Formal Specification and Verification

29.04.2014

Viorica Sofronie-Stokkermans

e-mail: sofronie@uni-koblenz.de

Mathematical foundations

Formal logic:

- Syntax: a formal language (formula expressing facts)
- Semantics: to define the meaning of the language, that is which facts are valid)
- Deductive system: made of axioms and inference rules to formaly derive theorems, that is facts that are provable

Last time

Propositional classical logic

- Syntax
- Semantics

  Models, Validity, and Satisfiability

  Entailment and Equivalence
- Checking Unsatisfiability

  Truth tables

  "Rewriting" using equivalences

  Proof systems: clausal/non-clausal

  - non-clausal: Hilbert calculus

    sequent calculus

  - clausal: Resolution

Today

Propositional classical logic

Proof systems: clausal/non-clausal

- non-clausal: Hilbert calculus

  sequent calculus

- clausal: Resolution; DPLL (translation to CNF needed)

- Binary Decision Diagrams

The DPLL Procedure

Goal:

Given a propositional formula in CNF (or alternatively, a finite set N of clauses), check whether it is satisfiable (and optionally: output *one* solution, if it is satisfiable).

Satisfiability of Clause Sets

$\mathcal{A} \models N$ if and only if $\mathcal{A} \models C$ for all clauses C in N.

$\mathcal{A} \models C$ if and only if $\mathcal{A} \models L$ for some literal $L \in C$.

Partial Valuations

Since we will construct satisfying valuations incrementally,
we consider partial valuations
(that is, partial mappings $\mathcal{A} : \Pi \to \{0, 1\}$).

We start with an empty valuation and try to extend it
step by step to all variables occurring in N.

If $\mathcal{A}$ is a partial valuation, then literals and clauses can be
true, false, or undefined under $\mathcal{A}$.

A clause is true under $\mathcal{A}$ if one of its literals is true;
it is false (or "conflicting") if all its literals are false;
otherwise it is undefined (or "unresolved").

Unit Clauses

Observation:

Let $\mathcal{A}$ be a partial valuation. If the set N contains a clause C, such that all literals but one in C are false under $\mathcal{A}$, then the following properties are equivalent:

- there is a valuation that is a model of N and extends $\mathcal{A}$.
- there is a valuation that is a model of N and extends $\mathcal{A}$ and makes the remaining literal L of C true.

C is called a unit clause; L is called a unit literal.

Pure Literals

One more observation:

Let $\mathcal{A}$ be a partial valuation and P a variable that is undefined under $\mathcal{A}$. If P occurs only positively (or only negatively) in the unresolved clauses in N, then the following properties are equivalent:

- there is a valuation that is a model of N and extends $\mathcal{A}$.
- there is a valuation that is a model of N and extends $\mathcal{A}$ and assigns true (false) to P.

P is called a pure literal.

The Davis-Putnam-Logemann-Loveland Proc.

```
boolean DPLL(clause set N, partial valuation 𝒜) {
    if (all clauses in N are true under 𝒜) return true;
    elsif (some clause in N is false under 𝒜) return false;
    elsif (N contains unit clause P) return DPLL(N, 𝒜 ∪ {P ↦ 1});
    elsif (N contains unit clause ¬P) return DPLL(N, 𝒜 ∪ {P ↦ 0});
    elsif (N contains pure literal P) return DPLL(N, 𝒜 ∪ {P ↦ 1});
    elsif (N contains pure literal ¬P) return DPLL(N, 𝒜 ∪ {P ↦ 0});
    else {
        let P be some undefined variable in N;
        if (DPLL(N, 𝒜 ∪ {P ↦ 0})) return true;
        else return DPLL(N, 𝒜 ∪ {P ↦ 1});
    }
}
```

The Davis-Putnam-Logemann-Loveland Proc.

Initially, DPLL is called with the clause set N and with an empty partial valuation $\mathcal{A}$.

The Davis-Putnam-Logemann-Loveland Proc.

In practice, there are several changes to the procedure:

The pure literal check is often omitted (it is too expensive).

The branching variable is not chosen randomly.

The algorithm is implemented iteratively;
the backtrack stack is managed explicitly
(it may be possible and useful to backtrack more than one level).

DPLL Iteratively

An iterative (and generalized) version:

```
status = preprocess();
if (status != UNKNOWN) return status;
while(1) {
    decide_next_branch();
    while(1) {
        status = deduce();
        if (status == CONFLICT) {
            blevel = analyze_conflict();
            if (blevel == 0) return UNSATISFIABLE;
            else backtrack(blevel); }
        else if (status == SATISFIABLE) return SATISFIABLE;
        else break;
    }
}
```

DPLL Iteratively

```
preprocess()
```

preprocess the input (as far as it is possible without branching);
return CONFLICT or SATISFIABLE or UNKNOWN.

```
decide_next_branch()
```

choose the right undefined variable to branch;
decide whether to set it to 0 or 1;
increase the backtrack level.

DPLL Iteratively

`deduce()`

make further assignments to variables (e.g., using the unit clause rule) until a satisfying assignment is found, or until a conflict is found, or until branching becomes necessary;
return CONFLICT or SATISFIABLE or UNKNOWN.

DPLL Iteratively

`analyze_conflict()`

check where to backtrack.

`backtrack(blevel)`

backtrack to `blevel`;

flip the branching variable on that level;

undo the variable assignments in between.

Branching Heuristics

Choosing the right undefined variable to branch is important for efficiency, but the branching heuristics may be expensive itself.

State of the art: use branching heuristics that need not be recomputed too frequently.

In general: choose variables that occur frequently.

The Deduction Algorithm

For applying the unit rule, we need to know the number of literals in a clause that are not false.

Maintaining this number is expensive, however.

The Deduction Algorithm

Better approach: "Two watched literals":

In each clause, select two (currently undefined) "watched" literals.

For each variable P, keep a list of all clauses in which P is watched and a list of all clauses in which $\neg P$ is watched.

If an undefined variable is set to 0 (or to 1), check all clauses in which P (or $\neg P$) is watched and watch another literal (that is true or undefined) in this clause if possible.

Watched literal information need not be restored upon backtracking.

Conflict Analysis and Learning

Goal: Reuse information that is obtained in one branch in further branches.

Method: Learning:

If a conflicting clause is found, use the resolution rule to derive a new clause and add it to the current set of clauses.

Problem: This may produce a large number of new clauses; therefore it may become necessary to delete some of them afterwards to save space.

Backjumping

Related technique:

non-chronological backtracking ("backjumping"):

If a conflict is independent of some earlier branch, try to skip that over that backtrack level.

Restart

Runtimes of DPLL-style procedures depend extremely on the choice of branching variables.

If no solution is found within a certain time limit, it can be useful to restart from scratch with another choice of branchings (but learned clauses may be kept).

A succinct formulation

State: $M||F$,

where:

- M partial assignment (sequence of literals),

  some literals are annotated (L^d: decision literal)

- F clause set.

