Binary Decision Diagrams

Gianpiero Cabodi

Politecnico di Torino

Torino, Italy

gianpiero.cabodi@polito.it

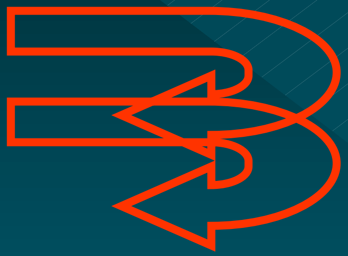
http://staff.polito.it/gianpiero.cabodi/

Outline

- Binary Decision Diagrams: Fundamentals
- Generation of BDDs from Network
- Variable Ordering Related Problems
- Complex Operations with BDDs
- Symbolic Simulation & STE
- Reachability analysis
- Symbolic Model Checking

Binary Decision Diagrams

- Restricted Form of Branching Program
  (graph representation of Boolean function)
- Canonical form (constant time comparison)
- Simple (Polynomial) algorithms to construct e manipulate (Boolean operations: and, or, not, etc.)
- Exponential but practically efficient algorithm for boolean quantification
- Starting Point
  1. If-Then-Else Decomposition
  2. Ordered Decision Tree
  3. Reduced Decision Tree



Decomposition

Reduction

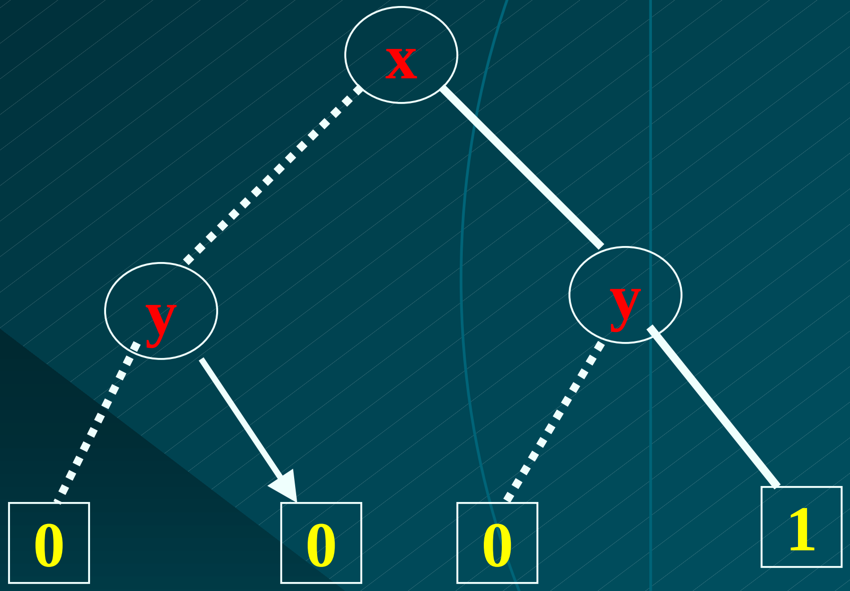
BDDs

- **Boolean functions can be (often)** *succinctly represented* **as** *boolean decision diagrams*.
- BDDs **are easy to manipulate.**
- *Not all boolean functions have a succinct representation*.
- *Use BDDs to represent and manipulate the boolean functions associated with the model checking process.*

Boolean Functions

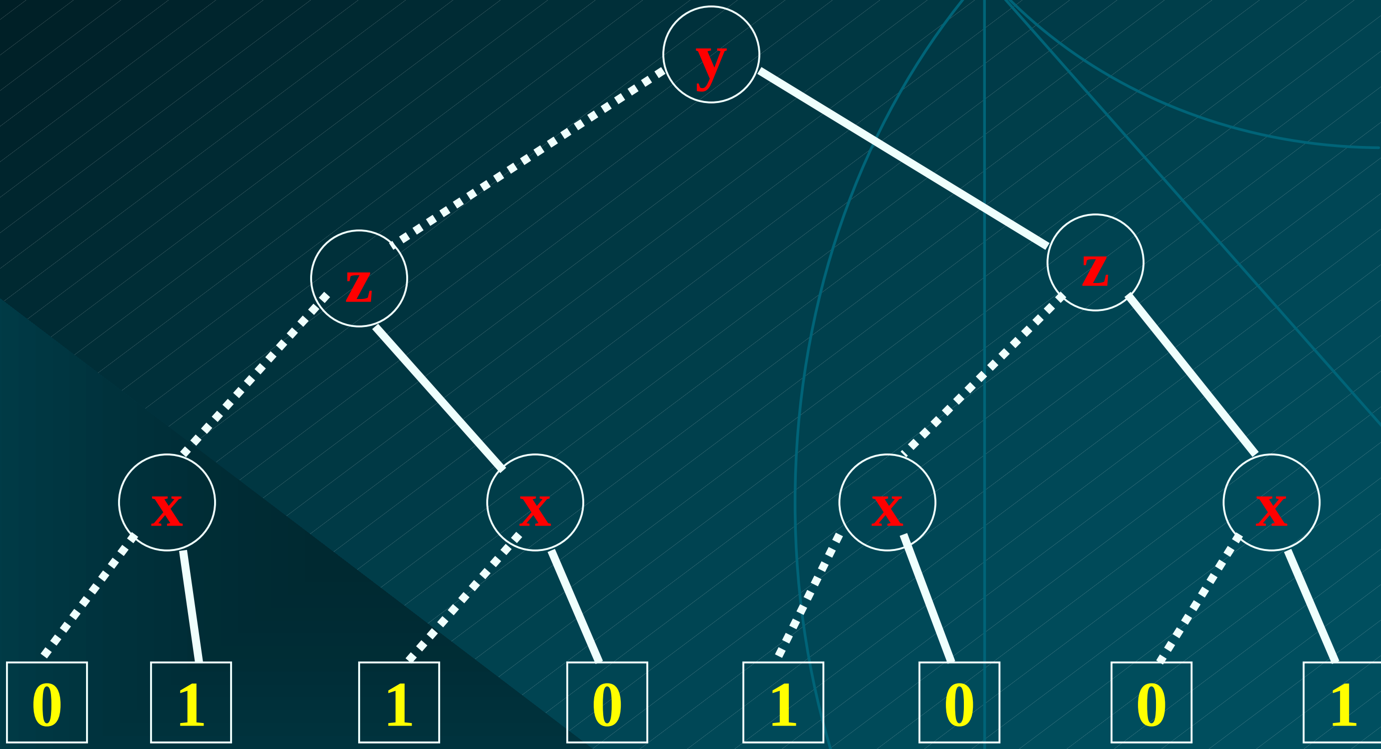
- f : Domain $ Range
- **Boolean function:**
  - Domain = $\{0, 1\}^n$ = {0,1} § ….§ {0,1}.
  - Range = {0, 1}
  - f **is a function of** n **boolean variables.**

Boolean decision trees.



