形式化方法导引

第 5 章 模型检测

5.2 理论

5.2.1 Fixpoint formulation | 5.2.2 BDD Algorithm

黄文超

http://staff.ustc.edu.cn/~huangwc/fm.html

往节内容

- Model checking
  - Modeling: Transition system
  - Specification: LTL, CTL
- Tool
  - NuSMV

本节内容:

- Basic idea of checking a model: fixpoint formulation
- Classical algorithms
  - Binary decisions diagram (BDD)
  - Bounded model checking (BMC)
  - Basic Inductive Techniques

往节内容

- Model checking
    - Modeling: Transition system
    - Specification: LTL, CTL
- Tool
    - NuSMV

本节内容:

- Basic idea of checking a model: fixpoint formulation
- Classical algorithms
    - Binary decisions diagram (BDD)
    - Bounded model checking (BMC)
    - Basic Inductive Techniques

回顾：定义：Transition system

A transition system $\mathcal{M} = (S, \rightarrow, L)$ is

- S: a set of states
- $\rightarrow$: a transition relation: every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$
- L: a label function: $L : S \rightarrow \mathcal{P}(\text{Atoms})$

定义：State space

The *state space*

$$S = V_1 \times \cdots \times V_n$$

is implied by the variables $v_1, \ldots, v_n$ from finite sets $V_1, \ldots, V_n$

问题：How to check a CTL property?

问题转换：We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

回顾: 定义: Transition system

A transition system $\mathcal{M} = (S, \rightarrow, L)$ is

- S: a set of states
- $\rightarrow$: a transition relation: every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$
- L: a label function: $L : S \rightarrow \mathcal{P}(\text{Atoms})$

定义: State space

The *state space*

$$S = V_1 \times \cdots \times V_n$$

is implied by the variables $v_1, \ldots, v_n$ from finite sets $V_1, \ldots, V_n$

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

回顾: 定义: Transition system

A transition system $\mathcal{M} = (S, \to, L)$ is

- S: a set of states
- $\to$: a transition relation: every $s \in S$ has some $s' \in S$ with $s \to s'$
- L: a label function: $L : S \to \mathcal{P}(\mathrm{Atoms})$

定义: State space

The *state space*

$$S = V_1 \times \cdots \times V_n$$

is implied by the variables $v_1, \ldots, v_n$ from finite sets $V_1, \ldots, V_n$

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

回顾: 定义: Transition system

A transition system $\mathcal{M} = (S, \rightarrow, L)$ is

- S: a set of states
- $\rightarrow$: a transition relation: every $s \in S$ has some $s' \in S$ with $s \rightarrow s'$
- L: a label function: $L : S \rightarrow \mathcal{P}(\mathrm{Atoms})$

定义: State space

The *state space*

$$S = V_1 \times \cdots \times V_n$$

is implied by the variables $v_1, \ldots, v_n$ from finite sets $V_1, \ldots, V_n$

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

基本思路:

1. *Compute the set* S_ϕ consisting of all states that satisfy ϕ
   - a state $s \in S$ satisfies ϕ if the set of all paths starting in s satisfies ϕ
2. Then the property to check is $I \subseteq S_\phi$

- $S_\perp = \emptyset$
- $S_\top = S$
- For an atomic proposition p:

$$S_p = \{s \in S \mid p(s)\}$$

- $S_{\neg\phi} = \{s \in S \mid s \notin S_\phi\}$
- $S_{\phi \vee \psi} = S_\phi \cup S_\psi$
- $S_{\phi \wedge \psi} = S_\phi \cap S_\psi$

It *remains* to describe S_ϕ for ϕ containing *CTL operator EX, EG, EU*

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

基本思路:

1. *Compute the set S_ϕ* consisting of all states that satisfy ϕ
   - a state $s \in S$ satisfies ϕ if the set of all paths starting in s satisfies ϕ
2. Then the property to check is $I \subseteq S_\phi$

- $S_\perp = \emptyset$
- $S_\top = S$
- For an atomic proposition p:

$$S_p = \{s \in S \mid p(s)\}$$

- $S_{\neg\phi} = \{s \in S \mid s \notin S_\phi\}$
- $S_{\phi\vee\psi} = S_\phi \cup S_\psi$
- $S_{\phi\wedge\psi} = S_\phi \cap S_\psi$

It *remains* to describe S_ϕ for ϕ containing *CTL operator EX, EG, EU*

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

基本思路:

1. *Compute the set* S_ϕ consisting of all states that satisfy ϕ
   - a state $s \in S$ satisfies ϕ if the set of all paths starting in s satisfies ϕ
2. Then the property to check is $I \subseteq S_\phi$

- $S_\perp = \emptyset$
- $S_\top = S$
- For an atomic proposition p:

$$S_p = \{s \in S \mid p(s)\}$$

- $S_{\neg\phi} = \{s \in S \mid s \notin S_\phi\}$
- $S_{\phi\vee\psi} = S_\phi \cup S_\psi$
- $S_{\phi\wedge\psi} = S_\phi \cap S_\psi$

It *remains* to describe S_ϕ for ϕ containing *CTL operator EX, EG, EU*

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

基本思路:

1. *Compute the set* S_ϕ consisting of all states that satisfy ϕ
   - a state $s \in S$ satisfies ϕ if the set of all paths starting in s satisfies ϕ
2. Then the property to check is $I \subseteq S_\phi$

- $S_\bot = \emptyset$
- $S_\top = S$
- For an atomic proposition p:

$$S_p = \{s \in S \mid p(s)\}$$

- $S_{\neg\phi} = \{s \in S \mid s \notin S_\phi\}$
- $S_{\phi\vee\psi} = S_\phi \cup S_\psi$
- $S_{\phi\wedge\psi} = S_\phi \cap S_\psi$

It *remains* to describe S_ϕ for ϕ containing *CTL operator EX, EG, EU*

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

基本思路:

1. *Compute the set S_ϕ* consisting of all states that satisfy ϕ
   - a state $s \in S$ satisfies ϕ if the set of all paths starting in s satisfies ϕ
2. Then the property to check is $I \subseteq S_\phi$

- $S_\perp = \emptyset$
- $S_\top = S$
- For an atomic proposition p:

$$S_p = \{s \in S \mid p(s)\}$$

- $S_{\neg\phi} = \{s \in S \mid s \notin S_\phi\}$
- $S_{\phi\vee\psi} = S_\phi \cup S_\psi$
- $S_{\phi\wedge\psi} = S_\phi \cap S_\psi$

It *remains* to describe S_ϕ for ϕ containing *CTL operator EX, EG, EU*

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

基本思路:

1. *Compute the set S_ϕ* consisting of all states that satisfy ϕ
   - a state $s \in S$ satisfies ϕ if the set of all paths starting in s satisfies ϕ
2. Then the property to check is $I \subseteq S_\phi$

- $S_\perp = \emptyset$
- $S_\top = S$
- For an atomic proposition p:

$$S_p = \{s \in S \mid p(s)\}$$

- $S_{\neg\phi} = \{s \in S \mid s \notin S_\phi\}$
- $S_{\phi\vee\psi} = S_\phi \cup S_\psi$
- $S_{\phi\wedge\psi} = S_\phi \cap S_\psi$

It *remains* to describe S_ϕ for ϕ containing *CTL operator EX, EG, EU*

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

基本思路:

1. *Compute the set* S_ϕ consisting of all states that satisfy ϕ
   - a state $s \in S$ satisfies ϕ if the set of all paths starting in s satisfies ϕ
2. Then the property to check is $I \subseteq S_\phi$

- $S_\bot = \emptyset$
- $S_\top = S$
- For an atomic proposition p:

$$S_p = \{s \in S \mid p(s)\}$$

- $S_{\neg\phi} = \{s \in S \mid s \notin S_\phi\}$
- $S_{\phi\vee\psi} = S_\phi \cup S_\psi$
- $S_{\phi\wedge\psi} = S_\phi \cap S_\psi$

It *remains* to describe S_ϕ for ϕ containing *CTL operator EX, EG, EU*

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

基本思路:

1. *Compute the set S_ϕ* consisting of all states that satisfy ϕ
   - a state $s \in S$ satisfies ϕ if the set of all paths starting in s satisfies ϕ
2. Then the property to check is $I \subseteq S_\phi$

- $S_\perp = \emptyset$
- $S_\top = S$
- For an atomic proposition p:

$$S_p = \{s \in S \mid p(s)\}$$

- $S_{\neg\phi} = \{s \in S \mid s \notin S_\phi\}$
- $S_{\phi\vee\psi} = S_\phi \cup S_\psi$
- $S_{\phi\wedge\psi} = S_\phi \cap S_\psi$

It *remains* to describe S_ϕ for ϕ containing *CTL operator EX, EG, EU*

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

基本思路:

1. *Compute the set S_ϕ* consisting of all states that satisfy ϕ
   - a state $s \in S$ satisfies ϕ if the set of all paths starting in s satisfies ϕ
2. Then the property to check is $I \subseteq S_\phi$

- $S_\perp = \emptyset$
- $S_\top = S$
- For an atomic proposition p:

$$S_p = \{s \in S \mid p(s)\}$$

- $S_{\neg\phi} = \{s \in S \mid s \notin S_\phi\}$
- $S_{\phi\vee\psi} = S_\phi \cup S_\psi$
- $S_{\phi\wedge\psi} = S_\phi \cap S_\psi$

It *remains* to describe S_ϕ for ϕ containing *CTL operator EX, EG, EU*

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

基本思路:

1. *Compute the set S_ϕ* consisting of all states that satisfy ϕ
   - a state $s \in S$ satisfies ϕ if the set of all paths starting in s satisfies ϕ
2. Then the property to check is $I \subseteq S_\phi$

- $S_\bot = \emptyset$
- $S_\top = S$
- For an atomic proposition p:

$$S_p = \{s \in S \mid p(s)\}$$

- $S_{\neg\phi} = \{s \in S \mid s \notin S_\phi\}$
- $S_{\phi\vee\psi} = S_\phi \cup S_\psi$
- $S_{\phi\wedge\psi} = S_\phi \cap S_\psi$

It *remains* to describe S_ϕ for ϕ containing *CTL operator EX, EG, EU*

问题: How to check a CTL property?

问题转换: We want to check whether a CTL formula ϕ holds for all states $s \in I$ for a given set of initial states $I \subseteq S$

基本思路:

1. *Compute the set S_ϕ* consisting of all states that satisfy ϕ
   - a state $s \in S$ satisfies ϕ if the set of all paths starting in s satisfies ϕ
2. Then the property to check is $I \subseteq S_\phi$

- $S_\bot = \emptyset$
- $S_\top = S$
- For an atomic proposition p:

$$S_p = \{s \in S \mid p(s)\}$$

- $S_{\neg\phi} = \{s \in S \mid s \notin S_\phi\}$
- $S_{\phi\vee\psi} = S_\phi \cup S_\psi$
- $S_{\phi\wedge\psi} = S_\phi \cap S_\psi$

It *remains* to describe S_ϕ for ϕ containing *CTL operator EX, EG, EU*

A state s satisfies *EX* ϕ, if there *exists* a path starting in s such that ϕ holds in the *next* state of that path. So:

$$S_{\mathrm{EX}\phi} = \{s \in S \mid \exists t \in \underline{S_\phi} : s \to t\}$$

EG and EU are harder: they deal with properties of paths *beyond a fixed finite part of the path*

思路:

Consider the first n steps for *increasing* n, *until* the corresponding set *does not change* anymore

A state s satisfies *EX* ϕ, if there *exists* a path starting in s such that ϕ holds in the *next* state of that path. So:

$$S_{\mathrm{EX}\phi} = \{s \in S \mid \exists t \in \underline{S_\phi} : s \to t\}$$

EG and EU are harder: they deal with properties of paths *beyond a fixed finite part of the path*

思路:

Consider the first n steps for *increasing* n, *until* the corresponding set *does not change* anymore

A state s satisfies *EX* ϕ, if there *exists* a path starting in s such that ϕ holds in the *next* state of that path. So:

$$S_{\mathrm{EX}\phi} = \{s \in S \mid \exists t \in \underline{S_\phi} : s \to t\}$$

EG and EU are harder: they deal with properties of paths *beyond a fixed finite part of the path*

思路*:*
Consider the first n steps for *increasing* n, *until* the corresponding set *does not change* anymore

$\mathrm{EG}\phi$ means: there *exists* a path $s_0 \to s_1 \to s_2 \cdots$ on which ϕ globally holds, that is, ϕ holds in s_i *for all i*

定义: T_n

For $n = 0, 1, 2, \ldots,$ let T_n = set of states s_0, for which there exists a path $s_0 \to s_1 \to s_2 \cdots$ on which ϕ holds *for all s_i with $i \le n$*

Then $T_0 = S_\phi$, and for all $n = 0, 1, \ldots,$ we have

$$T_{n+1} = T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$$

算法: 求解 T_n, i.e., $S_{\mathrm{EG}\phi} = T_n$ (Fixpoint formulation)

$T_0 := S_\phi$; $n := 0$;
repeat
  $T_{n+1} := T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$; $n = n + 1$;
until $T_n = T_{n-1}$

$\mathrm{EG}\phi$ means: there *exists* a path $s_0 \to s_1 \to s_2 \cdots$ on which ϕ globally holds, that is, ϕ holds in s_i *for all i*

定义: T_n

For $n = 0, 1, 2, \ldots$, let T_n = set of states s_0, for which there exists a path $s_0 \to s_1 \to s_2 \cdots$ on which ϕ holds *for all s_i with $i \le n$*

Then $T_0 = S_\phi$, and for all $n = 0, 1, \ldots$, we have

$$T_{n+1} = T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$$

算法: 求解 T_n, i.e., $S_{\mathrm{EG}\phi} = T_n$ (Fixpoint formulation)

$T_0 := S_\phi$; $n := 0$;
repeat
  $T_{n+1} := T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$; $n = n + 1$;
until $T_n = T_{n-1}$

$\mathrm{EG}\phi$ means: there *exists* a path $s_0 \to s_1 \to s_2 \cdots$ on which ϕ globally holds, that is, ϕ holds in s_i *for all i*

定义: T_n

For $n = 0, 1, 2, \ldots,$ let T_n = set of states s_0, for which there exists a path $s_0 \to s_1 \to s_2 \cdots$ on which ϕ holds *for all s_i with $i \leq n$*

Then $T_0 = S_\phi$, and for all $n = 0, 1, \ldots,$ we have

$$T_{n+1} = T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$$

算法: 求解 T_n, i.e., $S_{\mathrm{EG}\phi} = T_n$ (Fixpoint formulation)

$T_0 := S_\phi$; $n := 0$;

repeat

  $T_{n+1} := T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$; $n = n + 1$;

until $T_n = T_{n-1}$

$\mathrm{EG}\phi$ means: there *exists* a path $s_0 \to s_1 \to s_2 \cdots$ on which ϕ globally holds, that is, ϕ holds in s_i *for all i*

定义: T_n

For $n = 0, 1, 2, \ldots$, let T_n = set of states s_0, for which there exists a path $s_0 \to s_1 \to s_2 \cdots$ on which ϕ holds *for all s_i with $i \leq n$*

Then $T_0 = S_\phi$, and for all $n = 0, 1, \ldots$, we have

$$T_{n+1} = T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$$

算法: 求解 T_n, i.e., $S_{\mathrm{EG}\phi} = T_n$ (Fixpoint formulation)

$T_0 := S_\phi$; $n := 0$;

repeat

$\quad T_{n+1} := T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$; $n = n + 1$;

until $T_n = T_{n-1}$

算法: 求解 $S_{\mathrm{EG}\phi} = T_n$

$T_0 := S_\phi$; $n := 0$;
repeat
    $T_{n+1} := T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$; $n = n + 1$;
until $T_n = T_{n-1}$

The loop *terminates*, since

- the set T_n is finite
- $|T_n|$ decreases in every step

After running this algorithm we have $S_{\mathrm{EG}\phi} = T_n$

算法: 求解 $S_{\mathrm{EG}\phi} = T_n$

$T_0 := S_\phi$; $n := 0$;
repeat
  $T_{n+1} := T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$; $n = n + 1$;
until $T_n = T_{n-1}$

The loop *terminates*, since

- the set T_n is finite
- $|T_n|$ decreases in every step

After running this algorithm we have $S_{\mathrm{EG}\phi} = T_n$

算法: 求解 $S_{\mathrm{EG}\phi} = T_n$

$T_0 := S_\phi;\ n := 0;$
repeat
  $T_{n+1} := T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\};\ n = n + 1;$
until $T_n = T_{n-1}$

The loop *terminates*, since

- the set T_n is finite
- $|T_n|$ decreases in every step

After running this algorithm we have $S_{\mathrm{EG}\phi} = T_n$

s_0 satisfies $E[\phi\ U\ \psi]$ means: there *exists* n such that P_n holds, for

- P_n = there *exists* a path $s_0 \to s_1 \to s_2 \cdots$ on which ψ holds in s_n, and ϕ holds in s_i for all $i < n$

定义： U_n

U_n = set of states s_0 for which P_i holds for *some* $i \leq n$

Then $U_0 = S_\psi$, and for all n=0,1,..., we have

$$U_{n+1} = U_n \cup \{s \in S_\phi \mid \exists t \in U_n : s \to t\}$$

算法: 求解 $S_{\mathrm{E}[\phi \mathrm{U} \psi]} = U_n$

$U_0 := S_\psi; n := 0;$
repeat
  $U_{n+1} := U_n \cup \{s \in S_\phi \mid \exists t \in U_n : s \to t\}; n = n + 1;$
until $U_n = U_{n-1}$

s_0 satisfies $E[\phi \ U \ \psi]$ means: there *exists* n such that P_n holds, for

- P_n = there *exists* a path $s_0 \to s_1 \to s_2 \cdots$ on which ψ holds in s_n, and ϕ holds in s_i for all $i < n$

