an ordering** $[x_1, ..., x_n]$ if
    1. all variables of B occur in $[x_1, ..., x_n]$; and
    2. if x_j follows x_i on a path in B then $j > i$
- ▶ An **ordered BDD** (OBDD) is a BDD which has an ordering for some list of variables.
- ▶ The orderings of two OBDDs B and B' **are compatible** if there are no variables x, y such that
    - ▶ x is before y in the ordering for B, and
    - ▶ y is before x in the ordering for B'.

Theorem

*For a given ordering, the reduced OBDD (ROBDD) representing a given function f is **unique**.*

If B_1 and B_2 are two ROBDDs with compatible variable orderings representing the same boolean function, then they have identical structure.
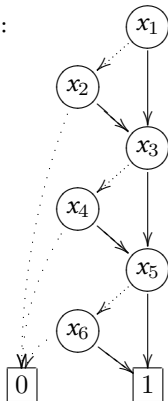
Impact of variable ordering on size (I)

Consider the boolean function
$$f(x_1, ..., x_{2n}) = (x_1 \lor x_2) \land (x_3 \lor x_4) \land ... \land (x_{2n-1} \lor x_{2n})$$

With variable ordering $[x_1, x_2, x_3, ..., x_{2n}]$ ROBDD has $2n + 2$ nodes
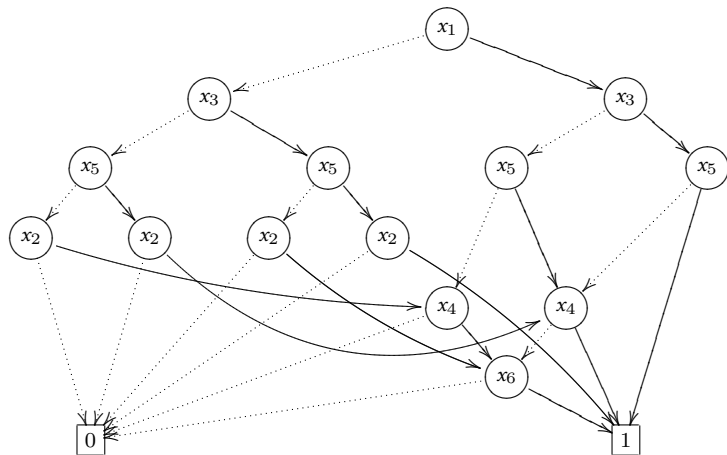
For $n = 3$:

Impact of variable ordering on size (II)

With $[x_1, x_3, ..., x_{2n-1}, x_2, x_4..., x_{2n}]$ ROBDD has 2^{n+1} nodes

For $n = 3$:

There are various heuristics that can help with choosing orderings.

However, improving a given ordering is NP-complete.

Importance of canonical representation

Having a canonical, i.e. unique, computable representation enables easy tests for

- **Absence of redundant variables.** A boolean function f does not depend on an input variable x if no nodes occur for x in the ROBDD for f.

- **Semantic equivalence.** Check $f \equiv g$ by checking whether or not the ROBDDs for f and g have identical structure.

- **Validity.** Check if the BDD is identical to the one with just the terminal node $\boxed{1}$ and nothing else.

- **Satisfiability.** Check if the BDD is not identical to the one with just the terminal node $\boxed{0}$ and nothing else.

- **Implication.** Check if $\forall \vec{x}.\, f(\vec{x}) \rightarrow g(\vec{x})$ by checking whether or not the ROBDD for $f \wedge \neg g$ is constant 0.

Representations of Boolean Functions

From H&R, Figure 6.1

Representation	compact?	test for		Operations		
		satisf'y	validity	$\wedge$	$\vee$	$\neg$
Prop. Formulas	often	hard	hard	easy	easy	easy
Formulas in DNF	sometimes	easy	hard	hard	easy	hard
Formulas in CNF	sometimes	hard	easy	easy	hard	hard
Truth Tables	never	hard	hard	hard	hard	hard
Reduced OBDDs	often	easy	easy	medium	medium	easy

Space complexity of representations and time complexities of operations on those representations.

Note: With a truth table representation, while operations are conceptually easy, especially when table rows are always listed in some standard order, the time complexities are hard, as table sizes and hence operation time complexities are always exponential in the number of input variables.

