

# Efficient techniques for model checking: Symbolic techniques (ROBDD)

dr. István Majzik

dr. András Pataricza

dr. Tamás Bartha

BME Department of Measurement and Information System

# Where are we?

- Lower-level formalisms (KS, LTS, KTS)
- Higher-level formalisms

Temporal logics:  
PLTL, CTL, CTL\*

Formal model

Formalized requirements

Basic algorithms

Model checker

OK

Counterexample

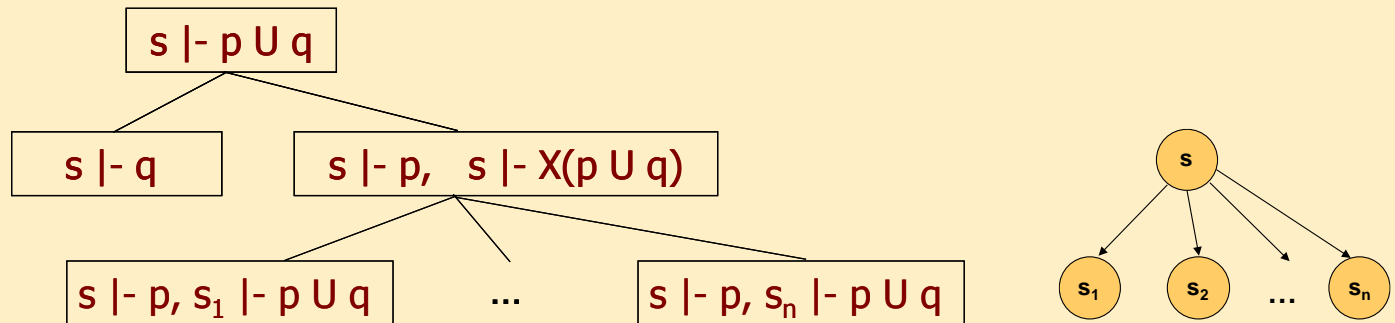
i

n

# Recap: Known techniques for model checking

- PLTL model checking:

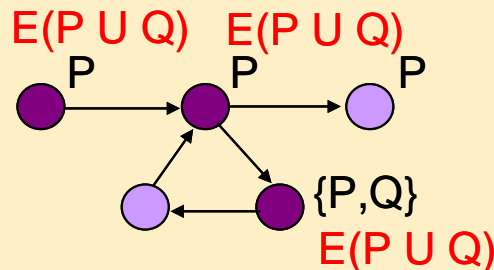
- Tableau method: Decomposition based on the model



- Automata-based approach (auxiliary)

- CTL model checking:

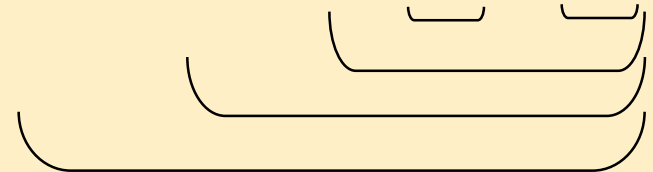
- Semantics-based approach: Iterative labeling of states



# Recap: CTL model checking with state labeling

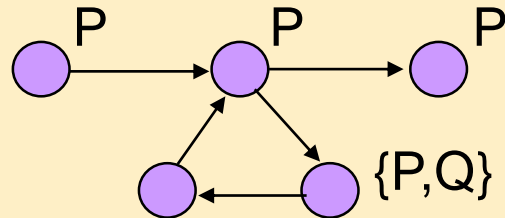
- Label states with subformulas based on  $\text{Sat}(\cdot)$  computation:

$AF ( P \wedge E ( Q \cup R ) )$



- State labeling: Where does a formula hold?
  - Initially: KS labeled with atomic propositions
  - Iteratively: Labeling with more complex formulas
    - If a state is labeled with  $p$  and  $q$ , then we can label with  $\neg p$ ,  $p \wedge q$ ,  $EX p$ ,  $AX p$ ,  $E(p \cup q)$ ,  $A(p \cup q)$
    - Incremental labeling algorithm based on the semantics of operators

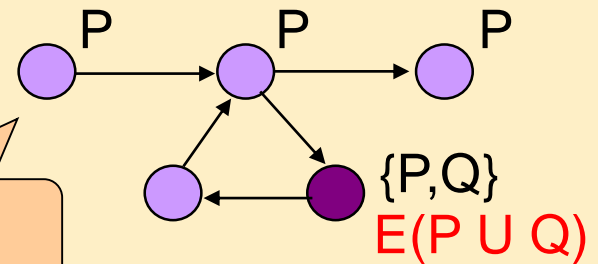
# Recap: Iteration of the E(P U Q) labeling



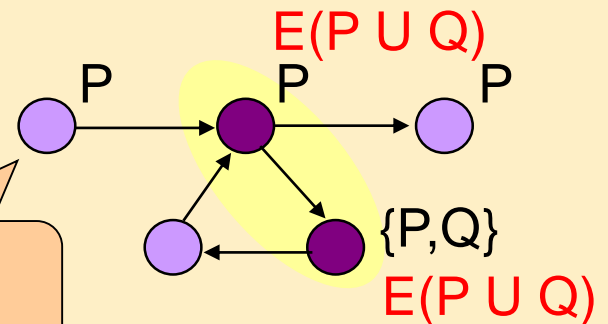
**Kripke structure  
with initial labeling**

- Exploiting:  
 $E(P \cup Q) =$   
 $Q \vee (P \wedge EX E(P \cup Q))$
- Iteration continues while set of states grows (until a fixed point is reached)

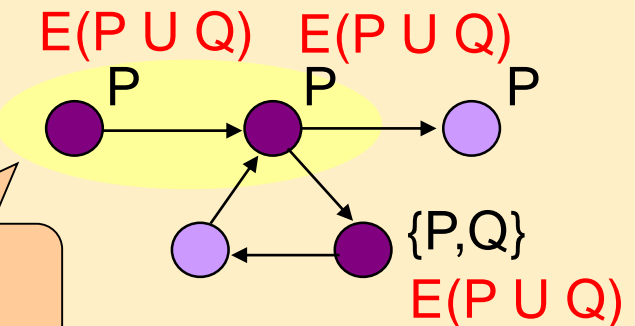
**First step: Q**



**Second  
step: P ∧ EX**

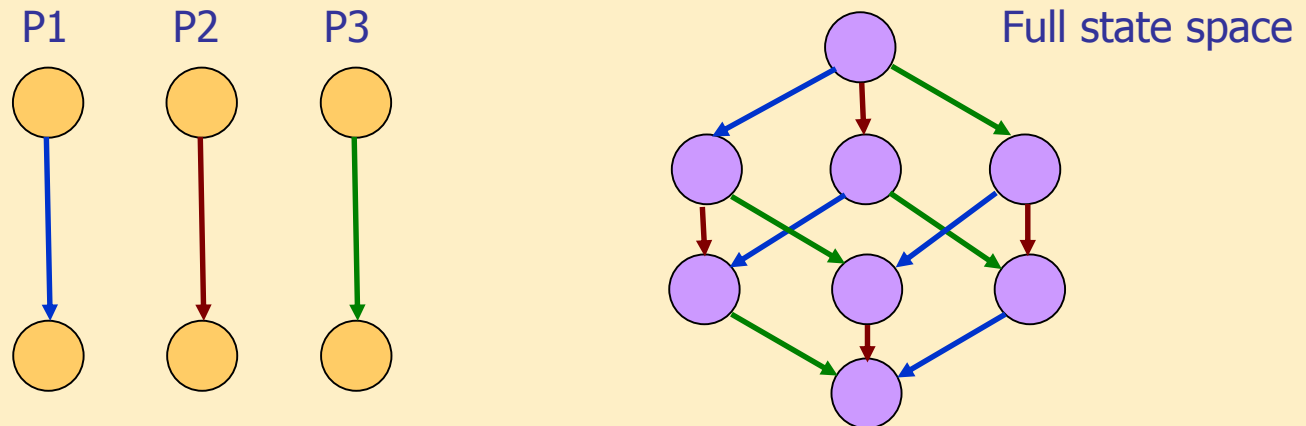


**Third step:  
P ∧ EX**



# Problems

- The state space to traverse can be huge
  - Concurrent systems exhibit a large state space: Combinatorial explosion in the number of possible interleavings of independent sequences



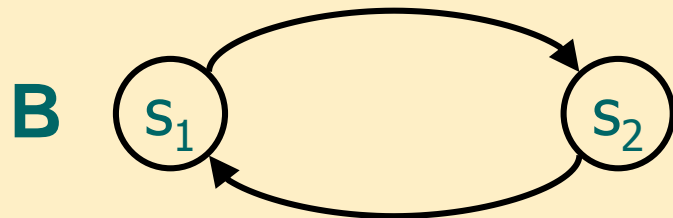
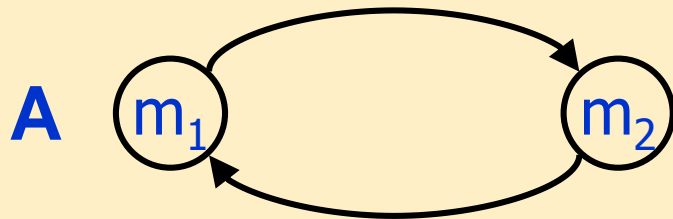
- How can we analyze large state spaces?
  - Promise: CTL model checking:  $10^{20}$ , sometimes even  $10^{100}$  states
  - What kind of technique can deliver this promise?

