Reactive Synthesis

Lecture 4

Swen Jacobs and Martin Zimmermann
(Saarland University)

Recap

Definition

A **symbolic representation** for a safety game is given as a tuple $(L, X_u, X_c, T(L, X_u, X_c, L'), \mathit{Unsafe}(L))$, where

- L is a set of (boolean) state variables
- X_u is a set of (boolean) uncontrollable input variables
- X_c is a set of (boolean) controllable input variables
- $T(L, X_u, X_c, L')$ is the transition relation, given as a quantifier-free boolean formula over variables from L, X_u, X_c and L', which represents the values of variables from L after the transition
- $\mathit{Unsafe}(L)$ defines the unsafe states, as a quantifier-free boolean formula over state variables L

Remark

We sometimes emphasize the variables X occurring in a formula F by writing $F(X)$.

Recap

To solve a safety game in symbolic representation, we can compute the (Player 1) attractor of the unsafe states.

The controllable predecessor of Player 1 becomes

$$\mathrm{CPre}_1(F) = \exists X_u \forall X_c \exists L'.\ F(L') \wedge T(L, X_u, X_c, L')$$

and the attractor is defined as (with $F = \mathit{Unsafe}$)

- $\mathrm{Attr}_1^0(F) = F$,
- $\mathrm{Attr}_1^{n+1}(F) = \mathrm{Attr}_1^n(F) \vee \mathrm{CPre}_1(\mathrm{Attr}_1^n(F))$, and

Recap

To solve a safety game in symbolic representation, we can compute the (Player 1) attractor of the unsafe states.

The controllable predecessor of Player 1 becomes

$$\mathrm{CPre}_1(F) = \exists X_u \forall X_c \exists L'.\ F(L') \wedge T(L, X_u, X_c, L')$$

and the attractor is defined as (with $F = \mathit{Unsafe}$)

- $\mathrm{Attr}_1^0(F) = F$,
- $\mathrm{Attr}_1^{n+1}(F) = \mathrm{Attr}_1^n(F) \vee \mathrm{CPre}_1(\mathrm{Attr}_1^n(F))$, and
- $\mathrm{Attr}_1(F) = \bigvee_{n \in \mathbb{N}} \mathrm{Attr}_1^n(F)$.

Problem: how can we efficiently compute this?
Need algorithms and data structures that allow:

- efficient combination of existing formulas with of $\exists, \forall, \wedge, \vee$
- efficient equivalence check of formulas (to determine attractor computation has finished),
- compact representation during these computations.

Recap

Definition

A **Binary Decision Diagram (BDD)** over a set of variables X is a directed, acyclic graph $G(V, E)$ with exactly one root $v_r \in V$ and the following properties:

- every node in V is either terminal or non-terminal
- each terminal node is labeled with a value from $\{0, 1\}$
- each non-terminal node is labeled with a variable $x \in X$ and has exactly two outgoing edges, denoted by $high(v) \in V$ and $low(v) \in V$, respectively.

Definition

A BDD G is a **Reduced Ordered Binary Decision Diagram (ROBDD)** if it is ordered and reduced, i.e., neither the isomorphism nor the Shannon reduction can be applied.

Recap

Theorem (Canonicity of ROBDDs)

Let $<$ be a fixed variable order on X. Then for each boolean formula F there exists (up to isomorphism) exactly one ROBDD over X with $<$.

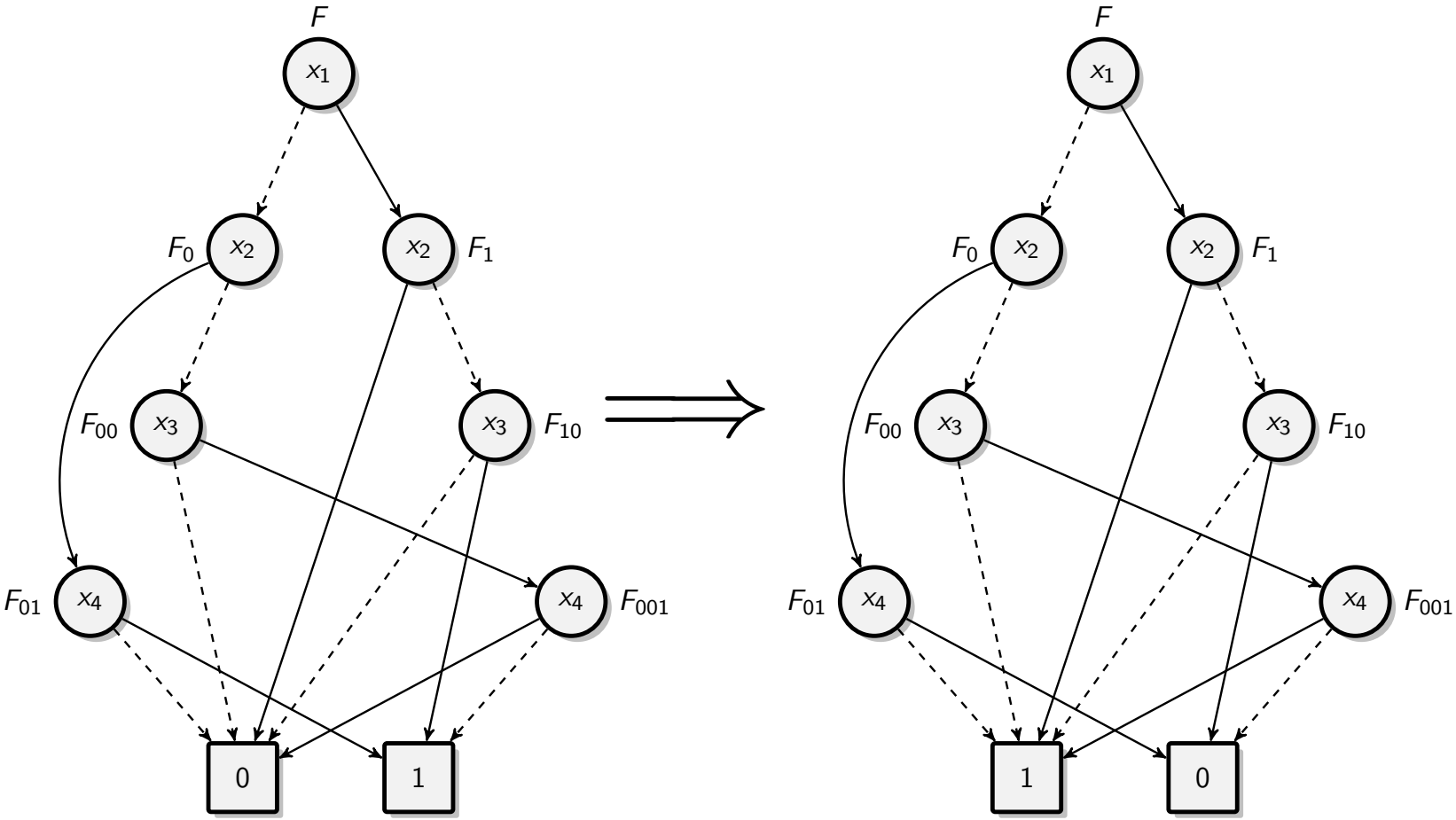
Remark

This means that two ROBDDs G_1, G_2 represent the same boolean formula if and only if they are isomorphic. Isomorphism, and thus equivalence, can be checked (by synchronous depth-first search) in time $O(|V_{G_1}| + |V_{G_2}|)$.

Recap

Let G be an OBDD for formula F. Then an OBDD for formula $\bar{F}$ is obtained by swapping the terminal nodes.

Example



Recap

Input: OBDDs G for F, H for D, binary operator $\circ$
Output: OBDD for $F \circ D$
Apply$(G, H, \circ)$

1. if (G and H are terminal nodes) then return $G \circ H$
2. if $(G, H) \in \mathit{HashTable}$ then return $\mathit{Hashtable}(G, H)$
3. let x be the maximal variable that G or H depend on
4. for $B \in \{G, H\}$ do: if $\mathit{var}(\mathit{root}(B)) = x$ then let $(B_0, B_1) = (B_{\mathit{low}(\mathit{root}(B))}, B_{\mathit{high}(\mathit{root}(B))})$ else let $B_0 = B_1 = B$
5. construct G_{new} with new node v_{new} as root and
   $\mathit{var}(v_{new}) = x$
   $\mathit{low}(v_{new}) = \textsc{Apply}(G_0, H_0, \circ)$
   $\mathit{high}(v_{new}) = \textsc{Apply}(G_1, H_1, \circ)$
6. add (G, H, G_{new}) to $\mathit{HashTable}$
7. return G_{new}

Plan for Today

- Basic Games
- Algorithms & Data Structures

- Advanced Games
- Temporal Logic Synthesis

Plan for Today

- Basic Games
- Algorithms & Data Structures
  - (More) Operations on BDDs
  - Size and Ordering of BDDs
  - Optimizations of BDD-based Algorithms
  - (Reactive Synthesis based on BDDs)
- Advanced Games
- Temporal Logic Synthesis

(More) Operations on BDDs

Operations on OBDDs: Quantification

One additional operation needed for attractor computation: **quantification of variables**.
For a boolean formula F and variable x:

$$\forall x.F \Leftrightarrow (F[x := 1] \wedge F[x := 0])$$

$$\exists x.F \Leftrightarrow (F[x := 1] \vee F[x := 0])$$

Operations on OBDDs: Quantification

One additional operation needed for attractor computation: **quantification of variables**.
For a boolean formula F and variable x:

$$\forall x.F \Leftrightarrow (F[x := 1] \wedge F[x := 0])$$

$$\exists x.F \Leftrightarrow (F[x := 1] \vee F[x := 0])$$

Example

Let F be the formula $\bar{x_1}\bar{x_2}x_3 \vee x_1\bar{x_3} \vee x_1x_2$. Then:

$$\begin{aligned} & \exists x_3.F \\ \Leftrightarrow\ & F[x_3 := 1] \vee F[x_3 := 0] \\ \Leftrightarrow\ & (\bar{x_1}\bar{x_2} \vee x_1x_2) \vee (x_1 \vee x_1x_2) \\ \Leftrightarrow\ & x_1 \vee \bar{x_2} \end{aligned}$$

Properties of Boolean Quantification

Commutativity:

$$\exists x_1 \exists x_2 . F \Leftrightarrow \exists x_2 \exists x_1 . F$$

$$\forall x_1 \forall x_2 . F \Leftrightarrow \forall x_2 \forall x_1 . F$$

Properties of Boolean Quantification

Commutativity:

$$\exists x_1 \exists x_2 . F \Leftrightarrow \exists x_2 \exists x_1 . F$$

$$\forall x_1 \forall x_2 . F \Leftrightarrow \forall x_2 \forall x_1 . F$$

Distributivity:

$$\exists x . (F_1 \vee F_2) \Leftrightarrow \exists x . F_1 \vee \exists x . F_2$$

$$\forall x . (F_1 \wedge F_2) \Leftrightarrow \forall x . F_1 \wedge \forall x . F_2$$

Properties of Boolean Quantification

Commutativity:

$$\exists x_1 \exists x_2 . F \Leftrightarrow \exists x_2 \exists x_1 . F$$

$$\forall x_1 \forall x_2 . F \Leftrightarrow \forall x_2 \forall x_1 . F$$

Distributivity:

$$\exists x . (F_1 \vee F_2) \Leftrightarrow \exists x . F_1 \vee \exists x . F_2$$

$$\forall x . (F_1 \wedge F_2) \Leftrightarrow \forall x . F_1 \wedge \forall x . F_2$$

But:

$$\exists x_1 \forall x_2 . F \not\Leftrightarrow \forall x_2 \exists x_1 . F$$

$$\forall x_1 \exists x_2 . F \not\Leftrightarrow \exists x_2 \forall x_1 . F$$

Properties of Boolean Quantification

Commutativity:

$$\exists x_1 \exists x_2 . F \Leftrightarrow \exists x_2 \exists x_1 . F$$

$$\forall x_1 \forall x_2 . F \Leftrightarrow \forall x_2 \forall x_1 . F$$

Distributivity:

$$\exists x . (F_1 \vee F_2) \Leftrightarrow \exists x . F_1 \vee \exists x . F_2$$

$$\forall x . (F_1 \wedge F_2) \Leftrightarrow \forall x . F_1 \wedge \forall x . F_2$$

But:

$$\exists x_1 \forall x_2 . F \not\Leftrightarrow \forall x_2 \exists x_1 . F$$

$$\forall x_1 \exists x_2 . F \not\Leftrightarrow \exists x_2 \forall x_1 . F$$

$$\exists x . (F_1 \wedge F_2) \not\Leftrightarrow \exists x . F_1 \wedge \exists x . F_2$$

$$\forall x . (F_1 \vee F_2) \not\Leftrightarrow \forall x . F_1 \vee \forall x . F_2$$

Operations on OBDDs: Quantification

Task: given a BDD G that represents formula F, and a BDD C that represents the (conjunction of variables) $\vec{x}$ that we want to quantify over, compute a BDD for $\exists \vec{x}.F$.