定义： U_n

U_n = set of states s_0 for which P_i holds for *some* $i \leq n$

Then $U_0 = S_\psi$, and for all n=0,1,..., we have

$$U_{n+1} = U_n \cup \{s \in S_\phi \mid \exists t \in U_n : s \to t\}$$

算法：求解 $S_{\mathrm{E}[\phi \mathrm{U} \psi]} = U_n$

$U_0 := S_\psi; n := 0;$
repeat
$\quad U_{n+1} := U_n \cup \{s \in S_\phi \mid \exists t \in U_n : s \to t\}; n = n + 1;$
until $U_n = U_{n-1}$

s_0 satisfies $E[\phi\ U\ \psi]$ means: there *exists* n such that P_n holds, for

- P_n = there *exists* a path $s_0 \to s_1 \to s_2 \cdots$ on which ψ holds in s_n, and ϕ holds in s_i for all $i < n$

定义： U_n

U_n = set of states s_0 for which P_i holds for *some* $i \le n$

Then $U_0 = S_\psi$, and for all n=0,1,..., we have

$$U_{n+1} = U_n \cup \{s \in S_\phi \mid \exists t \in U_n : s \to t\}$$

算法：求解 $S_{\mathrm{E}[\phi \mathrm{U} \psi]} = U_n$

$U_0 := S_\psi; n := 0;$
repeat
  $U_{n+1} := U_n \cup \{s \in S_\phi \mid \exists t \in U_n : s \to t\}; n = n + 1;$
until $U_n = U_{n-1}$

s_0 satisfies $E[\phi\ U\ \psi]$ means: there *exists* n such that P_n holds, for

- P_n = there *exists* a path $s_0 \to s_1 \to s_2 \cdots$ on which ψ holds in s_n, and ϕ holds in s_i for all $i < n$

定义： U_n

U_n = set of states s_0 for which P_i holds for *some* $i \leq n$

Then $U_0 = S_\psi$, and for all n=0,1,..., we have

$$U_{n+1} = U_n \cup \{s \in S_\phi \mid \exists t \in U_n : s \to t\}$$

算法: 求解 $S_{\mathrm{E}[\phi \mathrm{U} \psi]} = U_n$

$U_0 := S_\psi; n := 0;$
repeat
  $U_{n+1} := U_n \cup \{s \in S_\phi \mid \exists t \in U_n : s \to t\}; n = n + 1;$
until $U_n = U_{n-1}$

Concluding,

- For an *arbitrary CTL formula* ϕ we saw how to compute the set S_ϕ, being the set of states that satisfy ϕ
  - Basics: $S_\perp$, $S_\top$, S_p, $S_{\neg\phi}$, $S_{\phi\vee\psi}$, $S_{\phi\wedge\psi}$
  - CTL Related: $\underline{S_{\mathrm{EX}\phi}}$, $\underline{S_{\mathrm{EG}\phi}}$, $\underline{S_{\mathrm{EU}[\phi \mathrm{U} \psi]}}$
- In this computation we *only* needed the computation of the sets $T \cup U$, $T \cap U$, complements, and $\{s \in T \mid \exists t \in U : s \rightarrow t\}$, for given sets T, U

问题: In *explicit state based* model checking the complexity of this algorithm will be *at least the order of the size of the state space S*

可选方案: 2.2 Binary decisions diagram (BDD), 2.3 Bounded model checking (BMC), 2.4 k-induction

Concluding,

- For an *arbitrary CTL formula* ϕ we saw how to compute the set S_ϕ, being the set of states that satisfy ϕ
  - Basics: $S_\bot$, $S_\top$, S_p, $S_{\neg\phi}$, $S_{\phi\vee\psi}$, $S_{\phi\wedge\psi}$
  - CTL Related: $\underline{S_{\mathrm{EX}\phi}}$, $\underline{S_{\mathrm{EG}\phi}}$, $\underline{S_{\mathrm{EU}[\phi \mathrm{U} \psi]}}$
- In this computation we *only* needed the computation of the sets $T \cup U$, $T \cap U$, complements, and $\{s \in T \mid \exists t \in U : s \to t\}$, for given sets T, U

问题: In *explicit state based* model checking the complexity of this algorithm will be *at least the order of the size of the state space S*

可选方案: 2.2 Binary decisions diagram (BDD), 2.3 Bounded model checking (BMC), 2.4 k-induction

Concluding,

- For an *arbitrary CTL formula* ϕ we saw how to compute the set S_ϕ, being the set of states that satisfy ϕ
  - Basics: $S_\perp$, $S_\top$, S_p, $S_{\neg\phi}$, $S_{\phi\vee\psi}$, $S_{\phi\wedge\psi}$
  - CTL Related: $\underline{S_{\mathrm{EX}\phi}}$, $\underline{S_{\mathrm{EG}\phi}}$, $\underline{S_{\mathrm{EU}[\phi \mathrm{U} \psi]}}$
- In this computation we *only* needed the computation of the sets $T \cup U$, $T \cap U$, complements, and $\{s \in T \mid \exists t \in U : s \to t\}$, for given sets T, U

问题: In *explicit state based* model checking the complexity of this algorithm will be *at least the order of the size of the state space S*

可选方案: 2.2 Binary decisions diagram (BDD), 2.3 Bounded model checking (BMC), 2.4 k-induction

Concluding,

- For an *arbitrary CTL formula* ϕ we saw how to compute the set S_ϕ, being the set of states that satisfy ϕ
  - Basics: $S_\bot$, $S_\top$, S_p, $S_{\neg\phi}$, $S_{\phi\vee\psi}$, $S_{\phi\wedge\psi}$
  - CTL Related: $S_{\mathrm{EX}\phi}$, $S_{\mathrm{EG}\phi}$, $S_{\mathrm{EU}[\phi \mathrm{U} \psi]}$
- In this computation we *only* needed the computation of the sets $T \cup U$, $T \cap U$, complements, and $\{s \in T \mid \exists t \in U : s \to t\}$, for given sets T, U

问题: In *explicit state based* model checking the complexity of this algorithm will be *at least the order of the size of the state space S*

可选方案: 2.2 Binary decisions diagram (BDD), 2.3 Bounded model checking (BMC), 2.4 k-induction

回顾:

- In *explicit state based* model checking the complexity of this algorithm will be *at least the order of the size of the state space S*
  - Key words: state spaces

Now we study BDD

- a method of *symbolic* model checking.
- adopted by NuSMV

基本思路: *Firstly*, investigate desired *requirements* for *alternative representations* for *boolean functions*

回顾:

- In *explicit state based* model checking the complexity of this algorithm will be *at least the order of the size of the state space S*
  - Key words: state spaces

Now we study BDD

- a method of *symbolic* model checking.
- adopted by NuSMV

基本思路: *Firstly*, investigate desired *requirements* for *alternative representations* for *boolean functions*

回顾:

- In *explicit state based* model checking the complexity of this algorithm will be *at least the order of the size of the state space* S
  - Key words: state spaces

Now we study BDD

- a method of *symbolic* model checking.
- adopted by NuSMV

基本思路: *Firstly*, investigate desired *requirements* for *alternative representations* for *boolean functions*

Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
    - Stage 7.1: Compute $ROBDD(\phi)$
    - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
    - Stage 8.1: Express CTL operators
    - Stage 8.2: Compute EX, EG, EU
    - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \to t\}$

Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
   - Stage 7.1: Compute $ROBDD(\phi)$
   - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
   - Stage 8.1: Express CTL operators
   - Stage 8.2: Compute EX, EG, EU
   - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \to t\}$

第 *1* 阶: **Boolean Variables**:

In NuSMV, the variable types are finite sets, in particular *boolean* or *integers with a restricted range*, like

VAR
a : 1..100;

We may *assume* we *only have* **boolean variables**

- by representing variables in *binary* notation, e.g., using *7 bits* for the range 1..100,
- *Atomic propositions*, e.g., $a < b + 5$, can be expressed in *binary* notation too

第 *1* 阶: **Boolean Variables**:

In NuSMV, the variable types are finite sets, in particular *boolean* or *integers with a restricted range*, like

```
VAR
a : 1..100;
```

We may *assume* we *only have* **boolean variables**

- by representing variables in *binary* notation, e.g., using *7 bits* for the range 1..100,
- *Atomic propositions*, e.g., $a < b + 5$, can be expressed in *binary* notation too

第 *1* 阶: **Boolean Variables**:

In NuSMV, the variable types are finite sets, in particular *boolean* or *integers with a restricted range*, like

```
VAR
a : 1..100;
```

We may *assume* we *only have* **boolean variables**

- by representing variables in *binary* notation, e.g., using *7 bits* for the range 1..100,
- *Atomic propositions*, e.g., $a < b + 5$, can be expressed in *binary* notation too

Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
   - Stage 7.1: Compute $ROBDD(\phi)$
   - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
   - Stage 8.1: Express CTL operators
   - Stage 8.2: Compute EX, EG, EU
   - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \to t\}$

第 *2* 阶: **Boolean Functions:**

Write $B = \{0, 1\}$, then for n *boolean variables* the *state space* is $S = B^n$

We want to represent and manipulate *subsets* of $S = B^n$

A *subset* $U \subseteq B^n$ can be identified by a **boolean function**

$$f_U : B^n \to B$$

defined by

$$s \in U \leftrightarrow f_U(s) = 1$$

第 2 阶: Boolean Functions:

Write $B = \{0, 1\}$, then for n *boolean variables* the *state space* is $S = B^n$

We want to represent and manipulate *subsets* of $S = B^n$

A *subset* $U \subseteq B^n$ can be identified by a **boolean function**

$$f_U : B^n \to B$$

defined by

$$s \in U \leftrightarrow f_U(s) = 1$$

第 2 阶: **Boolean Functions:**

Write $B = \{0, 1\}$, then for n *boolean variables* the *state space* is $S = B^n$

We want to represent and manipulate *subsets* of $S = B^n$

A *subset* $U \subseteq B^n$ can be identified by a **boolean function**

$$f_U : B^n \to B$$

defined by

$$s \in U \leftrightarrow f_U(s) = 1$$

第 2 阶: **Boolean Functions:**

Write $B = \{0,1\}$, then for n *boolean variables* the *state space* is $S = B^n$

We want to represent and manipulate *subsets* of $S = B^n$

A *subset* $U \subseteq B^n$ can be identified by a **boolean function**

$$f_U : B^n \to B$$

defined by

$$s \in U \leftrightarrow f_U(s) = 1$$

问题: We want to *represent* and *manipulate* **boolean functions** efficiently, more precisely:

- Every boolean function has a *unique representation* (见后: 问题 2)
  (does not hold for formula representation: $p \wedge (p \wedge q)$ and $q \wedge p$ are distinct formulas representing the same boolean function)
- All operations needs for CTL model checking, including $\neg, \vee, \wedge$ should be *efficiently* computable
  (does not hold for *explicit state representation*: false corresponds to the empty set, but $\neg$ false $=$ true corresponds to the set $S = B^n$ *having* 2^n *elements, infeasible for* $n \geq 30$)
- *Many* boolean functions have an *efficient representation*
  - *Why not for all boolean functions*?

问题: We want to *represent* and *manipulate* **boolean functions** efficiently, more precisely:

- Every boolean function has a *unique representation* (见后: 问题 2)
(does not hold for formula representation: $p \wedge (p \wedge q)$ and $q \wedge p$ are distinct formulas representing the same boolean function)
- All operations needs for CTL model checking, including $\neg, \vee, \wedge$ should be *efficiently* computable
(does not hold for *explicit state representation*: false corresponds to the empty set, but $\neg$ false $=$ true corresponds to the set $S = B^n$ *having* 2^n *elements, infeasible for* $n \geq 30$)
- *Many* boolean functions have an *efficient representation*
    - *Why not for all boolean functions*?

问题: We want to *represent* and *manipulate* **boolean functions** efficiently, more precisely:

- Every boolean function has a *unique representation* (见后: 问题 2)
  (does not hold for formula representation: $p \wedge (p \wedge q)$ and $q \wedge p$ are distinct formulas representing the same boolean function)
- All operations needs for CTL model checking, including $\neg, \vee, \wedge$ should be *efficiently* computable
  (does not hold for *explicit state representation*: false corresponds to the empty set, but $\neg$ false $=$ true corresponds to the set $S = B^n$ *having* 2^n *elements, infeasible for* $n \geq 30$)
- *Many* boolean functions have an *efficient representation*
  - *Why not for all boolean functions*?

问题: We want to *represent* and *manipulate* **boolean functions** efficiently, more precisely:

- Every boolean function has a *unique representation* (见后: 问题 2)
  (does not hold for formula representation: $p \wedge (p \wedge q)$ and $q \wedge p$ are distinct formulas representing the same boolean function)

- All operations needs for CTL model checking, including $\neg, \vee, \wedge$ should be *efficiently* computable
  (does not hold for *explicit state representation*: false corresponds to the empty set, but $\neg$ false $=$ true corresponds to the set $S = B^n$ *having* 2^n *elements, infeasible for* $n \geq 30$)

- *Many* boolean functions have an *efficient representation*
  - *Why not for all boolean functions?*

问题: We want to *represent* and *manipulate* **boolean functions** efficiently, more precisely:

- Every boolean function has a *unique representation* (见后: 问题 2) (does not hold for formula representation: $p \wedge (p \wedge q)$ and $q \wedge p$ are distinct formulas representing the same boolean function)

- All operations needs for CTL model checking, including $\neg, \vee, \wedge$ should be *efficiently* computable
(does not hold for *explicit state representation*: false corresponds to the empty set, but $\neg$ false $=$ true corresponds to the set $S = B^n$ *having* 2^n *elements, infeasible for* $n \geq 30$)

- *Many* boolean functions have an *efficient representation*
  - *Why not for all boolean functions*?

Why not for all boolean functions?

On n variables a truth table consists of 2^n lines

- Hence on n variables there are 2^{2^n} distinct boolean functions
- Indeed, there are $2^{64} \approx 20,000,000,000,000,000,000$ distinct boolean functions on six variables
- If all of these 2^{2^n} distinct boolean functions should have a distinct representation, then *on average* at least 2^n bits are needed for that, begin untractable for $n > 30$

This information theoretic argument shows that it is *unavoidable* that most of the boolean functions have *untractable* (困难的) representation

So the *best* we may hope for is that we meet *in practice* is among the *minor part of all boolean functions* that have an *efficient representation*

结论: BDD provide a *data structure* for *boolean functions* —Decision Tree

Why not for all boolean functions?

On n variables a truth table consists of 2^n lines

- Hence on n variables there are 2^{2^n} distinct boolean functions
- Indeed, there are $2^{64} \approx 20,000,000,000,000,000,000$ distinct boolean functions on six variables
- If all of these 2^{2^n} distinct boolean functions should have a distinct representation, then *on average* at least 2^n bits are needed for that, begin untractable for $n > 30$

This information theoretic argument shows that it is *unavoidable* that most of the boolean functions have *untractable* (困难的) representation

So the *best* we may hope for is that we meet *in practice* is among the *minor part of all boolean functions* that have an *efficient representation*

结论: BDD provide a *data structure* for *boolean functions* —Decision Tree

Why not for all boolean functions?

On n variables a truth table consists of 2^n lines

- Hence on n variables there are 2^{2^n} distinct boolean functions
- Indeed, there are $2^{64} \approx 20,000,000,000,000,000,000$ distinct boolean functions on six variables
- If all of these 2^{2^n} distinct boolean functions should have a distinct representation, then *on average* at least 2^n bits are needed for that, begin untractable for $n > 30$

This information theoretic argument shows that it is *unavoidable* that most of the boolean functions have *untractable* (困难的) representation

So the *best* we may hope for is that we meet *in practice* is among the *minor part of all boolean functions* that have an *efficient representation*

结论: BDD provide a *data structure* for *boolean functions* —Decision Tree

Why not for all boolean functions?

On n variables a truth table consists of 2^n lines

- Hence on n variables there are 2^{2^n} distinct boolean functions
- Indeed, there are $2^{64} \approx 20,000,000,000,000,000,000$ distinct boolean functions on six variables
- If all of these 2^{2^n} distinct boolean functions should have a distinct representation, then *on average* at least 2^n bits are needed for that, begin untractable for $n > 30$

This information theoretic argument shows that it is *unavoidable* that most of the boolean functions have *untractable* (困难的) representation

So the *best* we may hope for is that we meet *in practice* is among the *minor part of all boolean functions* that have an *efficient representation*

结论: BDD provide a *data structure* for *boolean functions* —Decision Tree

Why not for all boolean functions?

On n variables a truth table consists of 2^n lines

- Hence on n variables there are 2^{2^n} distinct boolean functions
- Indeed, there are $2^{64} \approx 20,000,000,000,000,000,000$ distinct boolean functions on six variables
- If all of these 2^{2^n} distinct boolean functions should have a distinct representation, then *on average* at least 2^n bits are needed for that, begin untractable for $n > 30$

This information theoretic argument shows that it is *unavoidable* that most of the boolean functions have *untractable* (困难的) representation

So the *best* we may hope for is that we meet *in practice* is among the *minor part of all boolean functions* that have an *efficient representation*

结论: BDD provide a *data structure* for *boolean functions* —Decision Tree

Why not for all boolean functions?

On n variables a truth table consists of 2^n lines

- Hence on n variables there are 2^{2^n} distinct boolean functions
- Indeed, there are $2^{64} \approx 20,000,000,000,000,000,000$ distinct boolean functions on six variables
- If all of these 2^{2^n} distinct boolean functions should have a distinct representation, then *on average* at least 2^n bits are needed for that, begin untractable for $n > 30$

This information theoretic argument shows that it is *unavoidable* that most of the boolean functions have *untractable* (困难的) representation

So the *best* we may hope for is that we meet *in practice* is among the *minor part of all boolean functions* that have an *efficient representation*

结论: BDD provide a *data structure* for *boolean functions* —Decision Tree

Why not for all boolean functions?