A succinct formulation

UnitPropagation

$M||F, C \vee L \Rightarrow M, L||F, C \vee L$ if $M \models \neg C$, and L undef. in M

Decide

$M||F \Rightarrow M, L^d||F$ if L or $\neg L$ occurs in F, L undef. in M

Fail

$M||F, C \Rightarrow \text{Fail}$ if $M \models \neg C$, M contains no decision literals

Backjump

$M, L^d, N||F \Rightarrow M, L'||F$ if $\begin{cases} \text{there is some clause } C \vee L' \text{ s.t.:} \\ F \models C \vee L', M \models \neg C, \\ L' \text{ undefined in } M \\ L' \text{ or } \neg L' \text{ occurs in } F. \end{cases}$

Example

Assignment:	Clause set:	
$\emptyset$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (Decide)
$P_1{}^d$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (UnitProp
$P_1{}^d P_2$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (Decide)
$P_1{}^d P_2 P_3{}^d$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (UnitProp
$P_1{}^d P_2 P_3{}^d P_4$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (Decide)
$P_1{}^d P_2 P_3{}^d P_4 P_5{}^d$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (UnitProp
$P_1{}^d P_2 P_3{}^d P_4 P_5{}^d \neg P_6$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	$\Rightarrow$ (Backtrac
$P_1{}^d P_2 P_3{}^d P_4 \neg P_5$	$\|\|\neg P_1 \vee P_2, \neg P_3 \vee P_4, \neg P_5 \vee \neg P_6, P_6 \vee \neg P_5 \vee \neg P_2$	...

DPLL with learning

The DPLL system with learning consists of the four transition rules of the Basic DPLL system, plus the following two additional rules:

Learn

$M || F \Rightarrow M || F, C$ if all atoms of C occur in F and $F \models C$

Forget

$M || F, C \Rightarrow M || F$ if $F \models C$

In these two rules, the clause C is said to be learned and forgotten, respectively.

Further Information

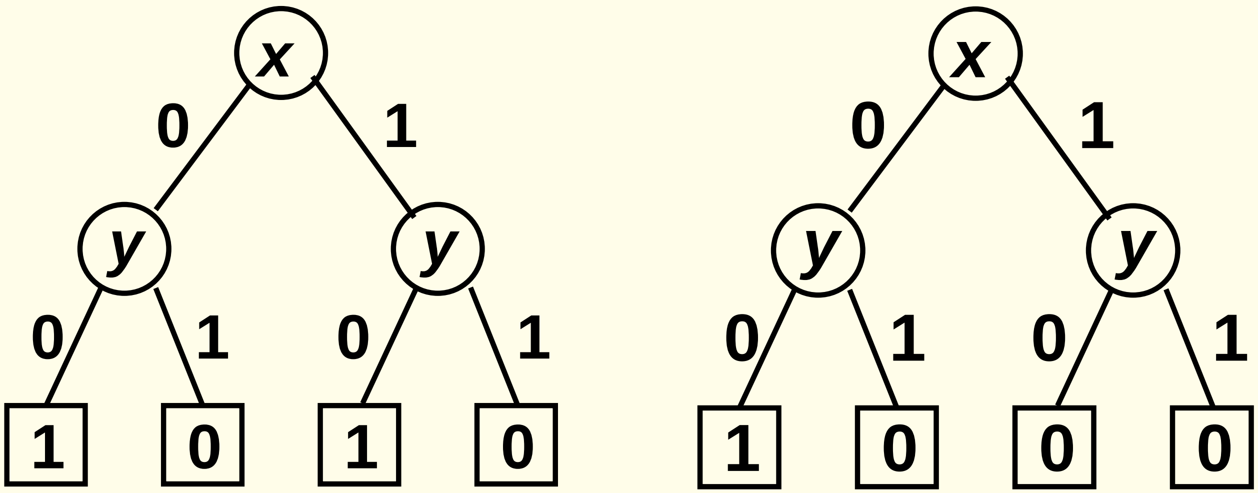
The ideas described so far heve been implemented in the SAT checker Chaff.

Further information:
Lintao Zhang and Sharad Malik:
The Quest for Efficient Boolean Satisfiability Solvers,
Proc. CADE-18, LNAI 2392, pp. 295–312, Springer, 2002.

Binary Decision Diagrams

Formulae	$\leftrightarrow$	Boolean functions
F (n Prop.Var)	$\mapsto$	$f_F : \{0,1\}^n \to \{0,1\}$

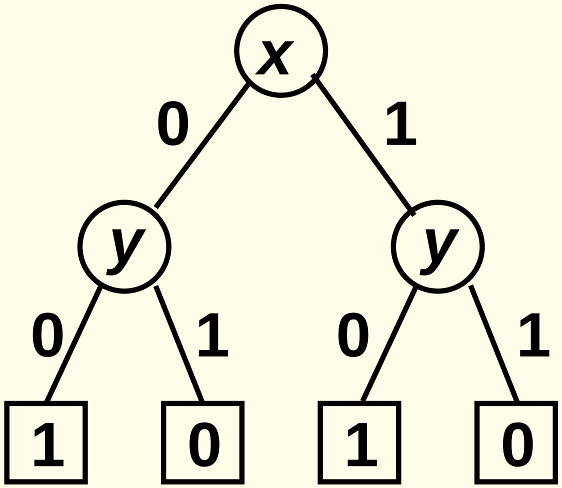
Binary decision trees:

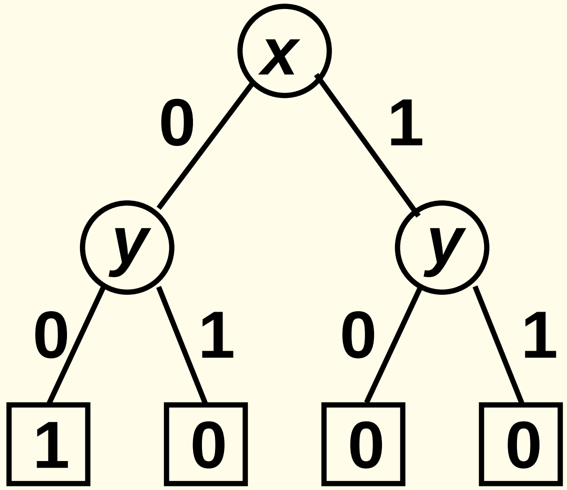


Binary Decision Diagrams

Formulae	$\leftrightarrow$	Boolean functions
F (n Prop.Var)	$\mapsto$	$f_F : \{0,1\}^n \to \{0,1\}$

Binary decision trees:





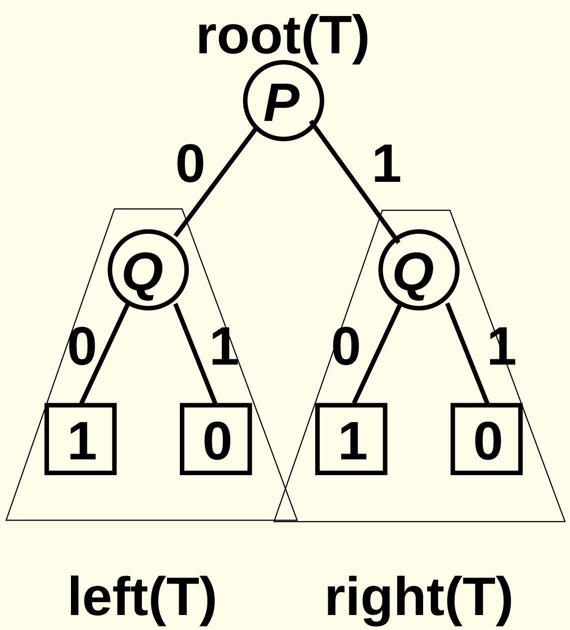
- exactly as inefficient as truth tables ($2^{n+1} - 1$ nodes if n prop.vars.)