Operations on OBDDs: Quantification

Task: given a BDD G that represents formula F, and a BDD C that represents the (conjunction of variables) $\vec{x}$ that we want to quantify over, compute a BDD for $\exists \vec{x}.F$.

Direct way: **quantify_direct**(G, C)

1. $B := G$
2. for $x \in \vec{x}$ do $\{$

Operations on OBDDs: Quantification

Task: given a BDD G that represents formula F, and a BDD C that represents the (conjunction of variables) $\vec{x}$ that we want to quantify over, compute a BDD for $\exists \vec{x}.F$.

Direct way: **quantify_direct**(G, C)

1. $B := G$
2. for $x \in \vec{x}$ do $\{$
3. $\quad B_0 := B[x := 0]$
4. $\quad B_1 := B[x := 1]$
5. $\quad B := APPLY(B_0, B_1, \vee)$ $\}$

Operations on OBDDs: Quantification

Task: given a BDD G that represents formula F, and a BDD C that represents the (conjunction of variables) $\vec{x}$ that we want to quantify over, compute a BDD for $\exists \vec{x}.F$.

Direct way: **quantify_direct**(G, C)

1. $B := G$
2. for $x \in \vec{x}$ do $\{$
3. $\quad B_0 := B[x := 0]$
4. $\quad B_1 := B[x := 1]$
5. $\quad B := APPLY(B_0, B_1, \vee)$ $\}$
6. return B

Operations on OBDDs: Quantification

Task: given a BDD G that represents formula F, and a BDD C that represents the (conjunction of variables) $\vec{x}$ that we want to quantify over, compute a BDD for $\exists \vec{x}.F$.

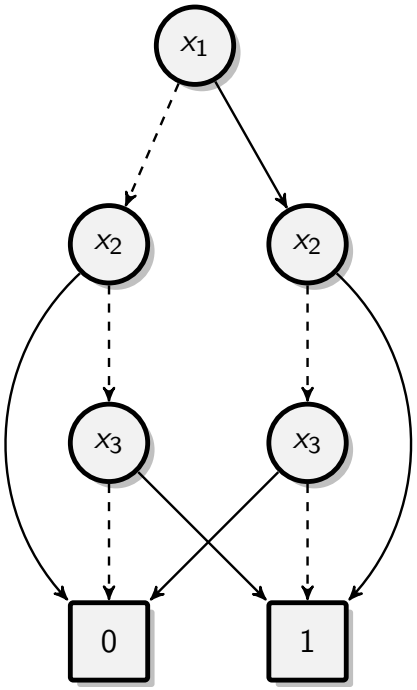
Direct way: **quantify_direct**(G, C)

1. $B := G$
2. for $x \in \vec{x}$ do $\{$
3. $\quad B_0 := B[x := 0]$
4. $\quad B_1 := B[x := 1]$
5. $\quad B := APPLY(B_0, B_1, \vee)$ $\}$
6. return B

Smart way: eliminate all variables recursively in one simultaneous pass over the BDDs (assuming that they have the same variable order).

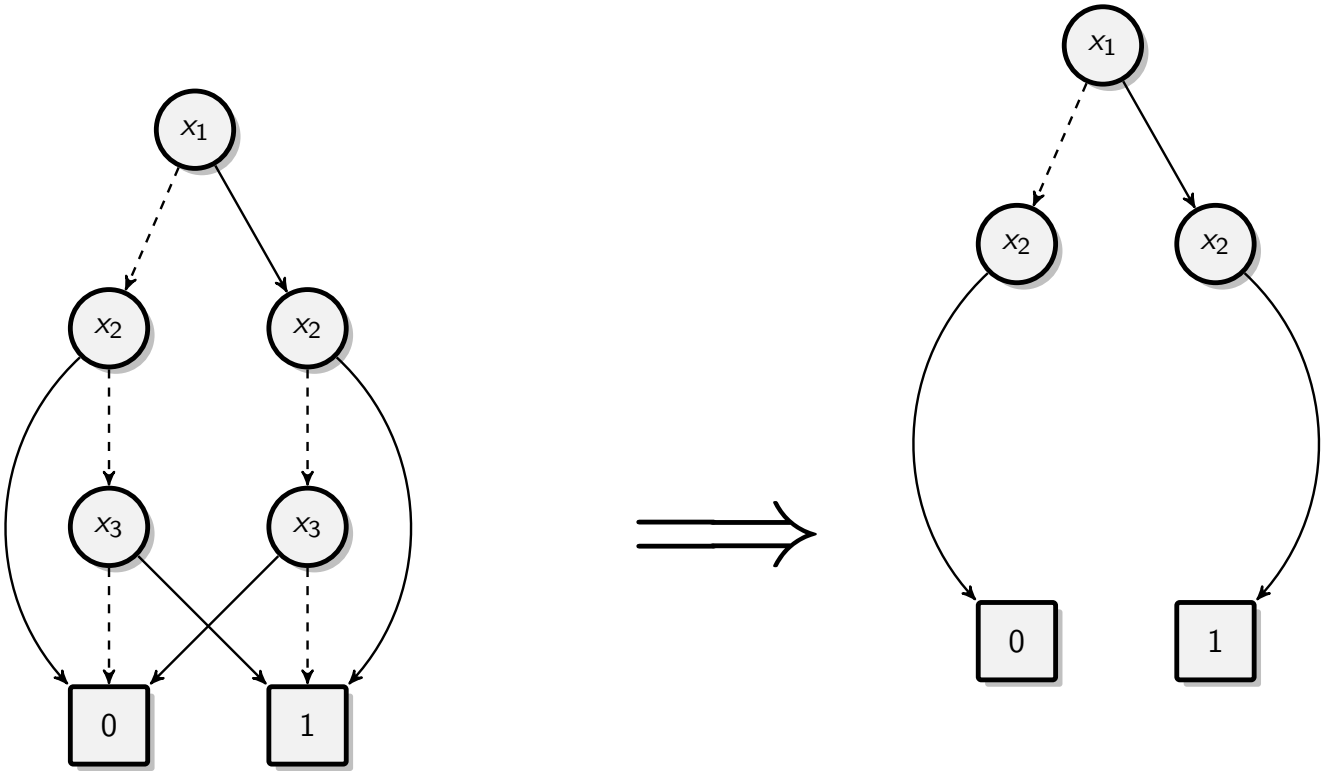
Example: Recursive Elimination of Variables

Input: a BDD G for $F = \bar{x_1}\bar{x_2}x_3 \vee x_1\bar{x_3} \vee x_1x_2$, a BDD C for x_3
Task: compute a BDD for $\exists x_3.F$.



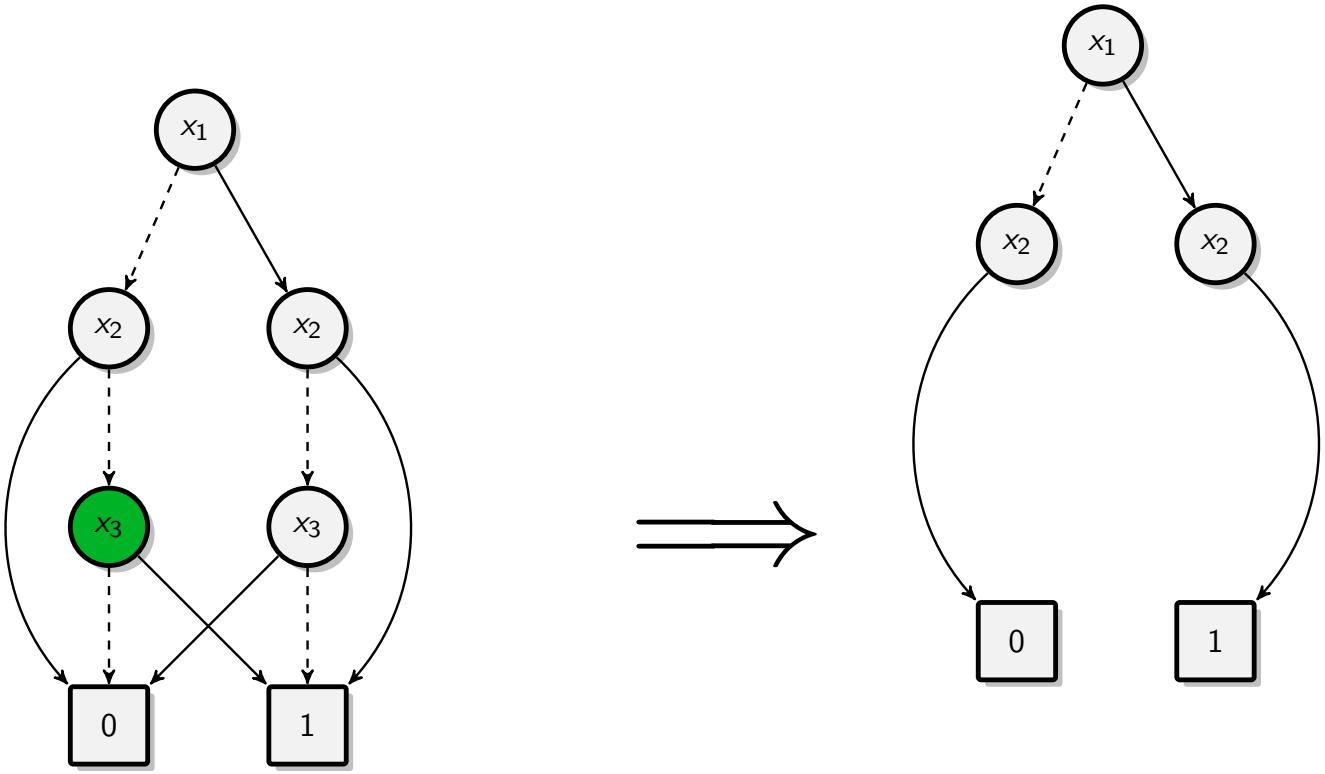
Example: Recursive Elimination of Variables

Input: a BDD G for $F = \bar{x_1}\bar{x_2}x_3 \vee x_1\bar{x_3} \vee x_1x_2$, a BDD C for x_3
Task: compute a BDD for $\exists x_3.F$.



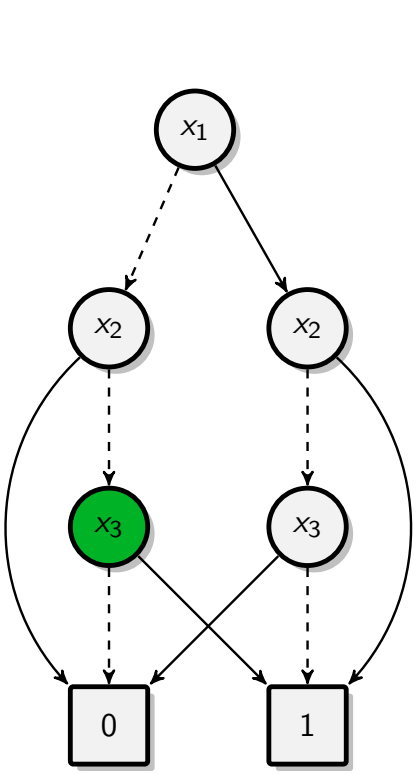
Example: Recursive Elimination of Variables

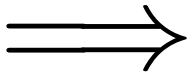
Input: a BDD G for $F = \bar{x_1}\bar{x_2}x_3 \vee x_1\bar{x_3} \vee x_1x_2$, a BDD C for x_3
Task: compute a BDD for $\exists x_3.F$.