On n variables a truth table consists of 2^n lines

- Hence on n variables there are 2^{2^n} distinct boolean functions
- Indeed, there are $2^{64} \approx 20,000,000,000,000,000,000$ distinct boolean functions on six variables
- If all of these 2^{2^n} distinct boolean functions should have a distinct representation, then *on average* at least 2^n bits are needed for that, begin untractable for $n > 30$

This information theoretic argument shows that it is *unavoidable* that most of the boolean functions have *untractable* (困难的) representation

So the *best* we may hope for is that we meet *in practice* is among the *minor part of all boolean functions* that have an *efficient representation*

结论: BDD provide a *data structure* for *boolean functions* —Decision Tree

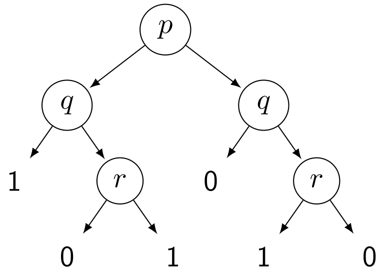
Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
    - Stage 7.1: Compute $ROBDD(\phi)$
    - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
    - Stage 8.1: Express CTL operators
    - Stage 8.2: Compute EX, EG, EU
    - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \to t\}$

第 *3* 阶: **Decision Tree:**

定义: A decision tree is a binary tree in which

- Every *node* is labeled by a *boolean variable*
- Every *leaf* is labeled by 1 or 0, representing true or false respectively



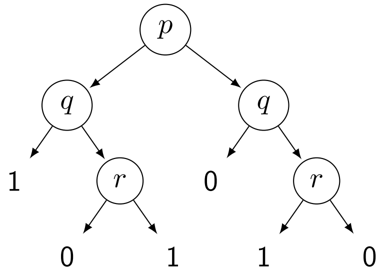
语义: If every variable has a boolean value then the corresponding function value is obtained by

- Start at the root
- For any node: go to the *left* if the corresponding variable is *true*; *otherwise*, go to the *right*
- Repeat until a *leaf* has been reached
- If the leaf is 1 then the result is true, otherwise false

第 3 阶: Decision Tree:

定义: A decision tree is a binary tree in which

- Every *node* is labeled by a *boolean variable*
- Every *leaf* is labeled by 1 or 0, representing true or false respectively



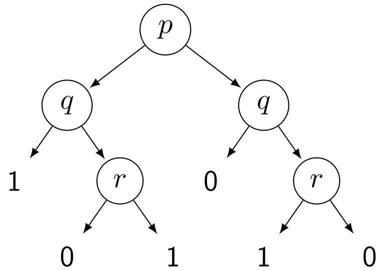
语义: If every variable has a boolean value then the corresponding function value is obtained by

- Start at the root
- For any node: go to the *left* if the corresponding variable is *true*; *otherwise*, go to the *right*
- Repeat until a *leaf* has been reached
- If the leaf is 1 then the result is true, otherwise false

第 *3* 阶: **Decision Tree:**

定义: A decision tree is a binary tree in which

- Every *node* is labeled by a *boolean variable*
- Every *leaf* is labeled by 1 or 0, representing true or false respectively



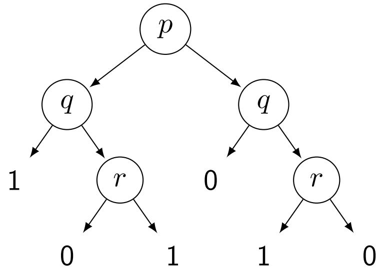
语义: If every variable has a boolean value then the corresponding function value is obtained by

- Start at the root
- For any node: go to the *left* if the corresponding variable is *true*; *otherwise*, go to the *right*
- Repeat until a *leaf* has been reached
- If the leaf is 1 then the result is true, otherwise false

第 3 阶: **Decision Tree:**

定义: A decision tree is a binary tree in which

- Every *node* is labeled by a *boolean variable*
- Every *leaf* is labeled by 1 or 0, representing true or false respectively



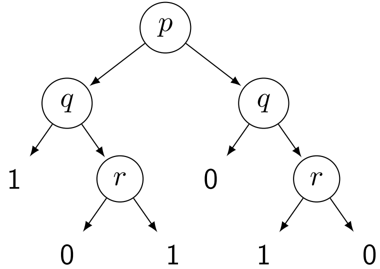
语义: If every variable has a boolean value then the corresponding function value is obtained by

- Start at the root
- For any node: go to the *left* if the corresponding variable is *true*; *otherwise*, go to the *right*
- Repeat until a *leaf* has been reached
- If the leaf is 1 then the result is true, otherwise false

第 *3* 阶: **Decision Tree:**

定义: A decision tree is a binary tree in which

- Every *node* is labeled by a *boolean variable*
- Every *leaf* is labeled by 1 or 0, representing true or false respectively



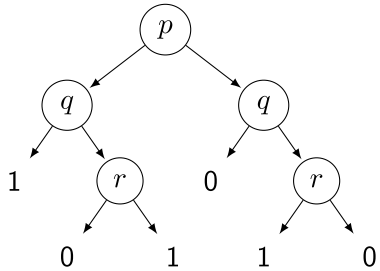
语义: If every variable has a boolean value then the corresponding function value is obtained by

- Start at the root
- For any node: go to the *left* if the corresponding variable is *true*; *otherwise*, go to the *right*
- Repeat until a *leaf* has been reached
- If the leaf is 1 then the result is true, otherwise false

第 *3* 阶: **Decision Tree:**

定义: A decision tree is a binary tree in which

- Every *node* is labeled by a *boolean variable*
- Every *leaf* is labeled by 1 or 0, representing true or false respectively

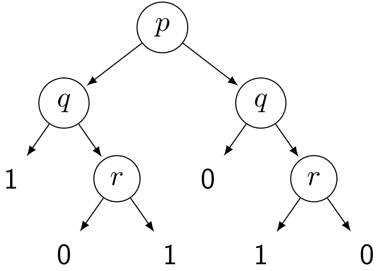


语义: If every variable has a boolean value then the corresponding function value is obtained by

- Start at the root
- For any node: go to the *left* if the corresponding variable is *true*; *otherwise*, go to the *right*
- Repeat until a *leaf* has been reached
- If the leaf is 1 then the result is true, otherwise false

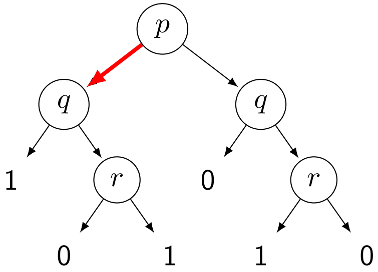
Hence every node is interpreted as an if-then-else

Consider our example for the values



Hence every node is interpreted as an if-then-else

Consider our example for the values
- p: true

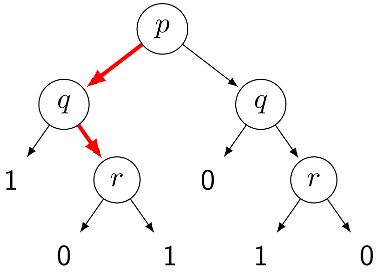


Hence every node is interpreted as an if-then-else

Consider our example for the values

p: true

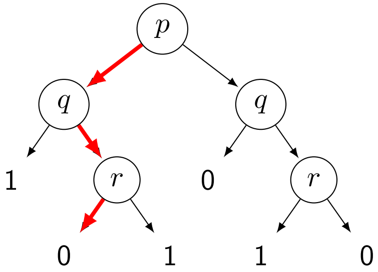
q: false



Hence every node is interpreted as an if-then-else

Consider our example for the values

- p: true
- q: false
- r: true

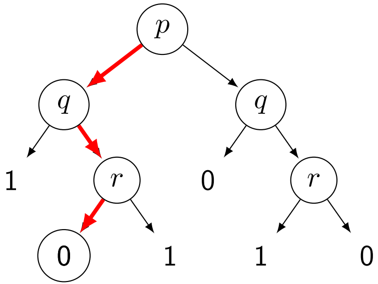


Hence every node is interpreted as an if-then-else

Consider our example for the values

- p: true
- q: false
- r: true

Hence, the result is 0



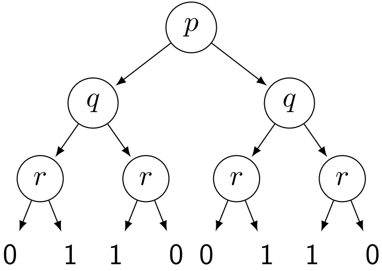
问题 1: Does every boolean function on finitely many boolean variables have a representation as decision tree?

答: *Yes*: it can be defined by a *truth table*, and any truth table on n variables having 2^n lines can be represented as a decision tree with 2^n leaves

问题 1: Does every boolean function on finitely many boolean variables have a representation as decision tree?
答: *Yes*: it can be defined by a *truth table*, and any truth table on n variables having 2^n lines can be represented as a decision tree with 2^n leaves

问题 1: Does every boolean function on finitely many boolean variables have a representation as decision tree?

答: *Yes*: it can be defined by a *truth table*, and any truth table on n variables having 2^n lines can be represented as a decision tree with 2^n leaves



问题 2: Does every boolean function on finitely many boolean variables have a *unique representation* as decision tree?

答: *No*

The following two decision trees both represent the boolean function on p, q that yields *true* in case p *is true* and q *is false*, and false otherwise

问题 2: Does every boolean function on finitely many boolean variables have a *unique representation* as decision tree?

答: *No*

The following two decision trees both represent the boolean function on p, q that yields *true* in case p *is true* and q *is false*, and false otherwise

问题 2: Does every boolean function on finitely many boolean variables have a *unique representation* as decision tree?

答: *No*

The following two decision trees both represent the boolean function on p, q that yields *true* in case p *is true* and q *is false*, and false otherwise

问题 2: Does every boolean function on finitely many boolean variables have a *unique representation* as decision tree?

答: *No*

The following two decision trees both represent the boolean function on p, q that yields *true* in case p *is true* and q *is false*, and false otherwise

Outline of a BDD algorithm

问题 2 的一种解决方法: Observe that in one case p is on top of q, while in the other case q is on top of p

Fix an order $<$ on the boolean variables, like $p < q$

第 4 阶: **Ordered decision tree**: An *ordered decision tree* with respect to $<$ is a decision tree such that if node n is on top of node n', then

$$\mathrm{label}(n) < \mathrm{label}(n')$$

So the left one is ordered with respect to $p < q$, the *right one is not*

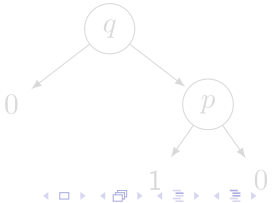
问题 2 的一种解决方法: Observe that in one case p is on top of q, while in the other case q is on top of p

Fix an order $<$ on the boolean variables, like $p < q$

第 *4* 阶*:* **Ordered decision tree**: An *ordered decision tree* with respect to $<$ is a decision tree such that if node n is on top of node n', then

$$\mathrm{label}(n) < \mathrm{label}(n')$$

So the left one is ordered with respect to $p < q$, the *right one is not*



问题 2 的一种解决方法: Observe that in one case p is on top of q, while in the other case q is on top of p

Fix an order $<$ on the boolean variables, like $p < q$

第 *4* 阶*:* **Ordered decision tree**: An *ordered decision tree* with respect to $<$ is a decision tree such that if node n is on top of node n', then

$$\mathrm{label}(n) < \mathrm{label}(n')$$

So the left one is ordered with respect to $p < q$, the *right one is not*

问题 3: Fixing the order $<$ on the boolean variables, does *every* boolean function have a *unique* representation as an *ordered* decision tree with respect to $<$?

No

Let T be any ordered decision tree, and let p be a variable less than the variables in T

Then T and

are two ordered decision trees representing *the same boolean function*

We are looking for a *small* representing, hence among these two we *prefer* T and exclude the latter.

问题 3: Fixing the order $<$ on the boolean variables, does *every* boolean function have a *unique* representation as an *ordered* decision tree with respect to $<$?

No

Let T be any ordered decision tree, and let p be a variable less than the variables in T

Then T and

are two ordered decision trees representing *the same boolean function*

We are looking for a *small* representing, hence among these two we *prefer* T and exclude the latter.

问题 3: Fixing the order $<$ on the boolean variables, does *every* boolean function have a *unique* representation as an *ordered* decision tree with respect to $<$?

No

Let T be any ordered decision tree, and let p be a variable less than the variables in T

Then T and

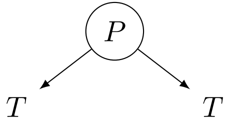
are two ordered decision trees representing *the same boolean function*

We are looking for a *small* representing, hence among these two we *prefer* T and exclude the latter.

问题 3: Fixing the order $<$ on the boolean variables, does *every* boolean function have a *unique* representation as an *ordered* decision tree with respect to $<$?

No

Let T be any ordered decision tree, and let p be a variable less than the variables in T

Then T and 

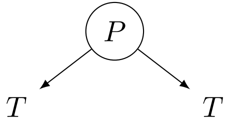
are two ordered decision trees representing *the same boolean function*

We are looking for a *small* representing, hence among these two we *prefer* T and exclude the latter.

问题 3: Fixing the order $<$ on the boolean variables, does *every* boolean function have a *unique* representation as an *ordered* decision tree with respect to $<$?

No

Let T be any ordered decision tree, and let p be a variable less than the variables in T

Then T and 

are two ordered decision trees representing *the same boolean function*

We are looking for a *small* representing, hence among these two we *prefer* T and exclude the latter.

Outline of a BDD algorithm

第 *5* 阶*:* **Reduced ordered decision tree:**

Replacing

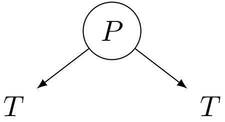
by T is called *elimination*

An ordered decision tree on which no elimination is possible is called a *reduced ordered decision tree*

2. 理论

2.2 Binary decisions diagram (BDD) | Stage 5: Reduced ordered decision tree (elimination)

第 *5* 阶*:* **Reduced ordered decision tree:**

Replacing 

by T is called *elimination*

An ordered decision tree on which no elimination is possible is called a *reduced ordered decision tree*

第 *5* 阶*:* **Reduced ordered decision tree:**

Replacing

$$\begin{tikzcd} & P \arrow[dl] \arrow[dr] & \\ T & & T \end{tikzcd}$$

by T is called *elimination*

An ordered decision tree on which no elimination is possible is called a *reduced ordered decision tree*

定理

For a fixed order $<$ on boolean variables, every boolean function has a *unique* representation as a *reduced ordered decision tree*

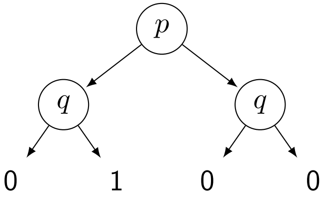
证明过程: 略

例: For the boolean function defined by the formula $p \wedge \neg q$, for the order $p < q$ the ordered decision tree reflecting the truth table is

Applying elimination on the right q yields

Now no elimination is possible any more, so this is a *reduced ordered decision tree*

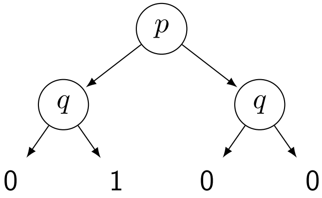
例: For the boolean function defined by the formula $p \wedge \neg q$, for the order $p < q$ the ordered decision tree reflecting the truth table is



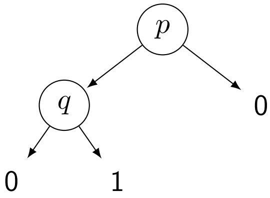
Applying elimination on the right q yields

Now no elimination is possible any more, so this is a *reduced ordered decision tree*

例: For the boolean function defined by the formula $p \wedge \neg q$, for the order $p < q$ the ordered decision tree reflecting the truth table is



Applying elimination on the right q yields

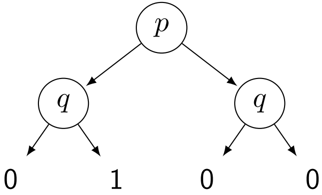


Now no elimination is possible any more, so this is a *reduced ordered decision tree*

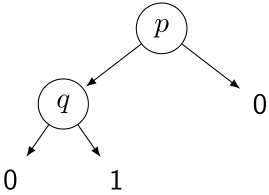
2. 理论

2.2 Binary decisions diagram (BDD) | Stage 5: Reduced ordered decision tree (elimination)

例: For the boolean function defined by the formula $p \wedge \neg q$, for the order $p < q$ the ordered decision tree reflecting the truth table is



Applying elimination on the right q yields



Now no elimination is possible any more, so this is a *reduced ordered decision tree*

Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
   - Stage 7.1: Compute $ROBDD(\phi)$
   - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
   - Stage 8.1: Express CTL operators
   - Stage 8.2: Compute EX, EG, EU
   - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \to t\}$

第 *6* 阶: *ROBDD*: Reduced Ordered Binary Decisions Diagrams

- A *particular* example of Binary Decisions Diagrams (*BDDs*)
- uniquely represent boolean functions by *merging* and *elimination*

The formula $(p \wedge q) \vee r$ describes the boolean function that yields true if both p and q are true, or r is true

With respect to the order $p < q < r$ its ROBDD is

第 *6* 阶: *ROBDD*: **Reduced Ordered Binary Decisions Diagrams**

- A *particular* example of Binary Decisions Diagrams (*BDDs*)
- uniquely represent boolean functions by *merging* and *elimination*

The formula $(p \wedge q) \vee r$ describes the boolean function that yields true if both p and q are true, or r is true

With respect to the order $p < q < r$ its ROBDD is

第 6 阶: *ROBDD*: Reduced Ordered Binary Decisions Diagrams

- A *particular* example of Binary Decisions Diagrams (*BDDs*)
- uniquely represent boolean functions by *merging* and *elimination*