Binary Decision Diagrams

Binary Decision Diagrams: DAGs, such that

- Unique root node
- Variables on non-terminal nodes
- Truth-values on terminal nodes
- Exactly two edges from each non-terminal node, labelled 0, 1

Some notation, for a given BDD node n:

- If n is a non-terminal node:
  $\text{var}(n)$ — the variable label on node n;
  $\text{lo}(n)$ — the node reached by following the 0 edge from n;
  $\text{hi}(n)$ — the node reached by following the 1 edge from n;
- If n is a terminal node:
  $\text{val}(n)$ — the truth value labelling n

For a BDD B, the root node is called $\text{root}(B)$.

reduce

reduce constructs a ROBDD from an OBDD.

1. Label each OBDD node n with an integer $\text{id}(n)$,
2. in a single bottom-up pass, such that:
3. two OBDD nodes m and n have the same label ($\text{id}(m) = \text{id}(n)$) if and only if m and n represent the same boolean function.

The ROBDD is then created by using one node from each class of nodes with the same label.
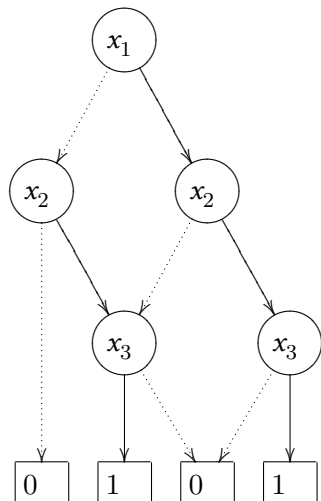
reduce

Assignment of labels follows the rules for performing reductions.

To label a node n:

- **Remove duplicate terminals:**
  if n is a terminal node (*i.e.*, $\boxed{0}$ or $\boxed{1}$), then set $\text{id}(n)$ to be $\text{val}(n)$.

- **Remove redundant tests:**
  if $\text{id}(\text{lo}(n)) = \text{id}(\text{hi}(n))$ then set $\text{id}(n)$ to be $\text{id}(\text{lo}(n))$.

- **Remove duplicate nodes:**
  if there exists a node m that has already been labelled such that
  $$\left\{ \begin{array}{l} \text{var}(m) = \text{var}(n) \\ \text{lo}(m) = \text{lo}(n) \\ \text{hi}(m) = \text{hi}(n) \end{array} \right\}, \text{ set } \text{id}(n) \text{ to } \text{id}(m).$$
  Use a hashtable with $\langle \text{var}(n), \text{lo}(n), \text{hi}(n) \rangle$ keys for $O(1)$ lookup time.

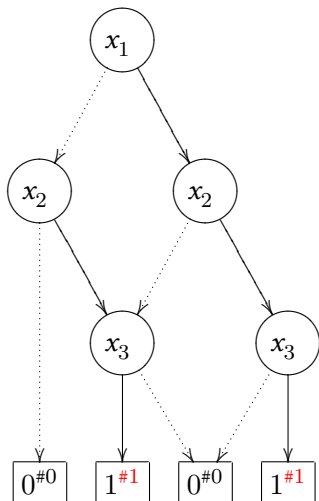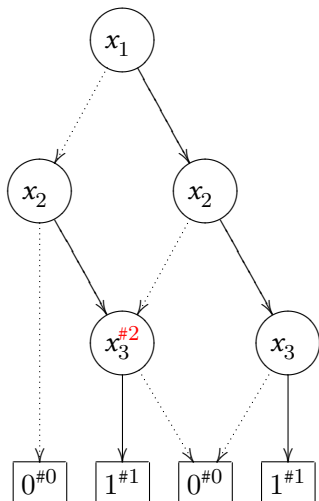- Otherwise, set $\text{id}(n)$ to an unused number.
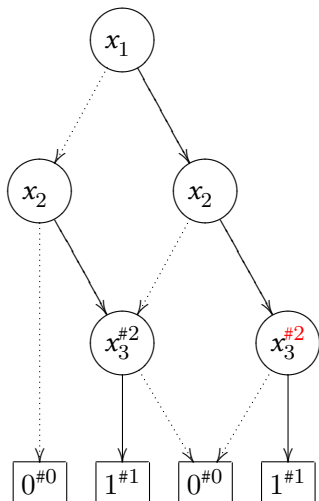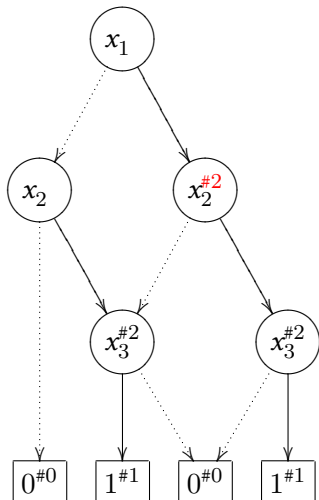
reduce Example