x a y

x	y	z	g
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

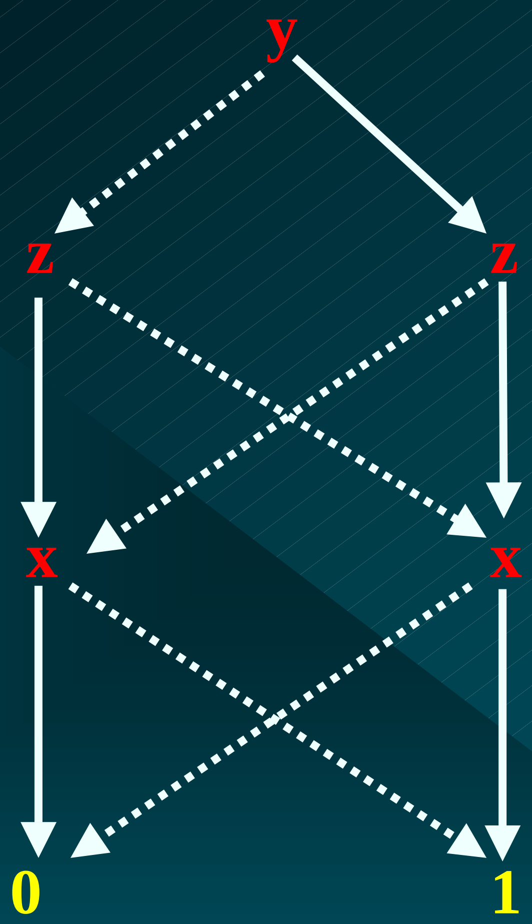


g = (y a (x ⇔ z)) b (¬y a (x ⇔ ¬z))

BDDs

A BDD is *finite rooted directed acyclic graph* in which:

- There is a *unique initial node* (the *root*)
- Each *terminal node* is labeled with a 0 or 1.
- Each *non-terminal* (internal) node *v* has 3 attributes:
  - *var(v)*, and
  - exactly *two successors low(v)* and *high(v)*: one labeled 0 (*dotted edge*, *low(v)*) and the other labeled 1 (*full edge*, *high(v)*).

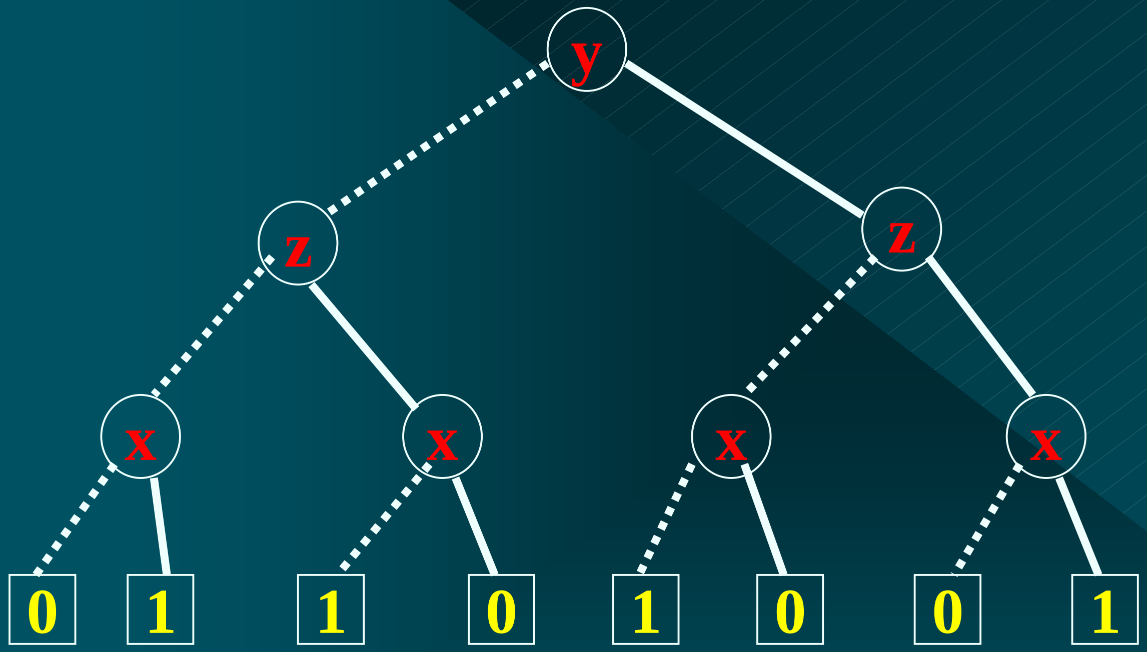


g = (y a (x ⇔ z)) b (¬y a (x ⇔ ¬z))

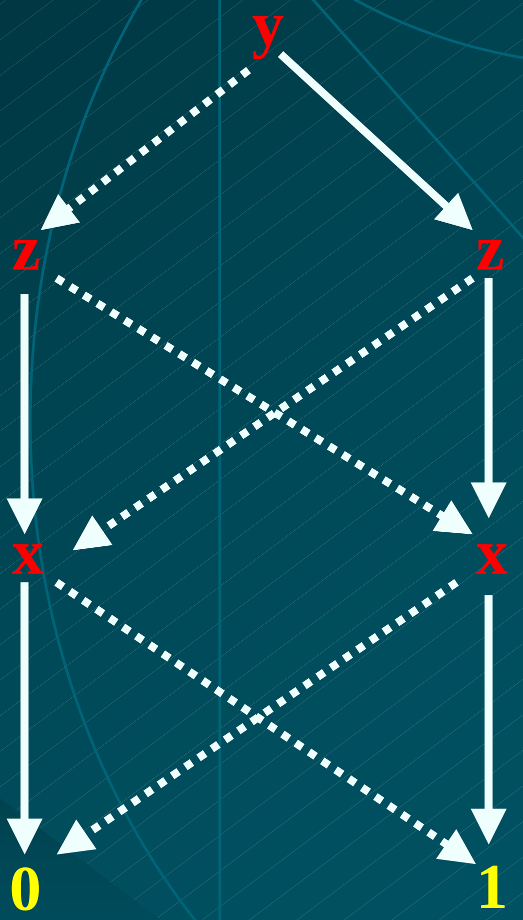
Reduced BDDs

- A BDD is *reduced iff* none of the three reduction rules can be applied to it.
- Start from the bottom layer (terminal nodes).
- *Apply* the *rules* repeatedly *to level* i. And then *move to level* i-1 (checking applicability of R3 only needs testing whether var(m)=var(n), low(m)=low(n) and high(m)=high(n)).
- Stop when the root node has been treated.
- This can be done efficiently.