The formula $(p \wedge q) \vee r$ describes the boolean function that yields true if both p and q are true, or r is true

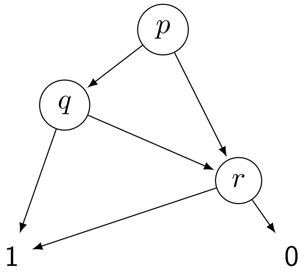
With respect to the order $p < q < r$ its ROBDD is

第 *6* 阶: *ROBDD*: **Reduced Ordered Binary Decisions Diagrams**

- A *particular* example of Binary Decisions Diagrams (*BDDs*)
- uniquely represent boolean functions by *merging* and *elimination*

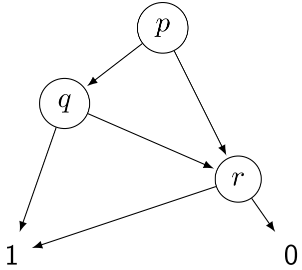
The formula $(p \wedge q) \vee r$ describes the boolean function that yields true if both p and q are true, or r is true

With respect to the order $p < q < r$ its ROBDD is



The formula $(p \wedge q) \vee r$ describes the boolean function that yields true if both p and q are true, or r is true

With respect to the order $p < q < r$ its ROBDD is



In such a ROBDD, *every node* represents a *boolean function* itself

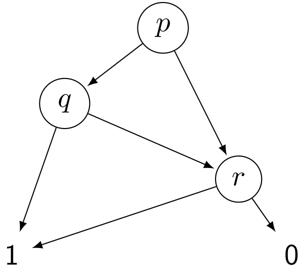
- The ROBDD of this function is the part of the original ROBDD of which the indicated node is the *root*

All nodes of a ROBDD represent *distinct* boolean functions

- since if two would represent the same, then they can be shared by applying *merging* and *elimination* steps, contradicting being *R*(educed)

The formula $(p \wedge q) \vee r$ describes the boolean function that yields true if both p and q are true, or r is true

With respect to the order $p < q < r$ its ROBDD is



In such a ROBDD, *every node* represents a *boolean function* itself

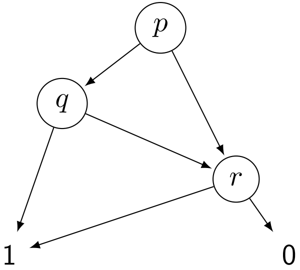
- The ROBDD of this function is the part of the original ROBDD of which the indicated node is the *root*

All nodes of a ROBDD represent *distinct* boolean functions

- since if two would represent the same, then they can be shared by applying *merging* and *elimination* steps, contradicting being *R*(educed)

The formula $(p \wedge q) \vee r$ describes the boolean function that yields true if both p and q are true, or r is true

With respect to the order $p < q < r$ its ROBDD is



In such a ROBDD, *every node* represents a *boolean function* itself

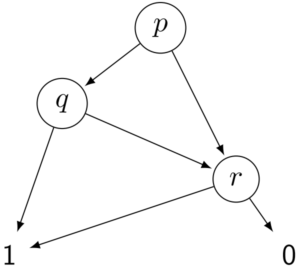
- The ROBDD of this function is the part of the original ROBDD of which the indicated node is the *root*

All nodes of a ROBDD represent *distinct* boolean functions

- since if two would represent the same, then they can be shared by applying *merging* and *elimination* steps, contradicting being *R*(educed)

The formula $(p \wedge q) \vee r$ describes the boolean function that yields true if both p and q are true, or r is true

With respect to the order $p < q < r$ its ROBDD is



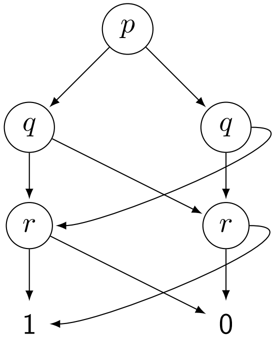
In such a ROBDD, *every node* represents a *boolean function* itself

- The ROBDD of this function is the part of the original ROBDD of which the indicated node is the *root*

All nodes of a ROBDD represent *distinct* boolean functions

- since if two would represent the same, then they can be shared by applying *merging* and *elimination* steps, contradicting being *R*(educed)

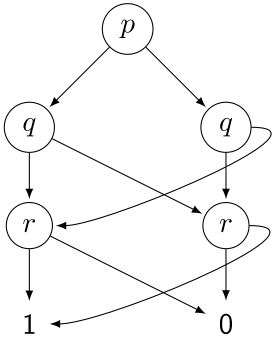
Merge and share: For $p < q < r$ the ROBDD of $p \leftrightarrow q \leftrightarrow r$ is



yields *true* if and only if the *number* of variables that is false, is *even*

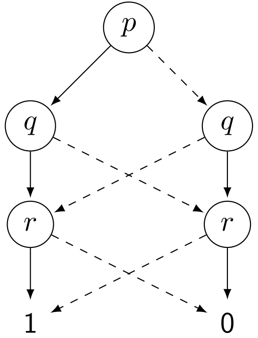
Alternative notation to avoid curved arrows: use *solid* arrows for true-branches and *dashed* arrows for false-branches

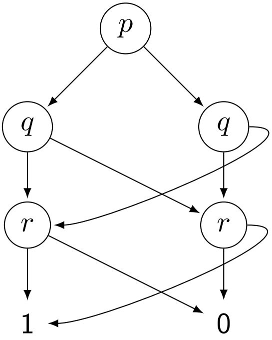
Merge and share: For $p < q < r$ the ROBDD of $p \leftrightarrow q \leftrightarrow r$ is



yields *true* if and only if the *number* of variables that is false, is *even*

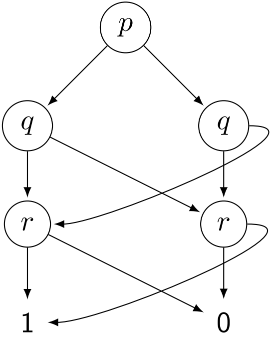
Alternative notation to avoid curved arrows: use *solid* arrows for true-branches and *dashed* arrows for false-branches



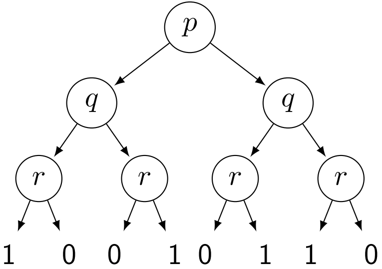


问: 为什么要 *merging?* Sharing really helps: without sharing (so the unique reduced ordered decision tree) for $p \leftrightarrow q \leftrightarrow r$ instead of, we would obtain

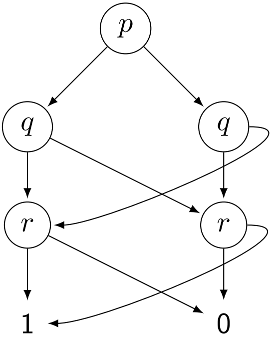
Doing the same for $p_1 \leftrightarrow p_2 \leftrightarrow \cdots \leftrightarrow p_n$ yields a ROBDD of $2n-1$ nodes, and a reduced ordered decision tree of 2^n-1 nodes: an *exponential* gap



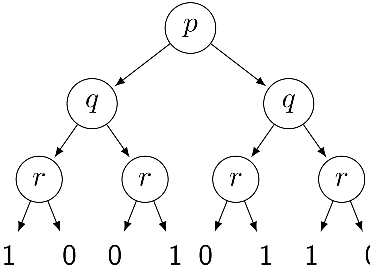
问: 为什么要 *merging?* Sharing really helps: without sharing (so the unique reduced ordered decision tree) for $p \leftrightarrow q \leftrightarrow r$ instead of, we would obtain



Doing the same for $p_1 \leftrightarrow p_2 \leftrightarrow \cdots \leftrightarrow p_n$ yields a ROBDD of $2n-1$ nodes, and a reduced ordered decision tree of 2^n-1 nodes: an *exponential* gap



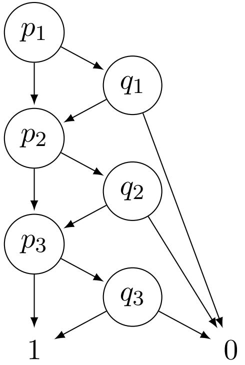
问: 为什么要 *merging?* Sharing really helps: without sharing (so the unique reduced ordered decision tree) for $p \leftrightarrow q \leftrightarrow r$ instead of, we would obtain



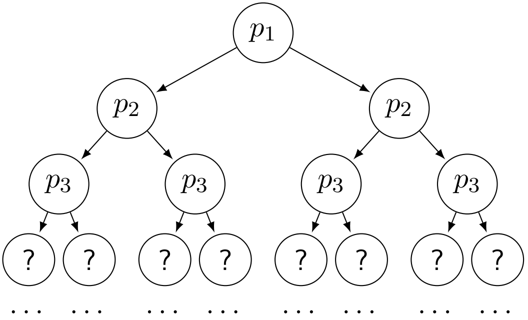
Doing the same for $p_1 \leftrightarrow p_2 \leftrightarrow \cdots \leftrightarrow p_n$ yields a ROBDD of $2n - 1$ nodes, and a reduced ordered decision tree of $2^n - 1$ nodes: an *exponential* gap

问题 *4:* 如何选择 *order?* The ROBDD of $(p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge (p_3 \vee q_3)$ with respect to

$p_1 < q_1 < p_2 < q_2 < p_3 < q_3$ is:



w.r.t. $p_1 < p_2 < p_3 < q_1 < q_2 < q_3$ it is:



where all ? nodes represent distinct boolean functions on q_1, q_2, q_3, so cannot be shared

More general, the ROBDD of

$$(p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge \cdots \wedge (p_3 \vee q_3)$$

with respect to the order

- $p_1 < q_1 < p_2 < q_2 < \cdots < p_n < q_n$: has exactly $2n$ nodes
- $p_1 < p_2 < \cdots < p_n < q_1 < q_2 < \cdots < q_n$: has more than 2^n nodes

So *distinct orders* may result in ROBDDs of sizes with an *exponential* gap in between

Heuristic

choose the order in such a way that *variables close to each other* in the formula are also *close in the order*

More general, the ROBDD of

$$(p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge \cdots \wedge (p_3 \vee q_3)$$

with respect to the order

- $p_1 < q_1 < p_2 < q_2 < \cdots < p_n < q_n$: has exactly $2n$ nodes
- $p_1 < p_2 < \cdots < p_n < q_1 < q_2 < \cdots < q_n$: has more than 2^n nodes

So *distinct orders* may result in ROBDDs of sizes with an *exponential* gap in between

Heuristic

choose the order in such a way that *variables close to each other* in the formula are also *close in the order*

More general, the ROBDD of

$$(p_1 \vee q_1) \wedge (p_2 \vee q_2) \wedge \cdots \wedge (p_3 \vee q_3)$$

with respect to the order

- $p_1 < q_1 < p_2 < q_2 < \cdots < p_n < q_n$: has exactly $2n$ nodes
- $p_1 < p_2 < \cdots < p_n < q_1 < q_2 < \cdots < q_n$: has more than 2^n nodes

So *distinct orders* may result in ROBDDs of sizes with an *exponential* gap in between

Heuristic

choose the order in such a way that *variables close to each other* in the formula are also *close in the order*

Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
   - Stage 7.1: Compute $ROBDD(\phi)$
   - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
   - Stage 8.1: Express CTL operators
   - Stage 8.2: Compute EX, EG, EU
   - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \rightarrow t\}$

第 *7* 阶: How to *compute* the Reduced Ordered Binary Decision Diagram (*ROBDD*) of a given formula?

问题 *5*: The methods in the former stages should *not* be used to compute ROBDDs in *practice*

- since the *size* of this *decision tree* is always *exponential*, so *unfeasible*

解决方法: operate directly on the *formula* $\underline{\phi}$, instead of *decision tree*

Observation: Every formula is of the shape:

- false or true, or
- p for a variable p, or
- $\neg\phi$, or
- $\phi \diamond \psi$ for $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

第 *7* 阶*:* How to *compute* the Reduced Ordered Binary Decision Diagram (*ROBDD*) of a given formula?

问题 *5*: The methods in the former stages should *not* be used to compute ROBDDs in *practice*

- since the *size* of this *decision tree* is always *exponential*, so *unfeasible*

解决方法: operate directly on the *formula* ϕ, instead of *decision tree*

Observation: Every formula is of the shape:

- false or true, or
- p for a variable p, or
- $\neg\phi$, or
- $\phi \diamond \psi$ for $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

第 *7* 阶*:* How to *compute* the Reduced Ordered Binary Decision Diagram (*ROBDD*) of a given formula?

问题 *5*: The methods in the former stages should *not* be used to compute ROBDDs in *practice*

- since the *size* of this *decision tree* is always *exponential*, so *unfeasible*

解决方法: operate directly on the *formula* ϕ, instead of *decision tree*

Observation: Every formula is of the shape:

- false or true, or
- p for a variable p, or
- $\neg\phi$, or
- $\phi \diamond \psi$ for $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

第 7 阶: How to *compute* the Reduced Ordered Binary Decision Diagram (*ROBDD*) of a given formula?

问题 *5*: The methods in the former stages should *not* be used to compute ROBDDs in *practice*

- since the *size* of this *decision tree* is always *exponential*, so *unfeasible*

解决方法: operate directly on the *formula* $\underline{\phi}$, instead of *decision tree*

Observation: Every formula is of the shape:

- false or true, or
- p for a variable p, or
- $\neg\phi$, or
- $\phi \diamond \psi$ for $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

第 7 阶: How to *compute* the Reduced Ordered Binary Decision Diagram (*ROBDD*) of a given formula?

问题 *5*: The methods in the former stages should *not* be used to compute ROBDDs in *practice*

- since the *size* of this *decision tree* is always *exponential*, so *unfeasible*

解决方法: operate directly on the *formula* $\underline{\phi}$, instead of *decision tree*

Observation: Every formula is of the shape:

- false or true, or
- p for a variable p, or
- $\neg\phi$, or
- $\phi \diamond \psi$ for $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$

Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
   - Stage 7.1: Compute $ROBDD(\phi)$
   - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
   - Stage 8.1: Express CTL operators
   - Stage 8.2: Compute EX, EG, EU
   - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \rightarrow t\}$

Stage 7.1: *Compute ROBDD*(ϕ). So, the ROBDD *ROBDD*(ϕ) of a formula $\underline{\phi}$ will be constructed recursively according this recursive structure of the formulas: Basic Idea

- *ROBDD*(**F**)=0, *ROBDD*(**T**)=1
- *ROBDD*$(\underline{p})$=

- *ROBDD*$(\neg\phi)$=*ROBDD*$(\phi \to$ **F**$)$
- *ROBDD*$(\phi \diamond \psi)$= *apply*(*ROBDD*(ϕ), *ROBDD*(ψ), $\diamond$)

So it remains to find an algorithm *apply* having two ROBDDs and a binary operation $\diamond \in \{\vee, \wedge, \to, \leftrightarrow\}$ as input, and having the desired ROBDD as its output

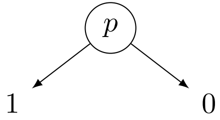
Stage 7.1: *Compute ROBDD*(ϕ). So, the ROBDD *ROBDD*(ϕ) of a formula $\underline{\phi}$ will be constructed recursively according this recursive structure of the formulas: ▸ Basic Idea

- *ROBDD*(**F**)=0, *ROBDD*(**T**)=1
- *ROBDD*$(\underline{p})$=

- *ROBDD*$(\neg\phi)$=*ROBDD*$(\phi \to$ **F**$)$
- *ROBDD*$(\phi \diamond \psi)$= *apply*(*ROBDD*(ϕ), *ROBDD*(ψ), $\diamond$)

So it remains to find an algorithm *apply* having two ROBDDs and a binary operation $\diamond \in \{\vee, \wedge, \to, \leftrightarrow\}$ as input, and having the desired ROBDD as its output

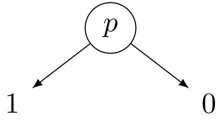
Stage 7.1: *Compute ROBDD*(ϕ). So, the ROBDD *ROBDD*(ϕ) of a formula $\underline{\phi}$ will be constructed recursively according this recursive structure of the formulas: Basic Idea

- *ROBDD*(**F**)=0, *ROBDD*(**T**)=1
- *ROBDD*($\underline{p}$)= 

- *ROBDD*($\neg\phi$)=*ROBDD*($\phi \to$ **F**)
- *ROBDD*($\phi \diamond \psi$)= *apply*(*ROBDD*(ϕ), *ROBDD*(ψ), $\diamond$)

So it remains to find an algorithm *apply* having two ROBDDs and a binary operation $\diamond \in \{\vee, \wedge, \to, \leftrightarrow\}$ as input, and having the desired ROBDD as its output

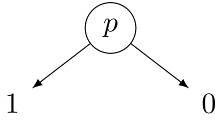
Stage 7.1: *Compute ROBDD*(ϕ). So, the ROBDD *ROBDD*(ϕ) of a formula $\underline{\phi}$ will be constructed recursively according this recursive structure of the formulas: ▸ Basic Idea

- *ROBDD*(**F**)=0, *ROBDD*(**T**)=1
- *ROBDD*$(\underline{p})$= 

- *ROBDD*$(\neg\phi)$=*ROBDD*$(\phi \to$ **F**$)$
- *ROBDD*$(\phi \diamond \psi)$= *apply*(*ROBDD*(ϕ), *ROBDD*(ψ), $\diamond$)

So it remains to find an algorithm *apply* having two ROBDDs and a binary operation $\diamond \in \{\vee, \wedge, \to, \leftrightarrow\}$ as input, and having the desired ROBDD as its output

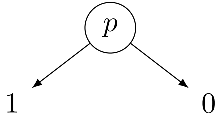
Stage 7.1: *Compute ROBDD*(ϕ). So, the ROBDD *ROBDD*(ϕ) of a formula ϕ will be constructed recursively according this recursive structure of the formulas: ▸ Basic Idea