reduce Example
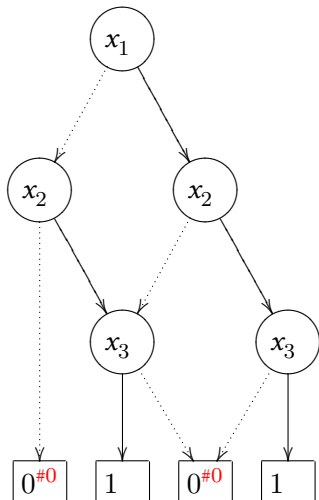
reduce Example

reduce Example
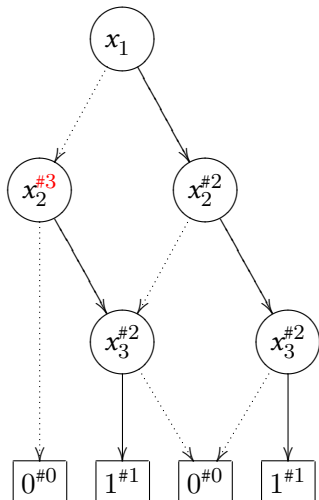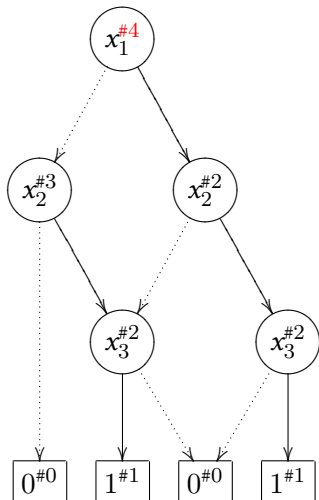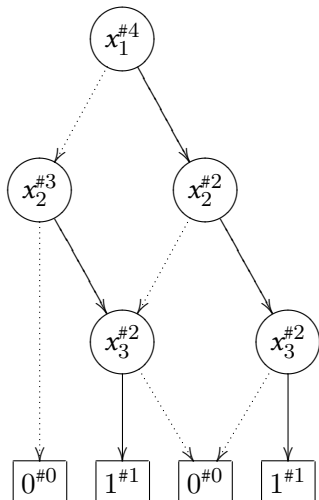
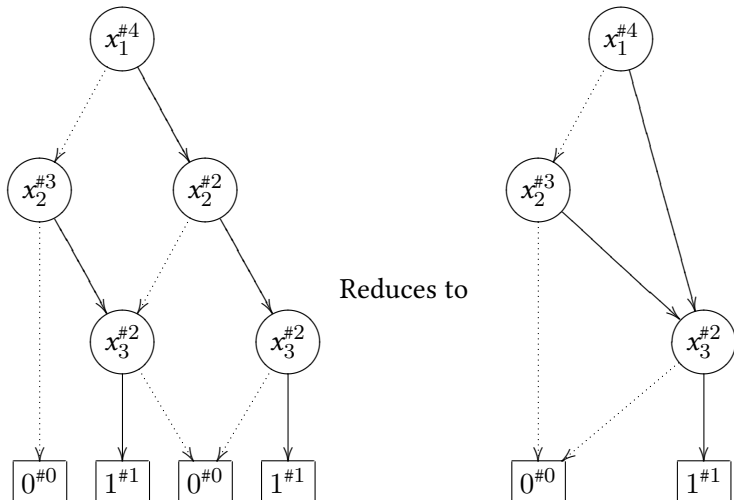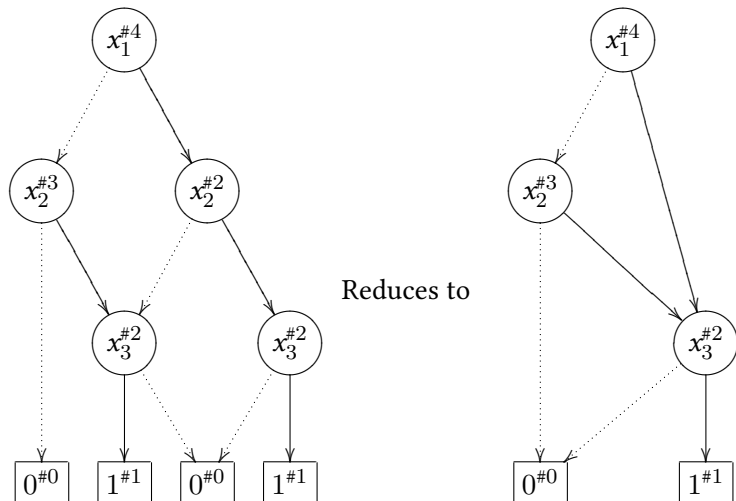# reduce Example

reduce Example

reduce Example

reduce Example

reduce Example

reduce Example

Reduces to

reduce Example

Reduces to

In practice, labelling and construction are interleaved.

apply

Given compatible OBDDs B_f and B_g that represent formulas f and g, `apply`$(\Box, B_f, B_g)$ computes an OBDD representing $f \Box g$.