Binary Decision Tree for



Reduced BDD



g = (y a **(x ⇔ z))** b **(¬y** a **(x ⇔ ¬z))**

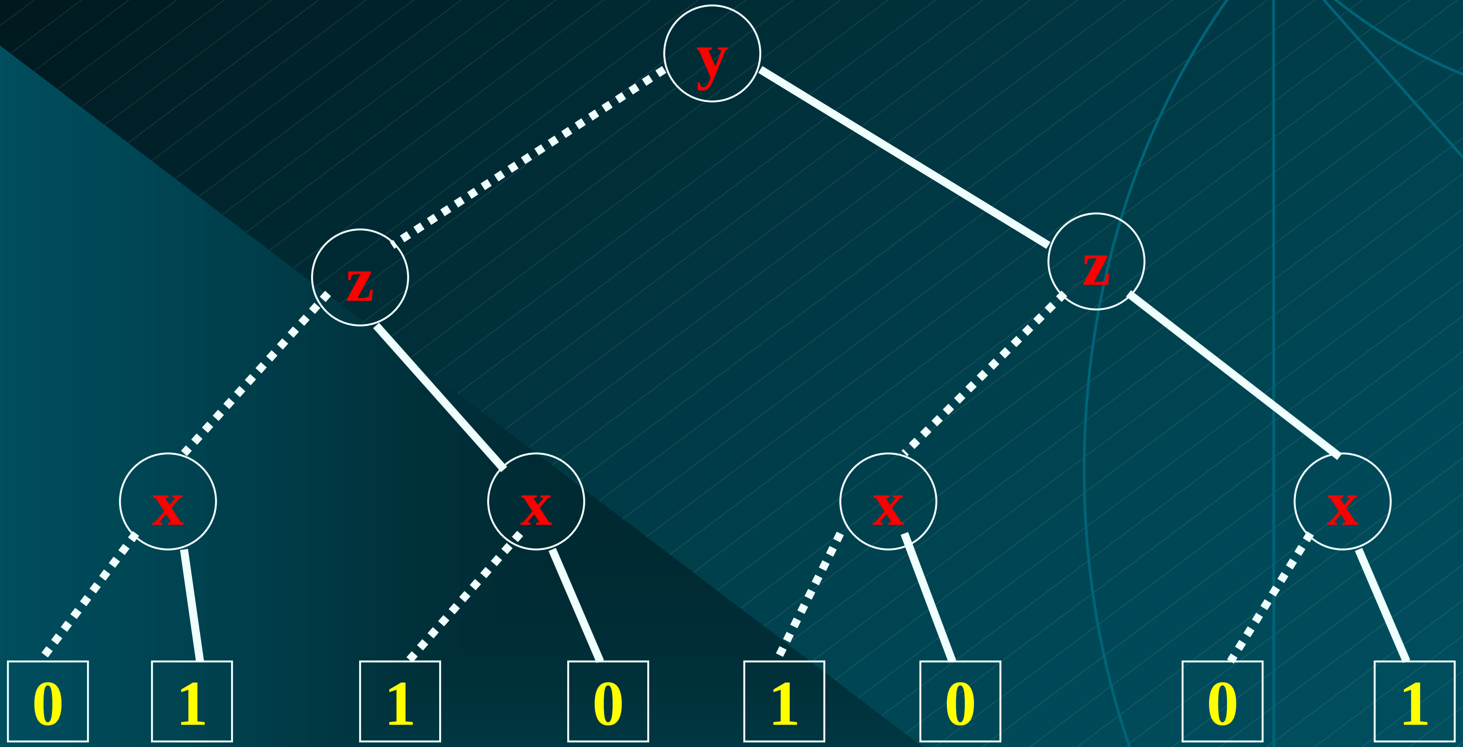
Ordered BDDs

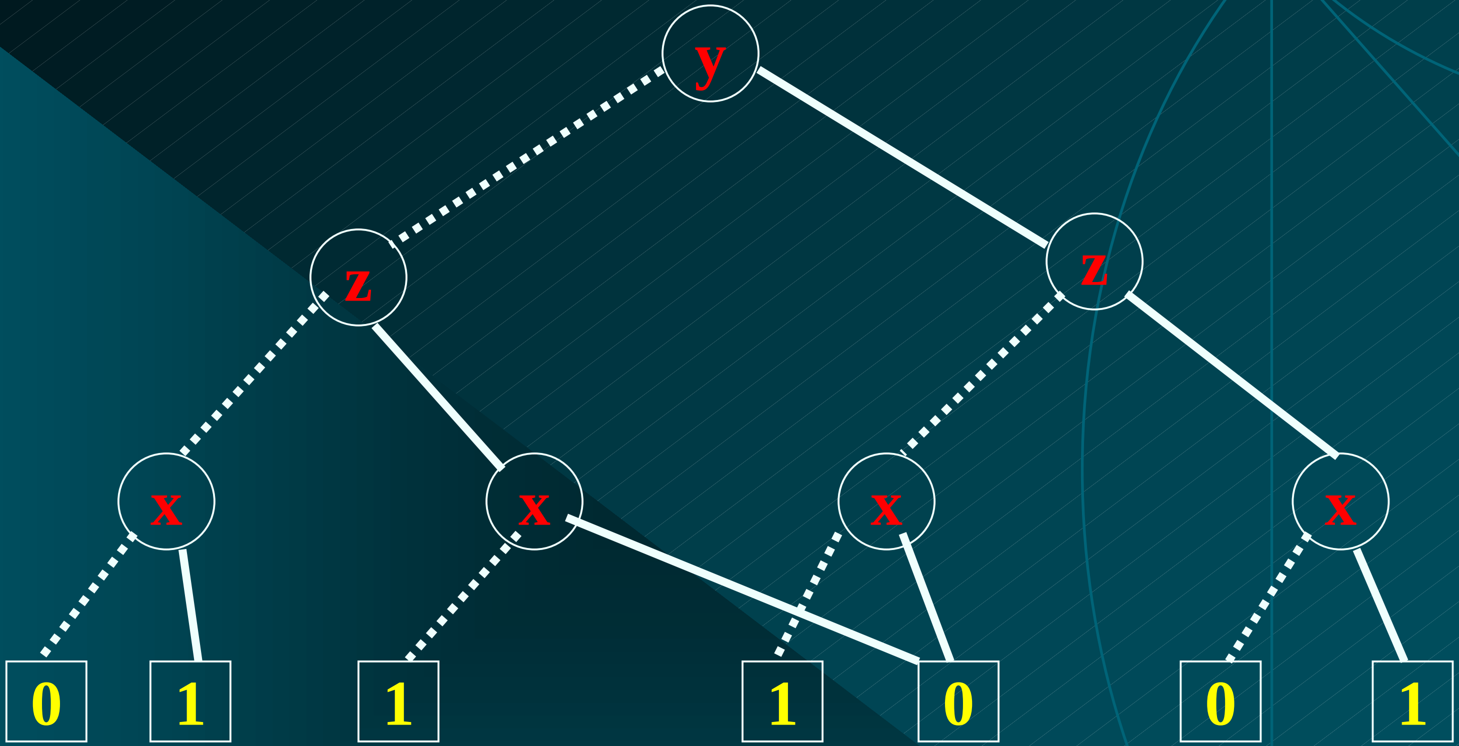
- $\{x_1, x_2, \ldots, x_n\}$
  - An indexed (ordered) set of boolean variables.
  - $x_1 < x_2 \ldots\ldots < x_n$
- G is an ordered BDD w.r.t. the above *variable ordering iff*:
  - Each variable that appears in G is in the above set. (but the converse may not be true).
  - If $i < j$ and x_i and x_j appear on a path then x_i appears before x_j.