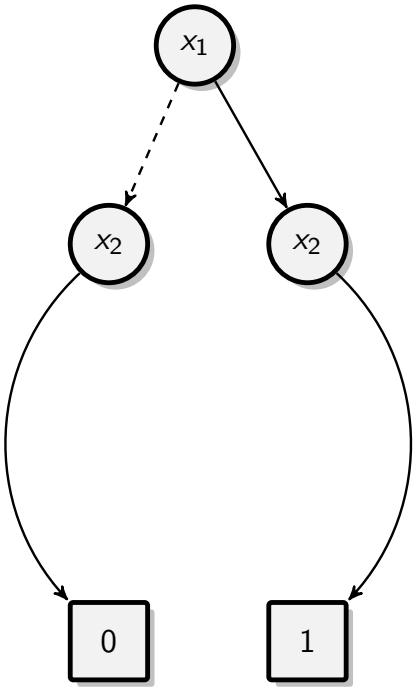


Example: Recursive Elimination of Variables

Input: a BDD G for $F = \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_3 \vee x_1x_2$, a BDD C for x_3
Task: compute a BDD for $\exists x_3.F$.

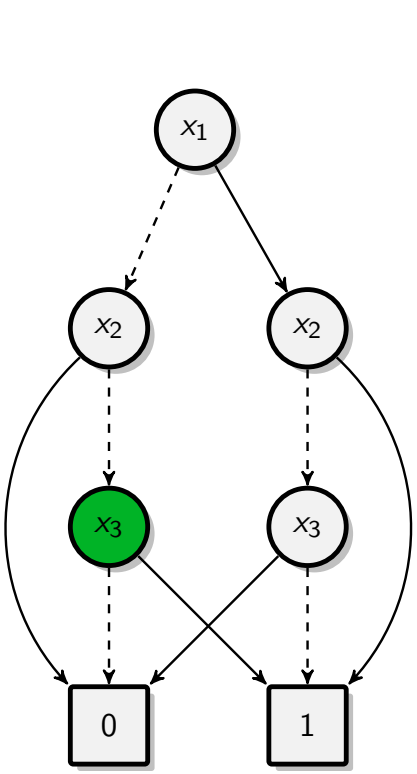


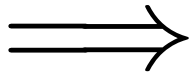


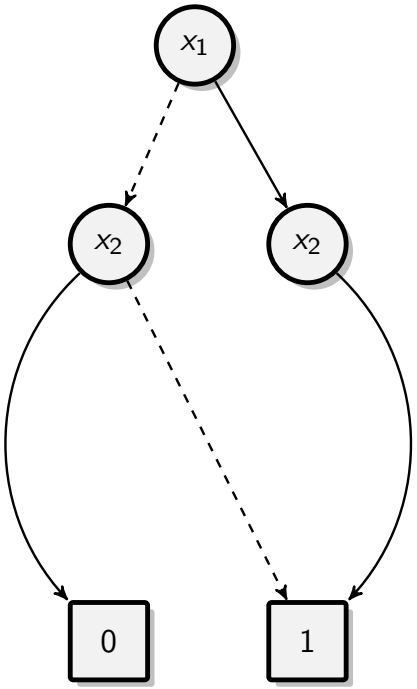


Example: Recursive Elimination of Variables

Input: a BDD G for $F = \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_3 \vee x_1x_2$, a BDD C for x_3
Task: compute a BDD for $\exists x_3.F$.

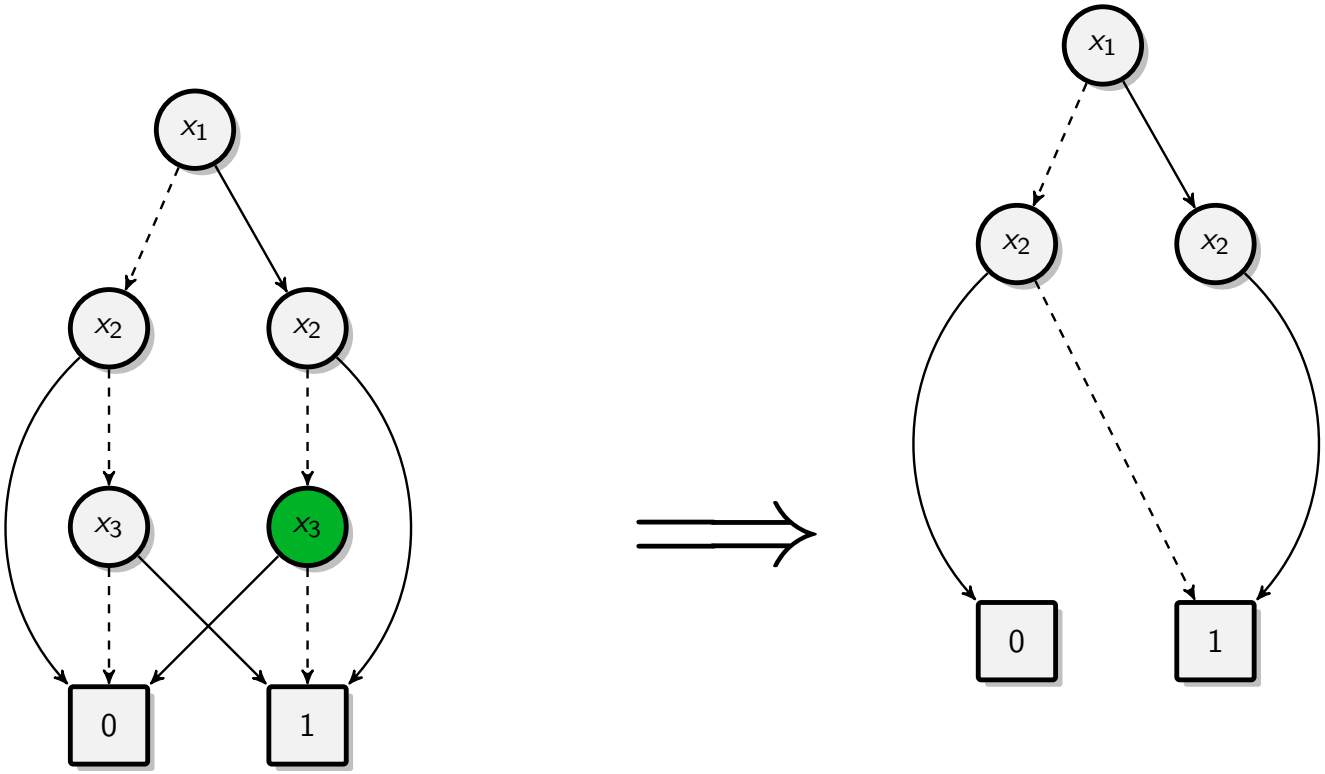






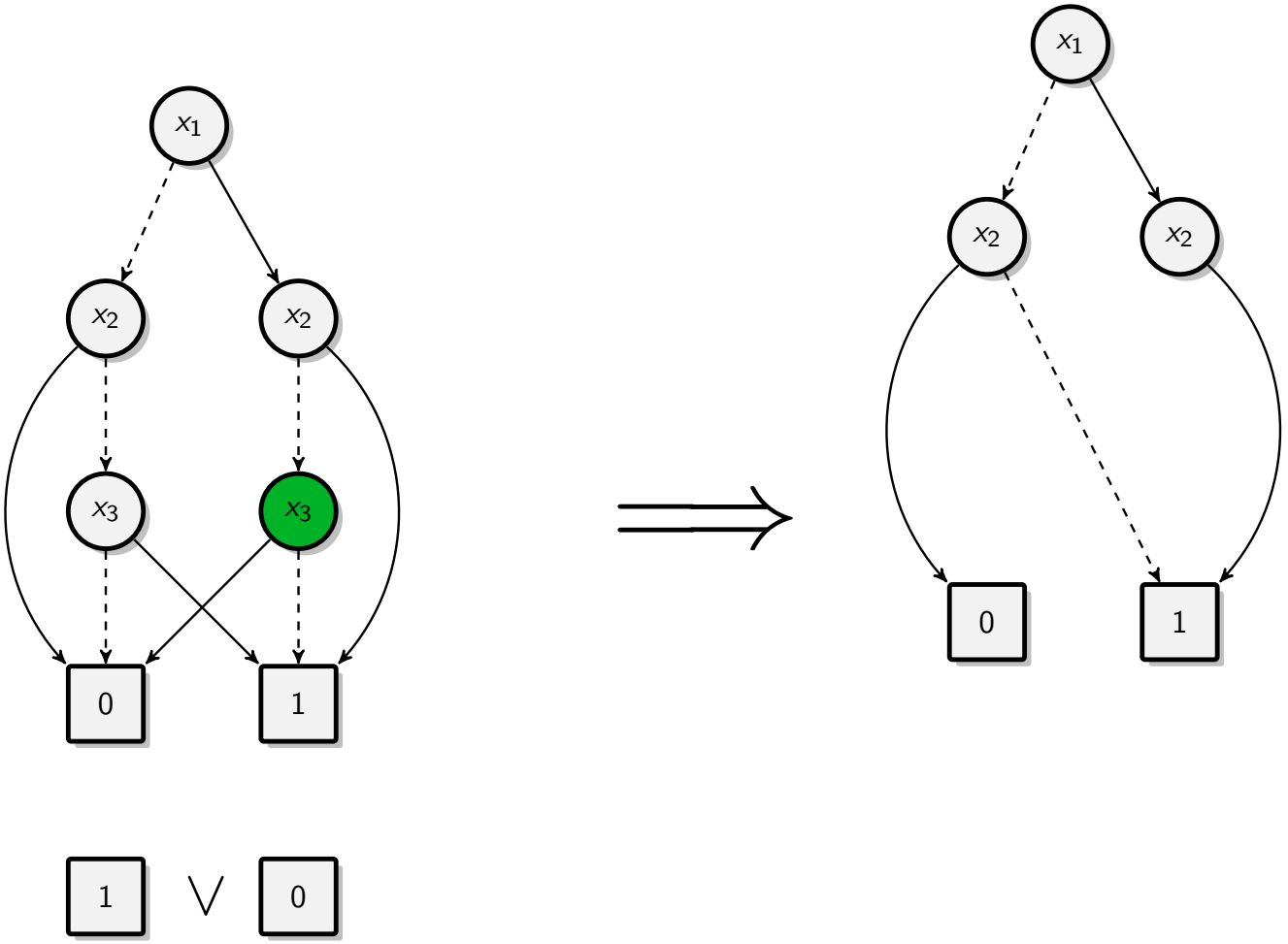
Example: Recursive Elimination of Variables

Input: a BDD G for $F = \bar{x_1}\bar{x_2}x_3 \vee x_1\bar{x_3} \vee x_1x_2$, a BDD C for x_3
Task: compute a BDD for $\exists x_3.F$.