# Outlook: Concurrent behavior of two automata

Direct product of automata,  
interleaving, synchronization

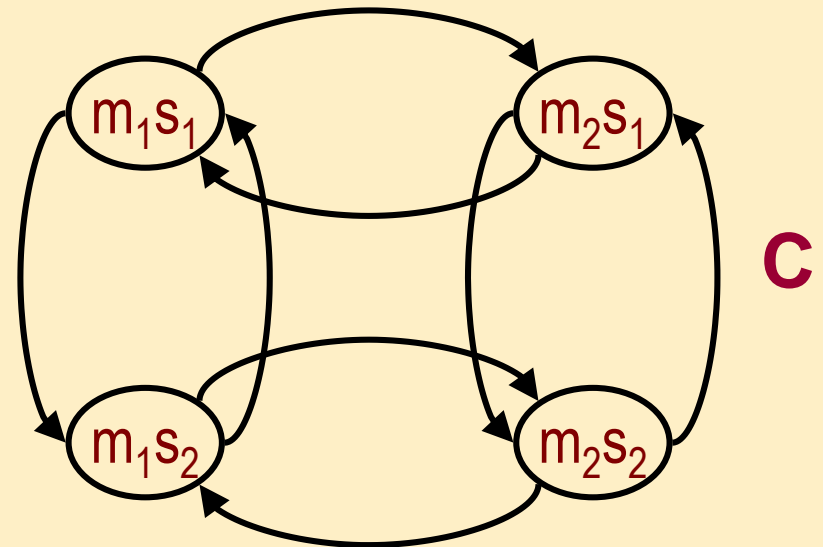
# Example: Operation of asynchronous automata

- System composed of two (independent) automata



- States of the automata:  
 $A = \{m_1, m_2\}$ ,  $B = \{s_1, s_2\}$

- (Direct) product automaton: state space of the system

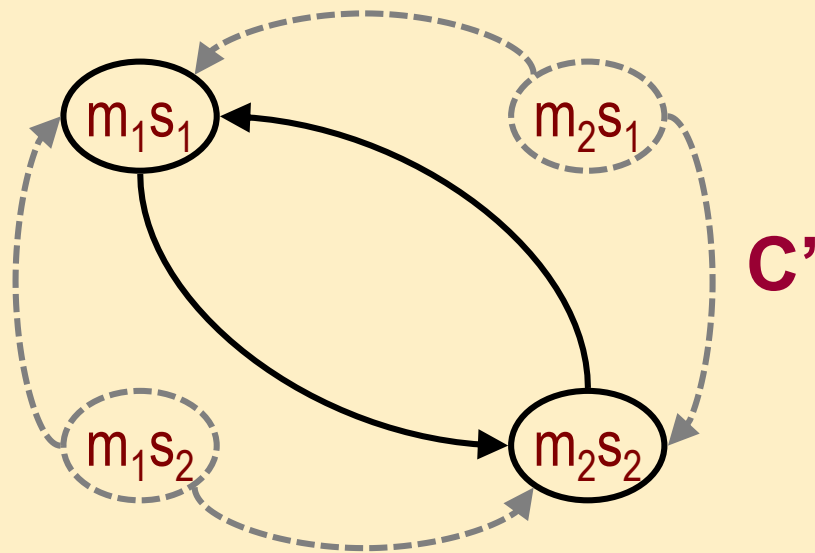


- Set of states:  
 $C = A \times B$   
 $C = \{m_1s_1, m_1s_2, m_2s_1, m_2s_2\}$



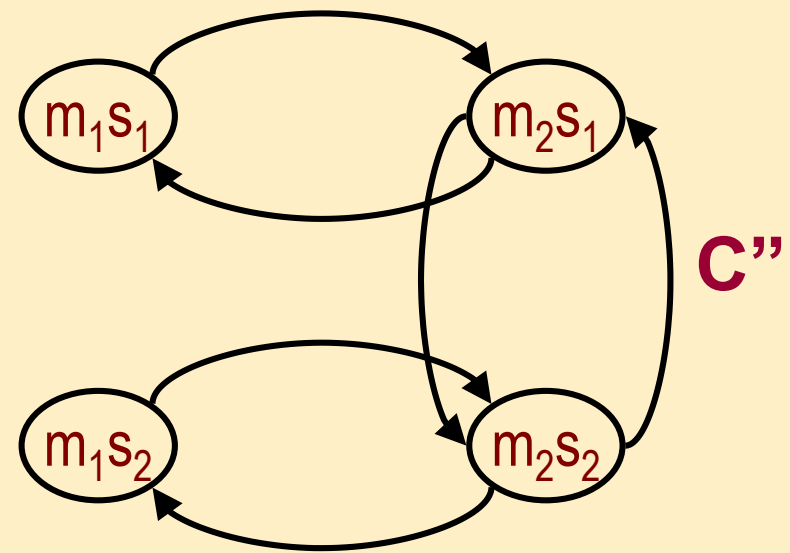
# Synchronizations and guards simplify the model

- Synchronization: taking the transitions at the same time



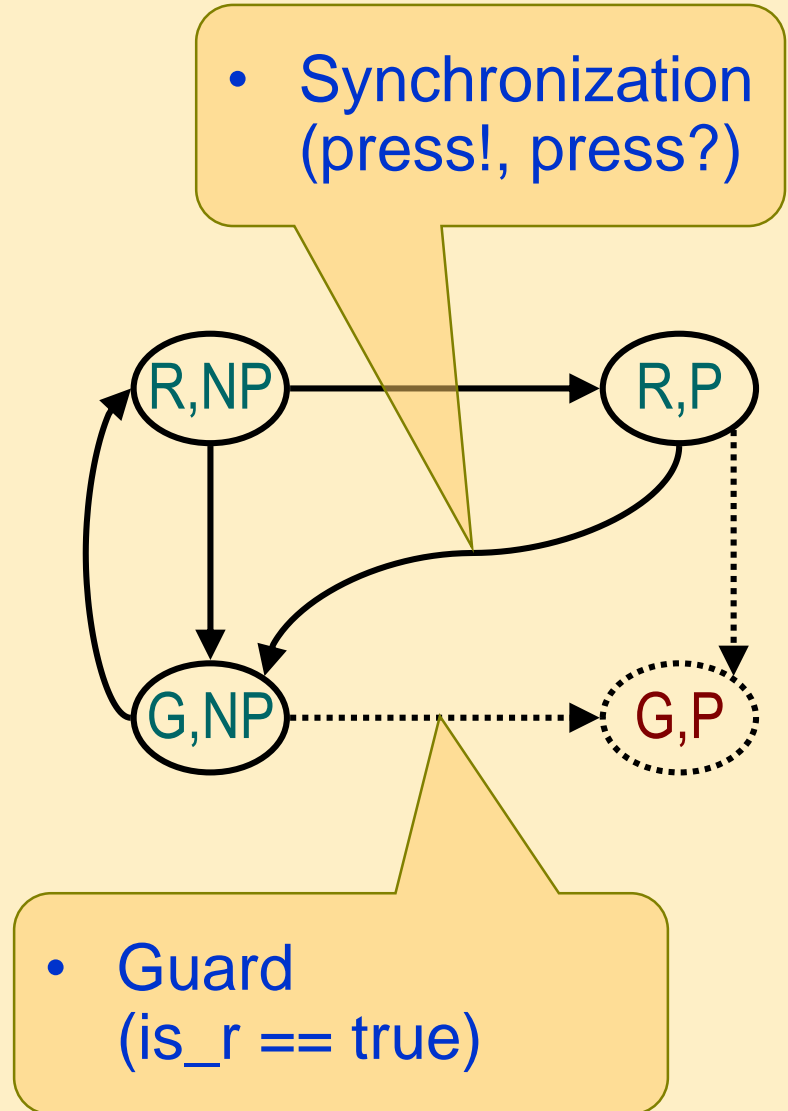
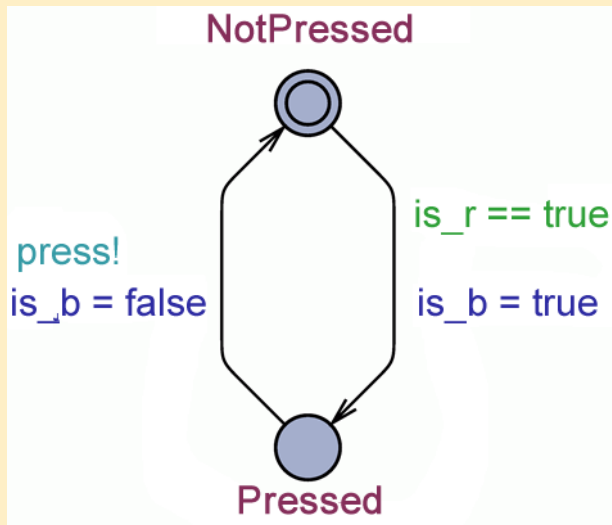
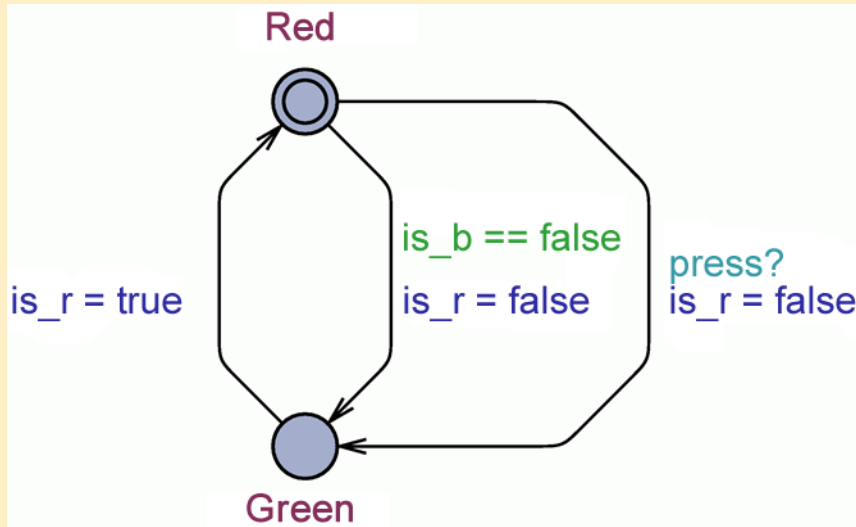
- E.g. "A and B takes the transition at the same time if their state index is the same"