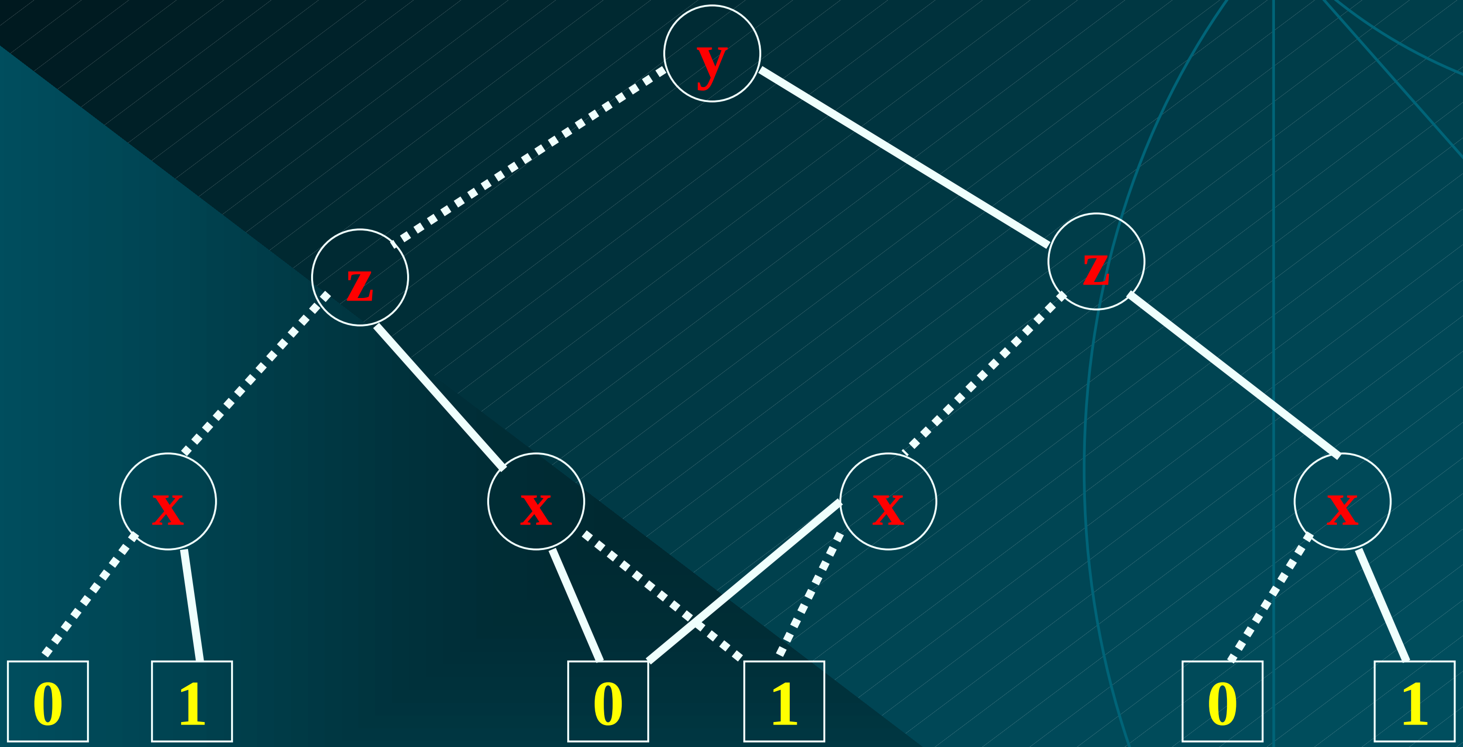
Ordered BDDS

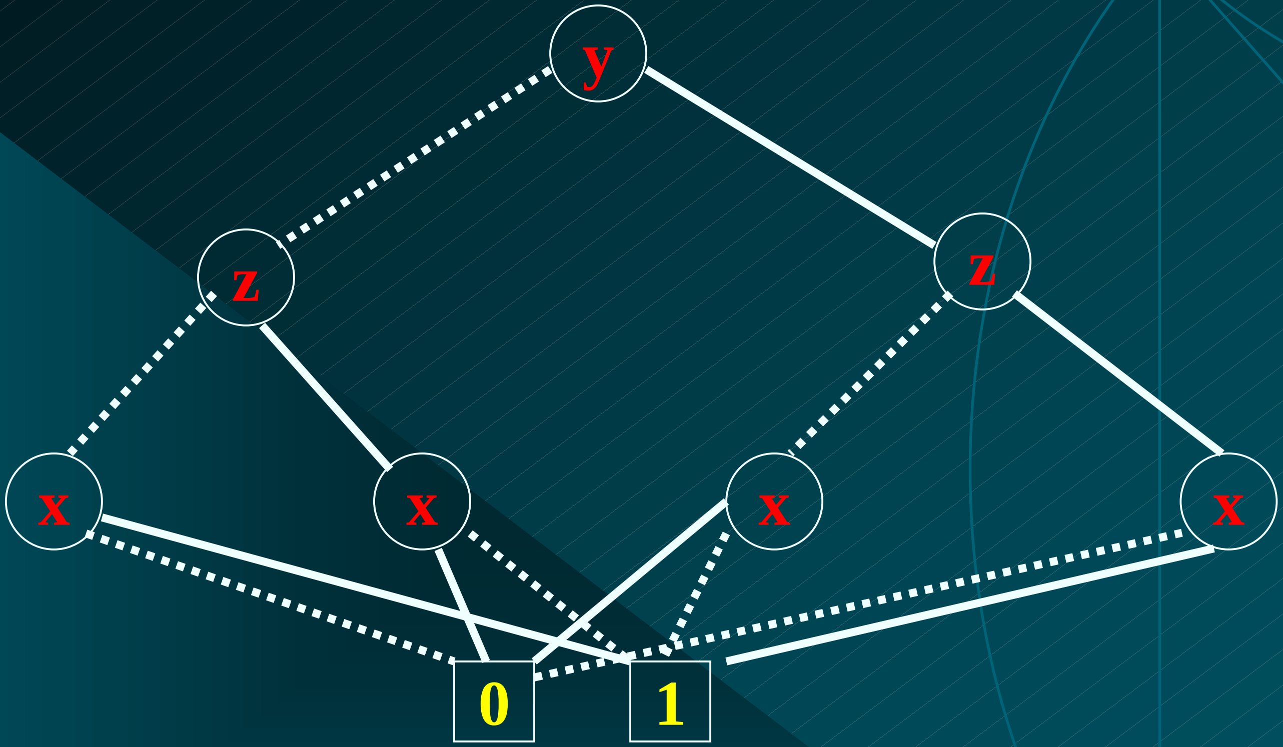
- **Fundamental Fact:**
  - **For a fixed variable ordering, each boolean function has** *exactly one* **reduced Ordered BDD!**
  - **Reduced OBDDs are** *canonical objects***.**
  - **To test if** *f* **and** *g* **are equal, we just have to check if** their **reduced** OBDD**s are** identical**.**
  - **This will be crucial for model checking!**