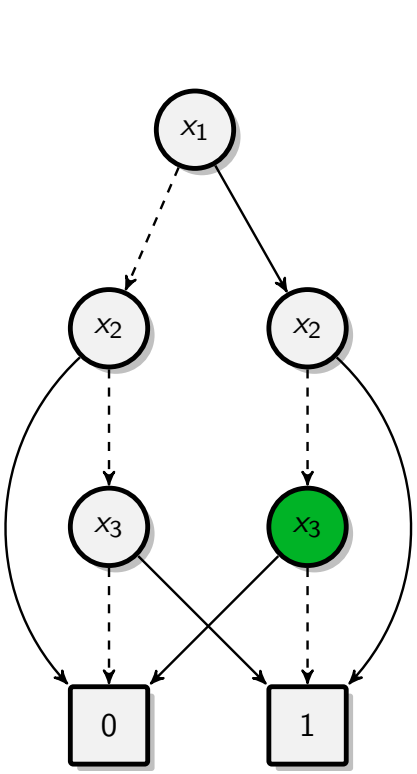
Example: Recursive Elimination of Variables

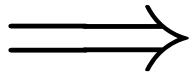
Input: a BDD G for $F = \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_3 \vee x_1x_2$, a BDD C for x_3
Task: compute a BDD for $\exists x_3.F$.

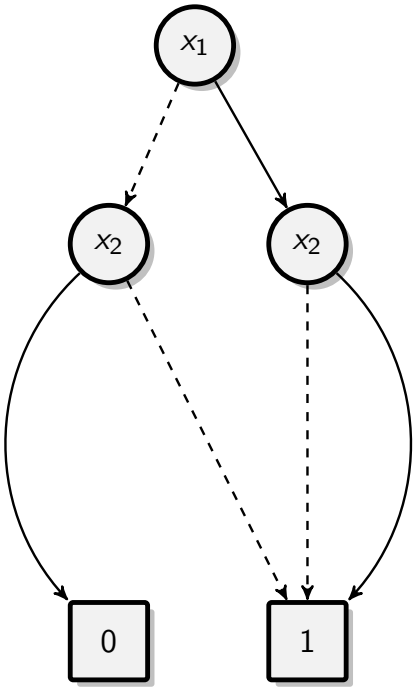


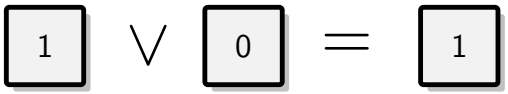
Example: Recursive Elimination of Variables

Input: a BDD G for $F = \bar{x_1}\bar{x_2}x_3 \vee x_1\bar{x_3} \vee x_1x_2$, a BDD C for x_3
Task: compute a BDD for $\exists x_3.F$.



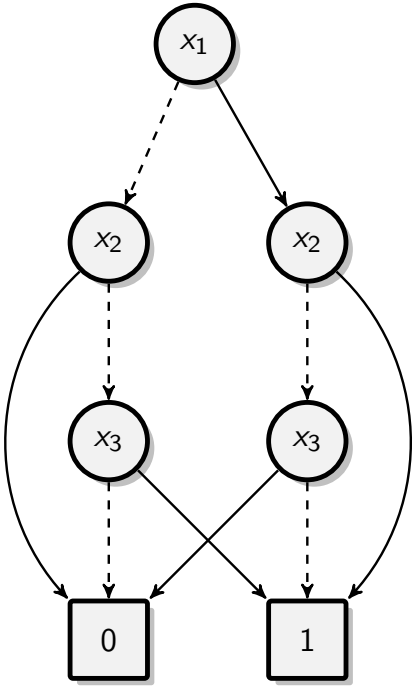






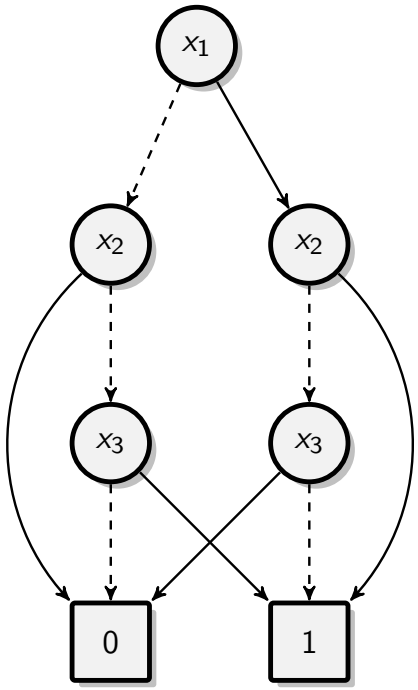
Example: Recursive Elimination of Variables

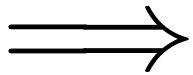
Input: a BDD G for $F = \bar{x_1}\bar{x_2}x_3 \vee x_1\bar{x_3} \vee x_1x_2$, a BDD C for x_2
Task: compute a BDD for $\exists x_2.F$.



Example: Recursive Elimination of Variables

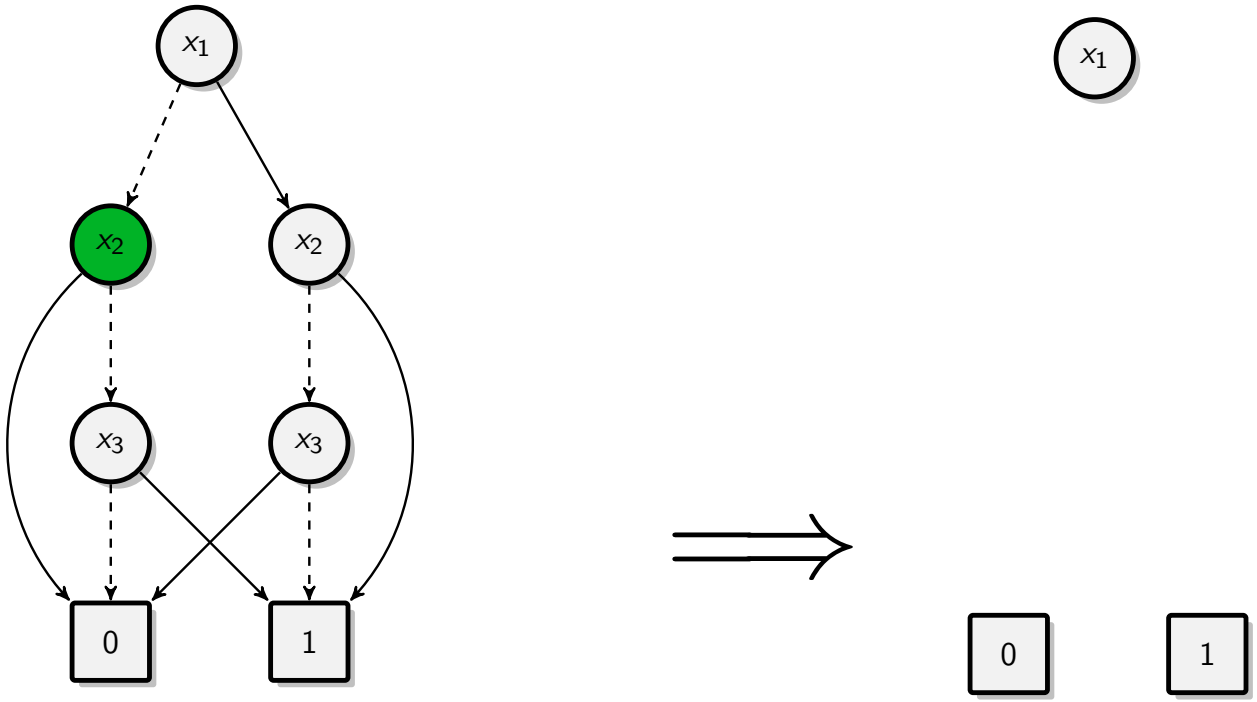
Input: a BDD G for $F = \bar{x_1}\bar{x_2}x_3 \vee x_1\bar{x_3} \vee x_1x_2$, a BDD C for x_2
Task: compute a BDD for $\exists x_2.F$.





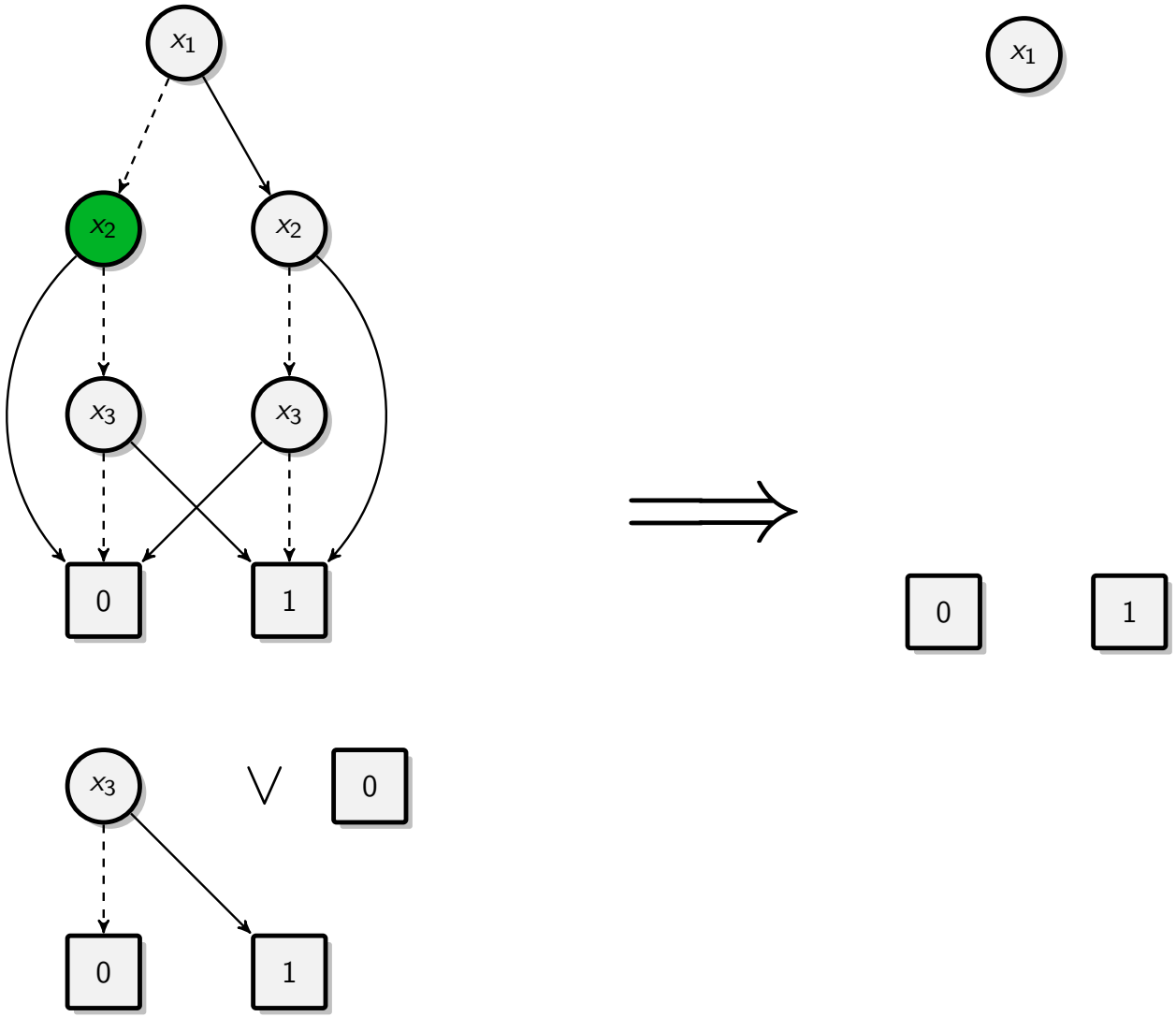
Example: Recursive Elimination of Variables

Input: a BDD G for $F = \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_3 \vee x_1x_2$, a BDD C for x_2
Task: compute a BDD for $\exists x_2.F$.