- Guards: disable certain transitions

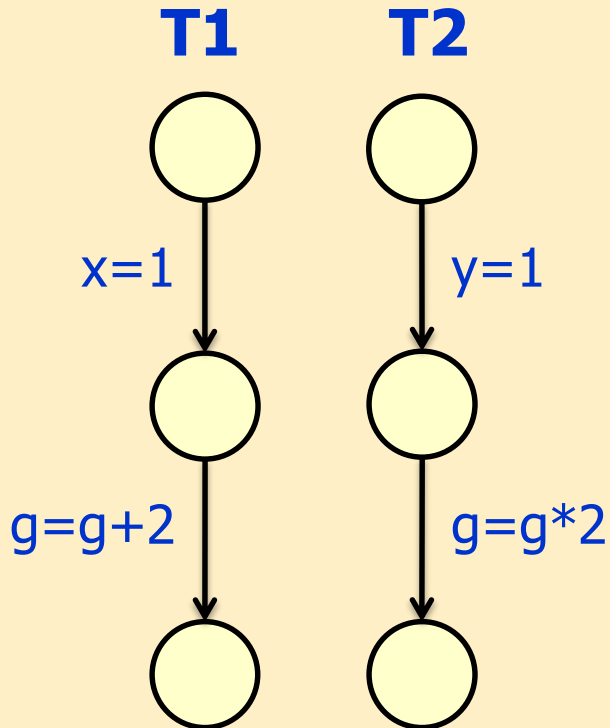


- E.g. "B can only take the transition if A is in state  $m_2$ "

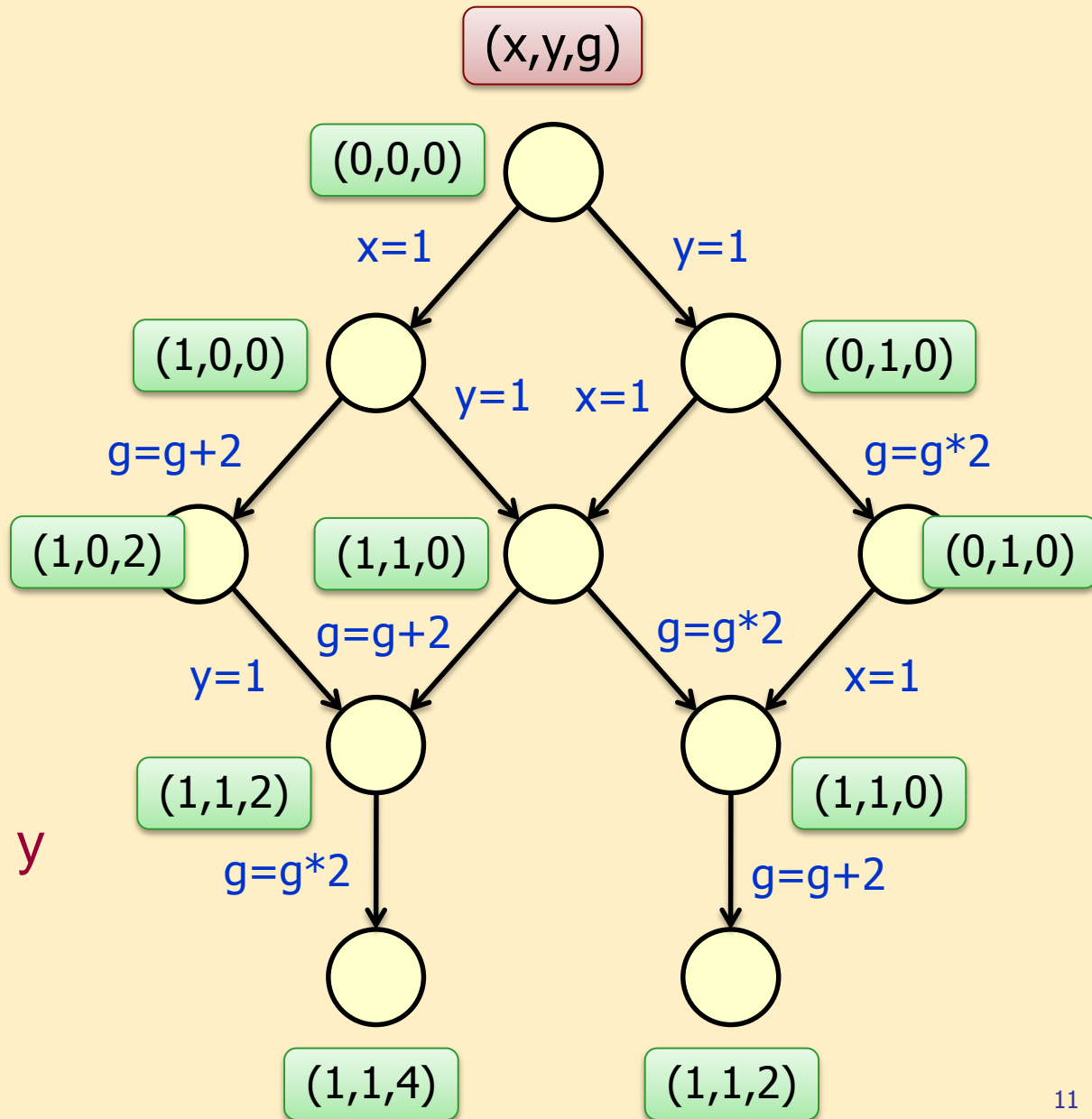
# Example: Pedestrian light with button



# Example: Alternative paths



Local variables:  $x$  and  $y$   
Global variable:  $g$



# Example for large state space: Dining philosophers

- Concurrent system
  - May have deadlock
  - May have livelock
- State space grows fast

#Philosophers	#States
16	$4,7 \cdot 10^{10}$
28	$4,8 \cdot 10^{18}$
...	...
200	$> 10^{40}$
1000	$> 10^{200}$
...	...

$$2^{64} = 1,8 \cdot 10^{19}$$



With smart (but not task-specific)  
state space representation:  
 $\sim 100\ 000$  philosophers, i.e.  
 $10^{62900}$  states can be checked!

# Overview of the techniques to learn

- CTL model checking: Symbolic technique

Semantics-based technique	Symbolic technique
Sets of labeled states	Characteristic functions (Boolean functions): ROBDD representation
Operations on sets of states	Efficient operations on ROBDDs

- Model checking of invariants: Bounded model checking
  - Searching satisfying valuations for Boolean formulas with SAT techniques
  - Model checking to a given depth:  
Searching for counterexamples with bounded length
    - A detected counterexample is always valid
    - No counterexamples does not imply correctness

# Symbolic model checking

# Recap: Iteration using set operations

- We expand the labeling using operations on sets
  - Initial set: states already labeled by subformulas
  - Expanding the labeling:
    - $E(p \cup q)$ : "At least one successor is labeled ..."
    - $A(p \cup q)$ : "All successors are labeled ..."
  - This way we can label preceding states

- How can we define the set of preceding states?

- Based on set of already labeled states  $Z$ :

$$\text{pre}_E(Z) = \{s \in S \mid \text{there exists } s' \text{ such that } (s, s') \in R \text{ and } s' \in Z\}$$

$$\text{pre}_A(Z) = \{s \in S \mid \text{for all } s' \text{ such that } (s, s') \in R \text{ we have } s' \in Z\}$$

At least one  
successor is  
labeled

All successors  
are labeled

- Example:  $E(P \cup Q)$ :

- Initial set:  $Z_0 = \{s \mid Q \in L(s)\}$
- Expansion:  $Z_{i+1} = Z_i \cup (\text{pre}_E(Z_i) \cap \{s \mid P \in L(s)\})$

Labeled so far

Predecessors of  
already labeled states

labeled P

- End of the iteration: if  $Z_{i+1} = Z_i$  (fixedpoint)

# Main idea

- Representation of and operations on sets of states:  
With **Boolean functions** instead of enumeration
  - Encoding a **state** with a **bit-vectors**
    - To encode a set of states  $S$  we need at least  $n = \lceil \log_2 |S| \rceil$  bits, so choose  $n$  such that  $2^n \geq |S|$
  - Encoding a **set of states** with an  $n$ -ary **Boolean function**:  
**Characteristic function**
    - The function should be true for a bit-vector *iff* the state encoded by the bit-vector is in the given set of states
    - Characteristic function:  $C: \{0,1\}^n \rightarrow \{0,1\}$
  - We will perform operations on characteristic functions instead of sets



# Characteristic functions

- For a state  $s$ :  $C_s(x_1, x_2, \dots, x_n)$

Let the encoding of  $s$  be the bit-vector  $(u_1, u_2, \dots, u_n)$ , where  $u_i \in \{0, 1\}$

Goal:  $C_s(x_1, x_2, \dots, x_n)$  should return be true only for  $(u_1, u_2, \dots, u_n)$

Construction of  $C_s(x_1, x_2, \dots, x_n)$ : with operator  $\wedge$

- $x_i$  is an operand if  $u_i=1$
- $\neg x_i$  is an operand if  $u_i=0$

Example: for state  $s$  with encoding  $(0,1)$ :  $C_s(x_1, x_2) = \neg x_1 \wedge x_2$

- For a set of state  $Y \subseteq S$ :  $C_Y(x_1, x_2, \dots, x_n)$

Goal:  $C_Y(x_1, x_2, \dots, x_n)$  should be true for parameters  $(u_1, u_2, \dots, u_n)$   
iff  $(u_1, u_2, \dots, u_n) \in Y$

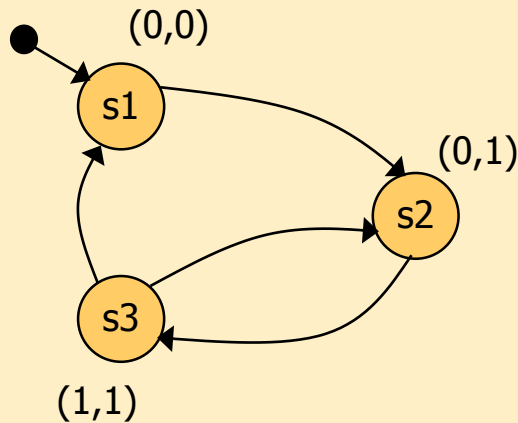
Construction of  $C_Y(x_1, x_2, \dots, x_n)$ :

$$C_Y(x_1, x_2, \dots, x_n) = \bigvee_{s \in Y} C_s(x_1, x_2, \dots, x_n)$$

- For sets of states in general:

$$C_{Y \cup W} = C_Y \vee C_W, \quad C_{Y \cap W} = C_Y \wedge C_W$$

# Example: Characteristic function of states



Variables:  $x, y$

Characteristic functions of states:

State s1:

$$C_{s1}(x,y) = (\neg x \wedge \neg y)$$

State s2:

$$C_{s2}(x,y) = (\neg x \wedge y)$$

State s3:

$$C_{s3}(x,y) = (x \wedge y)$$

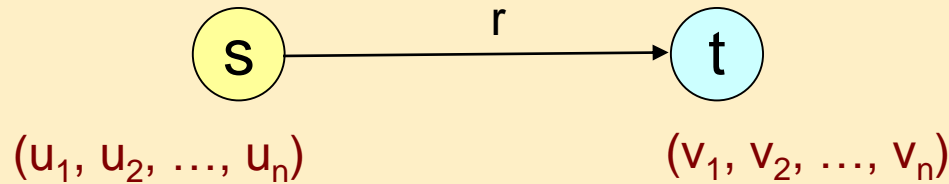
Characteristic function for a set of states:

Set of states  $\{s1,s2\}$ :

$$C_{\{s1,s2\}} = C_{s1} \vee C_{s2} = (\neg x \wedge \neg y) \vee (\neg x \wedge y)$$

# Characteristic functions (cont'd)

- For state transitions:  $C_r$



$r=(s,t)$  transition, where  $s=(u_1, u_2, \dots, u_n)$  and  $t=(v_1, v_2, \dots, v_n)$

- Characteristic function in the form  $C_r(x_1, x_2, \dots, x_n, x'_1, x'_2, \dots, x'_n)$

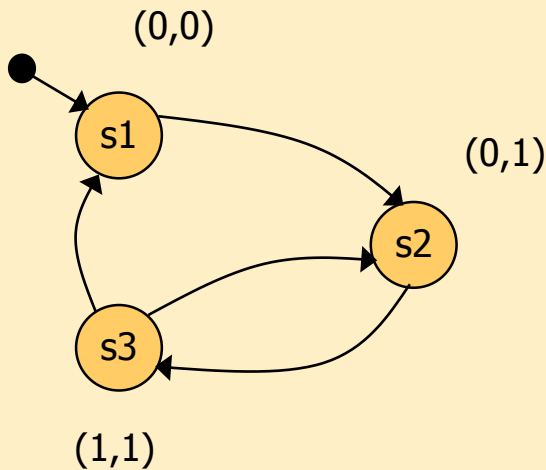
- „Primed“ variables denote the target state

Goal:  $C_r$  should be true *iff*  $x_i=u_i$  and  $x'_i=v_i$

Construction of  $C_r$ :

$$C_r = C_s(x_1, x_2, \dots, x_n) \wedge C_t(x'_1, x'_2, \dots, x'_n)$$

# Example: Characteristic functions of transitions



State s1:

$$C_{s1}(x,y) = (\neg x \wedge \neg y)$$

State s2:

$$C_{s2}(x,y) = (\neg x \wedge y)$$

Transition  $(s1,s2) \in R$ :

$$C_{(s1,s2)} = (\neg x \wedge \neg y) \wedge (\neg x' \wedge y')$$

Transition relation:

$$\begin{aligned} R(x,y,x',y') = & (\neg x \wedge \neg y \wedge \neg x' \wedge y') \vee \\ & \vee (\neg x \wedge y \wedge x' \wedge y') \vee \\ & \vee (x \wedge y \wedge \neg x' \wedge y') \vee \\ & \vee (x \wedge y \wedge \neg x' \wedge \neg y') \end{aligned}$$

# Characteristic functions (cont'd)

- Construction of  $\text{pre}_E(Z)$ :  $\text{pre}_E(Z) = \{s \mid \exists t: (s,t) \in R \text{ and } t \in Z\}$

Representation of  $Z$ :  $C_Z$

Representation of  $R$ :  $C_R = \bigvee_{r \in R} C_r$

$\text{pre}_E(Z)$ : find predecessor states for states of  $Z$

$$C_{\text{pre}_E(Z)} = \exists_{x'_1, x'_2, \dots, x'_n} C_R \wedge C'_Z$$

where  $\exists_x C = C[1/x] \vee C[0/x]$  („existential abstraction“)

- Model checking with set operations:  
reduced to operations with Boolean functions!
  - Union of sets: Disjunction of functions ( $\vee$ )
  - Intersection of sets: Conjunction of functions ( $\wedge$ )
  - Construction of  $\text{pre}_E(Z)$ : Complex operation (existential abstraction)

# Representation of Boolean functions

Canonical form: ROBDD

Reduced, Ordered Binary Decision Diagram

“Phases” (overview):

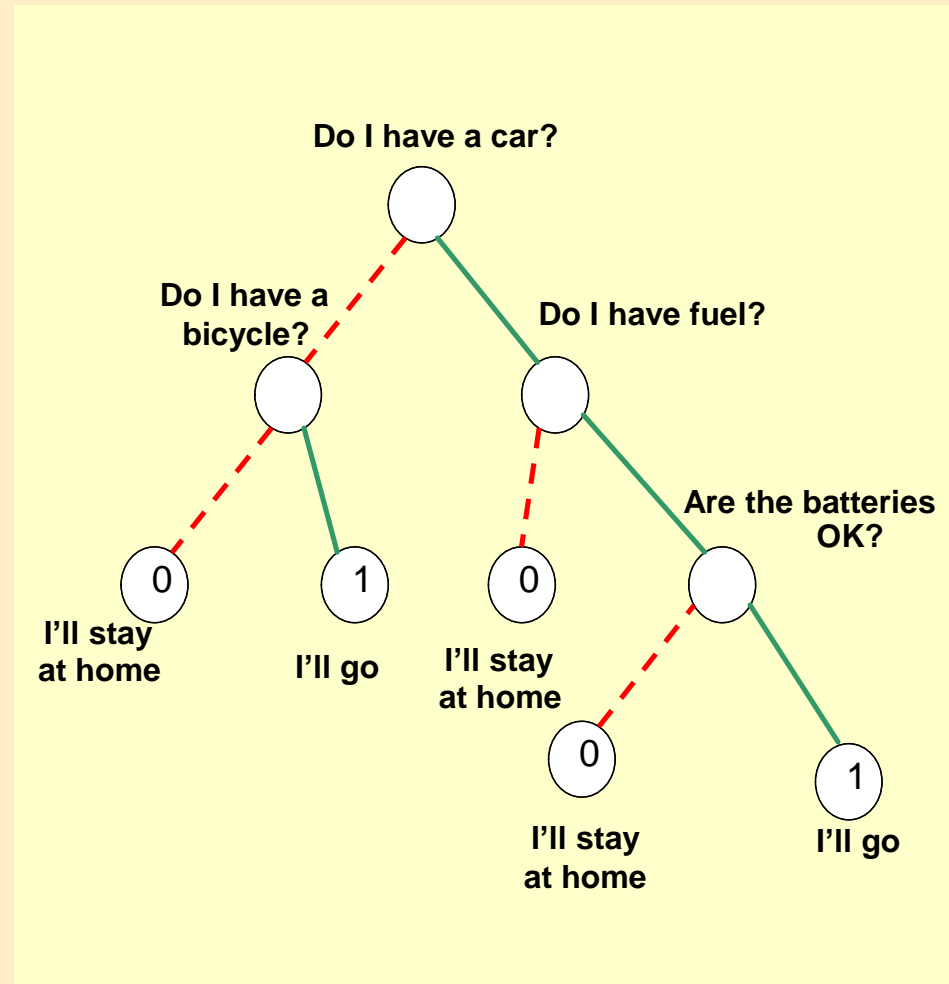
- Binary decision tree: to represent binary decisions
- BDD: identical subtrees are merged
- OBDD: evaluation of variables in the same order on every branch
- ROBDD: reduction of redundant nodes
  - If both two outcomes (branches) lead to the same node

# ROBDDs in detail

# Binary decision trees

- Final result is determined by a series of decisions
- Binary decisions in every node
  - Yes/**No** branches
- Final result after every necessary decision has been made:
  - Yes (1) / No (0)

There are multivalued extensions





# Boolean functions as binary decision trees

- Substitution (valuation) of a variable is a decision
- Notation: if-then-else

$$x \rightarrow f_1, f_0 = (x \wedge f_1) \vee (\neg x \wedge f_0)$$

- The result is the value of  $f_1$  if  $x$  is true (1)
  - The result is the value of  $f_0$  if  $x$  is false (0)
  - $x$  is the test variable, checking its value is a test
- Shannon decomposition of Boolean functions:

$$\left. \begin{array}{l} f = x \rightarrow f [1/x], f [0/x] \\ \text{let } f_x = f [1/x] ; f_{\underline{x}} = f [0/x] \end{array} \right\} f = x \rightarrow f_x, f_{\underline{x}}$$

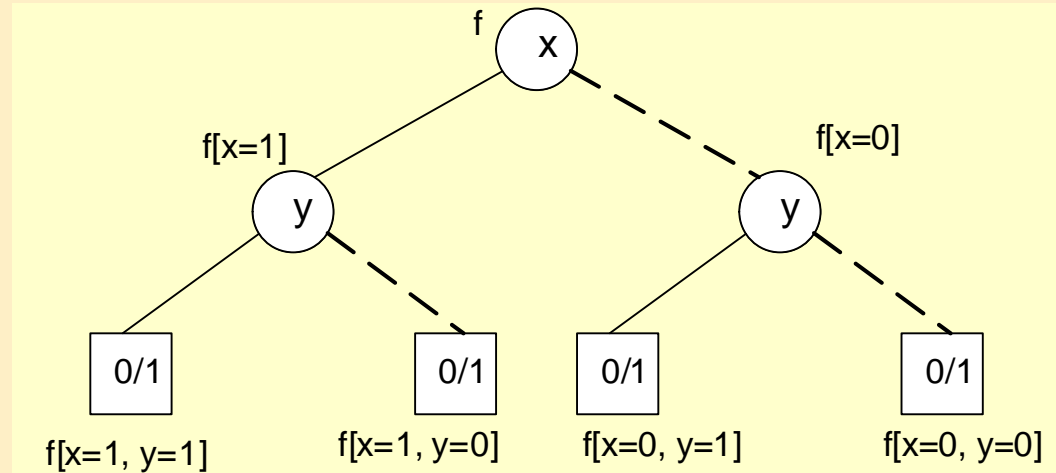
- The function is decomposed with if-then-else
- The test variable is reduced, will not appear in  $f_x, f_{\underline{x}}$
- Repeat until there is a variable left