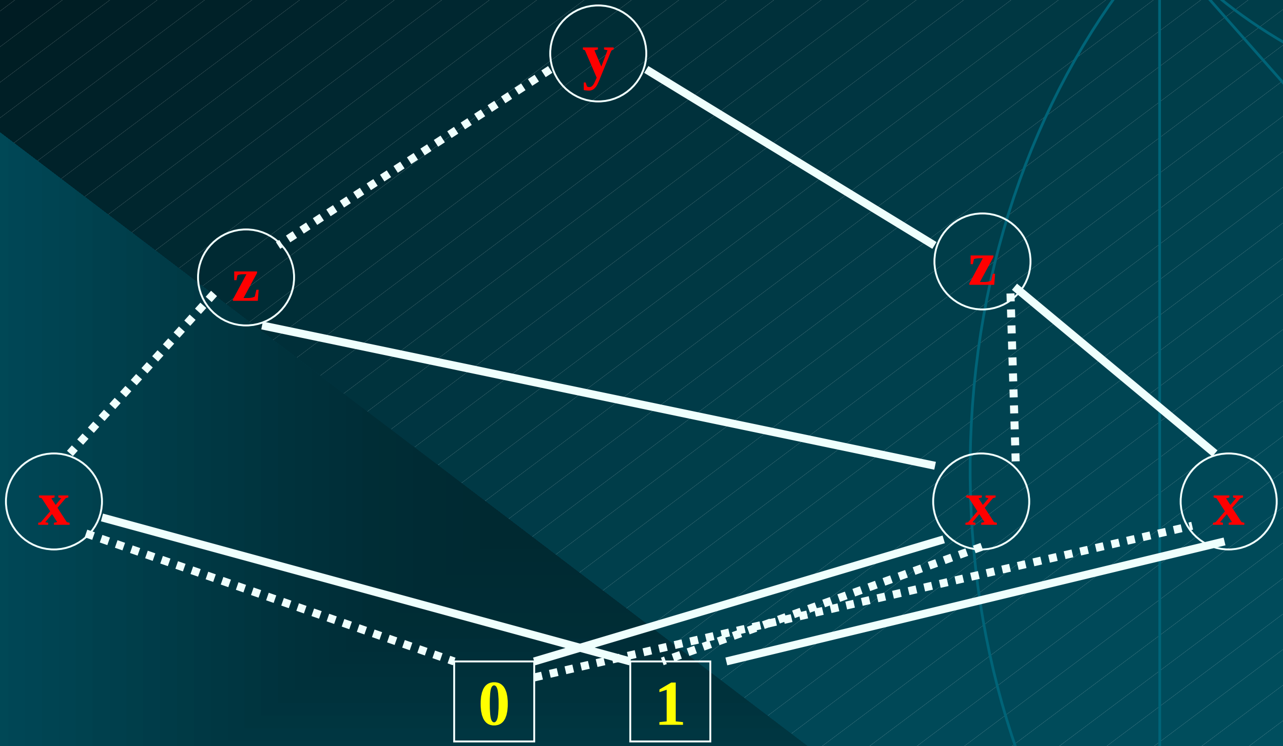
y < z < x

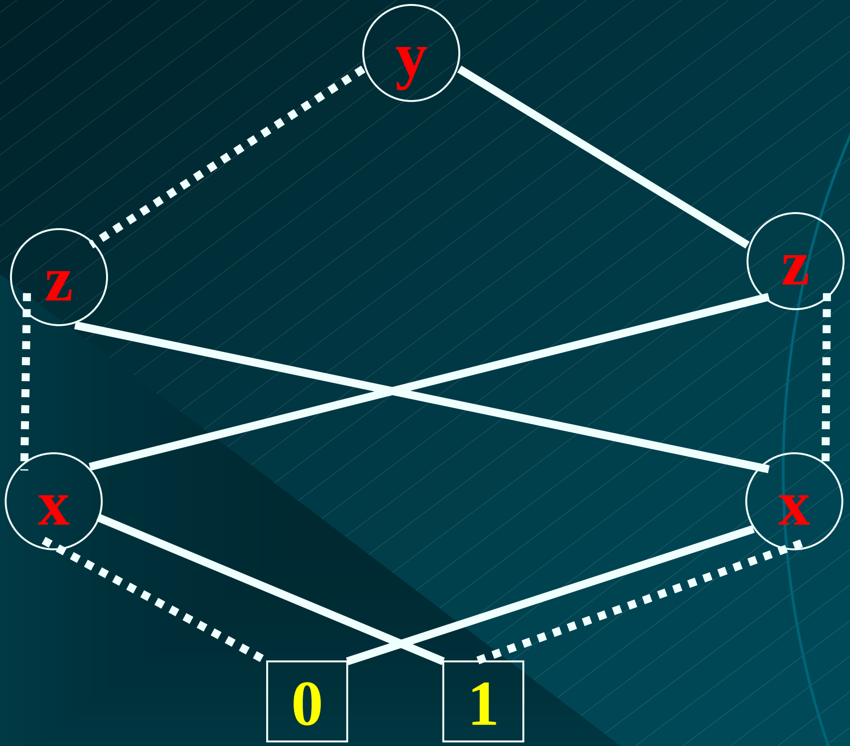



y
z
z
x
x
x
x
0
1
1
1
0
0
1


y
z
z
x
x
x
x
0
1
0
1
0
1


y
z
z
x
x
x
x
0
1


y
z
z
x
x
x
0
1


y
z
z
x
x
0
1

Canonicity of ROBDD

Let us denote an ROBDD **with its** *root node* **and the** *function* **represented by** *subgraph a rooted* **at node** *u* **with** f^u**. Then:**

Theorem: **For any function** $f:\{0,1\}^n \rightarrow \{0,1\}$ *there exists a unique* ROBDD *u* **with variable ordering** $x_1, x_2, \ldots, x_n$ **such that**

$$f^u = f(x_1, \ldots, x_n)$$

Consequences of canonicity

Theorem: **For any function** $f:\{0,1\}^n \rightarrow \{0,1\}$ **there exists a** *unique* ROBDD *u* **with variable ordering** $x_1, x_2,\ldots,x_n$ **such that**

$$f^u = f(x_1,\ldots,x_n)$$

Therefore we can say that:

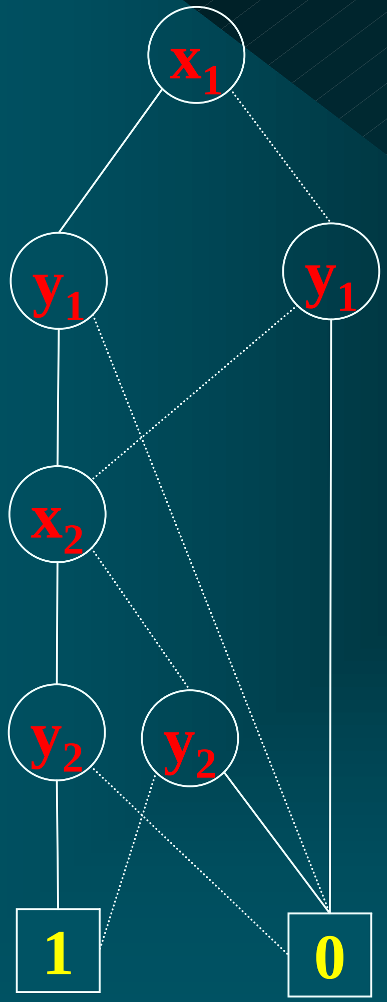
- **A function** f^u **is a** *tautology* **if its** ROBDD *u* **is** *equal* **to** 1**.**
- **A function** f^u **is a** *satisfiable* **if its** ROBDD *u* **is** *not equal* **to** 0**.**

Reduced OBDDs

- *The ordering is crucial!*
- $\{x_1, x_2, y_1, y_2\}$ x_1 x_2
  - $f(x_1, x_2, y_1, y_2)$ y_1 y_2
  - $f(x_1, x_2, y_1, y_2) = 1$ *iff* $(x_1 = y_1 \wedge x_2 = y_2)$
- **If** $x_1 < y_1 < x_2 < y_2$**, then the** OBDD **is of size** $3{\cdot}2 + 2 = 8$**.**
- **If** $x_1 < x_2 < y_1 < y_2$**, then the** OBDD **is of size** $3{\cdot}2^2 - 1 = 11$ **!**