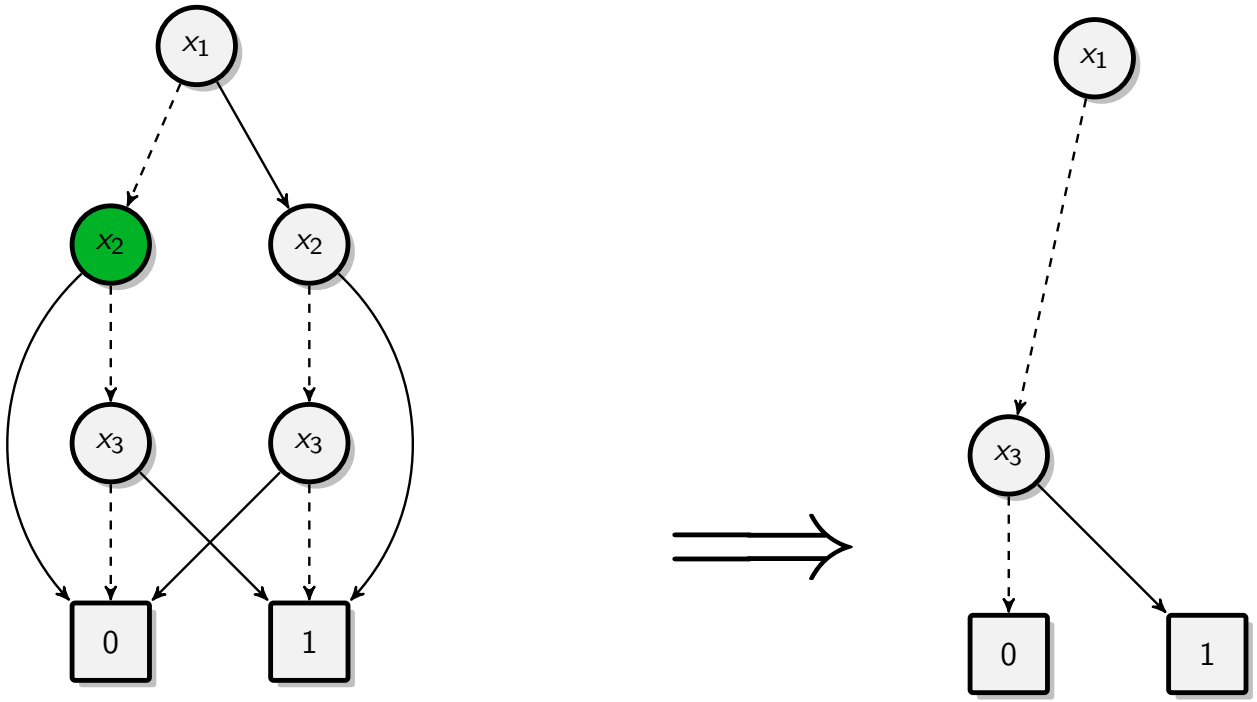
Example: Recursive Elimination of Variables

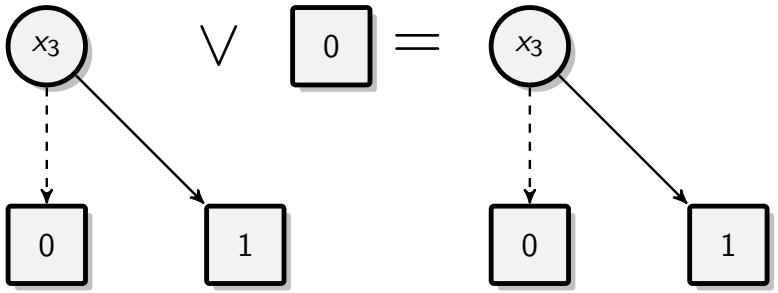
Input: a BDD G for $F = \bar{x_1}\bar{x_2}x_3 \vee x_1\bar{x_3} \vee x_1x_2$, a BDD C for x_2
Task: compute a BDD for $\exists x_2.F$.



Example: Recursive Elimination of Variables

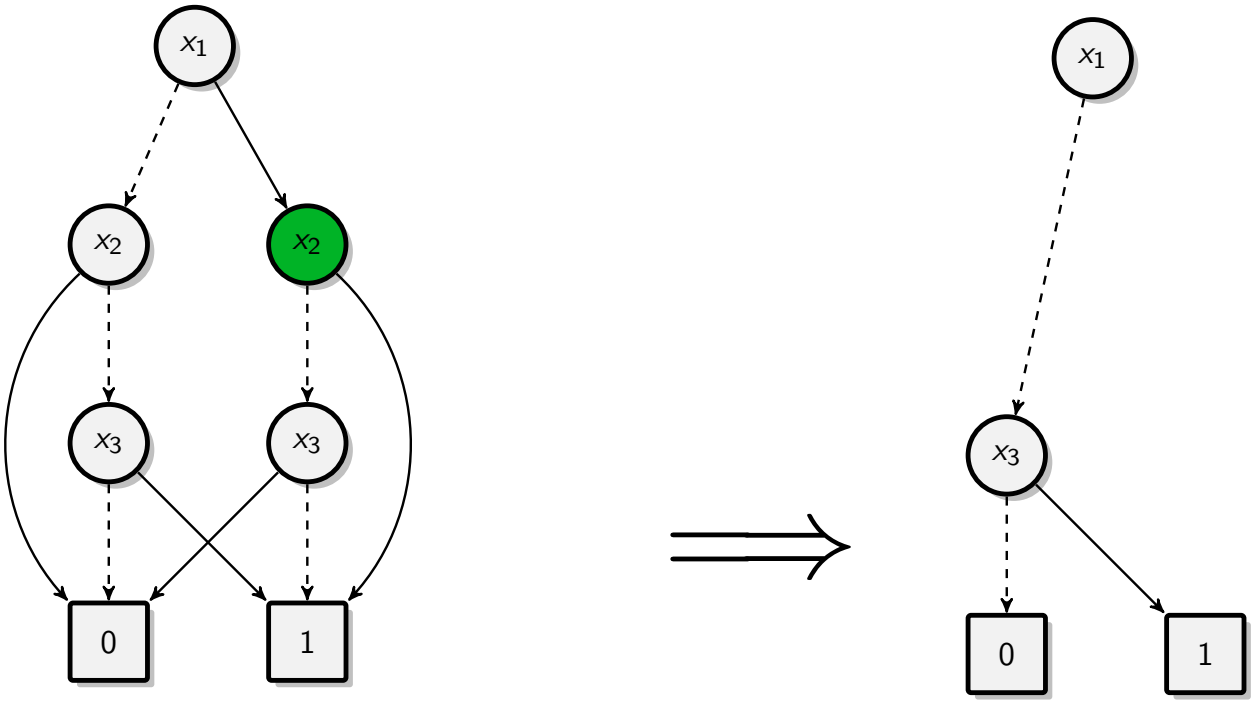
Input: a BDD G for $F = \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_3 \vee x_1x_2$, a BDD C for x_2
Task: compute a BDD for $\exists x_2.F$.





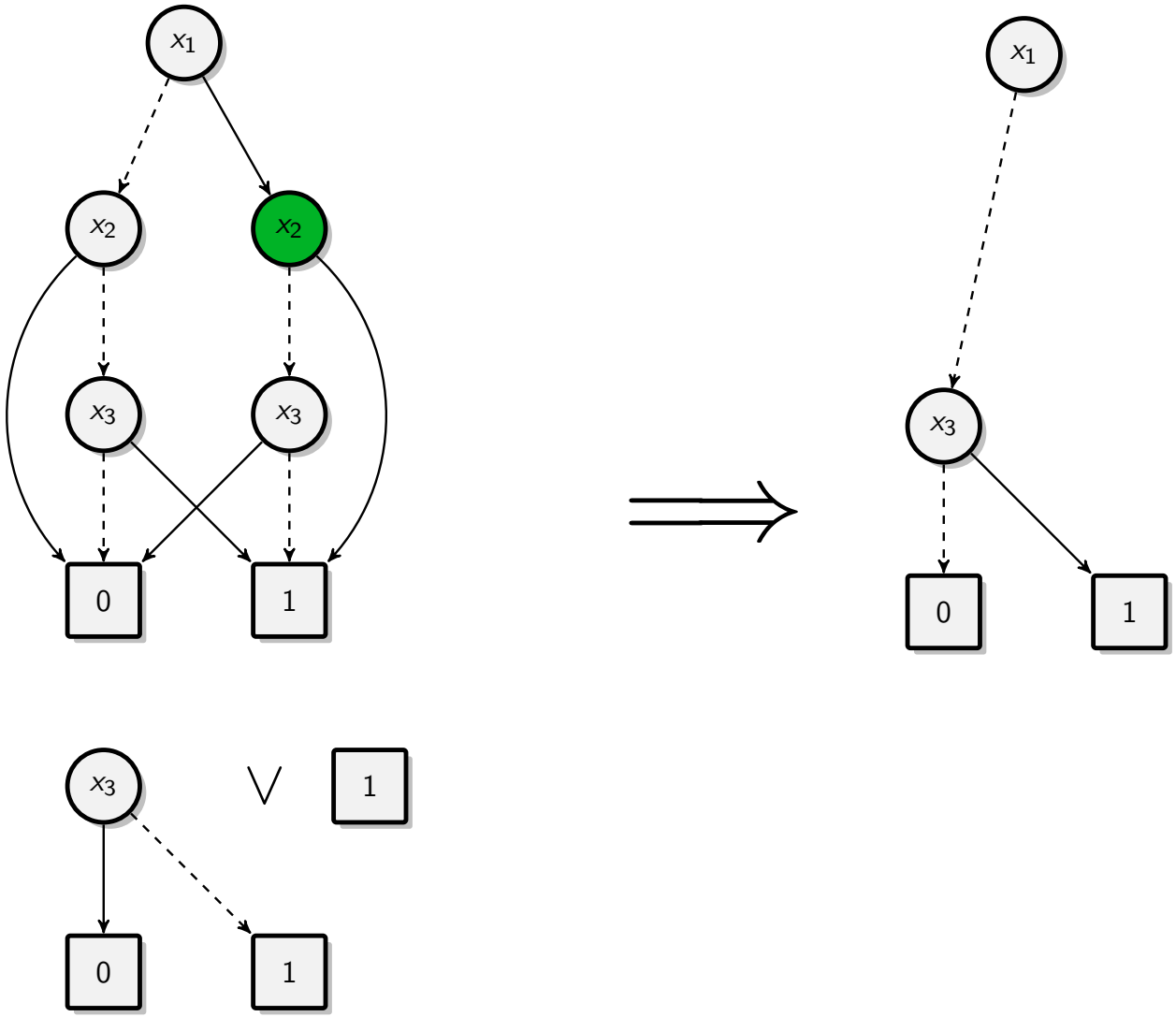
Example: Recursive Elimination of Variables

Input: a BDD G for $F = \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_3 \vee x_1x_2$, a BDD C for x_2
Task: compute a BDD for $\exists x_2.F$.