# Types of decision trees

Example:

$f(x,y)$

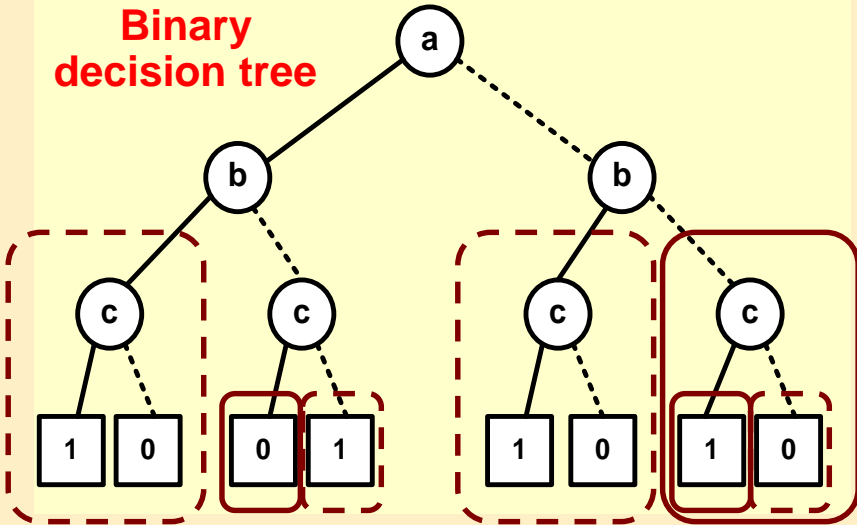
Potential values of  $f(x,y)$  should be specified in the boxes (leaf/terminal nodes)



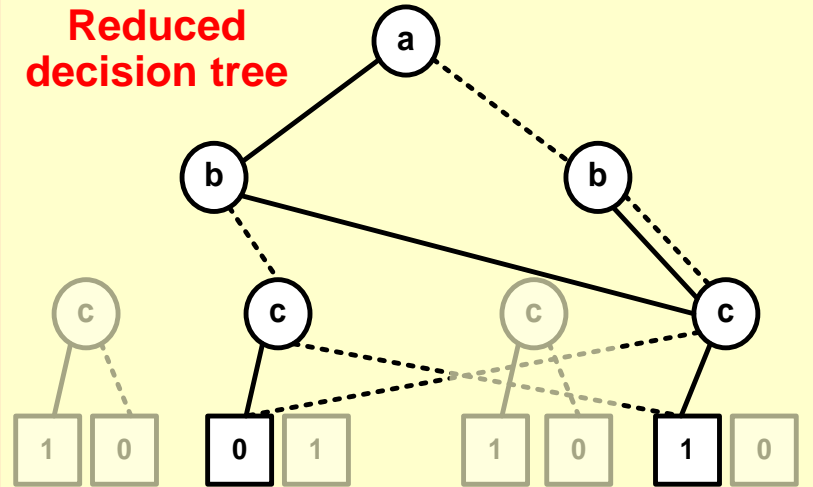
- We get a **binary decision diagram (BDD)**, if we merge all identical subtrees
- We get an **ordered binary decision diagram (OBDD)**, if we use test variables in the same order during decomposition
- We get a **reduced ordered binary decision diagram (ROBDD)**, if we remove redundant nodes (where both decisions lead to the same node)

# Example: Transformation of a binary decision diagram

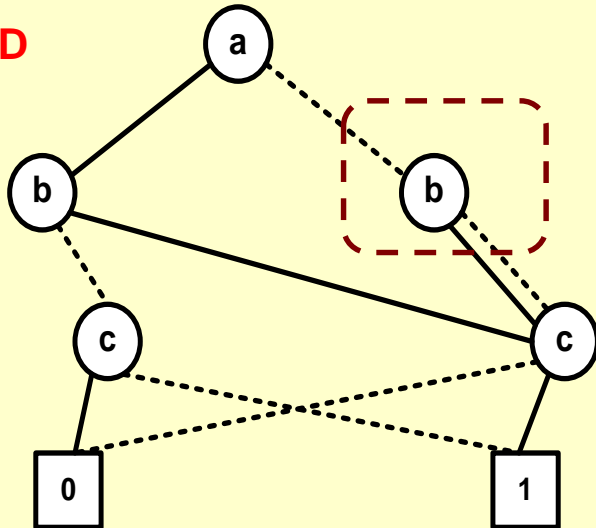
Binary decision tree



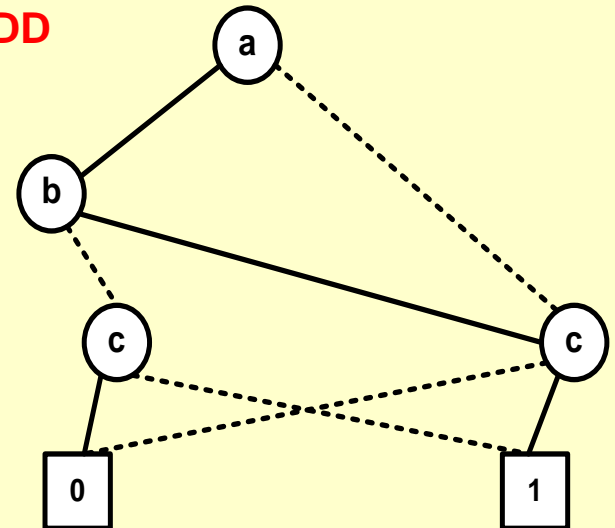
Reduced decision tree



BDD



ROBDD



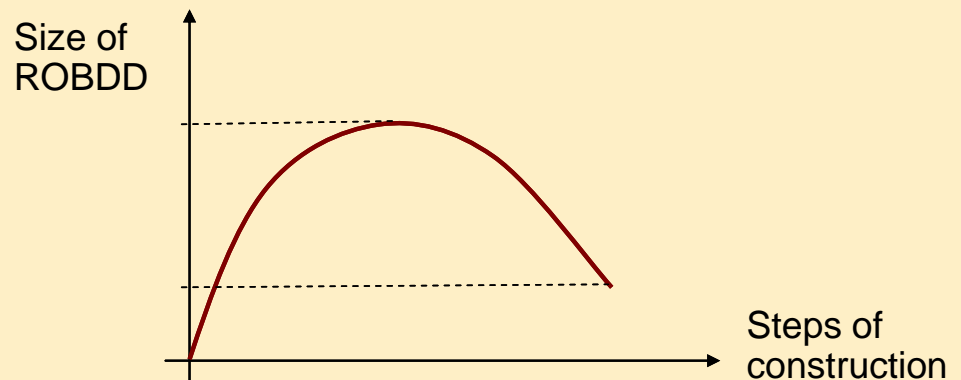
# ROBDD properties

- Directed, acyclic graph with one root and two leaves
  - Values of the two leaves are **1** and **0** (true and false)
  - Every node is assigned a test variable
- From every node, two edges leave
  - One for the value **0** (notation: dashed arrow)
  - The other for the value **1** (notation: solid arrow)
- On every path, test variables are in the same order
- Isomorphic subgraphs are merged
- Nodes from which both edges would point to the same node are reduced

For a given function, two ROBDDs with the same variable ordering are **isomorphic**

# Variable ordering for ROBDDs

- Size of ROBDD
  - For some functions (e.g. even number of 1's) very compact
  - For others (such as XOR) it may have an exponential size
- The order of variables has a great impact on the size!
  - A different order may cause an order of magnitude difference
  - Problem of finding an optimal ordering is NP-complete ( $\rightarrow$ heuristics)
- Memory requirements: If the ROBDD is built by combining functions one by one, we will store intermediate nodes which can be reduced later