- *ROBDD*(**F**)=0, *ROBDD*(**T**)=1
- *ROBDD*(p)= 

- *ROBDD*($\neg\phi$)=*ROBDD*($\phi \to$ **F**)
- *ROBDD*($\phi \diamond \psi$)= *apply*(*ROBDD*(ϕ), *ROBDD*(ψ), $\diamond$)

So it remains to find an algorithm *apply* having two ROBDDs and a binary operation $\diamond \in \{\vee, \wedge, \to, \leftrightarrow\}$ as input, and having the desired ROBDD as its output

Stage 7.1: *Compute ROBDD*(ϕ). So, the ROBDD *ROBDD*(ϕ) of a formula $\underline{\phi}$ will be constructed recursively according this recursive structure of the formulas: ▸ Basic Idea

- *ROBDD*(**F**)=0, *ROBDD*(**T**)=1
- *ROBDD*($\underline{p}$)= 

- *ROBDD*($\neg\phi$)=*ROBDD*($\phi \to$ **F**)
- *ROBDD*($\phi \diamond \psi$)= *apply*(*ROBDD*(ϕ), *ROBDD*(ψ), $\diamond$)

So it remains to find an algorithm *apply* having two ROBDDs and a binary operation $\diamond \in \{\vee, \wedge, \to, \leftrightarrow\}$ as input, and having the desired ROBDD as its output

续: So it remains to find an algorithm *apply* having two ROBDDs and a binary operation $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$ as input, and having the desired ROBDD as its output

write $p(T, U)$ for the BDD having root p for which the left branch is T and the right branch is U:

write $\diamond(T, U)$ instead of *apply*$(T, U, \diamond)$

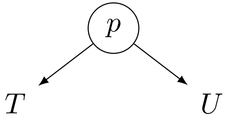
- As the basis of the recursive we define $\diamond(T, U)$ (见下页)

2. 理论

2.2 Binary decisions diagram (BDD) | Stage 7: Compute ROBDD

续: So it remains to find an algorithm *apply* having two ROBDDs and a binary operation $\diamond \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$ as input, and having the desired ROBDD as its output

write $p(T, U)$ for the BDD having root p for which the left branch is T and the right branch is U:

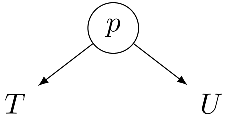


write $\diamond(T, U)$ instead of *apply*$(T, U, \diamond)$

- As the basis of the recursive we define $\diamond(T, U)$ (见下页)

续: So it remains to find an algorithm *apply* having two ROBDDs and a binary operation $\diamond \in \{\vee, \wedge, \to, \leftrightarrow\}$ as input, and having the desired ROBDD as its output

write $p(T, U)$ for the BDD having root p for which the left branch is T and the right branch is U:



write $\diamond(T, U)$ instead of *apply*$(T, U, \diamond)$

- As the basis of the recursive we define $\diamond(T, U)$ (见下页)

Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
   - Stage 7.1: Compute $ROBDD(\phi)$
   - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
   - Stage 8.1: Express CTL operators
   - Stage 8.2: Compute EX, EG, EU
   - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \rightarrow t\}$

Stage 7.2: *Compute* $\diamond(T,U)$ recursively for cases:

1. if $T,U \in \{\mathbf{T},\mathbf{F}\}$: return value according the *truth table of* $\diamond$
2. if T,U not both in $\{\mathbf{T},\mathbf{F}\}$: let p be the smallest variable occurring in T and U.
   1. p is on top of both T and U Details

      $$\diamond(p(T_1,T_2),p(U_1,U_2)) = p(\diamond(T_1,U_1),\diamond(T_2,U_2))$$

   2. p is on top of T but does not occur in U Details

      $$\diamond(p(T_1,T_2),U) = p(\diamond(T_1,U),\diamond(T_2,U)), \text{ if } p \text{ does not occur in } U$$

   3. p is on top of U but does not occur in T Details

      $$\diamond(T,p(U_1,U_2)) = p(\diamond(T,U_1),\diamond(T,U_2)), \text{ if } p \text{ does not occur in } T$$

Stage 7.2: *Compute* $\underline{\diamond(T,U)}$ recursively for cases:

1. if $T, U \in \{\mathbf{T}, \mathbf{F}\}$: return value according the *truth table of* $\diamond$
2. if T, U not both in $\{\mathbf{T}, \mathbf{F}\}$: let p be the smallest variable occurring in T and U.
   1. p is on top of both T and U ▸ Details

      $$\diamond(p(T_1,T_2),p(U_1,U_2)) = p(\diamond(T_1,U_1),\diamond(T_2,U_2))$$

   2. p is on top of T but does not occur in U ▸ Details

      $$\diamond(p(T_1,T_2),U) = p(\diamond(T_1,U),\diamond(T_2,U)), \text{ if } p \text{ does not occur in } U$$

   3. p is on top of U but does not occur in T ▸ Details

      $$\diamond(T,p(U_1,U_2)) = p(\diamond(T,U_1),\diamond(T,U_2)), \text{ if } p \text{ does not occur in } T$$

Stage 7.2: *Compute* $\underline{\diamond(T,U)}$ recursively for cases:

1. if $T, U \in \{\mathbf{T}, \mathbf{F}\}$: return value according the *truth table of* $\diamond$
2. if T, U not both in $\{\mathbf{T}, \mathbf{F}\}$: let p be the smallest variable occurring in T and U.
   1. p is on top of both T and U ▸ Details

   $$\diamond(p(T_1,T_2),p(U_1,U_2)) = p(\diamond(T_1,U_1),\diamond(T_2,U_2))$$

   2. p is on top of T but does not occur in U ▸ Details

   $$\diamond(p(T_1,T_2),U) = p(\diamond(T_1,U),\diamond(T_2,U)), \text{ if } p \text{ does not occur in } U$$

   3. p is on top of U but does not occur in T ▸ Details

   $$\diamond(T,p(U_1,U_2)) = p(\diamond(T,U_1),\diamond(T,U_2)), \text{ if } p \text{ does not occur in } T$$

Stage 7.2: *Compute* $\underline{\diamond(T, U)}$ recursively for cases:

1. if $T, U \in \{\mathbf{T}, \mathbf{F}\}$: return value according the *truth table of* $\diamond$
2. if T, U not both in $\{\mathbf{T}, \mathbf{F}\}$: let p be the smallest variable occurring in T and U.
    1. p is on top of both T and U ▸ Details

    $$\diamond(p(T_1, T_2), p(U_1, U_2)) = p(\diamond(T_1, U_1), \diamond(T_2, U_2))$$

    2. p is on top of T but does not occur in U ▸ Details

    $$\diamond(p(T_1, T_2), U) = p(\diamond(T_1, U), \diamond(T_2, U)), \text{ if } p \text{ does not occur in } U$$

    3. p is on top of U but does not occur in T ▸ Details

    $$\diamond(T, p(U_1, U_2)) = p(\diamond(T, U_1), \diamond(T, U_2)), \text{ if } p \text{ does not occur in } T$$

Stage 7.2: *Compute* $\underline{\diamond(T, U)}$ recursively for cases:

1. if $T, U \in \{\mathbf{T}, \mathbf{F}\}$: return value according the *truth table of* $\diamond$
2. if T, U not both in $\{\mathbf{T}, \mathbf{F}\}$: let p be the smallest variable occurring in T and U.
   1. p is on top of both T and U ▸ Details

$$\diamond(p(T_1, T_2), p(U_1, U_2)) = p(\diamond(T_1, U_1), \diamond(T_2, U_2))$$

   2. p is on top of T but does not occur in U ▸ Details

$$\diamond(p(T_1, T_2), U) = p(\diamond(T_1, U), \diamond(T_2, U)), \text{ if } p \text{ does not occur in } U$$

   3. p is on top of U but does not occur in T ▸ Details

$$\diamond(T, p(U_1, U_2)) = p(\diamond(T, U_1), \diamond(T, U_2)), \text{ if } p \text{ does not occur in } T$$

Stage 7.2: *Compute* $\underline{\diamond(T,U)}$ recursively for cases:

1. if $T, U \in \{\mathbf{T}, \mathbf{F}\}$: return value according the *truth table of* $\diamond$
2. if T, U not both in $\{\mathbf{T}, \mathbf{F}\}$: let p be the smallest variable occurring in T and U.
   1. p is on top of both T and U ▸ Details

      $$\diamond(p(T_1,T_2), p(U_1,U_2)) = p(\diamond(T_1,U_1), \diamond(T_2,U_2))$$

   2. p is on top of T but does not occur in U ▸ Details

      $$\diamond(p(T_1,T_2), U) = p(\diamond(T_1,U), \diamond(T_2,U)), \text{ if } p \text{ does not occur in } U$$

   3. p is on top of U but does not occur in T ▸ Details

      $$\diamond(T, p(U_1,U_2)) = p(\diamond(T,U_1), \diamond(T,U_2)), \text{ if } p \text{ does not occur in } T$$

Stage 7.2: *Compute* $\diamond(T,U)$ recursively for cases:

1. if $T, U \in \{\mathbf{T}, \mathbf{F}\}$: return value according the *truth table of* $\diamond$
2. if T, U not both in $\{\mathbf{T}, \mathbf{F}\}$: let p be the smallest variable occurring in T and U.
   1. p is on top of both T and U ▸ Details

      $$\diamond(p(T_1, T_2), p(U_1, U_2)) = p(\diamond(T_1, U_1), \diamond(T_2, U_2))$$

   2. p is on top of T but does not occur in U ▸ Details

      $$\diamond(p(T_1, T_2), U) = p(\diamond(T_1, U), \diamond(T_2, U)), \text{ if } p \text{ does not occur in } U$$

   3. p is on top of U but does not occur in T ▸ Details

      $$\diamond(T, p(U_1, U_2)) = p(\diamond(T, U_1), \diamond(T, U_2)), \text{ if } p \text{ does not occur in } T$$

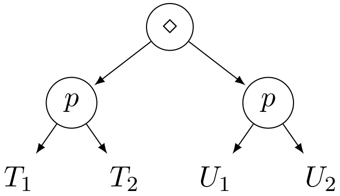
Intuitively: for two BDDs T, U computing $\diamond(T, U)$ is done by pushing $\diamond$ *downwards*, meanwhile *combining* T *and* U, until $\diamond$ applied to **T**/**F** has to be computed, which is replaced by its value according to the truth table

In a picture: (1) Replace

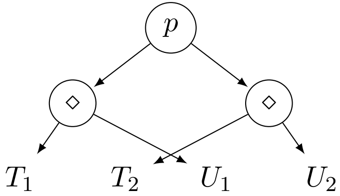
by

Intuitively: for two BDDs T, U computing $\diamond(T, U)$ is done by pushing $\diamond$ *downwards*, meanwhile *combining T and U*, until $\diamond$ applied to **T**/**F** has to be computed, which is replaced by its value according to the truth table

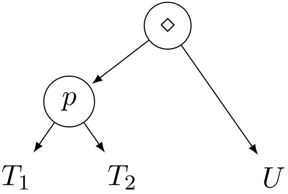
In a picture: (1) Replace



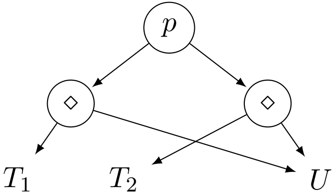
by



(2) If p not in U, then Replace



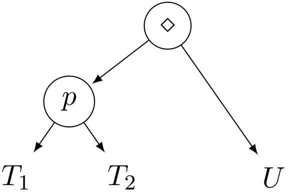
by



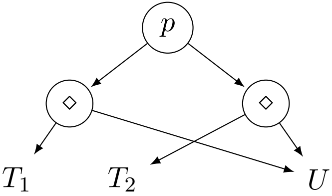
(3)If p not in T, then replace

by

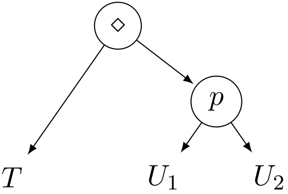
(2) If p not in U, then Replace



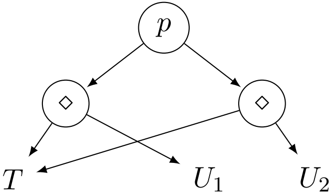
by



(3)If p not in T, then replace



by



例子: We choose the formula

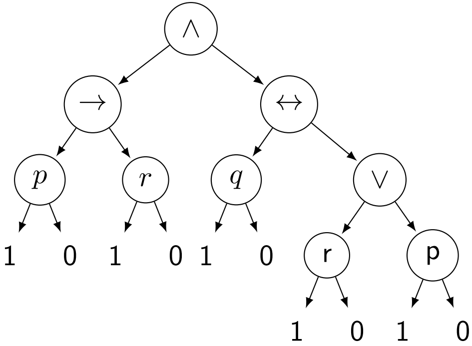
$$(p \to r) \land (q \leftrightarrow (r \lor p))$$

and the order $p < q < r$. in a picture:

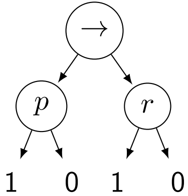
例子: We choose the formula

$$(p \to r) \wedge (q \leftrightarrow (r \vee p))$$

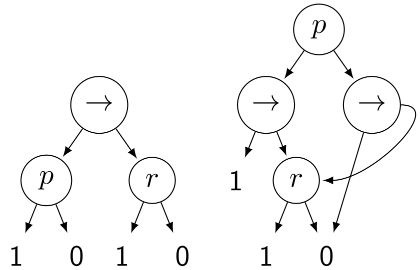
and the order $p < q < r$. in a picture:



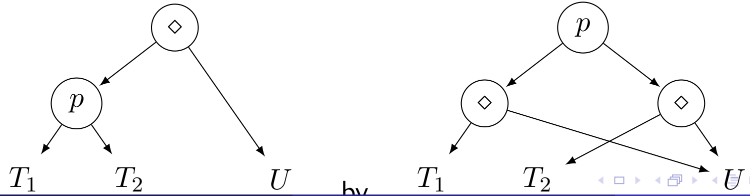
Left argument:



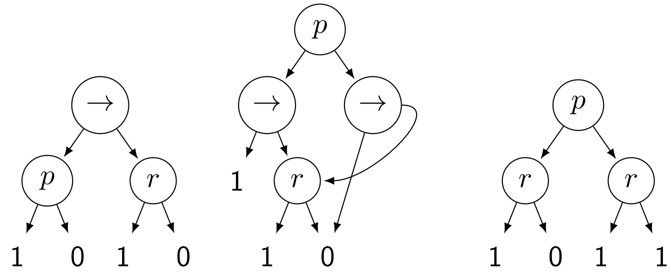
Left argument:



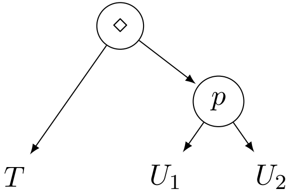
回顾:(2) If p not in U, then Replace



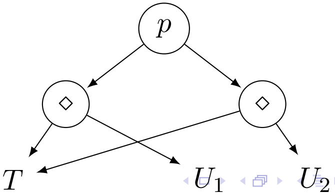
Left argument:



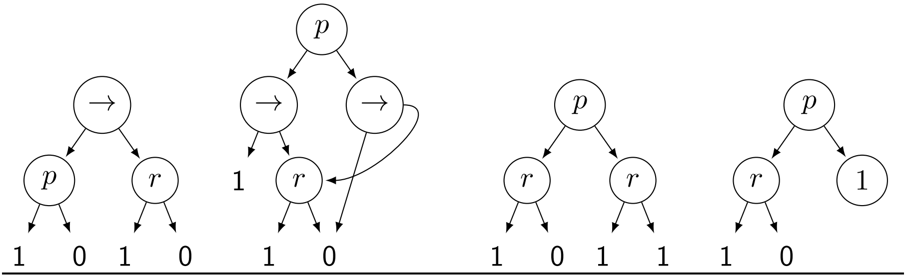
回顾:(3)If p not in T, then replace

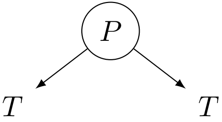


by

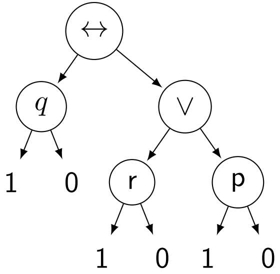


Left argument:

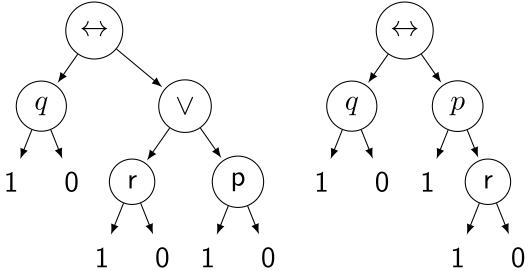


回顾:Replacing  by T is called *elimination*

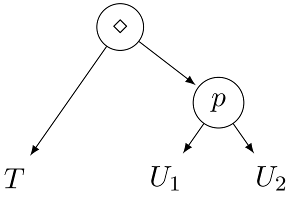
Right argument:



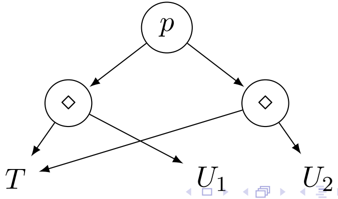
Right argument:



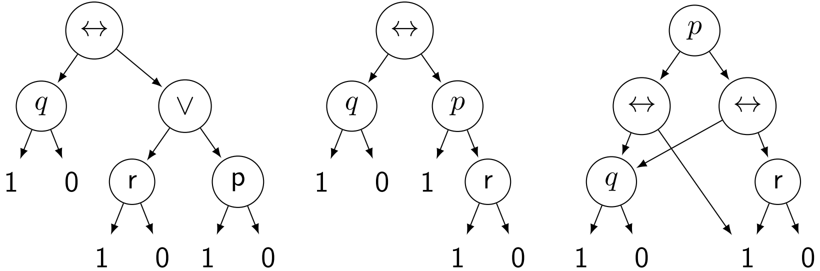
回顾:(3)If p not in T, then replace



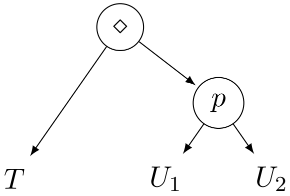
by



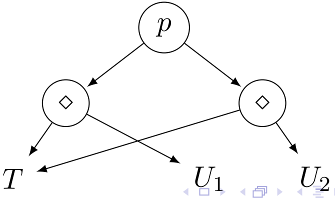
Right argument:



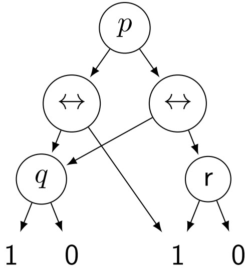
回顾:(3)If p not in T, then replace



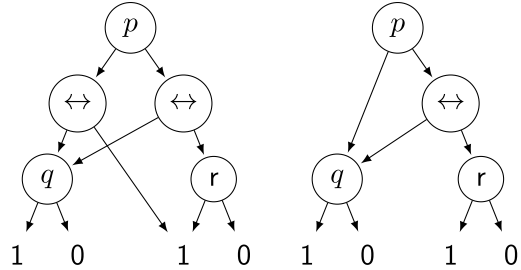
by



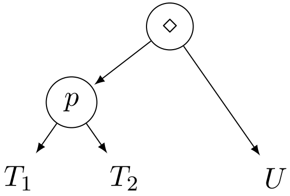
Right argument: (续上页):



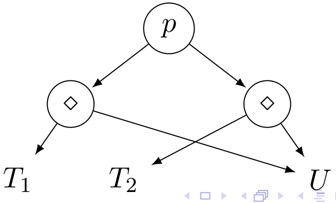
Right argument: (续上页):



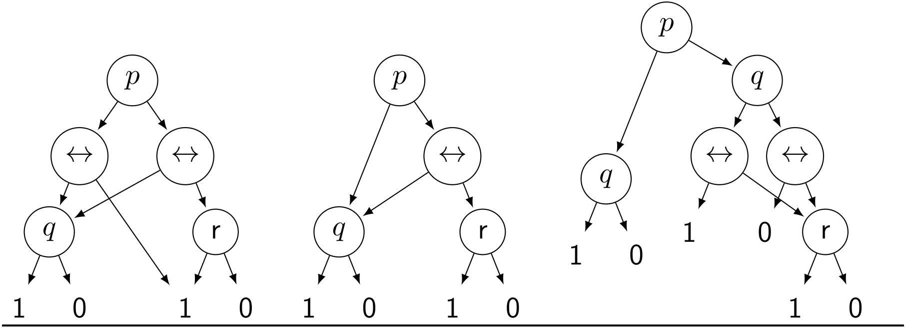
回顾:(2) If p not in U, then Replace



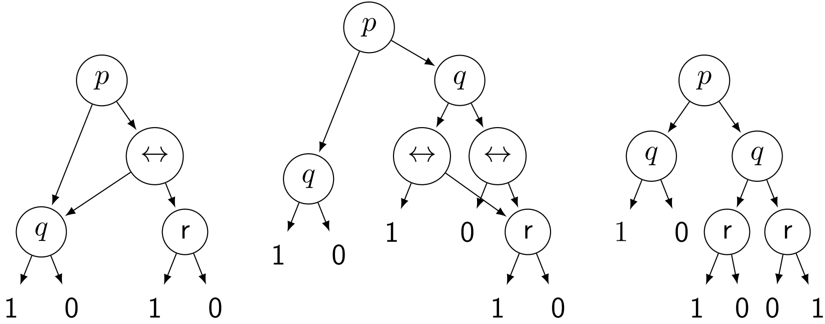
by



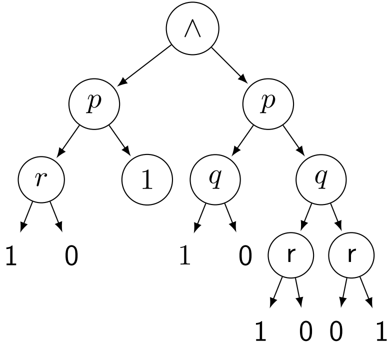
Right argument: (续上页):

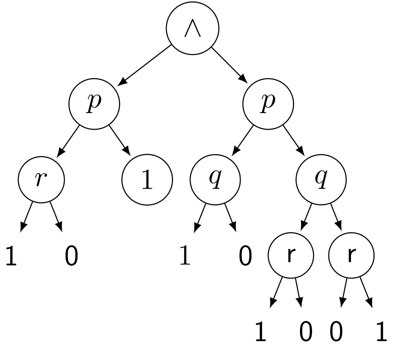


Right argument: (续上页):

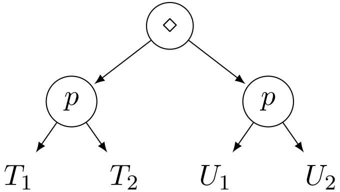


Now we have computed the ROBDDs of both arguments of $\wedge$ in the original formula, and it remains to apply $\wedge$ on these two

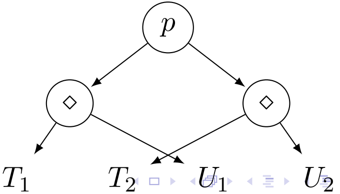


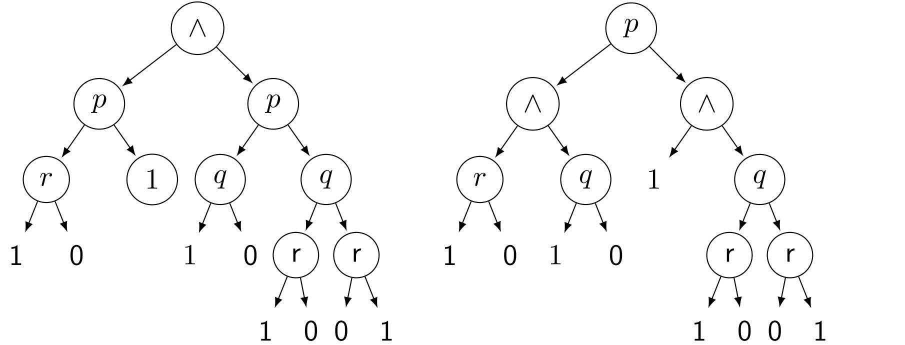


回顾:(1) Replace

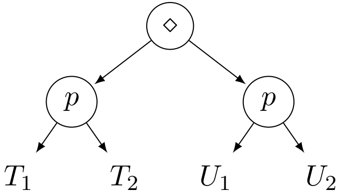


by

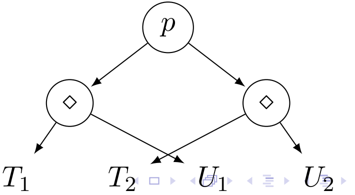


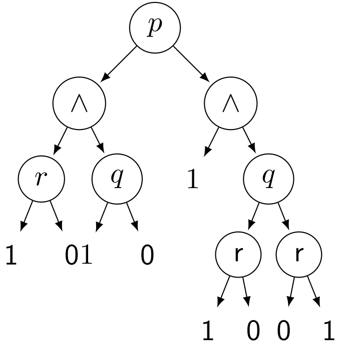


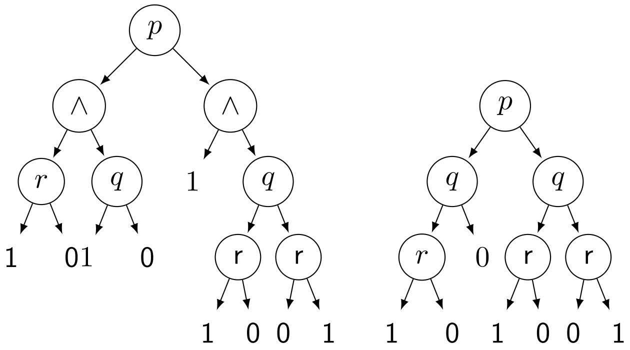
回顾:(1) Replace

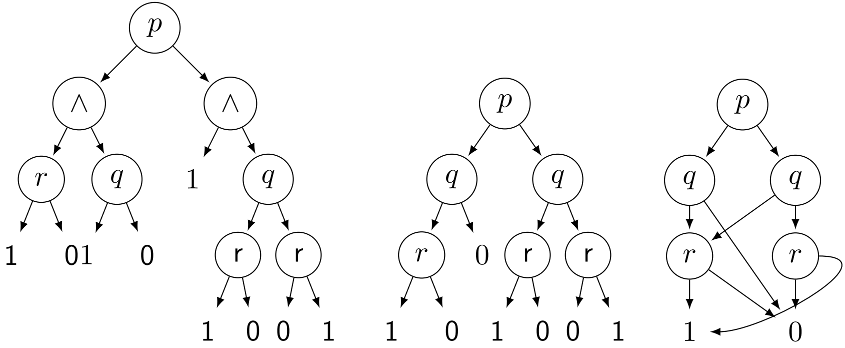


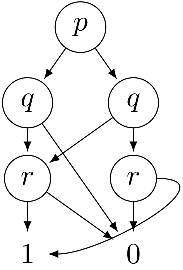
by




p
∧
∧
r
q
1
q
1
01
0
r
r
1
0
0
1


p
∧
∧
r
q
1
q
1
01
0
r
r
1
0
0
1
p
q
q
r
0
r
r
1
0
1
0
0
1


p
∧
∧
r
q
1
q
1
01
0
r
r
1
0
0
1
p
q
q
r
0
r
r
1
0
1
0
0
1
p
q
q
r
r
1
0



We computed the ROBDD of the formula

$$(p \to r) \wedge (q \leftrightarrow (r \vee p))$$

w.r.t the order $p < q < r$.

Doing this by hand in all detail is quite some work, but the steps are very systematic and suitable for implementation

Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
   - Stage 7.1: Compute $ROBDD(\phi)$
   - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
   - Stage 8.1: Express CTL operators
   - Stage 8.2: Compute EX, EG, EU
   - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \to t\}$

第 *8* 阶: CTL model checking by ROBDDs

How can we do CTL model checking by *representing large sets of states* by ROBDDs?

First *recall* the abstract algorithm for CTL model checking ▸ Basic Idea

第 *8* 阶: CTL model checking by ROBDDs

How can we do CTL model checking by *representing large sets of states* by ROBDDs?

First *recall* the abstract algorithm for CTL model checking Basic Idea

第 8 阶: **CTL model checking by ROBDDs**

How can we do CTL model checking by *representing large sets of states* by ROBDDs?

First *recall* the abstract algorithm for CTL model checking ▸ Basic Idea

核心思路: Iteratively update ROBDD of a *boolean function*, e.g., ROBDD(f_n), until ROBDD(f_n)=ROBDD(f_{n-1})

例: 求解 $S_{\mathrm{EG}\phi} = T_n$

回顾: 算法: 求解 $S_{\mathrm{EG}\phi} = T_n$

$T_0 := S_\phi$; $n := 0$;
repeat
  $T_{n+1} := T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$; $n = n + 1$;
until $T_n = T_{n-1}$

Here, $f_n(s) = 1 \leftrightarrow s \in T_n$

For convenience, define ROBDD(T_n) $\equiv$ ROBDD(f_n)

问题转换: How to compute ROBDD($T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$)?

核心思路: Iteratively update ROBDD of a *boolean function*, e.g., ROBDD(f_n), until ROBDD(f_n)=ROBDD(f_{n-1})

例: 求解 $S_{\mathrm{EG}\phi} = T_n$

回顾: 算法: 求解 $S_{\mathrm{EG}\phi} = T_n$

$T_0 := S_\phi;\ n := 0;$
repeat
  $T_{n+1} := T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\};\ n = n+1;$
until $T_n = T_{n-1}$

Here, $f_n(s) = 1 \leftrightarrow s \in T_n$

For convenience, define ROBDD(T_n) $\equiv$ ROBDD(f_n)

问题转换: How to compute ROBDD($T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$)?

核心思路: Iteratively update ROBDD of a *boolean function*, e.g., ROBDD(f_n), until ROBDD(f_n)=ROBDD(f_{n-1})

例: 求解 $S_{\mathrm{EG}\phi} = T_n$

回顾: 算法: 求解 $S_{\mathrm{EG}\phi} = T_n$

$T_0 := S_\phi$; $n := 0$;
repeat
  $T_{n+1} := T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$; $n = n + 1$;
until $T_n = T_{n-1}$

Here, $f_n(s) = 1 \leftrightarrow s \in T_n$

For convenience, define ROBDD(T_n) $\equiv$ ROBDD(f_n)

问题转换: How to compute ROBDD($T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$)?

核心思路: Iteratively update ROBDD of a *boolean function*, e.g., ROBDD(f_n), until ROBDD(f_n)=ROBDD(f_{n-1})

例: 求解 $S_{\mathrm{EG}\phi} = T_n$

回顾: 算法: 求解 $S_{\mathrm{EG}\phi} = T_n$

$T_0 := S_\phi;\ n := 0;$
repeat
    $T_{n+1} := T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\};\ n = n + 1;$
until $T_n = T_{n-1}$

Here, $f_n(s) = 1 \leftrightarrow s \in T_n$

For convenience, define ROBDD$(T_n) \equiv$ ROBDD(f_n)

问题转换: How to compute ROBDD($T_n \cap \{s \in S_\phi \mid \exists t \in T_n : s \to t\}$)?

Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
   - Stage 7.1: Compute $ROBDD(\phi)$
   - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
   - Stage 8.1: Express CTL operators
   - Stage 8.2: Compute EX, EG, EU
   - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \to t\}$

回顾:

$$\neg \mathrm{AF}\ \phi \equiv \mathrm{EG}\ \neg\phi$$

$$\neg \mathrm{EF}\ \phi \equiv \mathrm{AG}\ \neg\phi$$

$$\neg \mathrm{AX}\ \phi \equiv \mathrm{EX}\ \neg\phi$$

$$\mathrm{AF}\ \phi \equiv \mathrm{A}[\top\ \mathrm{U}\ \phi]$$

$$\mathrm{EF}\ \phi \equiv \mathrm{E}[\top\ \mathrm{U}\ \phi]$$

Stage 8.1: *All CTL operators* can be *expressed* in (1) *boolean operators* $(\neg, \wedge, \rightarrow, \vee)$ and (2) *EX, EG, EU*

- (1) solved in stage 7 Stage 7.1 Stage 7.2
- (2) to be solved in Stage 8.2 Stage 8.2

回顾:

$$\neg \mathrm{AF}\ \phi \equiv \mathrm{EG}\ \neg\phi$$

$$\neg \mathrm{EF}\ \phi \equiv \mathrm{AG}\ \neg\phi$$

$$\neg \mathrm{AX}\ \phi \equiv \mathrm{EX}\ \neg\phi$$

$$\mathrm{AF}\ \phi \equiv \mathrm{A}[\top\ \mathrm{U}\ \phi]$$

$$\mathrm{EF}\ \phi \equiv \mathrm{E}[\top\ \mathrm{U}\ \phi]$$

Stage 8.1: *All CTL operators* can be *expressed* in (1) *boolean operators* $(\neg, \wedge, \rightarrow, \vee)$ and (2) *EX, EG, EU*

- (1) solved in stage 7 Stage 7.1 Stage 7.2
- (2) to be solved in Stage 8.2 Stage 8.2

Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
   - Stage 7.1: Compute $ROBDD(\phi)$
   - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
   - Stage 8.1: Express CTL operators
   - Stage 8.2: Compute EX, EG, EU
   - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \to t\}$

Stage 8.2: For *computing* EX, EG, EU, we only needed the building blocks:

- set
- union $\cup$, intersection $\cap$
- computing
$$\{s \in T \mid \exists t \in U : s \to t\}$$
for a given transition relation $\to$ and given sets T, U

Stage 8.2: For *computing* EX, EG, EU, we only needed the building blocks:

- set
  - **基本思路:** Sets are described by **boolean functions**: an element is in the set if and only if the boolean function yields true $f_U : B^n \to B$

$$s \in U \leftrightarrow f_U(s) = 1$$

$$\mathrm{ROBDD}(U) \equiv \mathrm{ROBDD}(f_U(s)), \text{ a.k.a., } \mathrm{ROBDD}(S_\phi) \equiv \mathrm{ROBDD}(\phi)$$

- union $\cup$, intersection $\cap$
- computing

$$\{s \in T \mid \exists t \in U : s \to t\}$$

for a given transition relation $\to$ and given sets T, U

Stage 8.2: For *computing* EX, EG, EU, we only needed the building blocks:

- set
- union $\cup$, intersection $\cap$
  - **基本思路:** Union and intersection correspond to $\vee$ and $\wedge$, for which we already gave an algorithm for ROBDD representations
  - $S_{\phi\vee\psi} = S_\phi \cup S_\psi$, $S_{\phi\wedge\psi} = S_\phi \cap S_\psi$
  - $\mathrm{ROBDD}(S_\phi \cup S_\psi) = \mathrm{ROBDD}(S_{\phi\vee\psi}) \equiv \mathrm{ROBDD}(\phi \vee \psi)$
  - $\mathrm{ROBDD}(S_\phi \cap S_\psi) = \mathrm{ROBDD}(S_{\phi\wedge\psi}) \equiv \mathrm{ROBDD}(\phi \wedge \psi)$
- computing

$$\{s \in T \mid \exists t \in U : s \to t\}$$

for a given transition relation $\to$ and given sets T, U

Stage 8.2: For *computing* EX, EG, EU, we only needed the building blocks:

- set
- union $\cup$, intersection $\cap$
- computing
$$\{s \in T \mid \exists t \in U : s \to t\}$$
for a given transition relation $\to$ and given sets T, U
    - to be solve in *Stage 8.3* ▸ Stage 8.3

Stage 8.2: 小总结: 如何转换

Variables $a, b, \ldots \equiv$ *boolean variables* $a_1, a_2, \ldots, a_n$

- As all variables are from finite sets, they can be encoded *binary*
- So we assume that we *only* have *boolean variables* $a_1, \ldots, a_n$

Before

$a, b, \ldots$: 1..100;

After

$a_1, a_2, \ldots, a_n$: BOOL;

A *state* $s = (a_1, a_2, \ldots, a_n)$
The *sets* S, T and U of *states*

- boolean functions on the variables: $s \in U \leftrightarrow f_U(s) = 1$

ROBDD on S: $\text{ROBDD}(S_\phi) \equiv \text{ROBDD}(\phi)$
ROBDD on $\cup$ *and* $\cap$:

- $\text{ROBDD}(S_\phi \cup S_\psi) = \text{ROBDD}(S_{\phi\vee\psi}) \equiv \text{ROBDD}(\phi \vee \psi)$
- $\text{ROBDD}(S_\phi \cap S_\psi) = \text{ROBDD}(S_{\phi\wedge\psi}) \equiv \text{ROBDD}(\phi \wedge \psi)$

Stage 8.2: 小总结: 如何转换

Variables $a, b, \ldots \equiv$ *boolean variables* $a_1, a_2, \ldots, a_n$

- As all variables are from finite sets, they can be encoded *binary*
- So we assume that we *only* have *boolean variables* $a_1, \ldots, a_n$

Before

$a, b, \ldots$: 1..100;

After

$a_1, a_2, \ldots, a_n$: BOOL;

A *state* $s = (a_1, a_2, \ldots, a_n)$

The *sets* S, T and U of *states*

- boolean functions on the variables: $s \in U \leftrightarrow f_U(s) = 1$

ROBDD on S: $\mathrm{ROBDD}(S_\phi) \equiv \mathrm{ROBDD}(\phi)$

ROBDD on $\cup$ *and* $\cap$:

- $\mathrm{ROBDD}(S_\phi \cup S_\psi) = \mathrm{ROBDD}(S_{\phi\vee\psi}) \equiv \mathrm{ROBDD}(\phi \vee \psi)$
- $\mathrm{ROBDD}(S_\phi \cap S_\psi) = \mathrm{ROBDD}(S_{\phi\wedge\psi}) \equiv \mathrm{ROBDD}(\phi \wedge \psi)$

Stage 8.2: 小总结: 如何转换

Variables $a, b, \ldots \equiv$ *boolean variables* $a_1, a_2, \ldots, a_n$

- As all variables are from finite sets, they can be encoded *binary*
- So we assume that we *only* have *boolean variables* $a_1, \ldots, a_n$

Before

$a, b, \ldots$: 1..100;

After

$a_1, a_2, \ldots, a_n$: BOOL;

A *state* $s = (a_1, a_2, \ldots, a_n)$
The *sets* S, T and U of *states*

- boolean functions on the variables: $s \in U \leftrightarrow f_U(s) = 1$

ROBDD on S: $\mathrm{ROBDD}(S_\phi) \equiv \mathrm{ROBDD}(\phi)$
ROBDD on $\cup$ *and* $\cap$:

- $\mathrm{ROBDD}(S_\phi \cup S_\psi) = \mathrm{ROBDD}(S_{\phi\vee\psi}) \equiv \mathrm{ROBDD}(\phi \vee \psi)$
- $\mathrm{ROBDD}(S_\phi \cap S_\psi) = \mathrm{ROBDD}(S_{\phi\wedge\psi}) \equiv \mathrm{ROBDD}(\phi \wedge \psi)$

Stage 8.2: 小总结: 如何转换

Variables $a, b, \ldots \equiv$ *boolean variables* $a_1, a_2, \ldots, a_n$

- As all variables are from finite sets, they can be encoded *binary*
- So we assume that we *only* have *boolean variables* $a_1, \ldots, a_n$

Before

a, b,... : 1..100;

After

a_1,a_2,...,a_n : BOOL;

A *state* $s = (a_1, a_2, \ldots, a_n)$

The *sets* S, T and U of *states*

- boolean functions on the variables: $s \in U \leftrightarrow f_U(s) = 1$

ROBDD on S: $\mathrm{ROBDD}(S_\phi) \equiv \mathrm{ROBDD}(\phi)$

ROBDD on $\cup$ *and* $\cap$:

- $\mathrm{ROBDD}(S_\phi \cup S_\psi) = \mathrm{ROBDD}(S_{\phi\vee\psi}) \equiv \mathrm{ROBDD}(\phi \vee \psi)$
- $\mathrm{ROBDD}(S_\phi \cap S_\psi) = \mathrm{ROBDD}(S_{\phi\wedge\psi}) \equiv \mathrm{ROBDD}(\phi \wedge \psi)$

Stage 8.2: 小总结: 如何转换

Variables $a, b, \ldots \equiv$ *boolean variables* $a_1, a_2, \ldots, a_n$

- As all variables are from finite sets, they can be encoded *binary*
- So we assume that we *only* have *boolean variables* $a_1, \ldots, a_n$

Before

$a, b, \ldots$: 1..100;

After

$a_1, a_2, \ldots, a_n$: BOOL;

A *state* $s = (a_1, a_2, \ldots, a_n)$
The *sets* S, T and U of *states*

- boolean functions on the variables: $s \in U \leftrightarrow f_U(s) = 1$

ROBDD on S: $\mathrm{ROBDD}(S_\phi) \equiv \mathrm{ROBDD}(\phi)$
ROBDD on $\cup$ *and* $\cap$:

- $\mathrm{ROBDD}(S_\phi \cup S_\psi) = \mathrm{ROBDD}(S_{\phi\vee\psi}) \equiv \mathrm{ROBDD}(\phi \vee \psi)$
- $\mathrm{ROBDD}(S_\phi \cap S_\psi) = \mathrm{ROBDD}(S_{\phi\wedge\psi}) \equiv \mathrm{ROBDD}(\phi \wedge \psi)$

Outline of a BDD algorithm

1. Stage 1: Boolean variables
2. Stage 2: Boolean functions
3. Stage 3: Decision Tree
4. Stage 4: Ordered decision tree
5. Stage 5: Reduced ordered decision tree (elimination)
6. Stage 6: ROBDD (merging and elimination)
7. Stage 7: Compute ROBDD
   - Stage 7.1: Compute $ROBDD(\phi)$
   - Stage 7.2: Compute $\diamond(T, U)$
8. Stage 8: ROBDD-CTL
   - Stage 8.1: Express CTL operators
   - Stage 8.2: Compute EX, EG, EU
   - Stage 8.3: Compute $V = \{s \in T \mid \exists t \in U : s \to t\}$

Stage 8.3: Computing

$$V = \{s \in T \mid \exists t \in U : s \to t\}$$

for a given transition relation $\to$ and given sets T, U

准备:

- Write a_i' as shorthand for next(a_i)
- The transition relation $\to$ is given by a boolean function (P) on $a_1, \ldots, a_n, a_1', \ldots, a_n'$, again in ROBDD representation

$$P(a_1, \ldots, a_n, a_1', \ldots, a_n') \Leftrightarrow P_1 \wedge P_2$$

where

- $P_1 = (a_1, \ldots, a_n) \to (a_1', \ldots, a_n')$
- $P_2 = (a_1', \ldots, a_n') \in U$

Stage 8.3: Computing

$$V = \{s \in T \mid \exists t \in U : s \to t\}$$

for a given transition relation $\to$ and given sets T, U

准备:

- Write a'_i as shorthand for next(a_i)
- The transition relation $\to$ is given by a boolean function (P) on $a_1, \ldots, a_n, a'_1, \ldots, a'_n$, again in ROBDD representation

$$P(a_1, \ldots, a_n, a'_1, \ldots, a'_n) \Leftrightarrow P_1 \wedge P_2$$

where

- $P_1 = (a_1, \ldots, a_n) \to (a'_1, \ldots, a'_n)$
- $P_2 = (a'_1, \ldots, a'_n) \in U$

Step 1: Compute ROBDD of $P(\cdots)$, i.e., ROBDD(P)

$$P(a_1, \dots, a_n, a'_1, \dots, a'_n) \Leftrightarrow P_1 \wedge P_2$$

where

- $P_1 = (a_1, \dots, a_n) \rightarrow (a'_1, \dots, a'_n)$
  -
- $P_2 = (a'_1, \dots, a'_n) \in U$
  -
-

Step 1: Compute ROBDD of $P(\cdots)$, i.e., ROBDD(P)

$$P(a_1, \ldots, a_n, a'_1, \ldots, a'_n) \Leftrightarrow P_1 \wedge P_2$$

where

- $P_1 = (a_1, \ldots, a_n) \rightarrow (a'_1, \ldots, a'_n)$
  - ROBDD(P_1) was given by translation relation $\rightarrow$
- $P_2 = (a'_1, \ldots, a'_n) \in U$
  -
-

Step 1: Compute ROBDD of $P(\cdots)$, i.e., ROBDD(P)

$$P(a_1, \ldots, a_n, a_1', \ldots, a_n') \Leftrightarrow P_1 \wedge P_2$$

where

- $P_1 = (a_1, \ldots, a_n) \rightarrow (a_1', \ldots, a_n')$
  - ROBDD(P_1) was given by translation relation $\rightarrow$
- $P_2 = (a_1', \ldots, a_n') \in U$
  - When using the order $a_1 < \cdots < a_n < a_1' < \cdots < a_n'$, ROBDD($P_2$) is obtained by just replacing every a_i by a_i' in ROBDD(U)
-

Step 1: Compute ROBDD of $P(\cdots)$, i.e., ROBDD(P)

$$P(a_1, \ldots, a_n, a'_1, \ldots, a'_n) \Leftrightarrow P_1 \wedge P_2$$

where

- $P_1 = (a_1, \ldots, a_n) \rightarrow (a'_1, \ldots, a'_n)$
  - ROBDD(P_1) was given by translation relation $\rightarrow$
- $P_2 = (a'_1, \ldots, a'_n) \in U$
  - When using the order $a_1 < \cdots < a_n < a'_1 < \cdots < a'_n$, ROBDD($P_2$) is obtained by just replacing every a_i by a'_i in ROBDD(U)
- Now, ROBDD(P)=ROBDD($P_1 \wedge P_2$)=*apply*(*ROBDD*(P_1), *ROBDD*(P_2), $\wedge$)

Step 2: Compute ROBDD(S_{P_e})

$$S_{P_e} = \{(a_1, \ldots, a_n) \mid \exists a'_1, \ldots, a'_n : P(a_1, \ldots, a_n, a'_1, \ldots, a'_n)\}$$

Observe that for every boolean variable x:

$$\exists x : \phi \equiv \underbrace{\phi[x := \mathbf{T}] \vee \phi[x := \mathbf{F}]}_{\text{computable}}$$

Applying this n times, for $x = a'_1, a'_2, \ldots, a'_n$, all '$\exists$'s are eliminated, yielding an ROBDD over $a_1, \ldots, a_n$

Step 3: Compute ROBDD of V

$$V = \{s \in T \mid \exists t \in U : s \to t\} = T \cap S_{P_e}$$

ROBDD(V)=ROBDD($T \cap S_{P_e}$)=*apply*(*ROBDD*(T), *ROBDD*(S_{P_e}), $\wedge$)

Step 2: Compute ROBDD(S_{P_e})

$$S_{P_e} = \{(a_1, \ldots, a_n) \mid \exists a'_1, \ldots, a'_n : P(a_1, \ldots, a_n, a'_1, \ldots, a'_n)\}$$

Observe that for every boolean variable x:

$$\exists x : \phi \equiv \underbrace{\phi[x := \mathbf{T}] \vee \phi[x := \mathbf{F}]}_{\text{computable}}$$

Applying this n times, for $x = a'_1, a'_2, \ldots, a'_n$, all '$\exists$'s are eliminated, yielding an ROBDD over $a_1, \ldots, a_n$

Step 3: Compute ROBDD of V

$$V = \{s \in T \mid \exists t \in U : s \to t\} = T \cap S_{P_e}$$

ROBDD(V)=ROBDD($T \cap S_{P_e}$)=*apply*(*ROBDD*(T), *ROBDD*(S_{P_e}), $\wedge$)

Stage 8 步骤小结:

- Stage 8.1: Express *all CTL operators* in
  1. *boolean operators* $(\neg, \wedge, \rightarrow, \vee)$
     - Compute ROBDD of boolean operators: ▸ Stage 7.1 ▸ Stage 7.2
  2. *EX, EG, EU*
- Stage 8.2: Compute ROBDD of *EX, EG, EU* ▸ Example: $S_{\mathrm{EG}\phi} = t_n$
  - solved: t_0, S_ϕ, $\cap$
  - problems left: ROBDD(V), where $V = \{s \in T \mid \exists t \in U : s \rightarrow t\}$
- Stage 8.3:
  - step 1: compute ROBDD(P), where $P = P_1 \wedge P_2$, $P_1 = (a_1, \ldots, a_n) \rightarrow (a'_1, \ldots, a'_n)$, $P_2 = (a'_1, \ldots, a'_n) \in U$
  - step 2: compute ROBDD(S_{P_e}), where

$$S_{P_e} = \{(a_1, \ldots, a_n) \mid \exists a'_1, \ldots, a'_n : P(a_1, \ldots, a_n, a'_1, \ldots, a'_n)\}$$

  - step 3: compute ROBDD(V), where
    - $V = T \cap S_{P_e}$

Stage 8 步骤小结:

- Stage 8.1: Express *all CTL operators* in
  1. *boolean operators* $(\neg, \wedge, \rightarrow, \vee)$
     - Compute ROBDD of boolean operators: Stage 7.1 Stage 7.2
  2. *EX, EG, EU*
- Stage 8.2: Compute ROBDD of *EX, EG, EU* Example: $S_{\mathrm{EG}\phi} = t_n$
  - solved: t_0, S_ϕ, $\cap$
  - problems left: ROBDD(V), where $V = \{s \in T \mid \exists t \in U : s \rightarrow t\}$
- Stage 8.3:
  - step 1: compute ROBDD(P), where $P = P_1 \wedge P_2$,
    $P_1 = (a_1, \ldots, a_n) \rightarrow (a'_1, \ldots, a'_n)$, $P_2 = (a'_1, \ldots, a'_n) \in U$
  - step 2: compute ROBDD(S_{P_e}), where

$$S_{P_e} = \{(a_1, \ldots, a_n) \mid \exists a'_1, \ldots, a'_n : P(a_1, \ldots, a_n, a'_1, \ldots, a'_n)\}$$

  - step 3: compute ROBDD(V), where
    - $V = T \cap S_{P_e}$

Stage 8 步骤小结:

- Stage 8.1: Express *all CTL operators* in
  1. *boolean operators* $(\neg, \wedge, \rightarrow, \vee)$
     - Compute ROBDD of boolean operators: Stage 7.1 Stage 7.2
  2. *EX, EG, EU*
- Stage 8.2: Compute ROBDD of *EX, EG, EU* Example: $S_{\mathrm{EG}\phi} = t_n$
  - solved: t_0, S_ϕ, $\cap$
  - problems left: $\mathrm{ROBDD}(V)$, where $V = \{s \in T \mid \exists t \in U : s \rightarrow t\}$
- Stage 8.3:
  - step 1: compute ROBDD(P), where $P = P_1 \wedge P_2$, $P_1 = (a_1, \ldots, a_n) \rightarrow (a'_1, \ldots, a'_n)$, $P_2 = (a'_1, \ldots, a'_n) \in U$
  - step 2: compute ROBDD(S_{P_e}), where

$$S_{P_e} = \{(a_1, \ldots, a_n) \mid \exists a'_1, \ldots, a'_n : P(a_1, \ldots, a_n, a'_1, \ldots, a'_n)\}$$

  - step 3: compute ROBDD(V), where
    - $V = T \cap S_{P_e}$

Stage 8 步骤小结:

- Stage 8.1: Express *all CTL operators* in
  1. *boolean operators* $(\neg, \wedge, \rightarrow, \vee)$
     - Compute ROBDD of boolean operators: ▸ Stage 7.1 ▸ Stage 7.2
  2. *EX, EG, EU*
- Stage 8.2: Compute ROBDD of *EX, EG, EU* ▸ Example: $S_{\mathrm{EG}\phi} = t_n$
  - solved: t_0, S_ϕ, $\cap$
  - problems left: $\mathrm{ROBDD}(V)$, where $V = \{s \in T \mid \exists t \in U : s \rightarrow t\}$
- Stage 8.3:
  - step 1: compute ROBDD(P), where $P = P_1 \wedge P_2$, $P_1 = (a_1, \ldots, a_n) \rightarrow (a'_1, \ldots, a'_n)$, $P_2 = (a'_1, \ldots, a'_n) \in U$
  - step 2: compute ROBDD(S_{P_e}), where
    $$S_{P_e} = \{(a_1, \ldots, a_n) \mid \exists a'_1, \ldots, a'_n : P(a_1, \ldots, a_n, a'_1, \ldots, a'_n)\}$$
  - step 3: compute ROBDD(V), where
    - $V = T \cap S_{P_e}$

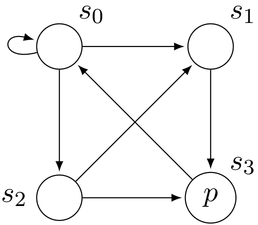
例: ROBDD-CTL 求解

Given a transition system $\mathcal{M} = (S, \to, L)$, where $S = \{s_0, s_1, s_2, s_3\}$ $\to = \{(s_0, s_0), (s_0, s_1), (s_0, s_2), (s_1, s_3), (s_2, s_1), (s_2, s_3), (s_3, s_0)\}$, $L(s_3) = \{p\}$.