- where $\Box$ represents some binary operation on boolean formulas
  for example, $\wedge$, $\vee$, $\oplus$
- Unary operations can be handled too.
  for example, negation: $\neg x = x \oplus 1$

apply: Shannon expansions

For any boolean formula f and variable x, it can be written as:

$$f \equiv (\neg x \wedge f[0/x]) \vee (x \wedge f[1/x])$$

This is the **Shannon expansion** of f (originally due to G. Boole).

`apply`: Shannon expansions

For any boolean formula f and variable x, it can be written as:

$$f \equiv (\neg x \wedge f[0/x]) \vee (x \wedge f[1/x])$$

This is the **Shannon expansion** of f (originally due to G. Boole).

In particular: $f \,\square\, g$ can be expanded like so:

$$f \,\square\, g \equiv (\neg x \wedge (f[0/x] \,\square\, g[0/x])) \vee (x \wedge (f[1/x] \,\square\, g[1/x]))$$
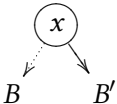
`apply`: Shannon expansions

For any boolean formula f and variable x, it can be written as:

$$f \equiv (\neg x \wedge f[0/x]) \vee (x \wedge f[1/x])$$

This is the **Shannon expansion** of f (originally due to G. Boole).

In particular: $f \square g$ can be expanded like so:

$$f \square g \equiv (\neg x \wedge (f[0/x] \square g[0/x])) \vee (x \wedge (f[1/x] \square g[1/x]))$$

If a BDD represents a boolean function f, then:

1. B represents $f[0/x]$ and B' represents $f[1/x]$; and
2. The BDD is effectively a compressed representation of f in Shannon normal form.

So: implement `apply` recursively on the structure of the BDDs.

apply: cases

$$\mathtt{apply}(\square, \underset{B \quad B'}{x}, \underset{C \quad C'}{x}) = \underset{\mathtt{apply}(\square, B, C) \quad \mathtt{apply}(\square, B', C')}{x}$$

$$\mathtt{apply}(\square, \underset{B \quad B'}{x}, C) = \underset{\mathtt{apply}(\square, B, C) \quad \mathtt{apply}(\square, B', C)}{x}$$

when C is terminal node, or non-terminal with $\mathrm{var}(\mathrm{root}(C)) > x$

$$\mathtt{apply}(\square, B, \underset{C \quad C'}{x}) = \underset{\mathtt{apply}(\square, B, C) \quad \mathtt{apply}(\square, B, C')}{x}$$

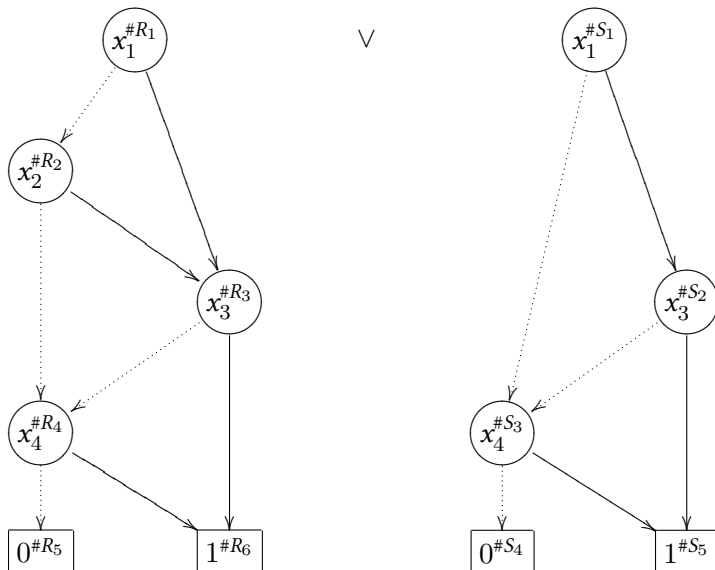when B is terminal node, or non-terminal with $\mathrm{var}(\mathrm{root}(B)) > x$