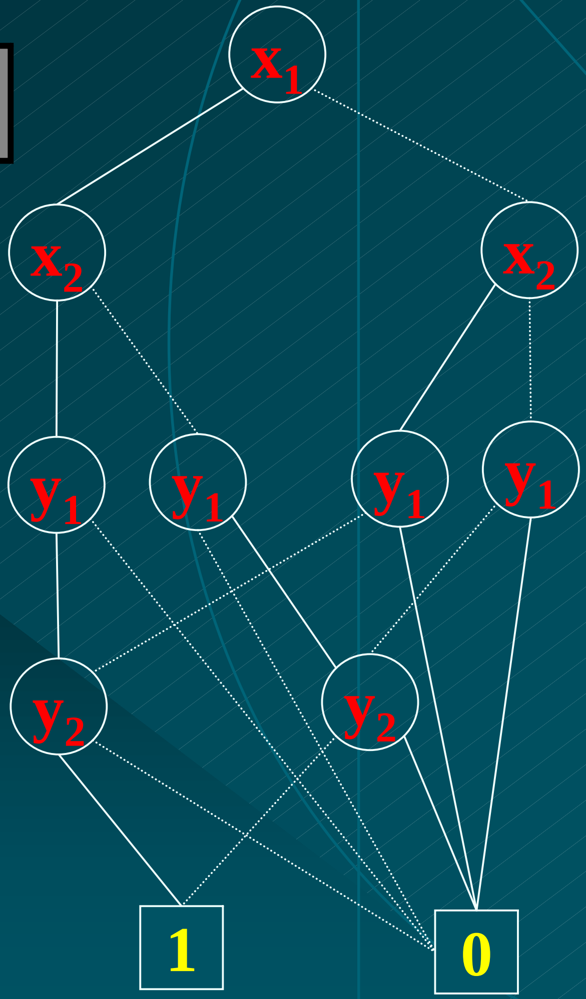
Reduced OBDDs

$x_1 < y_1 < x_2 < y_2$



$(x_1 = y_1 \wedge x_2 = y_2)$

$x_1 < x_2 < y_1 < y_2$



Reduced OBDDs

- *The ordering is crucial!*
- $\{x_1, x_2,\ldots,x_n,y_1,y_2,\ldots,y_n\}$ $\qquad$ $x_1 \;\; x_2 \ldots x_n$

  $f(x_1, x_2,\ldots,x_n,y_1,y_2,\ldots,y_n)$ $\qquad$ $y_1\, y_2 \ldots y_n$

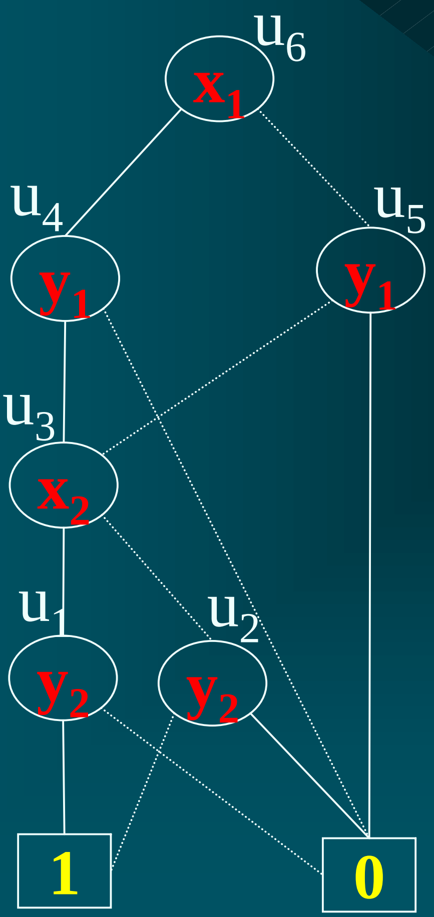
  - $f(x_1, x_2,\ldots,x_n,y_1,y_2,\ldots,y_n) = 1$ *iff* $\displaystyle \grave{O}_{i=1}^{n} (x_i = y_i)$
- **If** $x_1 < y_1 < x_2 < y_2\ldots<x_n < y_n$**, then the** OBDD **is of size** $3n + 2$**.**
- **If** $x_1 < x_2 <\ldots<x_n < y_1 <\ldots< y_n$**, then the** OBDD **is of size** $3 \,.\, 2^n - 1$ **!**

ROBDDs

- **Finding the** *optimal variable ordering* **is** *computationally expensive* **(**NP-complete**).**
- **There are** *heuristics* **for finding "***good orderings***".**
- **There exist boolean functions whose sizes are** *exponential* **(in the number of variables) for any ordering.**
- **Functions encountered in practice are** rarely **of this kind.**

Implementation of ROBDDs

Array-based implementation



root = u_6

***T*[]=**

	Var	Low	High
0	**?**	**?**	**?**
1	**?**	**?**	**?**
u_1	y_2	**0**	**1**
u_2	y_2	**1**	**0**
u_3	x_2	u_2	u_1
u_4	y_2	**0**	u_3
u_5	y_1	**0**	u_3
u_6	x_1	u_5	u_4

The function MK

- The function MK searches for a node *u* with *$var(u)=x_i$*, *low(u)=l* and *high(u)=h*. If the node does not exists, then creates the new node after inserting it. The running time is *O(1)*.

H(i,l,h) is a hash function mapping a triple *<i,l,h>* into a node index in T.

```
mk(i,l,h)
  if  l=h then
       return l
  else if T[H(i,l,h)] $\neq$ empty then
       return T[H(i,l,h)]
  else u = add(T,H(i,l,h),i,l,h)
        return u
```

Operations on ROBDDs.

- **Boolean operations will have to be performed on** ROBDD**s.**
- **These operations can be implemented efficiently.**
- $f \mathtt{b} g$ ---------- $G_f \; op_{\mathtt{b}} \; G_g = G_{f \mathtt{b} g}$
- **There is a procedure called** APPLY **to do this.**

Operations on ROBDDs

- When performing an operation on G and G' we assume their variable orderings are *compatible*.
- $X = X_G \hat{} \; X_{G'}$
- There is an ordering $<$ on X such that:
  - $<$ restricted to X_G is $<_G$
  - $<$ restricted to $X_{G'}$ is $<_{G'}$.

Operations on OBDDs

- The basic idea (**Shannon Expansion**):
- $f(x_1, x_2, \ldots, x_n)$
  - $f\mathrm{m}_{x_1 = 0} = f(0, x_2,\ldots,x_n)$
    - $f = x_1 \;\mathrm{b}\; (x_2 \;\mathrm{a}\; x_3)$
    - $f\mathrm{m}_{x_1 = 0} = x_2 \;\mathrm{a}\; x_3$
  - **Similarly,** $f\ \mathrm{m}_{x1 = 1} = f(1, x_2,\ldots,x_n)$

$$f(x_1, x_2,\ldots,x_n) = (\neg x_1 \;\mathrm{a}\; f\mathrm{m}_{x_1 = 0}) \;\mathrm{b}\; (x_1 \;\mathrm{a}\; f\mathrm{m}_{x_1 =1})$$

- **This is true even if** x_1 **does not appear in** f **!**

Operations on OBDDs: Negation

- The basic idea (**Shannon Expansion**):

$$f(x_1, x_2,\ldots,x_n) = (\neg x_1 \text{ a } f\text{m}_{x_1 = 0}) \text{ b } (x_1 \text{ a } f\text{m}_{x_1 =1})$$

- Therefore, assuming $x_1 < x_2 < \ldots < x_n,$

$$\neg f(x_1, x_2,\ldots,x_n) = \neg ((\neg x_1 \text{ a } f\text{m}_{x_1 = 0}) \text{ b } (x_1 \text{ a } f\text{m}_{x_1 =1}))$$

$$= (\neg(\neg x_1 \text{ a } f\text{m}_{x_1 = 0}) \text{ a } \neg(x_1 \text{ a } f\text{m}_{x_1 =1}))$$

$$= ((x_1 \text{ b } \neg f\text{m}_{x_1 = 0}) \text{ a } (\neg x_1 \text{ b } \neg f\text{m}_{x_1 =1})$$

$$= (x_1 \text{ a } \neg x_1) \text{ b } (\neg x_1 \text{ a } \neg f\text{m}_{x_1 = 0}) \text{ b}$$

$$\text{b } (x_1 \text{ a } \neg f\text{m}_{x_1 = 1}) \text{ b } (\neg f\text{m}_{x_1 = 0} \text{ a } \neg f\text{m}_{x_1 = 1})$$

$$= (\neg x_1 \text{ a } \neg f\text{m}_{x_1 = 0}) \text{ b } (x_1 \text{ a } \neg f\text{m}_{x_1 = 1})$$

Operations on ROBDDs.