Verify: $\mathcal{M}, s_0 \vDash \mathsf{AF}\ p$

例: ROBDD-CTL 求解

Given a transition system $\mathcal{M} = (S, \rightarrow, L)$, where $S = \{s_0, s_1, s_2, s_3\}$ $\rightarrow = \{(s_0, s_0), (s_0, s_1), (s_0, s_2), (s_1, s_3), (s_2, s_1), (s_2, s_3), (s_3, s_0)\}$, $L(s_3) = \{p\}$.
Verify: $\mathcal{M}, s_0 \models \mathsf{AF}\ p$



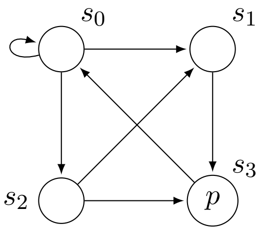
Stage 8.1: compute ROBDD(AF p)

1. AF $p = \neg$EG$\neg p$
2. ROBDD(AF p) = ROBDD(EG$\neg p \rightarrow$ **F**)
   =apply(ROBDD(EG$\neg p$), ROBDD(**F**), $\rightarrow$)
3. compute ROBDD(EG$\neg p$) in *Stage 8.2*

例: ROBDD-CTL 求解

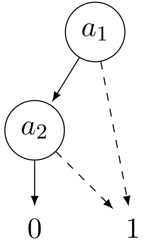
Given a transition system $\mathcal{M} = (S, \rightarrow, L)$, where $S = \{s_0, s_1, s_2, s_3\}$ $\rightarrow = \{(s_0, s_0), (s_0, s_1), (s_0, s_2), (s_1, s_3), (s_2, s_1), (s_2, s_3), (s_3, s_0)\}$, $L(s_3) = \{p\}$.

Verify: $\mathcal{M}, s_0 \vDash \text{AF } p$



Stage 8.2: Compute ROBDD(EG$\neg p$) ▸ $S_{\text{EG}\phi} = t_n$

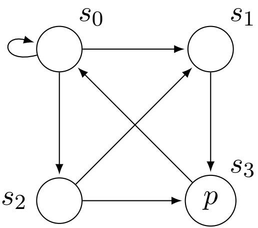
1. Define a state as a pair of variables (a_1, a_2), where $a_1, a_2 \in \{0, 1\}$
   - i.e., $s_0 = (0, 0)$, $s_1 = (0, 1)$, $s_2 = (1, 0)$, $s_3 = (1, 1)$
2. $S_\phi = S_{\neg p} = \{s_0, s_1, s_2\} = \{(0, 0), (0, 1), (1, 0)\}$
3. $t_0 = \text{ROBDD}(S_\phi)$



t_0

例: ROBDD-CTL 求解

Given a transition system $\mathcal{M} = (S, \rightarrow, L)$, where $S = \{s_0, s_1, s_2, s_3\}$ $\rightarrow = \{(s_0, s_0), (s_0, s_1), (s_0, s_2), (s_1, s_3), (s_2, s_1), (s_2, s_3), (s_3, s_0)\}$, $L(s_3) = \{p\}$.
Verify: $\mathcal{M}, s_0 \vDash$ AF p

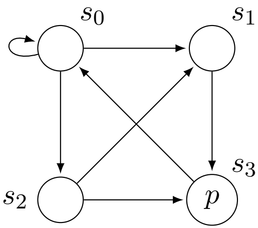


Stage 8.2: Compute ROBDD(EG$\neg p$) ▸ $S_{\text{EG}\phi} = t_n$

4. Iteratively compute t_{n+1}, where
   - t_{n+1} :=apply(t_n,ROBDD(V), $\wedge$)
   - $V = \{s \in T \mid \exists t \in U : s \rightarrow t\}$ (ROBDD(V) computed in *Stage 8.2*)
     - $T = S_\phi = S_{\neg p} = \{s_0, s_1, s_2\} = \{(0,0), (0,1), (1,0)\}$
     - ROBDD(U)=t_n

Stage 8.3: compute ROBDD(V), where

- $V = \{s \in T \mid \exists t \in U : s \to t\}$
- $T = \{(0,0),(0,1),(1,0)\}$, ROBDD($U$)=$t_n$

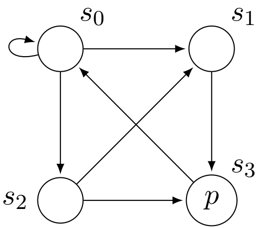


Stage 8.3: compute ROBDD(V), where

- $V = \{s \in T \mid \exists t \in U : s \to t\}$
- $T = \{(0,0),(0,1),(1,0)\}$, ROBDD($U$)=$t_n$

Step 1:

- $P_1 = (a_1, a_2) \to (a'_1, a'_2)$
- $P_2 = (a'_1, a'_2) \in U$
- ROBDD(P)=ROBDD($P_1 \wedge P_2$)=*apply*(*ROBDD*(P_1), *ROBDD*(P_2), $\wedge$)

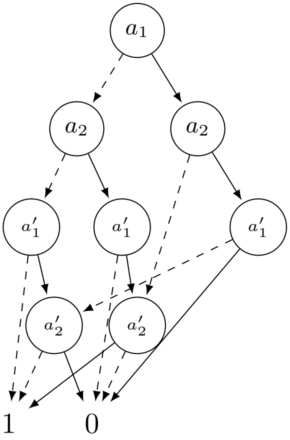


Stage 8.3: compute ROBDD(V), where

- $V = \{s \in T \mid \exists t \in U : s \to t\}$
- $T = \{(0,0),(0,1),(1,0)\}$, ROBDD($U$)=$t_n$

Step 1:

- $P_1 = (a_1, a_2) \to (a'_1, a'_2)$
  - $P_1 = \{(s_0,s_0),(s_0,s_1),(s_0,s_2),(s_1,s_3),(s_2,s_1),(s_2,s_3),(s_3,s_0)\} = \{0000,0001,0010,0111,1001,1011,1100\}$
- $P_2 = (a'_1, a'_2) \in U$
- ROBDD(P)=ROBDD($P_1 \wedge P_2$)=*apply*(*ROBDD*(P_1), *ROBDD*(P_2), $\wedge$)



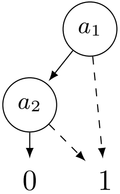
ROBDD(P_1)

Stage 8.3: compute ROBDD(V), where

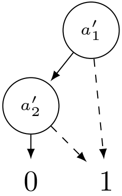
- $V = \{s \in T \mid \exists t \in U : s \rightarrow t\}$
- $T = \{(0,0),(0,1),(1,0)\}$, ROBDD($U$)=$t_n$

Step 1:

- $P_1 = (a_1, a_2) \rightarrow (a_1', a_2')$
- $P_2 = (a_1', a_2') \in U$
    - 0th iteration:
- ROBDD(P)=ROBDD($P_1 \wedge P_2$)=*apply*(*ROBDD*(P_1), *ROBDD*(P_2), $\wedge$)



t_0



ROBDD(P_2)

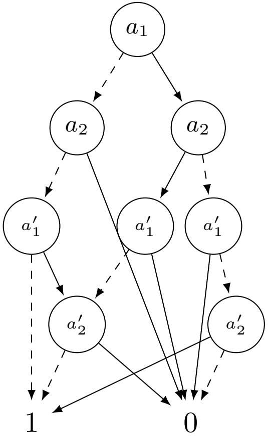
Stage 8.3: compute ROBDD(V), where

- $V = \{s \in T \mid \exists t \in U : s \rightarrow t\}$
- $T = \{(0,0),(0,1),(1,0)\}$, ROBDD($U$)=$t_n$

Step 1:

- $P_1 = (a_1, a_2) \rightarrow (a_1', a_2')$
- $P_2 = (a_1', a_2') \in U$
- ROBDD(P)=ROBDD($P_1 \wedge P_2$)=*apply*(*ROBDD*(P_1), *ROBDD*(P_2), $\wedge$)

  - 0th iteration:



ROBDD(P)

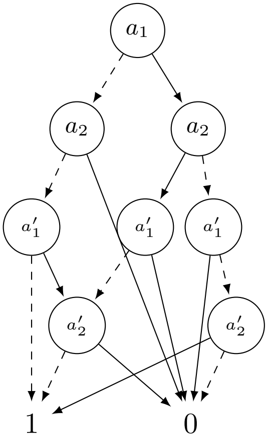
Stage 8.3: compute ROBDD(V), where

- $V = \{s \in T \mid \exists t \in U : s \to t\}$
- $T = \{(0,0),(0,1),(1,0)\}$, ROBDD($U$)=$t_n$

Step 2: Compute ROBDD(S_{P_e}) *(0th step)*

$S_{P_e} = \{(a_1,a_2) \mid \exists a_1', a_2' : P(a_1,a_2,a_1',a_2')\}$

- $S_{P_e} = \{(0,0),(1,0),(1,1)\}$



ROBDD(P)

Stage 8.3: compute ROBDD(V), where

- $V = \{s \in T \mid \exists t \in U : s \to t\}$
- $T = \{(0,0),(0,1),(1,0)\}$, ROBDD($U$)=$t_n$

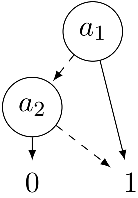
Step 2: Compute ROBDD(S_{P_e}) *(0th step)*

$S_{P_e} = \{(a_1,a_2) \mid \exists a_1', a_2' : P(a_1,a_2,a_1',a_2')\}$

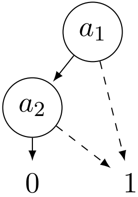
- $S_{P_e} = \{(0,0),(1,0),(1,1)\}$

Step 3: ROBDD(V)=*apply*(*ROBDD*(T), *ROBDD*(S_{P_e}), $\wedge$)

- $T = \{(0,0),(0,1),(1,0)\}$



ROBDD(S_{P_e})



ROBDD(T)

Stage 8.3: compute ROBDD(V), where

- $V = \{s \in T \mid \exists t \in U : s \to t\}$
- $T = \{(0,0),(0,1),(1,0)\}$, ROBDD($U$)=$t_n$

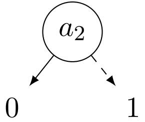
Step 2: Compute ROBDD(S_{P_e}) *(0th step)*

$S_{P_e} = \{(a_1, a_2) \mid \exists a_1', a_2' : P(a_1, a_2, a_1', a_2')\}$

- $S_{P_e} = \{(0,0),(1,0),(1,1)\}$

Step 3: ROBDD(V)=*apply*(*ROBDD*(T), *ROBDD*(S_{P_e}), $\wedge$)

- $T = \{(0,0),(0,1),(1,0)\}$



ROBDD(V)

Stage 8.3: compute ROBDD(V), where

- $V = \{s \in T \mid \exists t \in U : s \to t\}$
- $T = \{(0,0),(0,1),(1,0)\}$, ROBDD($U$)=$t_n$

Step 2: Compute ROBDD(S_{P_e}) *(0th step)*

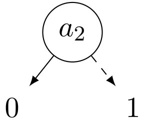
$S_{P_e} = \{(a_1,a_2) \mid \exists a_1', a_2' : P(a_1,a_2,a_1',a_2')\}$

- $S_{P_e} = \{(0,0),(1,0),(1,1)\}$

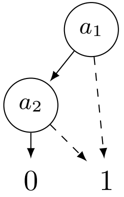
Step 3: ROBDD(V)=*apply*(*ROBDD*(T), *ROBDD*(S_{P_e}), $\wedge$)

- $T = \{(0,0),(0,1),(1,0)\}$

Back to Stage 8.2: t_1 :=apply(t_0,ROBDD(V), $\wedge$)



ROBDD(V)



t_0

Stage 8.3: compute ROBDD(V), where

- $V = \{s \in T \mid \exists t \in U : s \to t\}$
- $T = \{(0,0),(0,1),(1,0)\}$, ROBDD($U$)=$t_n$

Step 2: Compute ROBDD(S_{P_e}) *(0th step)*

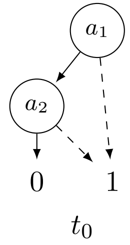
$S_{P_e} = \{(a_1, a_2) \mid \exists a_1', a_2' : P(a_1, a_2, a_1', a_2')\}$

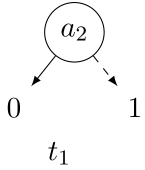
- $S_{P_e} = \{(0,0),(1,0),(1,1)\}$

Step 3: ROBDD(V)=*apply*(*ROBDD*(T), *ROBDD*(S_{P_e}), $\wedge$)

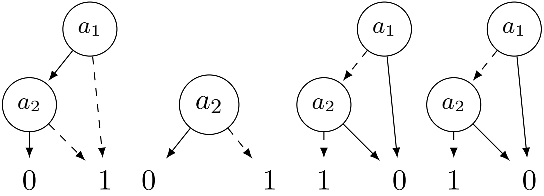
- $T = \{(0,0),(0,1),(1,0)\}$

Back to Stage 8.2: t_1 :=apply(t_0,ROBDD(V), $\wedge$)





Similarly, compute t_2,t_3... (作业：计算 t_2, t_3)



t_0 t_1 t_2 t_3

Observe that $t_2 = t_3$

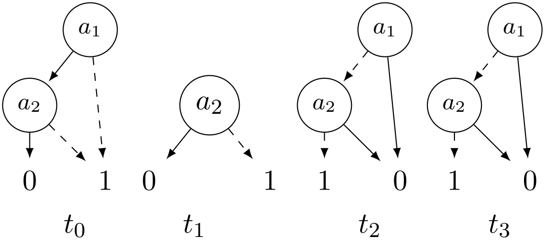
Back to Stage 8.1: ROBDD(EG $\neg p$)=t_2

$S_{\mathrm{EG}\neg p} = \{(0,0)\} = \{s_0\}$

$S_{\mathrm{AF}p} = S_{\neg\mathrm{EG}\neg p} = \{s_1, s_2, s_3\}$

So, $\mathcal{M}, s_0 \not\models$ AF p

Similarly, compute t_2,t_3... (作业: 计算 t_2, t_3)



Observe that $t_2 = t_3$

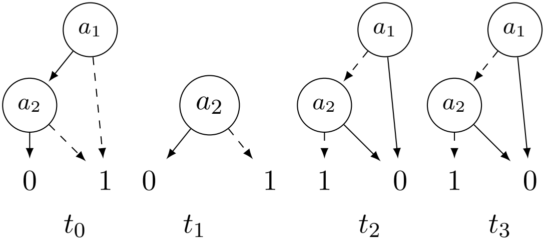
Back to Stage 8.1: ROBDD(EG $\neg p$)=t_2

$S_{\mathrm{EG}\neg p} = \{(0,0)\} = \{s_0\}$

$S_{\mathrm{AF}p} = S_{\neg\mathrm{EG}\neg p} = \{s_1, s_2, s_3\}$

So, $\mathcal{M}, s_0 \not\models$ AF p

Similarly, compute t_2,t_3... (作业: 计算 t_2, t_3)



Observe that $t_2 = t_3$

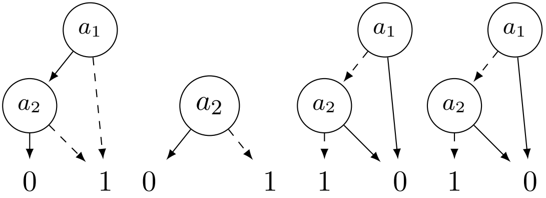
Back to Stage 8.1: ROBDD(EG $\neg p$)=t_2

$S_{\mathrm{EG}\neg p} = \{(0,0)\} = \{s_0\}$

$S_{\mathrm{AF}p} = S_{\neg\mathrm{EG}\neg p} = \{s_1, s_2, s_3\}$

So, $\mathcal{M}, s_0 \not\models$ AF p

Similarly, compute t_2,t_3... (作业: 计算 t_2, t_3)



t_0 t_1 t_2 t_3

Observe that $t_2 = t_3$

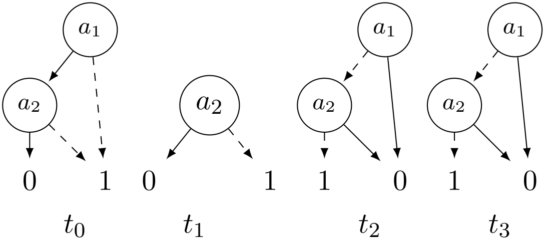
Back to Stage 8.1: ROBDD(EG $\neg p$)=t_2

$S_{\mathrm{EG}\neg p} = \{(0,0)\} = \{s_0\}$

$S_{\mathrm{AF}p} = S_{\neg\mathrm{EG}\neg p} = \{s_1, s_2, s_3\}$

So, $\mathcal{M}, s_0 \not\models$ AF p

Similarly, compute t_2,t_3... (作业: 计算 t_2, t_3)



Observe that $t_2 = t_3$

Back to Stage 8.1: ROBDD(EG $\neg p$)=t_2

$S_{\text{EG}\neg p} = \{(0,0)\} = \{s_0\}$

$S_{\text{AF}p} = S_{\neg\text{EG}\neg p} = \{s_1, s_2, s_3\}$

So, $\mathcal{M}, s_0 \nvDash$ AF p

Stage 8 小结:

- Combining this gives an algorithm to compute the ROBDD of the set states satisfying any CTL formula
- This is essentially the algorithm as it is used in tools like *NuSMV* to do *symbolic model checking*
- In contrast to *explicit state based model checking*, it can deal with very large state spaces.

作业

作业: 模仿 t_1 的求解过程，手工运算 t_2, t_3，给出运算过程.

实验大作业 (可选): 实现 ROBDD 算法, 要求:

- 可以实现至不同 stage，例如，可实现至 ROBDD, 或 ROBDD-CTL。实现的越完整，给分越高。
- 提供源代码、可执行程序、测试文件、相关文档