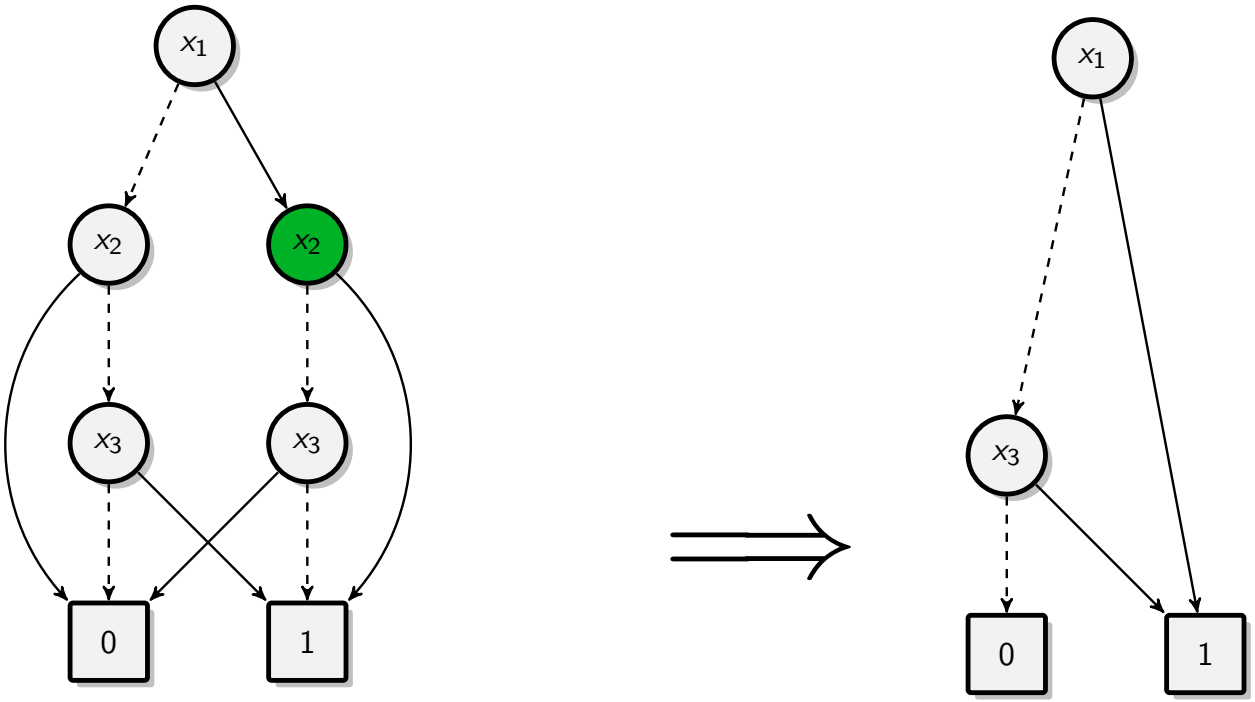
Example: Recursive Elimination of Variables

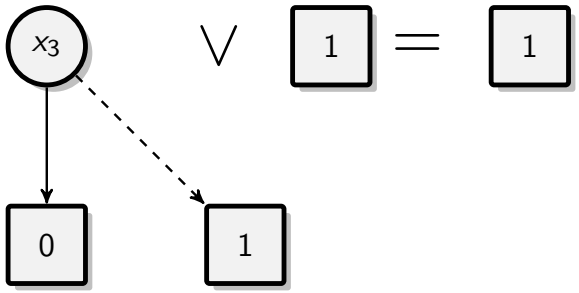
Input: a BDD G for $F = \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_3 \vee x_1x_2$, a BDD C for x_2
Task: compute a BDD for $\exists x_2.F$.



Example: Recursive Elimination of Variables

Input: a BDD G for $F = \bar{x}_1\bar{x}_2x_3 \vee x_1\bar{x}_3 \vee x_1x_2$, a BDD C for x_2
Task: compute a BDD for $\exists x_2.F$.





$\vee$ 1 $=$ 1

Operations on OBDDs: Quantification

Recursive elimination of multiple variables: after a variable from C appeared, remove it from C in recursive call

Operations on OBDDs: Quantification

Recursive elimination of multiple variables: after a variable from C appeared, remove it from C in recursive call

Universal quantification: similar, but with $\wedge$ instead of $\vee$

Operations on OBDDs: Quantification

Recursive elimination of multiple variables: after a variable from C appeared, remove it from C in recursive call

Universal quantification: similar, but with $\wedge$ instead of $\vee$

Algorithm: Exercises!

Size and Ordering of BDDs

Size of BDDs

Definition

Let $G = (V, E)$ be a BDD. Then the **size** of G is the number of non-terminal nodes in V.

Remark

The size of an ROBDD depends strongly on the variable order.

Size of BDDs

Definition

Let $G = (V, E)$ be a BDD. Then the **size** of G is the number of non-terminal nodes in V.

Remark

The size of an ROBDD depends strongly on the variable order.

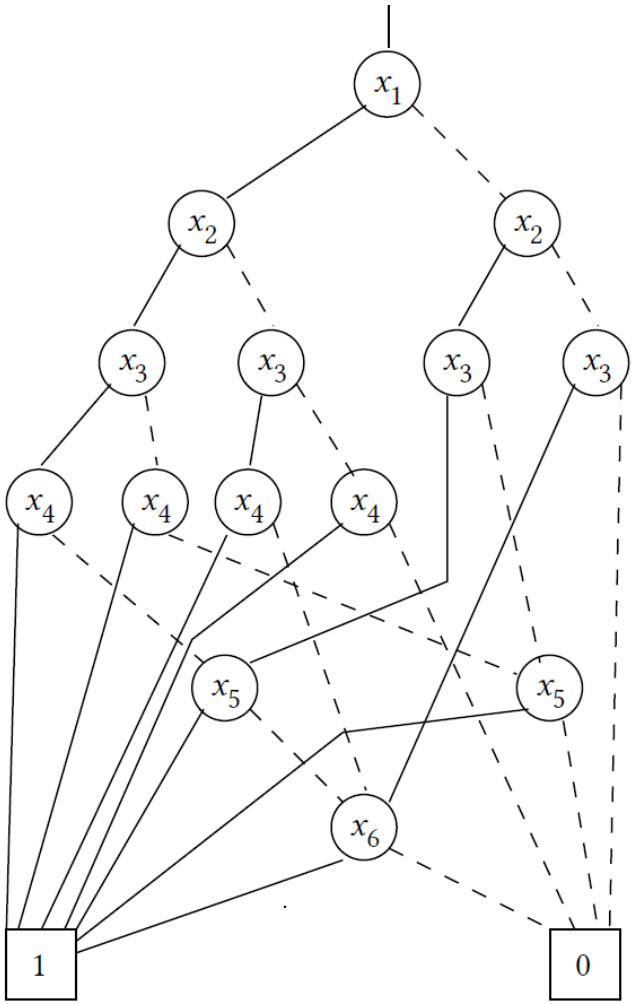
Example

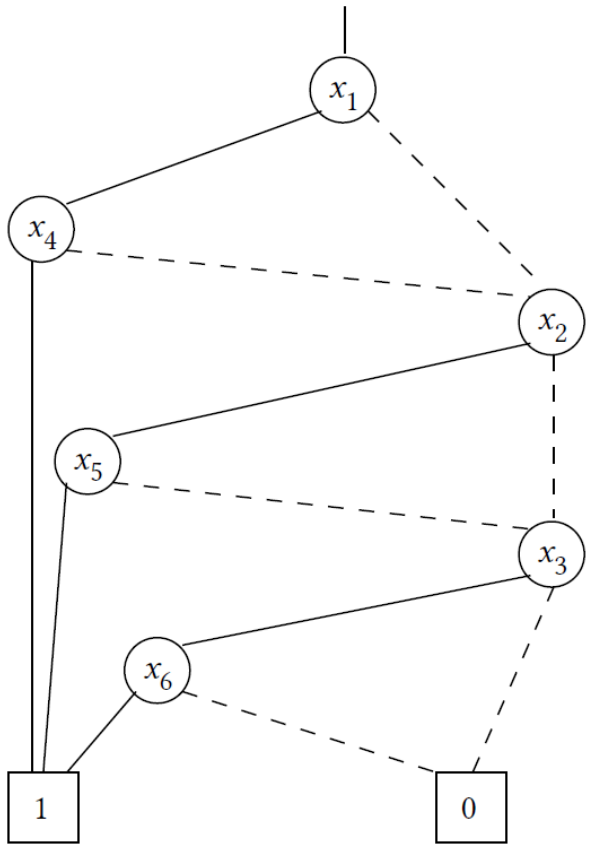
Consider formula $x_1 x_{n+1} \vee x_2 x_{n+2} \vee \ldots \vee x_n x_{2n}$.
Size of the ROBDD is

- exponential in n for variable order $x_1 > x_2 > \ldots > x_{2n}$,
- linear in n for $x_1 > x_{n+1} > x_2 > x_{n+2} > \ldots > x_n > x_{2n}$.

Example: Size of ROBDDs

ROBBDs for $x_1x_4 \vee x_2x_5 \vee x_3x_6$:





Size of ROBDDs

Theorem (Optimal Variable Order)

Given an ROBDD for formula F, the problem of finding a new variable order with minimal ROBDD-size for F is NP-complete.

Size of ROBDDs

Theorem (Optimal Variable Order)

Given an ROBDD for formula F, the problem of finding a new variable order with minimal ROBDD-size for F is NP-complete.

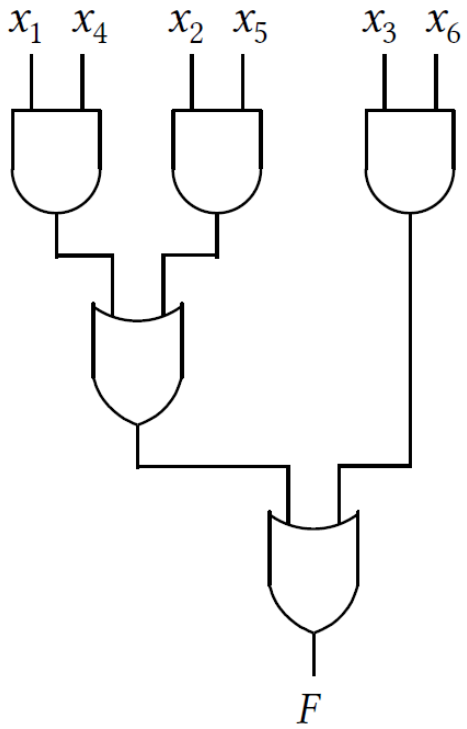
Instead: use heuristics such as

- choose an **initial variable ordering** based on properties of the input circuit or formula
- perform **dynamic reordering** to change an existing variable order and reduce the size of the ROBDD.

Obtaining an ROBDD from a (Sequential) Circuit

Procedure:

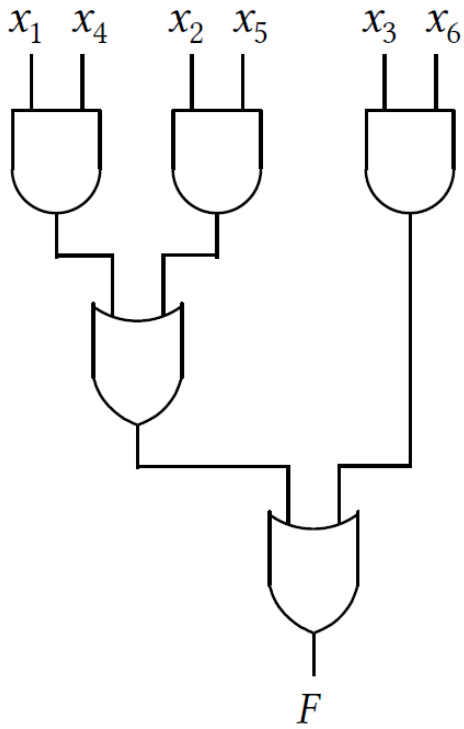
1. **For every output bit** o, compute ROBDD G_o that defines its current value, based on current values of inputs $x \in X$ and latches $l \in L$ (i.e., representing some formula $o \Leftrightarrow F_o(X, L)$)
2. **For every latch** l, compute ROBDD G_l that defines its next-state value, based on current values of inputs $x \in X$ and latches $l \in L$ (i.e., representing some formula $l' \Leftrightarrow F_{l'}(X, L)$)



Obtaining an ROBDD from a (Sequential) Circuit

Procedure:

1. **For every output bit** o, compute ROBDD G_o that defines its current value, based on current values of inputs $x \in X$ and latches $l \in L$ (i.e., representing some formula $o \Leftrightarrow F_o(X, L)$)
2. **For every latch** l, compute ROBDD G_l that defines its next-state value, based on current values of inputs $x \in X$ and latches $l \in L$ (i.e., representing some formula $l' \Leftrightarrow F_{l'}(X, L)$)



But which variable order to use in the ROBDD?