- optimization possible: remove redundancies

Binary Decision Diagrams

With every function $f : \{0,1\}^n \to \{0,1\}$ we can associate a decision tree

With every decision tree T we can associate a Boolean function:



Sei $\mathcal{A} : \{P_1, \ldots, P_n\} \to \{0,1\}$, mit $\mathcal{A}(P_i) = a_i$

P marks the root of T:

if $\mathcal{A}(P) = 0$: $f_T(\overline{a}) := f_{\text{left}(T)}(\overline{a})$

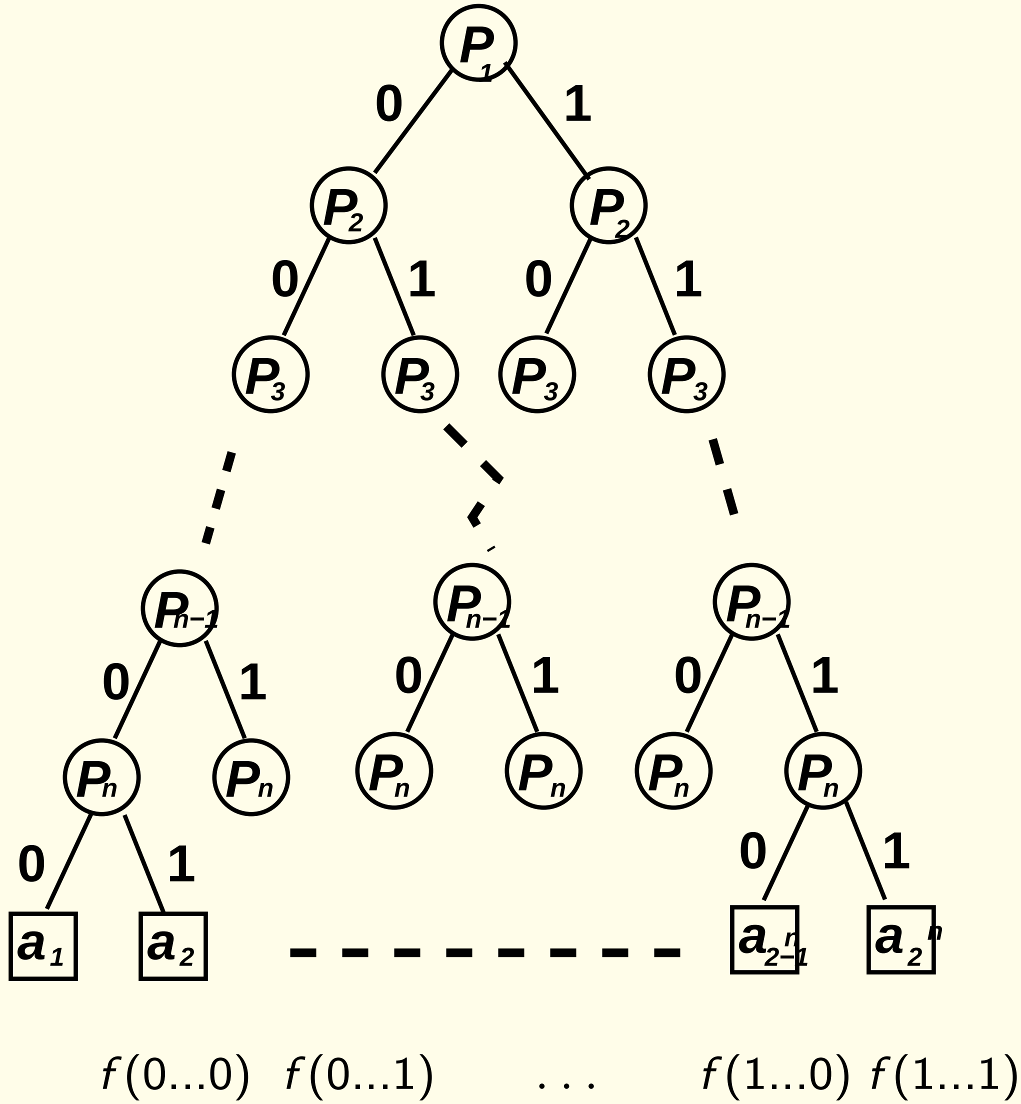
is $\mathcal{A}(P) = 1$: $f_T(\overline{a}) := f_{\text{right}(T)}(\overline{a})$

0 marks the root of T: $f_T(\overline{a}) := 0$

1 marks the root of T: $f_T(\overline{a}) := 1$

Binary Decision Trees

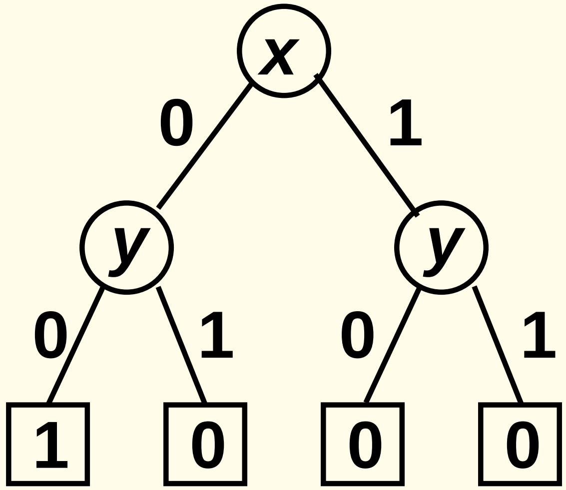
$f : \{0,1\}^n \to \{0,1\} \quad \mapsto$



Binary Decision Diagrams

Formulae	$\leftrightarrow$	Boolean functions
F (n Prop.Var)	$\mapsto$	$f_F : \{0,1\}^n \to \{0,1\}$

Binary decision trees:



- exactly as inefficient as truth tables ($2^{n+1} - 1$ nodes if n prop.vars.)

- optimization possible: remove redundancies

Binary Decision Diagrams

Optimization: remove redundancies

1. remove duplicate leaves

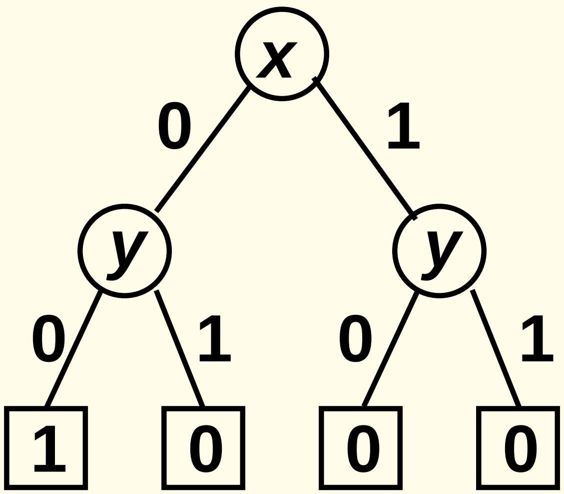
2. remove unnecessary tests

3. remove duplicate nodes

Binary Decision Diagrams

1. remove duplicate leaves

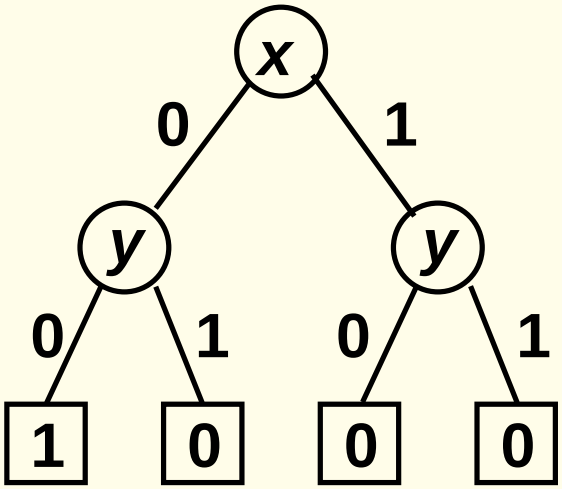
Only one copy of 0 and 1 necessary:

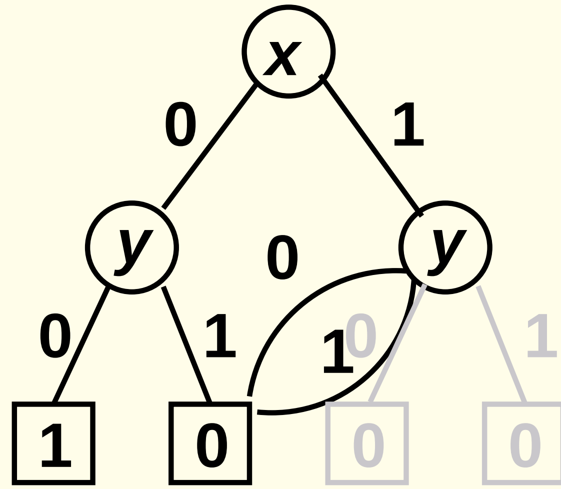


Binary Decision Diagrams

1. remove duplicate leaves

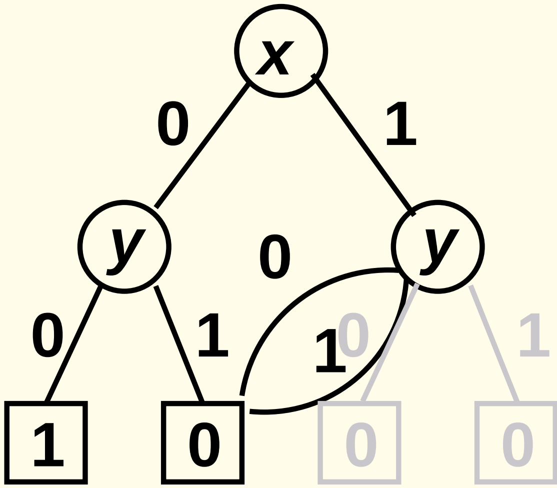
Only one copy of 0 and 1 necessary:





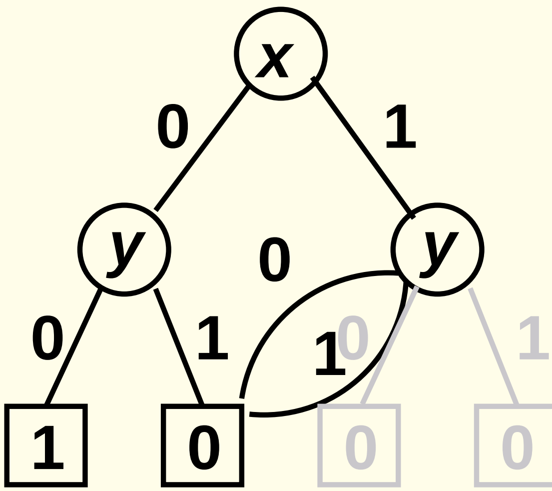
Binary Decision Diagrams

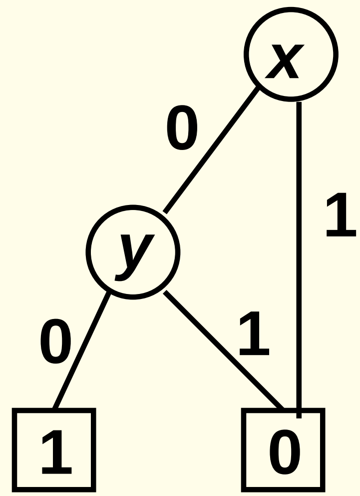
2. remove unnecessary tests



Binary Decision Diagrams

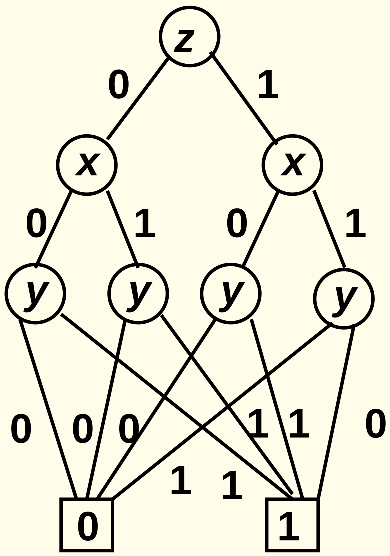
2. remove unnecessary tests





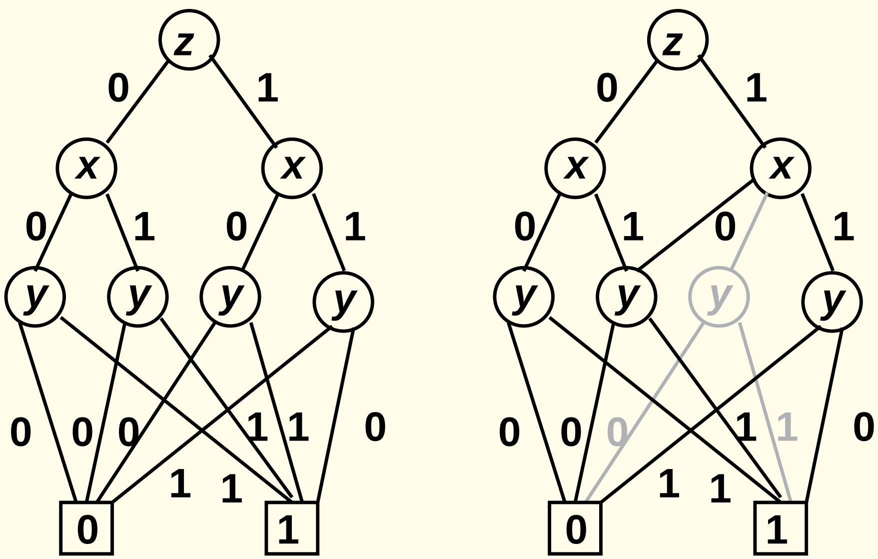
Binary Decision Diagrams

3. remove duplicate non-terminal nodes:



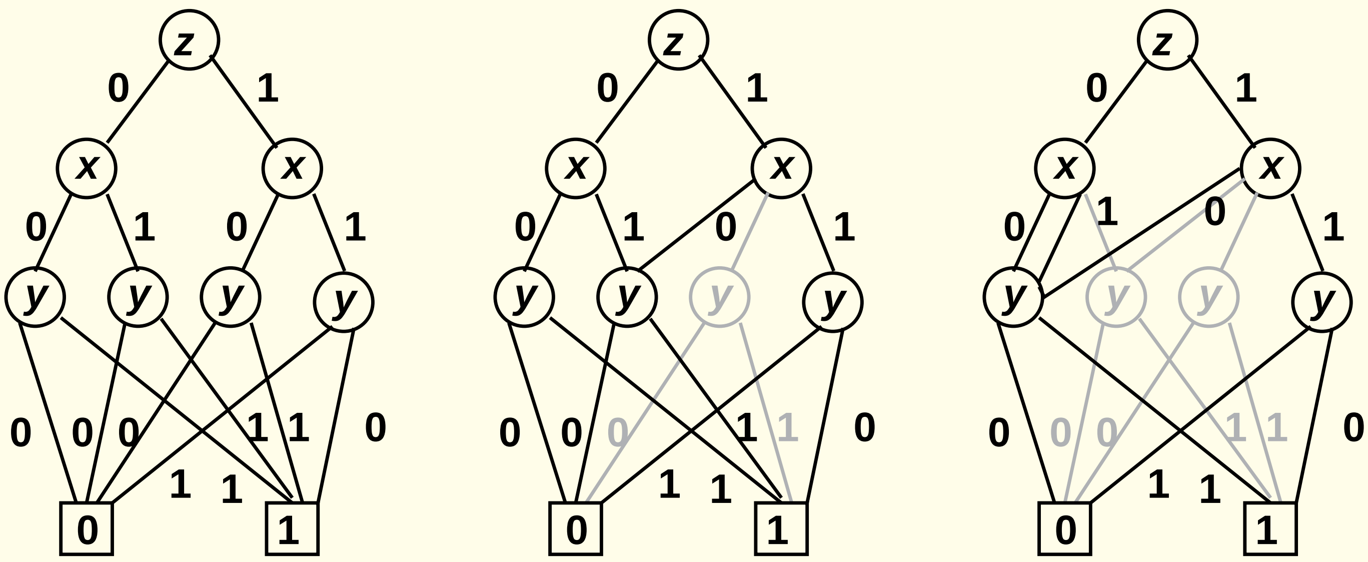
Binary Decision Diagrams

3. remove duplicate non-terminal nodes:



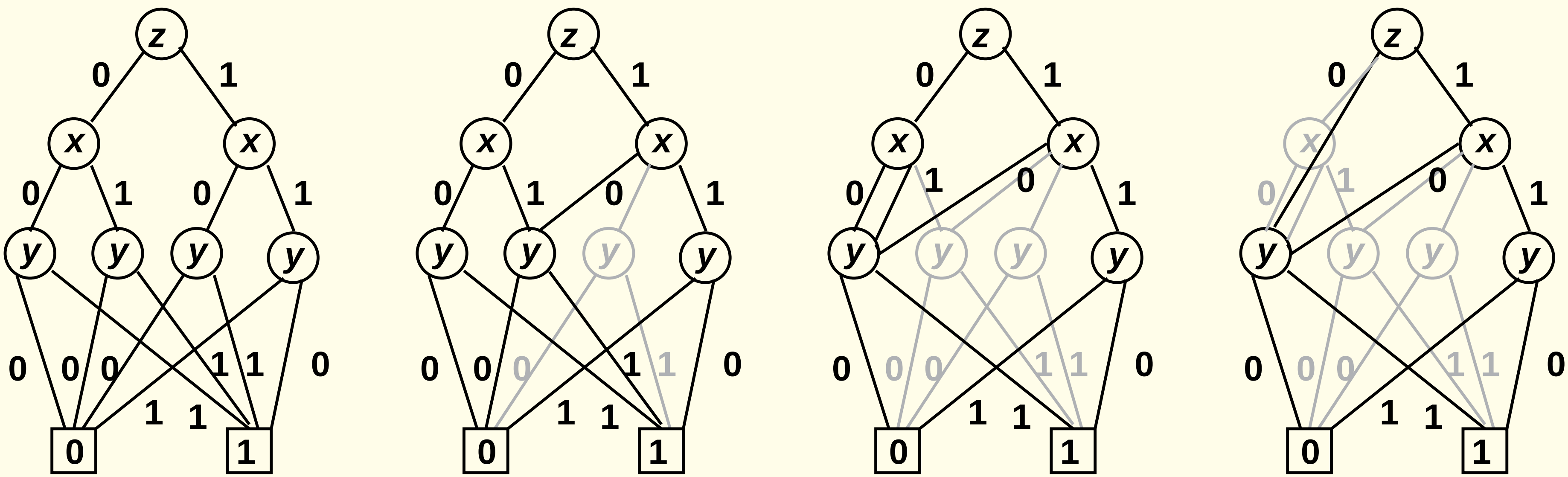
Binary Decision Diagrams

3. remove duplicate non-terminal nodes:



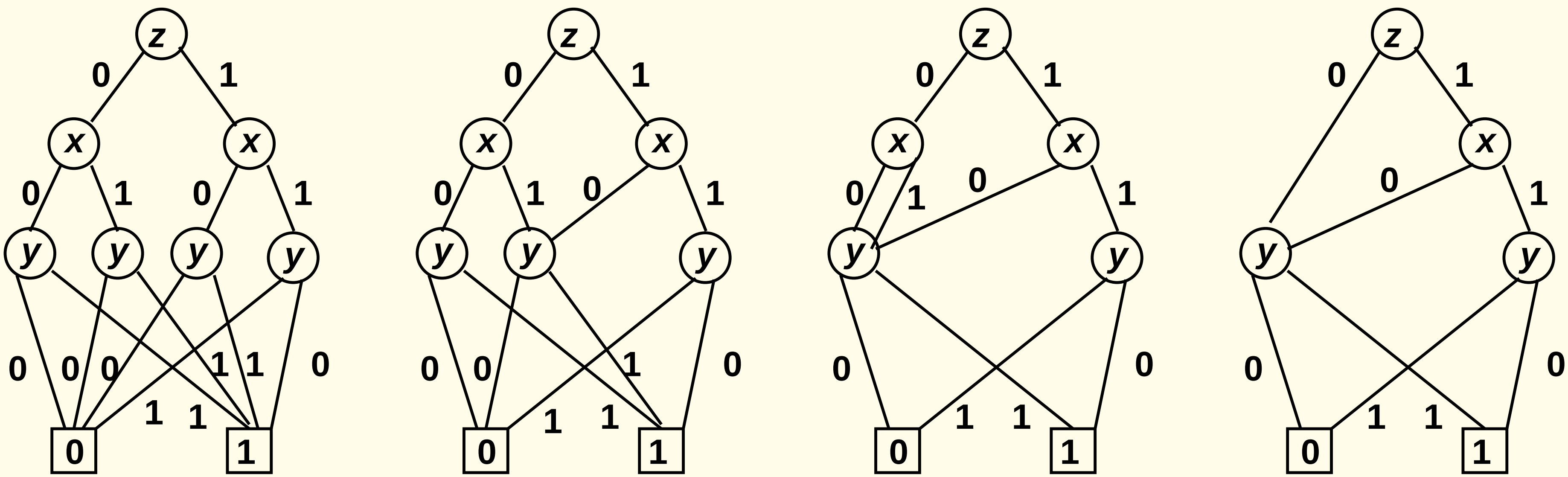
Binary Decision Diagrams

3. remove duplicate non-terminal nodes:



Binary Decision Diagrams

3. remove duplicate non-terminal nodes:



Operations with BDDs

$f \mapsto B_f$ (BDD associated with f)

$g \mapsto B_g$ (BDD associated with g)

BDD for $f \wedge g$: replace all 1-leaves in B_f with B_g

BDD for $f \vee g$: replace all 0-leaves in B_f with B_g

BDD for $\neg f$: replace all 1-leaves in B_f with 0-leaves and all 0-leaves with 1 leaves.