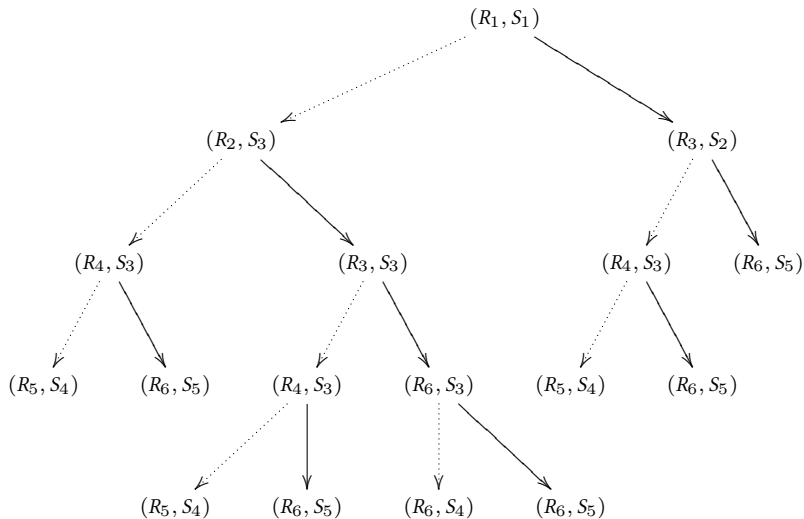$$\mathtt{apply}(\square, \boxed{u}, \boxed{v}) = \boxed{u \,\square\, v}$$

`apply`: example

Compute $\mathtt{apply}(\vee, B_f, B_g)$, where B_f and B_g are:

apply: recursive calls

`apply`: memoisation

The recursive `apply` implementation will generate an OBBD.

- Apply `reduce` to convert it back to a ROBDD.

However, as can be seen from the tree of recursive calls, there are many calls to `apply` with the same arguments.

- Each invocation of `apply` where at least one of the arguments is non-terminal generates two further calls to `apply`: the number of calls is worst-case exponential in the sizes of the original diagrams.

We are not taking into account the **sharing** in BDDs.

We can greatly improve the run-time by using **memoisation**: remembering the results of previous calls.
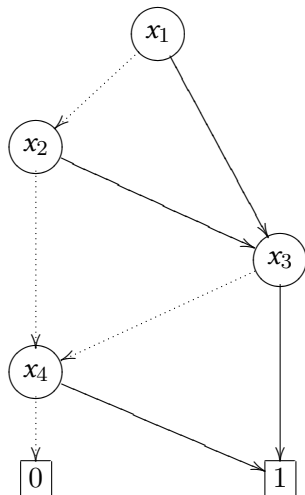
`apply`: memoised recursive calls

Memoisation results in at most $|B_f| \cdot |B_g|$ calls to `apply`.
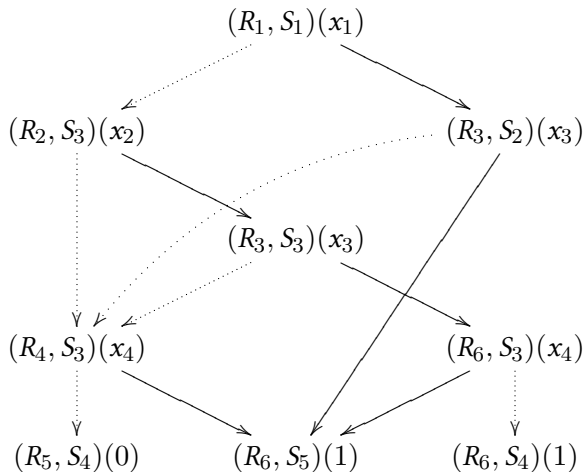
apply: Result

If we are careful to never create the same BDD node twice (using the same lookup table technique as reduce), then with memoisation, we automatically get a reduced BDD:

Other Operations

`restrict(0, x, B_f)` computes ROBDD for $f[0/x]$

1. For each node n labelled with x, incoming edges are redirected to $lo(n)$, and the node n is removed.

2. Resulting BDD then reduced with `reduce`.

3. (again, `reduce` can be interleaved with the removal.)

`exists(x, B_f)` computes ROBDD for $\exists x.\ f$.

1. Uses the identity

$$(\exists x.\ f) \equiv f[0/x] \lor f[1/x]$$

2. Realised using the `restrict` and `apply` functions:

$$\texttt{apply}(\lor, \texttt{restrict}(0, x, B_f), \texttt{restrict}(1, x, B_f))$$