Size of BDDs: Initial Variable Order

Method of Malik et al., 1998:

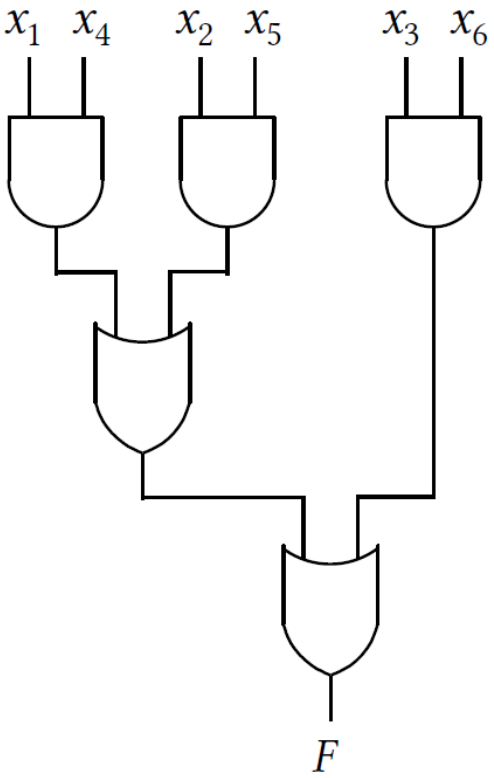
- Use the topology of the given circuit to compute an **initial variable order**:
  Carry out a depth-first search in the circuit starting from the outputs/latches, and sort the input variables according to the sequence in which they are visited

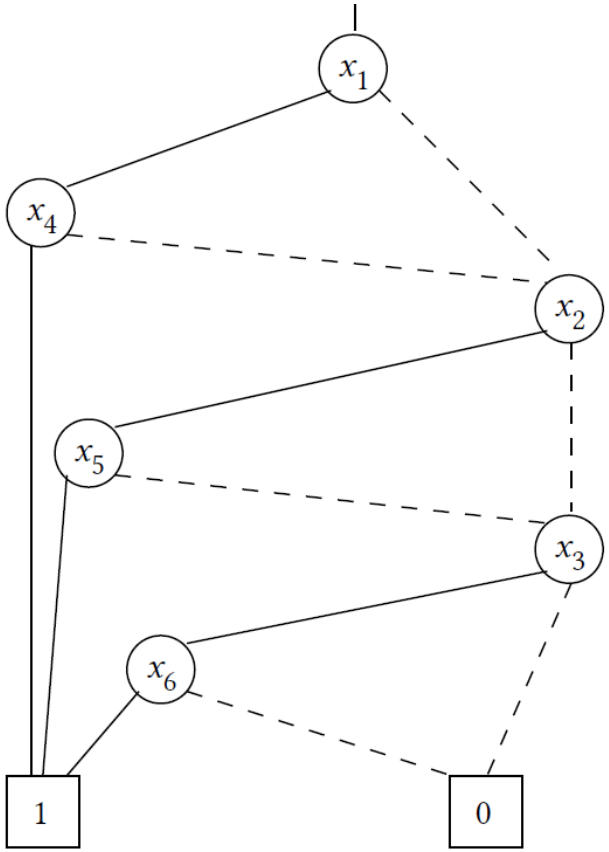
Size of BDDs: Initial Variable Order

Method of Malik et al., 1998:

- Use the topology of the given circuit to compute an **initial variable order**:
  Carry out a depth-first search in the circuit starting from the outputs/latches, and sort the input variables according to the sequence in which they are visited
- **Why this should work**: variables which are common inputs of shared components are close to each other in the resulting order.

Initial Variable Order: Example





$$F = x_1x_4 \vee x_2x_5 \vee x_3x_6$$

Dynamic Modification of Variable Order

Idea: combine initial variable order with re-ordering when necessary

Dynamic Modification of Variable Order

Idea: combine initial variable order with re-ordering when necessary

Procedure:

1. Determine an initial variable order by analysis of the circuit (or formula)

Dynamic Modification of Variable Order

Idea: combine initial variable order with re-ordering when necessary

Procedure:

1. Determine an initial variable order by analysis of the circuit (or formula)
2. Compute ROBDD that corresponds to circuit based on initial ordering

Dynamic Modification of Variable Order

Idea: combine initial variable order with re-ordering when necessary

Procedure:

1. Determine an initial variable order by analysis of the circuit (or formula)
2. Compute ROBDD that corresponds to circuit based on initial ordering
3. If during the construction (or during following computations) the size of the BDD **exceeds some threshold**, then the current ROBDD operation is interrupted and the **variable order is changed** in order to reduce the ROBDD size.

Dynamic Reordering: Sifting

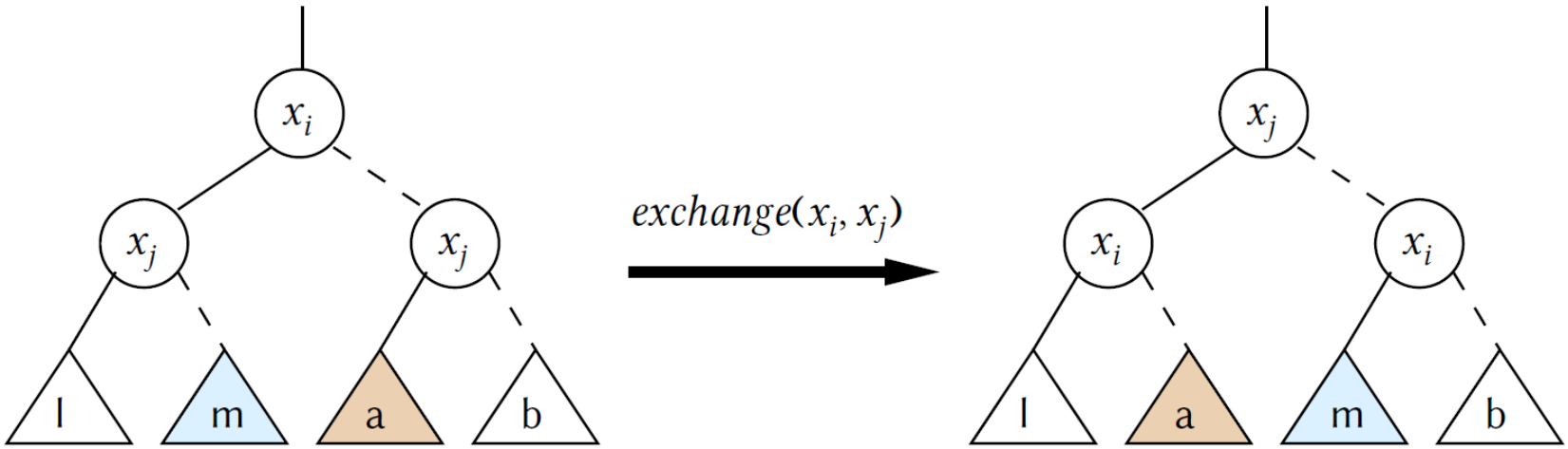
Well-known technique for dynamic re-ordering:
Sifting (Rudell, 1993)

Idea: swap pairs of variables that are next to each other in current variable order

Dynamic Reordering: Sifting

Well-known technique for dynamic re-ordering:
Sifting (Rudell, 1993)

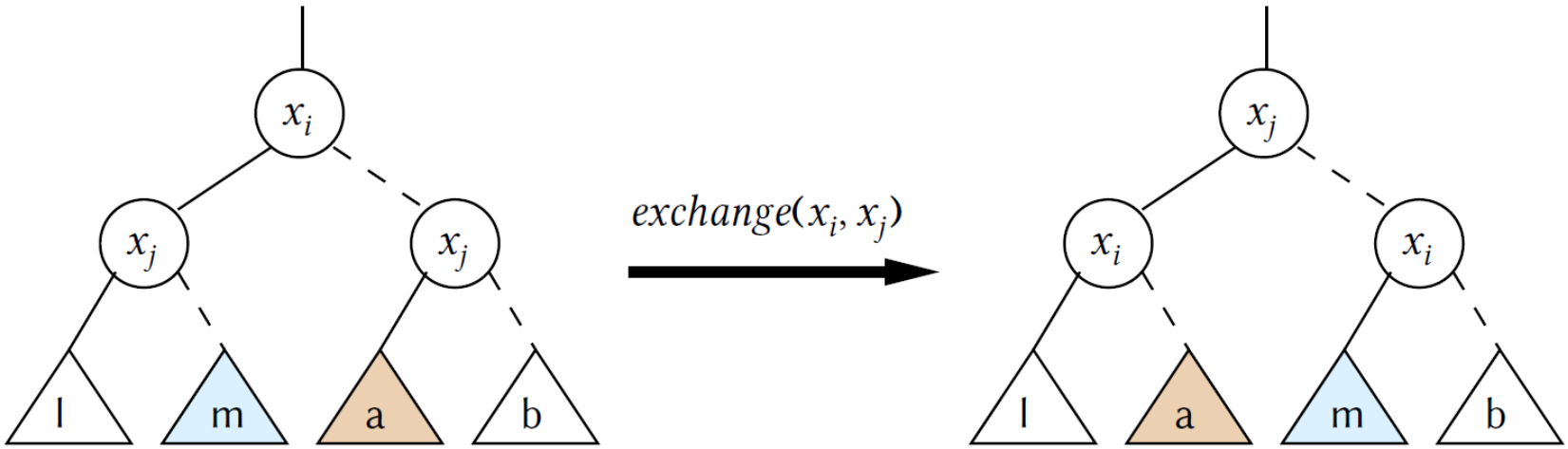
Idea: swap pairs of variables that are next to each other in current variable order



Dynamic Reordering: Sifting

Well-known technique for dynamic re-ordering:
Sifting (Rudell, 1993)

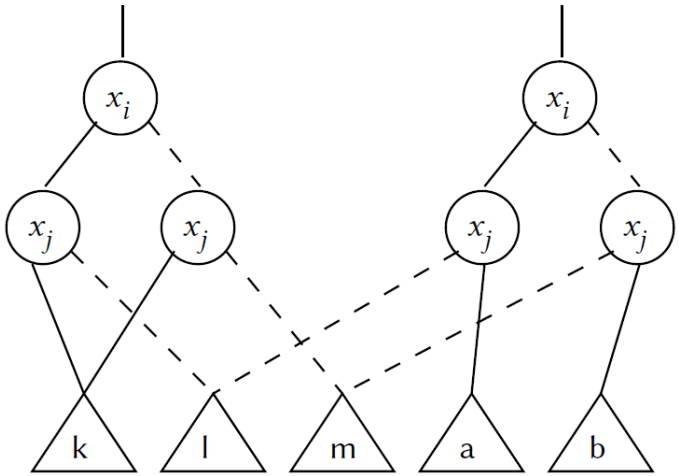
Idea: swap pairs of variables that are next to each other in current variable order

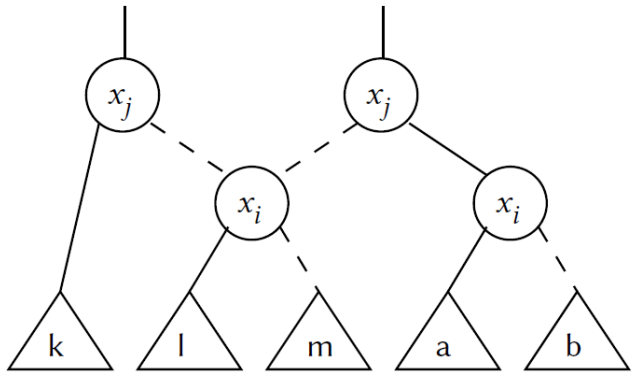


Property: variable swapping is a **local operation** that involves only the nodes that are labeled with the variables being swapped

Sifting: Example

Sifting can lead to smaller BDDs:





Size of ROBDDs: Limitations

ROBDDs provide compact representations for many practically relevant formulas. However, in some cases the minimal BDD is exponential in the number of variables:

Lemma (Bryant, 1986)

Regardless of variable order, multiplication is representable with ROBDDs only with exponential complexity in the bit-width.

Size of ROBDDs: Limitations

ROBDDs provide compact representations for many practically relevant formulas. However, in some cases the minimal BDD is exponential in the number of variables:

Lemma (Bryant, 1986)

Regardless of variable order, multiplication is representable with ROBDDs only with exponential complexity in the bit-width.

"With the exception of integer multiplication, our experience has been that such functions seldom arise in digital logic design applications." (Bryant, 1986)

Optimizations of BDD-based algorithms

Efficient Representation of OBDDs: Unique Table

To avoid keeping identical (sub-)BDDs in memory, implementations keep not only a *HashTable* for operations that have already been computed, but also a **unique table** of BDDs.