Binary Decision Diagrams

Binary decision diagram (BDD): finite directed acyclic graph with:

- a unique initial node
- terminal nodes marked with 0 or 1
- non-terminal nodes marked with propositional variables
- in each non-terminal node: two vertices (marked 0/1)

Reduced BDD: Optimizations 1-3 cannot be applied.

Binary Decision Diagrams

Binary decision diagram (BDD): finite directed acyclic graph with:

- a unique initial node
- terminal nodes marked with 0 or 1
- non-terminal nodes marked with propositional variables
- in each non-terminal node: two vertices (marked 0/1)

Reduced BDD: Optimizations 1-3 cannot be applied.

Problem: Variables may occur several times on a path.

Solution: Ordered BDDs.

Ordered BDDs

$[P_1, \ldots, P_n]$ ordered list of variables (without repetitions)

Let B be a BDD with variables $\{P_1, \ldots, P_n\}$

B has the order $[P_1, \ldots, P_n]$
if for every path $v_1 \to v_2 \to \cdots \to v_m$ in B,
if - $i < j$,
- v_i is marked with P_{k_i}
- v_j ist marked with P_{k_j}

then $k_i < k_j$.

A ordered BDD (Notation: OBDD) is a BDD which has an order, for a certain ordered list of variables.

Reduced OBDDs

Let $[P_1, \ldots, P_n]$ be an order on variables.

The reduced OBDD, which represents a given function f is unique.

Theorem:

Let B_1, B_2 be two reduced OBDDs with the same variable ordering.

If B_1 and B_2 represent the same function, then B_1 and B_2 are equal.

OBDDs have a canonical form, namely the reduced OBDD.

The role of the ordering on variables

Example $(P_1 \vee P_2) \wedge (P_3 \vee P_4) \wedge \cdots \wedge (P_{2n-1} \vee P_{2n})$

$[P_1, P_2, \ldots, P_{2n-1}, P_{2n}]$: OBDD with $2n + 2$ nodes

$[P_1, P_3, \ldots, P_{2n-1}, P_2, \ldots, P_{2n}]$: OBDD with 2^{n+1} nodes

Advantages of canonical representations

- Absence of redundant variables

  If the value of f does not depend on the i-argument (P_i)
  then no reduced OBDD contains the variable P_i

- Equivalence test

  $F_i \mapsto f_i \mapsto B_i$ (OBDDs with compatible variable ordering), $i = 1, 2$

  Reduce B_i, $i = 1, 2$. $F_1 \equiv F_2$ iff. B_1 and B_2 identical.

Advantages of canonical representations

- Validity test

  $F \mapsto f \mapsto B$ (OBDD)

  F valid iff its reduced OBDD is $B_1 := \boxed{1}$

- Entailment test

  $F \models G$ iff the reduced OBDD for $F \wedge \neg G$ is $B_0 := \boxed{0}$

- Satisfiability test

  F satisfiable iff its reduced OBDD is not B_0.

Operations with OBDDs

- Reduce

  Apply reduction steps 1–3

- Apply

  Boolean operations

- Restrict

  Compute OBDD for $F[0/P_i]$ and $F[1/P_i]$

- Exists

  Compute OBDD for $\exists P_i F(P_1, \ldots, P_n)$

Operations with OBDDs

- Reduce

  Apply reduction steps 1–3

- Apply

  Boolean operations

- Restrict

  Compute OBDD for $F[0/P_i]$ and $F[1/P_i]$

- Exists

  Compute OBDD for $\exists P_i F(P_1, \ldots, P_n)$

Reduce

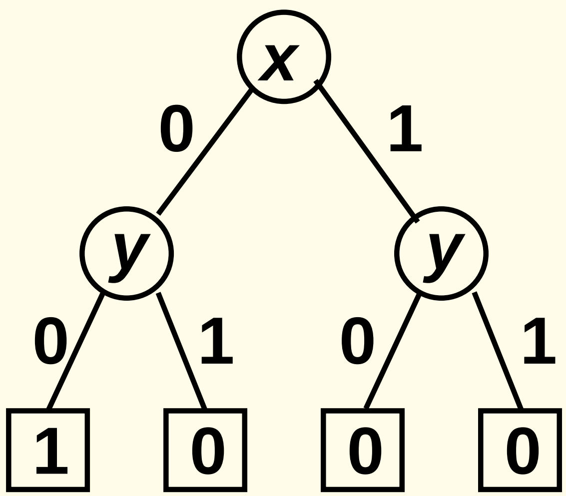
remove redundancies

1. remove duplicate leaves
2. remove unnecessary tests
3. remove duplicate nodes

Reduce

1. remove duplicate leaves

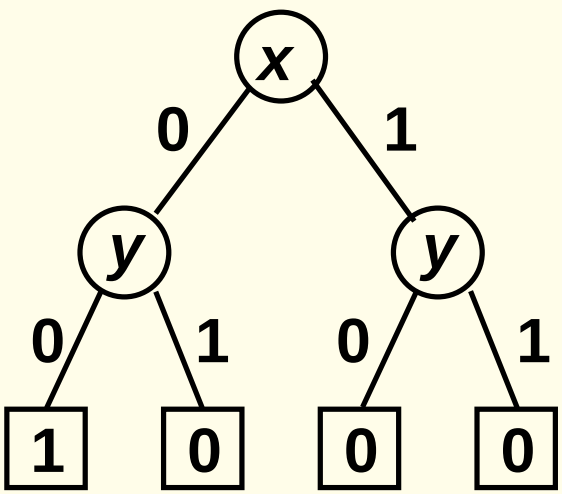
Only one copy of 0 and 1 necessary:

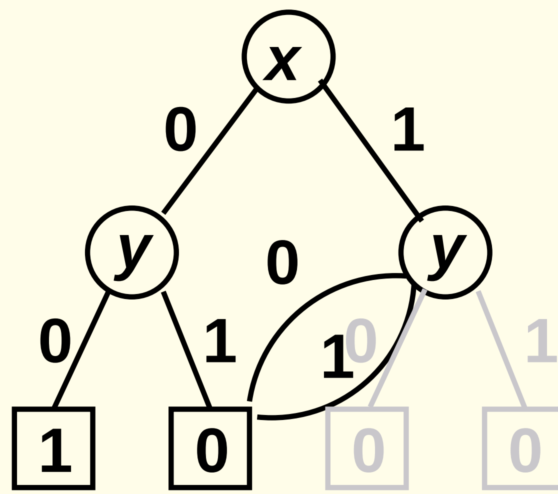


Reduce

1. remove duplicate leaves

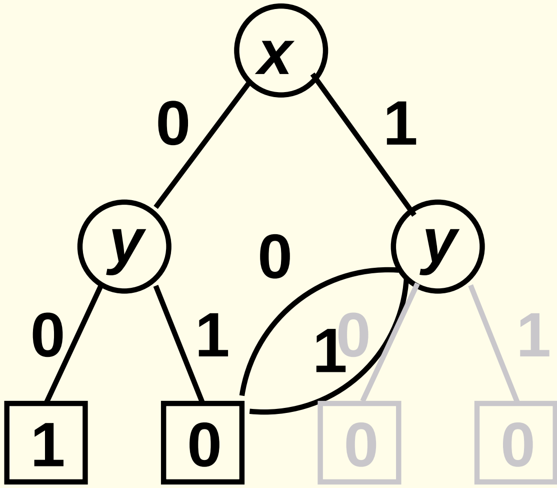
Only one copy of 0 and 1 necessary:





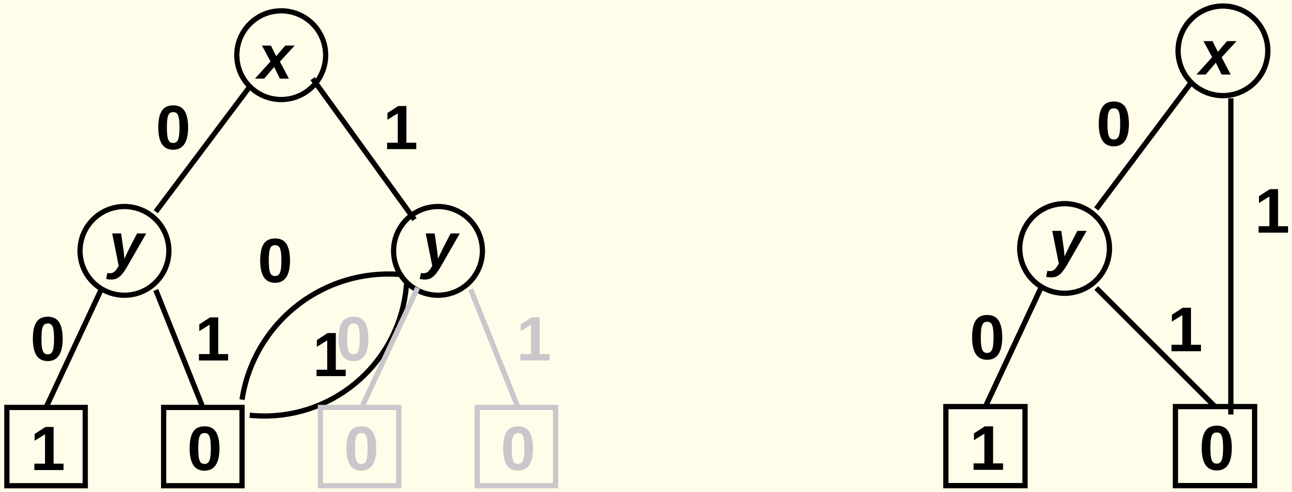
Reduce

2. remove unnecessary tests



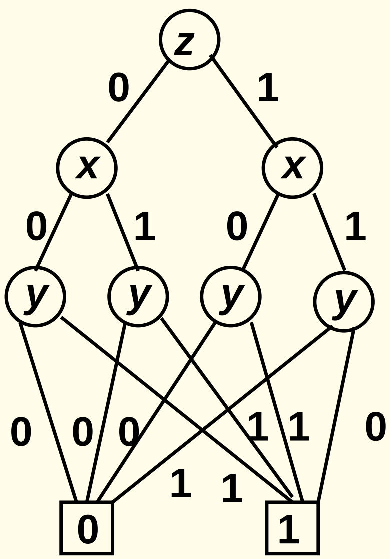
Reduce

2. remove unnecessary tests



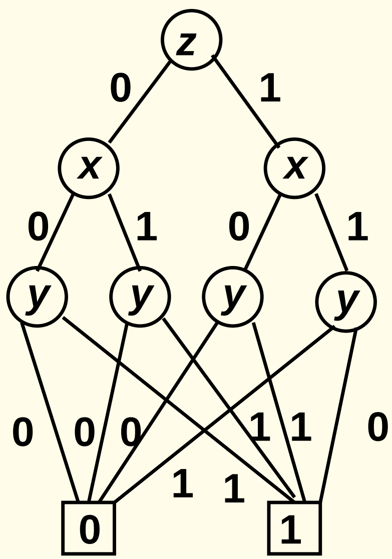
Reduce

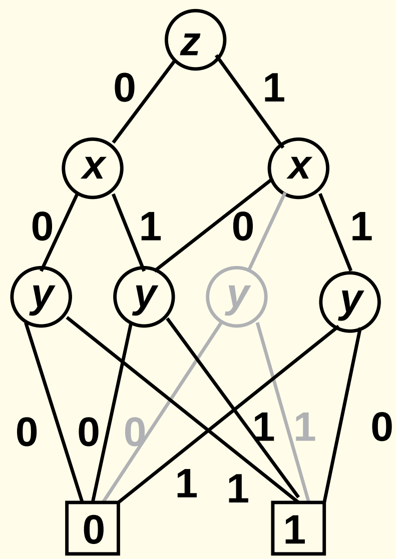
3. remove duplicate non-terminal nodes:



Reduce

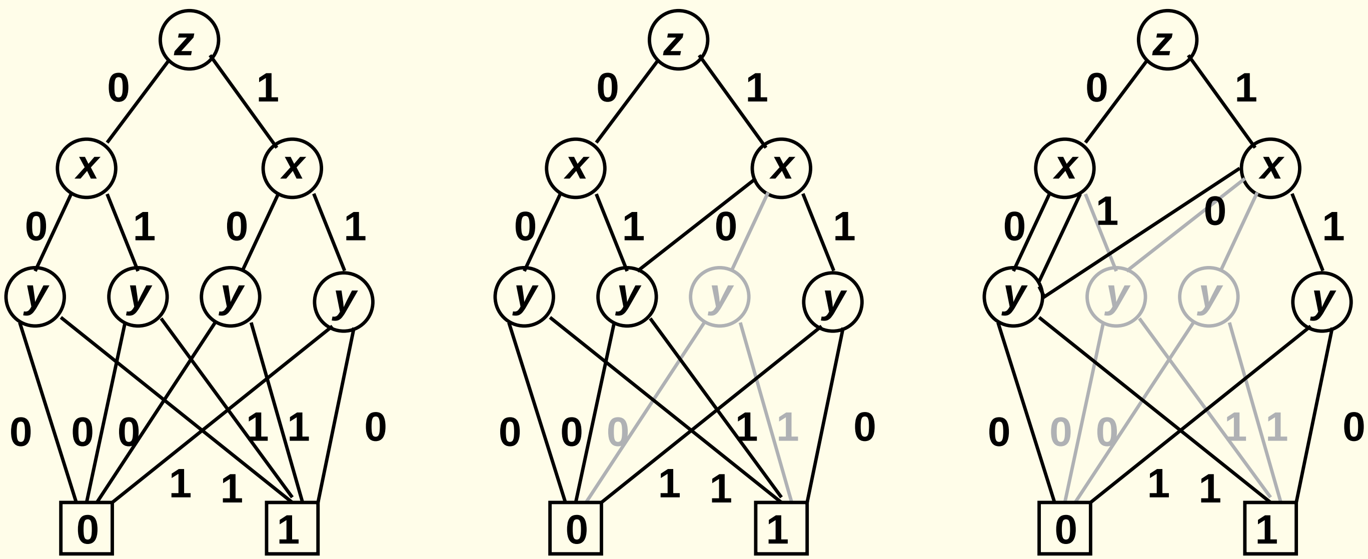
3. remove duplicate non-terminal nodes:





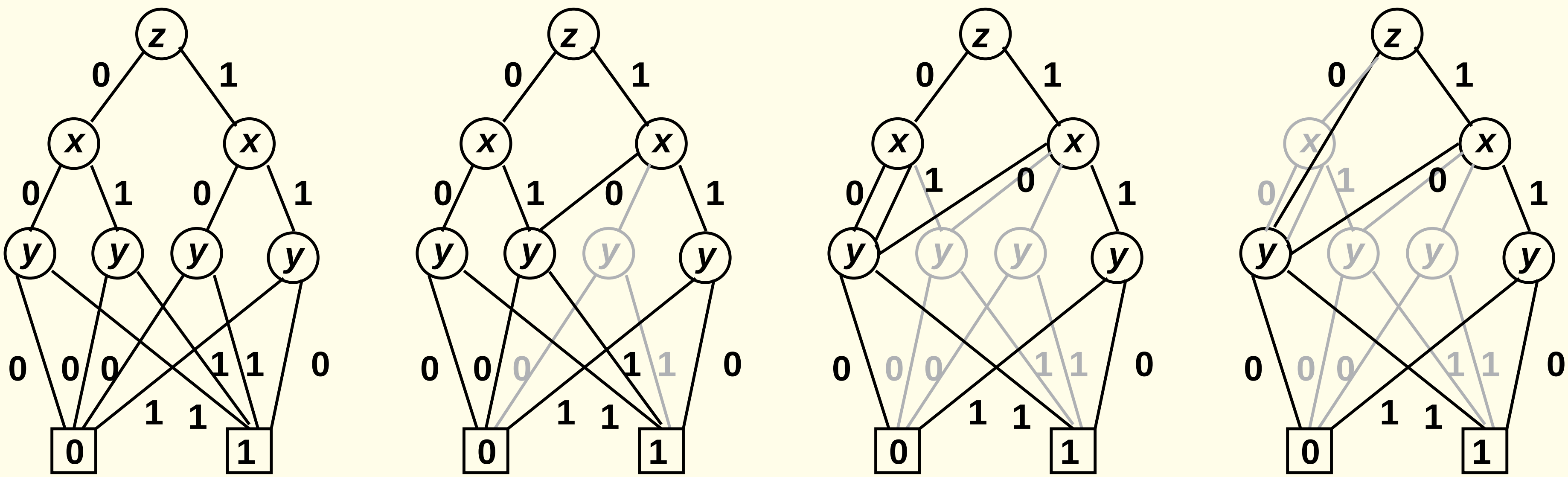
Reduce

3. remove duplicate non-terminal nodes:



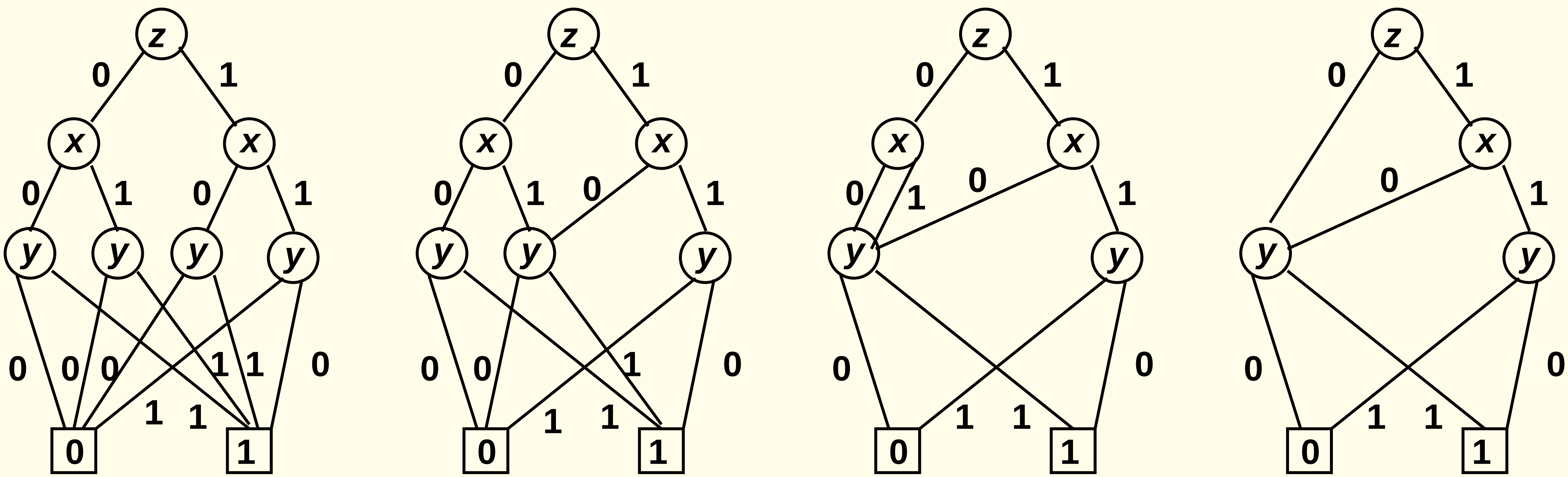
Reduce

3. remove duplicate non-terminal nodes:



Reduce

3. remove duplicate non-terminal nodes:



Reduce

The algorithm reduce traverses an OBDD B layer by layer in a bottom-up fashion, beginning with the terminal nodes.

In traversing B, it assigns an integer label $id(n)$ to each node n of B, in such a way that the subOBDDs with root nodes n and m denote the same boolean function iff, $id(n) = id(m)$.

Reduce

Terminal nodes:

Since reduce starts with the layer of terminal nodes, it assigns the first label (say #0) to the first 0-node it encounters. All other terminal 0-nodes denote the same function as the first 0-node and therefore get the same label (compare with reduction 1).

Similarly, the 1-nodes all get the next label, say #1.

Reduce

Non-terminal nodes

Now let us inductively assume that reduce has already assigned integer labels to all nodes of a layer $> i$ (i.e. all terminal nodes and P_j-nodes with $j > i$).

We describe how nodes of layer i (i.e. P_i-nodes) are being handled.

$n \mapsto lo(n)$ node reached on branch labelled with 0
$\quad hi(n)$ node reached on branch labelled with 1

Given an P_i-node n, there are three ways in which it may get its label:

- If $id(lo(n)) = id(hi(n))$, we set $id(n)$ to be that label (reduction 2)
- If there is another node m s.t. n and m have same variable P_i, and $id(lo(n)) = id(lo(m))$ and $id(hi(n)) = id(hi(m))$, then we set $id(n) := id(m)$ (reduction 3)
- Otherwise, we set $id(n)$ to the next unused integer label.