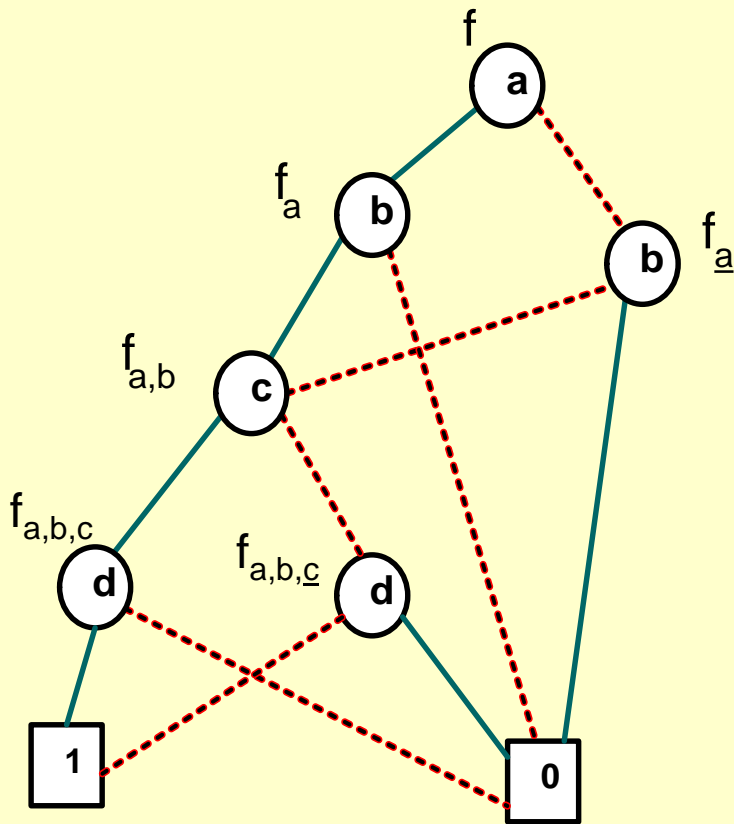


# Example: Manual construction of an ROBDD

Let

$$f = (a \Leftrightarrow b) \wedge (c \Leftrightarrow d)$$

Variable ordering: a, b, c, d



- $f = a \rightarrow f_a, \underline{f_a}$   
 $f_a = (1 \Leftrightarrow b) \wedge (c \Leftrightarrow d), \underline{f_a} = (0 \Leftrightarrow b) \wedge (c \Leftrightarrow d)$
- $f_a = b \rightarrow f_{a,b}, \underline{f_{a,b}}$   
 $f_{a,b} = (1 \Leftrightarrow 1) \wedge (c \Leftrightarrow d) = (c \Leftrightarrow d)$   
 $\underline{f_{a,b}} = (1 \Leftrightarrow 0) \wedge (c \Leftrightarrow d) = 0$
- $f_a = b \rightarrow f_{\underline{a},b}, \underline{f_{\underline{a},b}}$   
 $f_{\underline{a},b} = (0 \Leftrightarrow 1) \wedge (c \Leftrightarrow d) = 0$   
 $\underline{f_{\underline{a},b}} = (0 \Leftrightarrow 0) \wedge (c \Leftrightarrow d) = (c \Leftrightarrow d)$
- $f_{a,b} = c \rightarrow f_{a,b,c}, \underline{f_{a,b,c}}$   
 $f_{a,b,c} = (1 \Leftrightarrow d), \underline{f_{a,b,c}} = (0 \Leftrightarrow d)$
- $f_{a,b,c} = d \rightarrow f_{a,b,c,d}, \underline{f_{a,b,c,d}}$   
 $f_{a,b,c,d} = (1 \Leftrightarrow 1) = 1,$   
 $\underline{f_{a,b,c,d}} = (1 \Leftrightarrow 0) = 0$
- $f_{a,b,c} = d \rightarrow f_{a,b,c,\underline{d}}, \underline{f_{a,b,c,\underline{d}}}$   
 $f_{a,b,c,\underline{d}} = (0 \Leftrightarrow 1) = 0, \underline{f_{a,b,c,\underline{d}}} = (0 \Leftrightarrow 0) = 1$

$f_{a,b}$  and  $f_{\underline{a},b}$  are isomorphic!

## Storing an ROBDD in memory

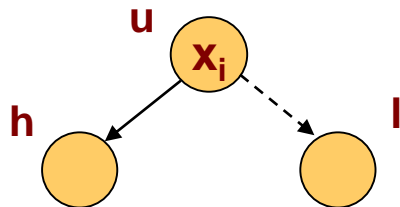
- Nodes of the ROBDD are identified by Ids (indices)
- The ROBDD is stored in a table

$T: u \rightarrow (i,l,h):$

- $u$ : index of node
- $i$ : index of variable ( $x_i, i=1\dots n$ )
- $l$ : index of the node reachable through edge corresponding to 0
- $h$ : index of the node reachable through edge corresponding to 1

low

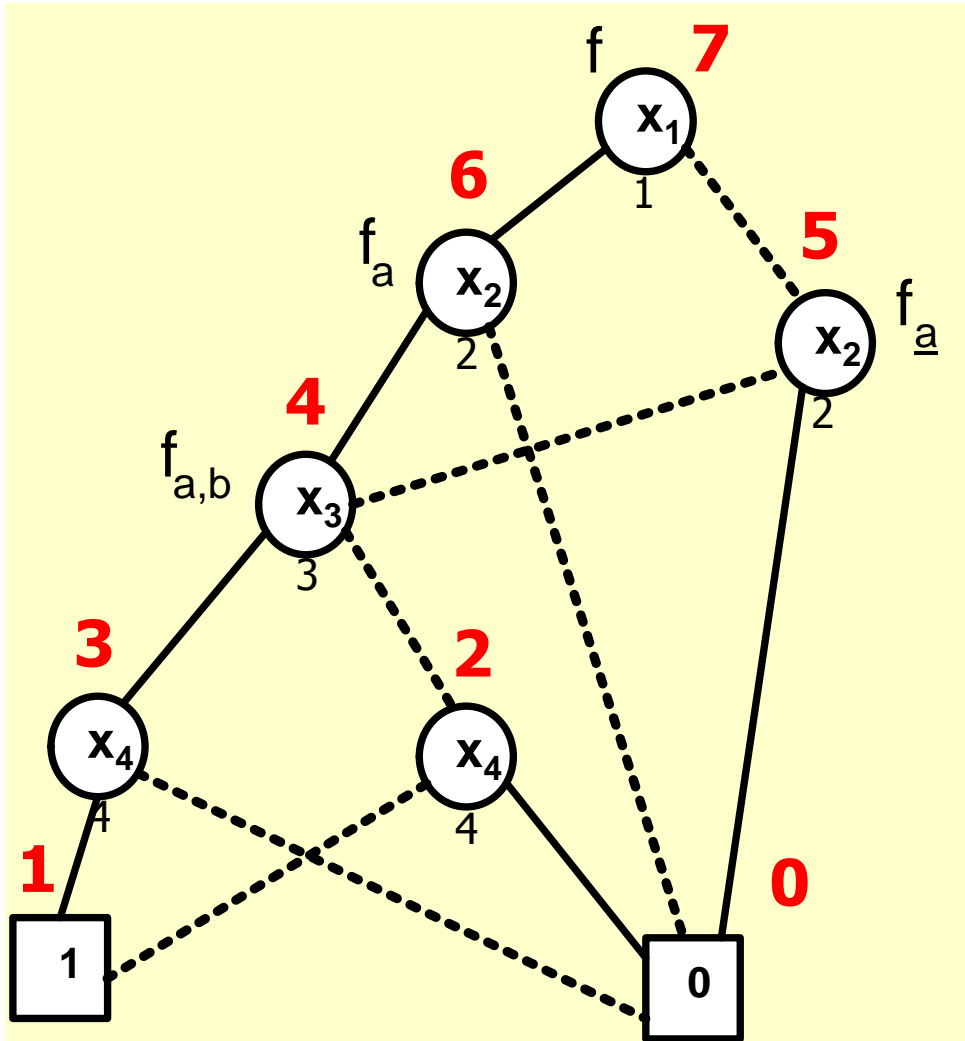
high



<b>u</b>	<b>i</b>	<b>l</b>	<b>h</b>
<b>0</b>			
<b>1</b>			
<b>2</b>	4	1	0
<b>3</b>	4	0	1
<b>4</b>	3	2	3
<b>5</b>	2	4	0
<b>6</b>	2	0	4
<b>7</b>	1	5	6

# Storing an ROBDD in memory

Auxiliary



<b>u</b>	<b>i</b>	<b>l</b>	<b>h</b>
<b>0</b>			
<b>1</b>			
<b>2</b>	4	1	0
<b>3</b>	4	0	1
<b>4</b>	3	2	3
<b>5</b>	2	4	0
<b>6</b>	2	0	4
<b>7</b>	1	5	6



# Handling ROBDDs 1.

- Defined operations:
  - $\text{init}(T)$ 
    - Initializes table  $T$
    - Only the terminal nodes  $0$  and  $1$  are in the table
  - $\text{add}(T, i, l, h): u$ 
    - Creates a new node in  $T$  with the provided parameters
    - Returns its index  $u$
  - $\text{var}(T, u): i$ 
    - Returns from  $T$  the index  $i$  of the node  $u$
  - $\text{low}(T, u): l$  and  $\text{high}(T, u): h$ 
    - Returns the index  $l$  (or  $h$ ) of the node reachable from the node with index  $u$  through the edge corresponding to  $0$  (or  $1$ )

## Handling ROBDDs 2.

- To look up ROBDD nodes, we use another table  
 $H: (i,l,h) \rightarrow u$
- Operations:
  - $\text{init}(H)$ 
    - Initializes an empty  $H$
  - $\text{member}(H,i,l,h):t$ 
    - Checks if the triple  $(i,l,h)$  is in  $H$ ;  $t$  is a Boolean value
  - $\text{lookup}(H,i,l,h):u$ 
    - Looks up the triple  $(i,l,h)$  from table  $H$
    - Returns the index  $u$  of the matching node
  - $\text{insert}(H,i,l,h,u)$ 
    - Inserts a new entry into the table

## Handling ROBDDs 3.

Creating nodes:  $Mk(i,l,h)$

- Where  $i$  is the index of variable,  $l$  and  $h$  are the branches
- If  $l=h$ , i.e. the branches would lead to the same node
  - then we don't need new a node
  - we can return any branch
- If  $H$  already contains a triple  $(i,l,h)$ 
  - then we don't need a new node  
⇒ There exists an isomorphic subtree, return that
- If  $H$  does not contain such a triple  $(i,l,h)$ 
  - then we need to create it and return its index

```
Mk(i,l,h) {  
    if l=h then  
        return l;  
    else if member(H,i,l,h) then  
        return lookup(H,i,l,h);  
    else {  
        u=add(T,i,l,h);  
        insert(H,i,l,h,u);  
        return u;  
    }  
}
```

## Handling ROBDDs 4.

Building an ROBDD: `Build(f)` and `Build'(t,i)` recursive helper function

```
Build(f) {  
    init(T); init(H);  
    return Build'(f,1);  
}
```

Will traverse variables recursively

```
Build'(t,i) {  
    if i>n then  
        if t==false then return 0 else return 1  
    else {v0 = Build'(t[0/xi],i+1);  
        v1 = Build'(t[1/xi],i+1);  
        return Mk(i,v0,v1)}  
}
```

Reached a terminal node  
(every variable bound)

Recursive building;  
Mk() will check  
isomorphic subtrees



# Operations on ROBDDs (cont'd)

- Boolean operators can be evaluated directly on ROBDDs
  - Variables of the functions should be the same in the same order
- Equivalence for functions  $f, t$  ( $op$  is a Boolean operator):
  1.  $f \text{ op } t = (x \rightarrow f_x, f_{\underline{x}}) \text{ op } (x \rightarrow t_x, t_{\underline{x}}) = x \rightarrow (f_x \text{ op } t_x), (f_{\underline{x}} \text{ op } t_{\underline{x}})$
- Additional rules (missing variables due to reduction):
  2.  $f \text{ op } t = (x \rightarrow f_x, f_{\underline{x}}) \text{ op } t = x \rightarrow (f_x \text{ op } t), (f_{\underline{x}} \text{ op } t)$
  3.  $f \text{ op } t = f \text{ op } (x \rightarrow t_x, t_{\underline{x}}) = x \rightarrow (f \text{ op } t_x), (f \text{ op } t_{\underline{x}})$
- Based on these rules  $App(op, i, j)$  can be defined recursively
  - where  $i, j$ : indices of the root nodes of operands
- Drawback: slow
  - worst-case  $2^n$  exponential

# Accelerated operation

- Let  $G(op,i,j)$  be a cache table that contains the results of  $App(op,i,j)$  (these are nodes)
- The four cases of the algorithm:
  - Both nodes are **terminal**: return a terminal based on the Boolean operation (e.g.  $0 \wedge 1 = 0$ )
  - If the **variable indices** for both operands are the **same**, then call  $App(op,i,j)$  with the **0** branches and with the **1** one branches based on **equivalence 1**.
  - If one **variable index is less**, then that node is paired with the **0** and **1** branches of the other based on **equivalence 2. or 3**.