Efficient Representation of OBDDs: Unique Table

To avoid keeping identical (sub-)BDDs in memory, implementations keep not only a *HashTable* for operations that have already been computed, but also a **unique table** of BDDs.

$\Rightarrow$ **modify all recursive algorithms** that generate new BDDs: replace construction of BDD from node v and successors G_0, G_1 with call to function $FIND\_OR\_ADD\_UNIQUE\_TABLE(v, G_0, G_1)$, which returns the BDD if it already exists in the table, and otherwise generates and adds it to the table.

Efficient Representation of OBDDs: Unique Table

To avoid keeping identical (sub-)BDDs in memory, implementations keep not only a *HashTable* for operations that have already been computed, but also a **unique table** of BDDs.

$\Rightarrow$ **modify all recursive algorithms** that generate new BDDs: replace construction of BDD from node v and successors G_0, G_1 with call to function $FIND\_OR\_ADD\_UNIQUE\_TABLE(v, G_0, G_1)$, which returns the BDD if it already exists in the table, and otherwise generates and adds it to the table.

$\Rightarrow$ also affects sorting: aim is not to minimize one BDD, but the whole unique table

Substitution

In many computations, a substitution operator that allows us to "insert" one BDD into another is useful.

Definition (Substitution Operator Compose)

Let $f, g : \mathbb{B}^n \to \mathbb{B}$ be two boolean functions over the set of variables $X = \{x_1, \ldots, x_n\}$. Then the result of the substitution operator **compose** for f, g and $x_i \in X$ is defined as follows:

$$compose(f, g, x_i)(x_1, ..., x_n) = f(x_1, ..., x_{i-1}, g(x_1, ..., x_n), x_{i+1}, ..., x_n)$$

Also written as: $f(x_1, ..., x_n)[x_i \leftarrow g(x_1, ..., x_n)]$.

Substitution

In many computations, a substitution operator that allows us to "insert" one BDD into another is useful.

Definition (Substitution Operator Compose)

Let $f, g : \mathbb{B}^n \to \mathbb{B}$ be two boolean functions over the set of variables $X = \{x_1, \ldots, x_n\}$. Then the result of the substitution operator **compose** for f, g and $x_i \in X$ is defined as follows:

$$compose(f, g, x_i)(x_1, ..., x_n) = f(x_1, ..., x_{i-1}, g(x_1, ..., x_n), x_{i+1}, ..., x_n)$$

Also written as: $f(x_1, ..., x_n)[x_i \leftarrow g(x_1, ..., x_n)]$.

A generalization is **vectorCompose**, which takes as input $f, \vec{g}, \vec{x}$ with $\vec{g} = (g_1, ..., g_m)$ and $\vec{x} = (x_{i_1}, ..., x_{i_m})$, and returns the simultaneous substitution of each x_{i_j} with the function g_j in f, also written as $f(x_1, ...x_n)[\forall x_{i_j} \in \vec{x}.\ x_{i_j} \leftarrow g_j(x_1, ..., x_n)]$.

Computation of Transitions

The transition relation for a circuit with inputs X and latches L can be represented as a BDD for formula $T(L, X, L') = \bigwedge_{l \in L} l' \Leftrightarrow F_{l'}(X, L)$.

Then, for a set of states $S \subseteq \mathbb{B}^{L'}$ (given as a formula $F_S(L')$), we can compute the pre-image as the set $Pre(S) \subseteq \mathbb{B}^{L}$ that satisfies $Pre(S) = \exists X, L'.F_S(L') \wedge T(L, X, L')$.

Computation of Transitions

The transition relation for a circuit with inputs X and latches L can be represented as a BDD for formula
$T(L, X, L') = \bigwedge_{l \in L} l' \Leftrightarrow F_{l'}(X, L)$.

Then, for a set of states $S \subseteq \mathbb{B}^{L'}$ (given as a formula $F_S(L')$), we can compute the pre-image as the set $Pre(S) \subseteq \mathbb{B}^{L}$ that satisfies $Pre(S) = \exists X, L'.F_S(L') \wedge T(L, X, L')$.

Example

Let $X = \{u_0, c_0\}$, $L = \{l_0, l_1, l_2\}$, and
$T(L, X, L') = (l_0' \Leftrightarrow u_0 \wedge \bar{c_0}) \wedge (l_1' \Leftrightarrow c_0) \wedge (l_2' \Leftrightarrow l_0)$.

For $S = \bar{l_0'} \bar{l_1'} l_2'$, we compute $Pre(S) = \exists X, L'.F_S(L') \wedge T(L, X, L')$.

Computation of Transitions

The transition relation for a circuit with inputs X and latches L can be represented as a BDD for formula
$T(L, X, L') = \bigwedge_{l \in L} l' \Leftrightarrow F_{l'}(X, L)$.

Then, for a set of states $S \subseteq \mathbb{B}^{L'}$ (given as a formula $F_S(L')$), we can compute the pre-image as the set $Pre(S) \subseteq \mathbb{B}^{L}$ that satisfies $Pre(S) = \exists X, L'.F_S(L') \wedge T(L, X, L')$.

Example

Let $X = \{u_0, c_0\}$, $L = \{l_0, l_1, l_2\}$, and
$T(L, X, L') = (l'_0 \Leftrightarrow u_0 \wedge \bar{c_0}) \wedge (l'_1 \Leftrightarrow c_0) \wedge (l'_2 \Leftrightarrow l_0)$.

For $S = \bar{l'_0} \bar{l'_1} l'_2$, we compute $Pre(S) = \exists X, L'.F_S(L') \wedge T(L, X, L')$.

Problem: this may be costly since it involves global manipulation of potentially large BDDs.

Substitution in Computation of Transitions

With *compose*, we can instead keep a **functional transition relation**: for every $l \in L$, keep a BDD that defines the update for each latch, $F_{l'}(X, L)$. Then, we can directly substitute each l' against its update definition when computing the pre-image: $Pre(S) = \exists X. F_S(L')[\forall l' \in L'.\ l' \leftarrow F_{l'}(X, L)]$

Substitution in Computation of Transitions

With *compose*, we can instead keep a **functional transition relation**: for every $l \in L$, keep a BDD that defines the update for each latch, $F_{l'}(X, L)$. Then, we can directly substitute each l' against its update definition when computing the pre-image: $Pre(S) = \exists X. F_S(L')[\forall l' \in L'.\ l' \leftarrow F_{l'}(X, L)]$

Benefits:

- many small BDDs instead of few big BDDs
- there is no BDD that contains variables from both L and L'
- many local operations instead of few global operations
- since all variables in L' are substituted (by terms without variables from L'), quantification is only needed for X.

Reactive Synthesis based on BDDs

Reactive Synthesis based on BDDs

Consider synthesis problems defined by circuits.

A standard format are And-Inverter-Graphs (AIGs, also used in hardware model checking): circuits defined only by inputs X, latches L, AND-gates, and negations (aka. inverters).

Reactive Synthesis based on BDDs

Consider synthesis problems defined by circuits.

A standard format are And-Inverter-Graphs (AIGs, also used in hardware model checking): circuits defined only by inputs X, latches L, AND-gates, and negations (aka. inverters).

To obtain a symbolic two-player game from a circuit:

- partition inputs X into controllable inputs X_c (Player 0) and uncontrollable inputs X_u (Player 1)
- translate the circuit into a transition relation $T(L, X_u, X_c, L')$
- define a winning condition, e.g., that a certain set of states S should never be visited.

Remark

Note that this winning condition is equivalent to the condition that a certain output of the circuit should never become 1*, since the output can defined to be* 1 *exactly for the set of states* S*.*

Correspondence of Two-Player Games and Symbolic Games

In (explicit-state) two-player games, we distinguish between states that belong to the different players.

In symbolic games, there is no such distinction, and both players make their move in one transition.

Correspondence of Two-Player Games and Symbolic Games

In (explicit-state) two-player games, we distinguish between states that belong to the different players.

In symbolic games, there is no such distinction, and both players make their move in one transition.

However, the two types of games are equivalent, since also in symbolic games one of the players has to decide first, i.e., the single transition can be divided into two steps, where always the same player moves first.

BDD-based Computation of Attractor

For symbolic safety games, we need to compute the attractor of unsafe states S, and now have defined all necessary operations.

BDD-based Computation of Attractor

For symbolic safety games, we need to compute the attractor of unsafe states S, and now have defined all necessary operations. For a set of unsafe states S, the **controllable predecessors for Player 1** are those states where there exists a valuation of the uncontrollable inputs such that for all valuations of the controllable inputs, we move to S. In formulas:

$$\mathrm{CPre}_1(S) = \exists X_u \forall X_c \exists L'.S(L') \wedge T(L, X_u, X_c, L')$$

BDD-based Computation of Attractor

For symbolic safety games, we need to compute the attractor of unsafe states S, and now have defined all necessary operations. For a set of unsafe states S, the **controllable predecessors for Player 1** are those states where there exists a valuation of the uncontrollable inputs such that for all valuations of the controllable inputs, we move to S. In formulas:

$$\mathrm{CPre}_1(S) = \exists X_u \forall X_c \exists L'.S(L') \wedge T(L, X_u, X_c, L')$$

The **attractor** of the safety game is defined as:

- $\mathrm{Attr}_1^0(S) = S$
- $\mathrm{Attr}_1^{n+1}(S) = \mathrm{Attr}_1^n(S) \vee \mathrm{CPre}_1(\mathrm{Attr}_1^n(S))$, and
- $\mathrm{Attr}_1(S) = \bigvee_{n \in \mathbb{N}} \mathrm{Attr}_1^n(S)$.

BDD-based Computation of Attractor

For symbolic safety games, we need to compute the attractor of unsafe states S, and now have defined all necessary operations. For a set of unsafe states S, the **controllable predecessors for Player 1** are those states where there exists a valuation of the uncontrollable inputs such that for all valuations of the controllable inputs, we move to S. In formulas:

$$\mathrm{CPre}_1(S) = \exists X_u \forall X_c \exists L'.S(L') \wedge T(L, X_u, X_c, L')$$

The **attractor** of the safety game is defined as:

- $\mathrm{Attr}_1^0(S) = S$
- $\mathrm{Attr}_1^{n+1}(S) = \mathrm{Attr}_1^n(S) \vee \mathrm{CPre}_1(\mathrm{Attr}_1^n(S))$, and
- $\mathrm{Attr}_1(S) = \bigvee_{n \in \mathbb{N}} \mathrm{Attr}_1^n(S)$.

In practice, a **fixpoint** of that computation can be detected by checking equivalence between $\mathrm{Attr}_1^i(S)$ and $\mathrm{Attr}_1^{i+1}(S)$.

Winning Region and Winning Strategy

The fixpoint of the attractor computation is the **winning region of Player 1**, and (since safety games are determined) the negation of the attractor is the **winning region of Player 0**.

Winning Region and Winning Strategy

The fixpoint of the attractor computation is the **winning region of Player 1**, and (since safety games are determined) the negation of the attractor is the **winning region of Player 0**.

If the initial state is in the winning region of Player 0, we know that there exists a **strategy for Player 0** to win the game.

Winning Region and Winning Strategy

The fixpoint of the attractor computation is the **winning region of Player 1**, and (since safety games are determined) the negation of the attractor is the **winning region of Player 0**.

If the initial state is in the winning region of Player 0, we know that there exists a **strategy for Player 0** to win the game.

How to find a strategy? What is a good strategy?

Organization

Next week: Project Kickoff

- a bit more theory, directly related to project
- you will get a partially implemented synthesis tool
- you implement the missing parts ...
- ... and then improve them, based on our suggestions and your own ideas
- lectures, tutorials and exercises are replaced with project work for a few weeks
- in the end, tools will compete on challenging benchmarks

Please:

- do project work in groups of 2
- register on `https://courses.react.uni-saarland.de/rs1718/` if you have not done so