- Let x be the **top variable** of G_f and y the **top variable** of G_g.
- To compute $G_{f \text{ op } g}$ we consider:
  - CASE1: $x = y$
    - $f \text{ op } g = (\neg x \text{ a } (f \text{ m}_{x=0} \text{ op } g \text{ m}_{x=0}) \text{ b }$
      $(x \text{ a } (f \text{ m}_{x=1} \text{ op } g \text{ m}_{x=1})$
  - We have to solve now two smaller problems!

Operations on ROBDDs.

- Let x be the **top variable** of G_f and y the **top variable** of G_g.
- To compute $G_{f\ op\ g}$ we consider:

  CASE2: $x < y$.

  - Then x does not appear in G_g (why?).
  - $g\ m_{x=0} = g = g\ m_{x=1}$
    - ✧ $f\ op\ g = (\neg x\ a\ (f\ m_{x=0}\ op\ g\ )\ b\ (x\ a\ (f\ m_{x=1}\ op\ g)$
  - We have to solve now two smaller problems!

  CASE2: $x > y$ is symmetric.

Operations on ROBDDs.

- To compute $G_{f \text{ op } g}$ we consider:

  Base (terminal) cases depend upon op

  Eg.: if op = b then {0,0 → 0; 1}

  if op = a then {1,1 → 1; 0}

  ....

Build BDDs: The Apply Procedure

- Given:
  - two BDDs one for f and one for g
  - the logical operator op
- To build
  - r = f op g

    (and of two BDDs, or of two BDDs etc.) call:
- Do the following:
  - Init computed table CT
  - r = APPLY (f, g)

with:

Algorithm for Apply

```
Algorithm Apply(op,u,v)

  Function App(u,v)
     if terminal_case(op,u,v) then return op(u,v)
     else if var(u) = var(v) then
         u = mk(var(u), App(op,low(u),low(v)),
                                    App(op,high(u),high(v)))
     else if var(u) < var(v) then
         u = mk(var(u),App(op,low(u), v), App(op,high(u),v))
     else /* var(u) > var(v) */
         u = mk(var(u),App(op,u,low(v)), App(op,u,high(v)))
     return u

return App(u,v)
```

running time = **O(2^n)**. Why?
n = number of variables.

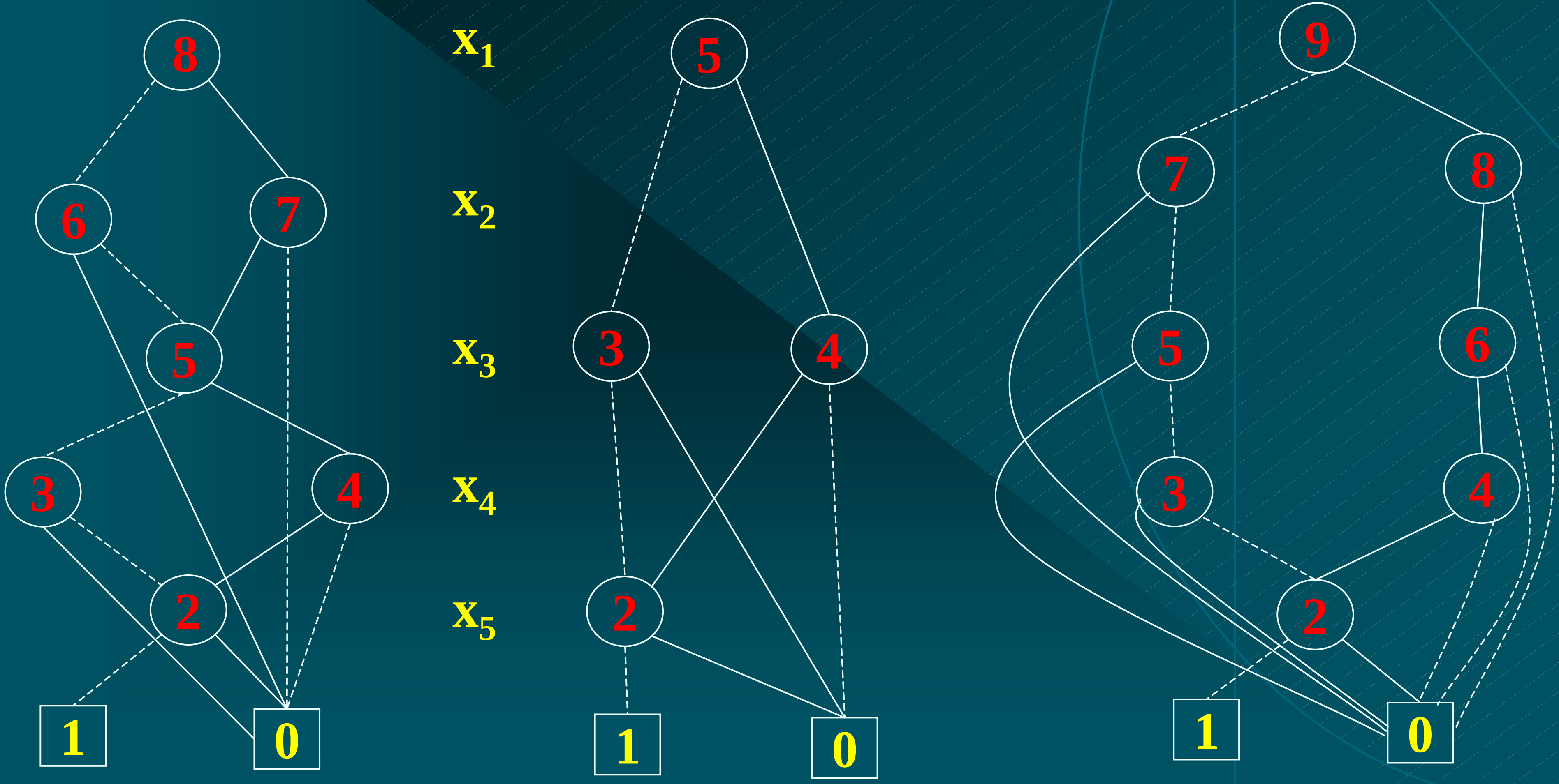
Efficient algorithm for Apply

```
Algorithm Apply(op,u,v)
  init(G)
   Function App(u,v)
      if  G(u,v) $\neq$ empty then return G(u,v)
      else if terminal_case(op,u,v) then return op(u,v)
      else if var(u)=var(v) then
          r = mk(var(u), App(op,low(u),low(v)),
                                  App(op,high(u),high(v)))
      else if var(u) < var(v) then
          r = mk(var(u),App(op,low(u), v), App(op,high(u),v))
      else /* var(u) > var(v) */
          r = mk(var(u),App(op,u,low(v)), App(op,u,high(v)))
      G(u,v) = r
      return r
return App(u,v)
```

running time = $O(|G_u||G_v|)$. Why?