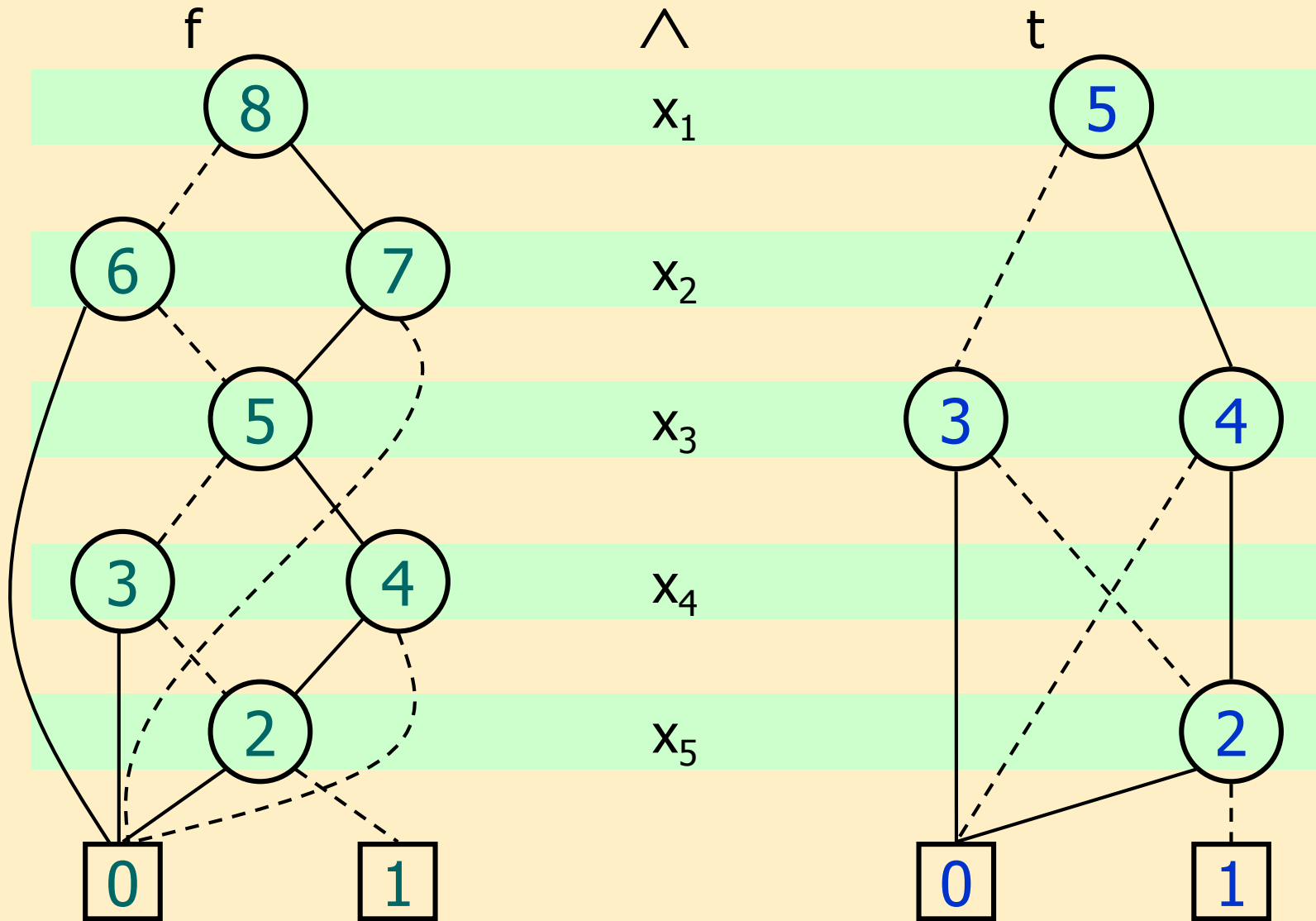
## Pseudo-code of the operation

```
Apply(op, f, t) {
    init(G);
    return App(op, f, t);
}

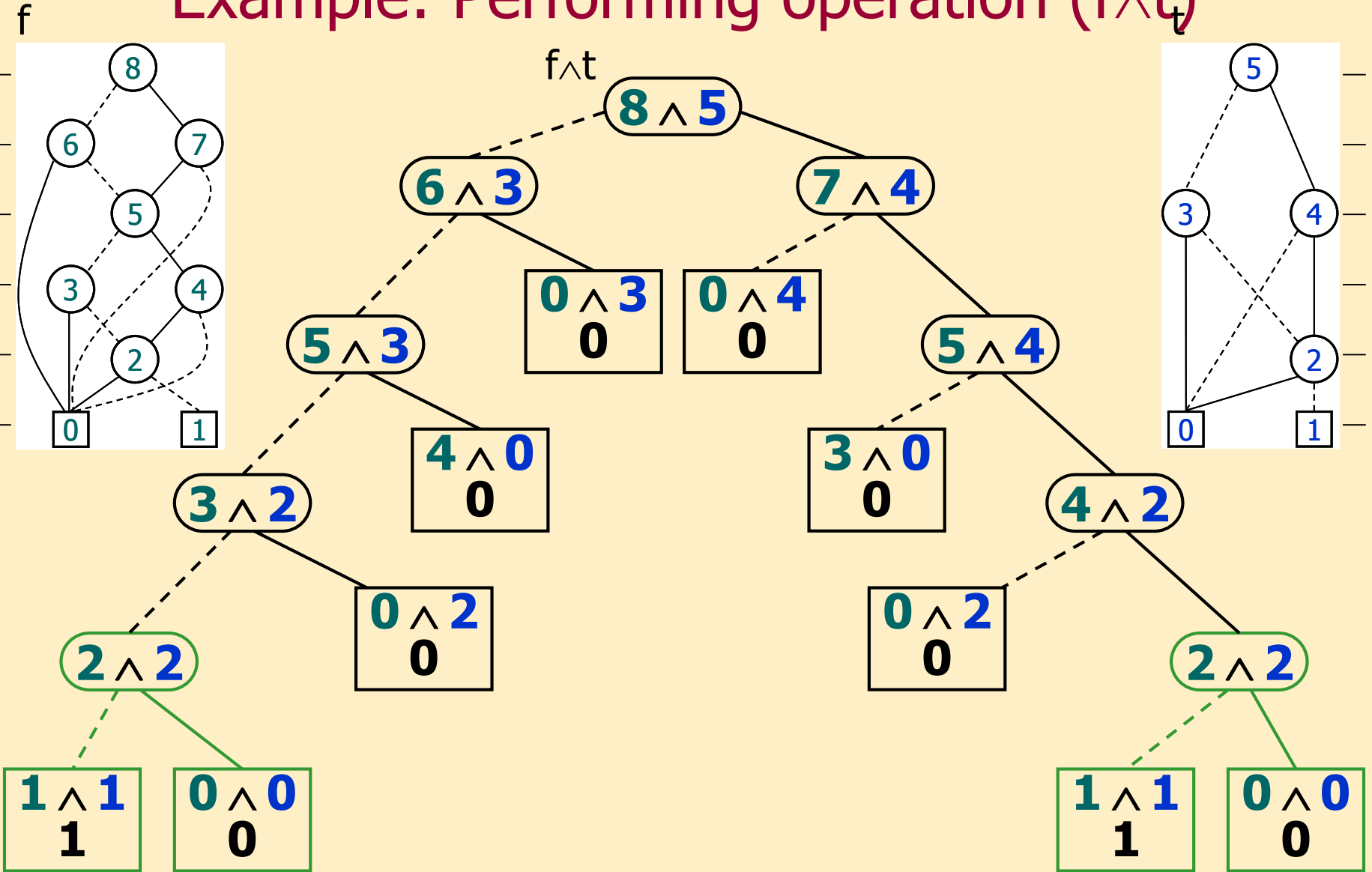
App(op, u1, u2) {
    if (G(op, u1, u2) != empty) then return G(op, u1, u2);
    else if (u1 in {0, 1} and u2 in {0, 1}) then u = op(u1, u2);
    else if (var(u1) = var(u2)) then
        u = Mk(var(u1), App(op, low(u1), low(u2)),
              App(op, high(u1), high(u2)));
    else if (var(u1) < var(u2)) then
        u = Mk(var(u1), App(op, low(u1), u2), App(op, high(u1), u2));
    else (* if (var(u1) > var(u2)) then *)
        u = Mk(var(u2), App(op, u1, low(u2)), App(op, u1, high(u2)));
    G(op, u1, u2) = u;
    return u;
}
```



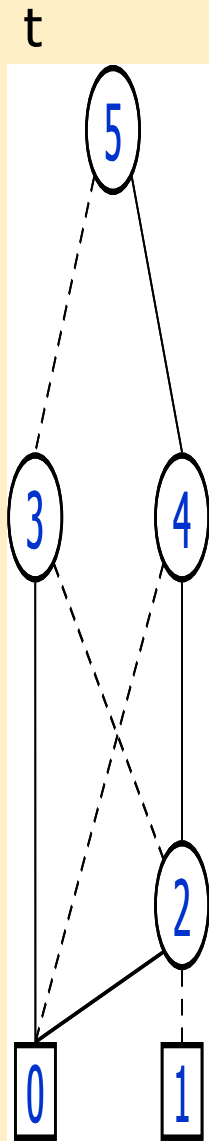
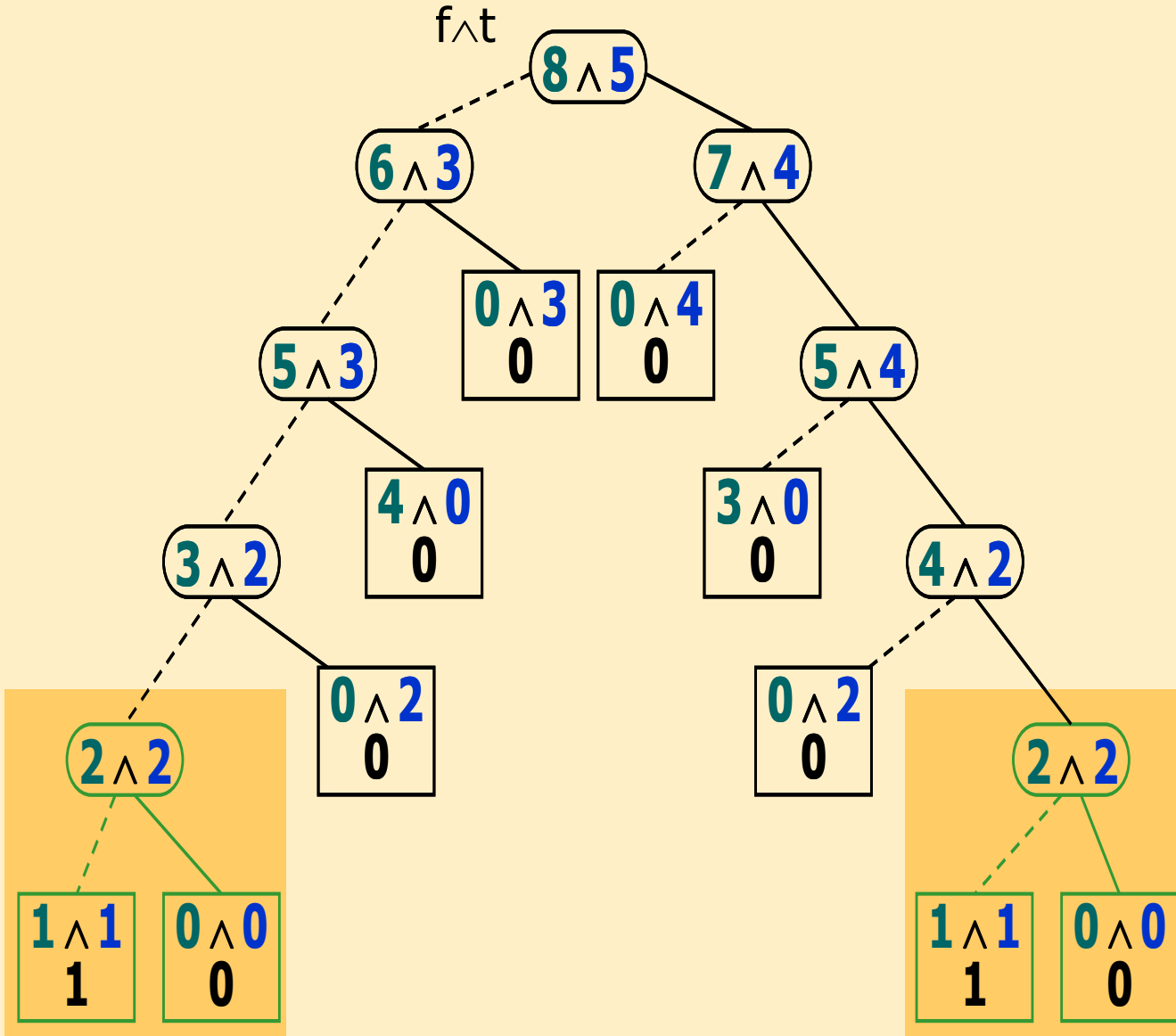
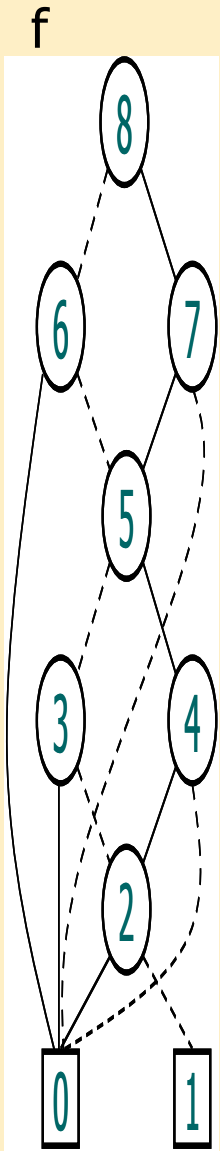
# Example: Performing operation $(f \wedge t)$



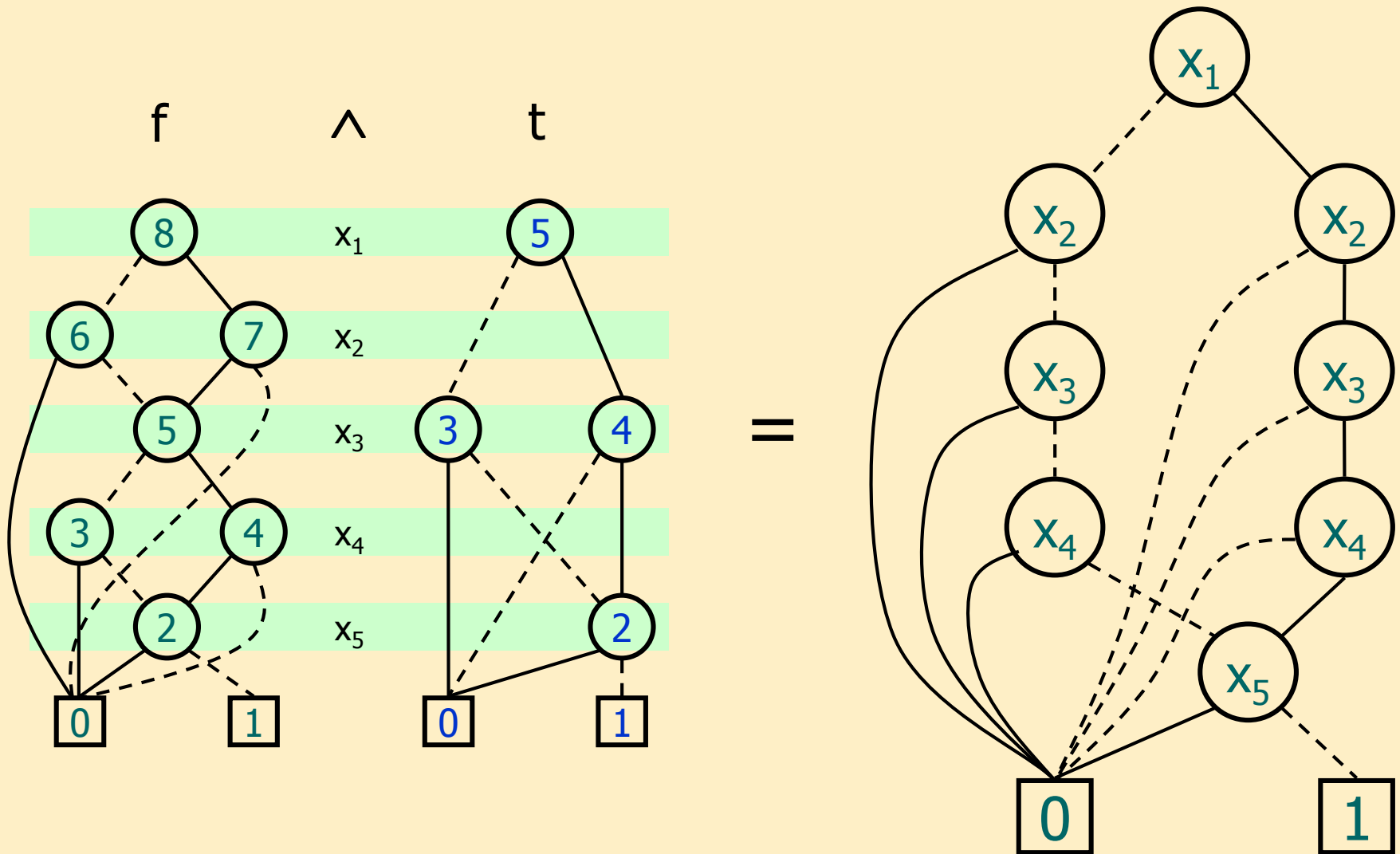
# Example: Performing operation $(f \wedge t)$



# Example: Performing operation $(f \wedge t)$



# Example: Result of operation $(f \wedge t)$



# Restricting a variable in an ROBDD

Bind variables with constants (e.g.  $(\neg x \wedge y)^{[y=1]} = \neg x$ ):

The value of  $x_j$  should be  $b$  in the ROBDD rooted in  $u$

```
Restrict(u, j, b) {  
    return Res(u, j, b);  
}  
  
Res(u, j, b) {  
    if var(u) > j then return u;  
    else if var(u) < j then  
        return Mk(var(u),  
                  Res(low(u), j, b),  
                  Res(high(u), j, b));  
    else  
        if b=0 then  
            return Res(low(u), j, b)  
        else  
            return Res(high(u), j, b);  
}
```

If we are lower than the variable to bind, the original subtree is returned

If we are higher, then we need recursive building

If we are at the variable to bind, we process only the branch  $b$

# Summary: Model checking with ROBDDs

- Realizing model checking:
  - Model checking algorithm: Operations on sets of states (labeling)
  - Symbolic technique: Instead of sets, use Boolean characteristic functions
  - Efficient implementation: Boolean functions handled as ROBDDs
- Benefits
  - ROBDD is a canonical form (equivalence of functions is easy to check)
  - Algorithms can be accelerated (with caching)
  - Reduced storage requirements (depends on variable ordering!)

## Dining philosophers:

Number of Philosophers	Size of state space	Number of ROBDD nodes
16	$4,7 \cdot 10^{10}$	747
28	$4,8 \cdot 10^{18}$	1347

Instead of storing  $10^{18}$  states the ROBDD takes ~21kB!