Example of Apply

$$(x_1 \equiv x_2) \wedge (x_3 \equiv x_4) \wedge \neg x_5 \quad \wedge \quad (x_1 \equiv x_3) \wedge \neg x_5 \quad = \quad ((x_1 \wedge x_2 \wedge x_3 \wedge x_4) \vee \vee (\neg x_1 \wedge \neg x_2 \wedge \neg x_3 \wedge \neg x_4)) \wedge \neg x_5$$



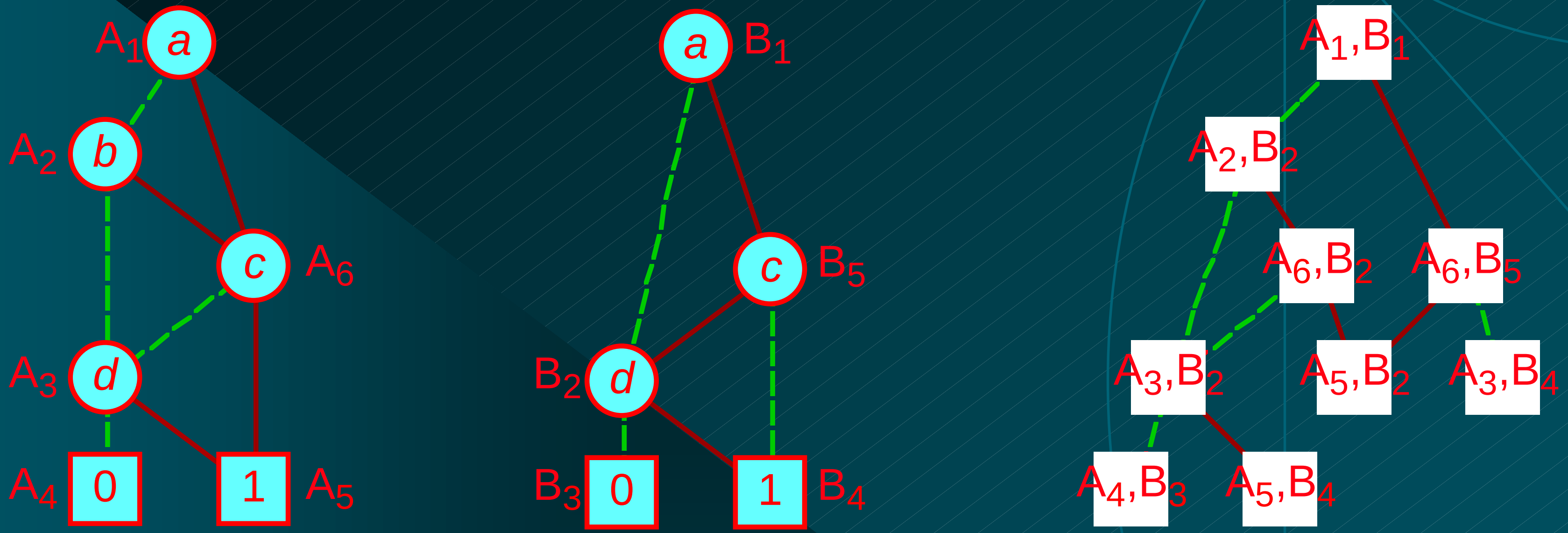
APPLY (f, g)

```
1. IF CT(f, g) $\neq$ empty THEN return (CT (f, g))
2. ELSE if f and g $\in$ { 0, 1} THEN r = op (f, g)
3. ELSE if topVar(f) = topVar(g) THEN
     ◆ r = ITE (topVar (f), APPLY (T(f), T(g)), APPLY (E(f), E(g)))
4. ELSE if topVar(f) < topVar(g) THEN
     ◆ r = ITE (topVar (f), APPLY (T(f), g), APPLY (E(f), g))
5. ELSE /* topVar(f) > topVar(g) */
     ◆ r = ITE (topVar (g), APPLY (f, T(g)), APPLY (f, E(g)))
6. put r in G
7. return (r)
```

Execution Example

Argument *A* ∧ Argument *B*

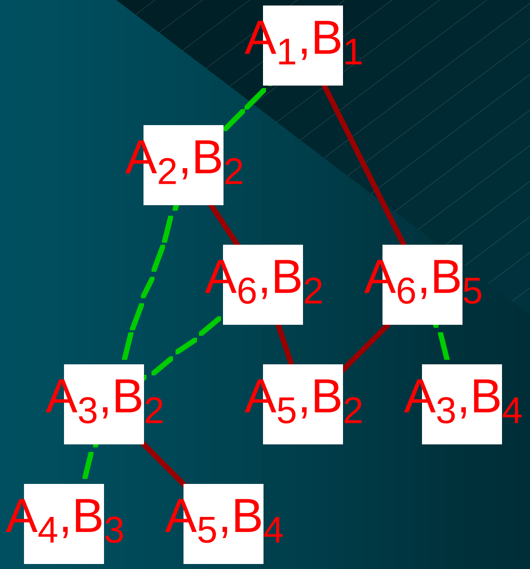
Recursive Calls



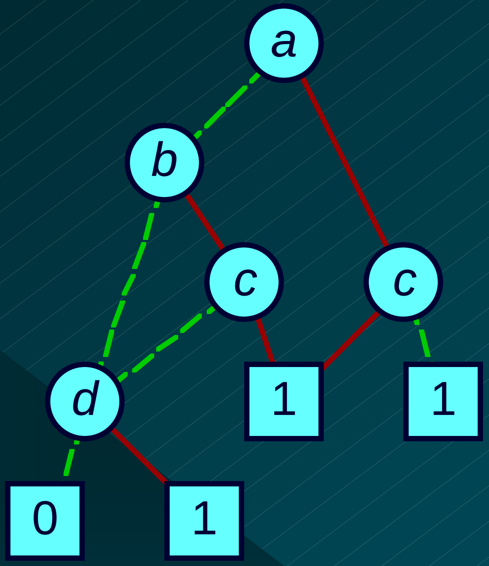
- **Optimizations**
  - **Dynamic programming**
  - **Early termination rules**

Result Generation

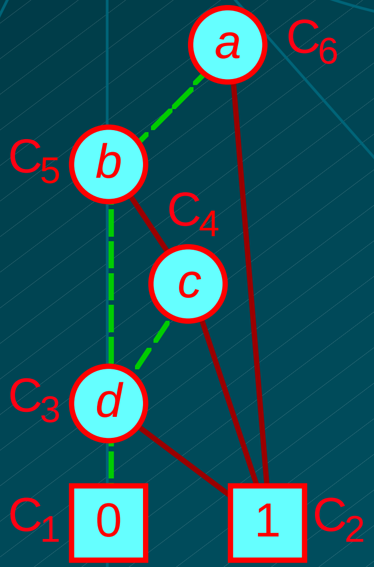
Recursive Calls



Without Reduction

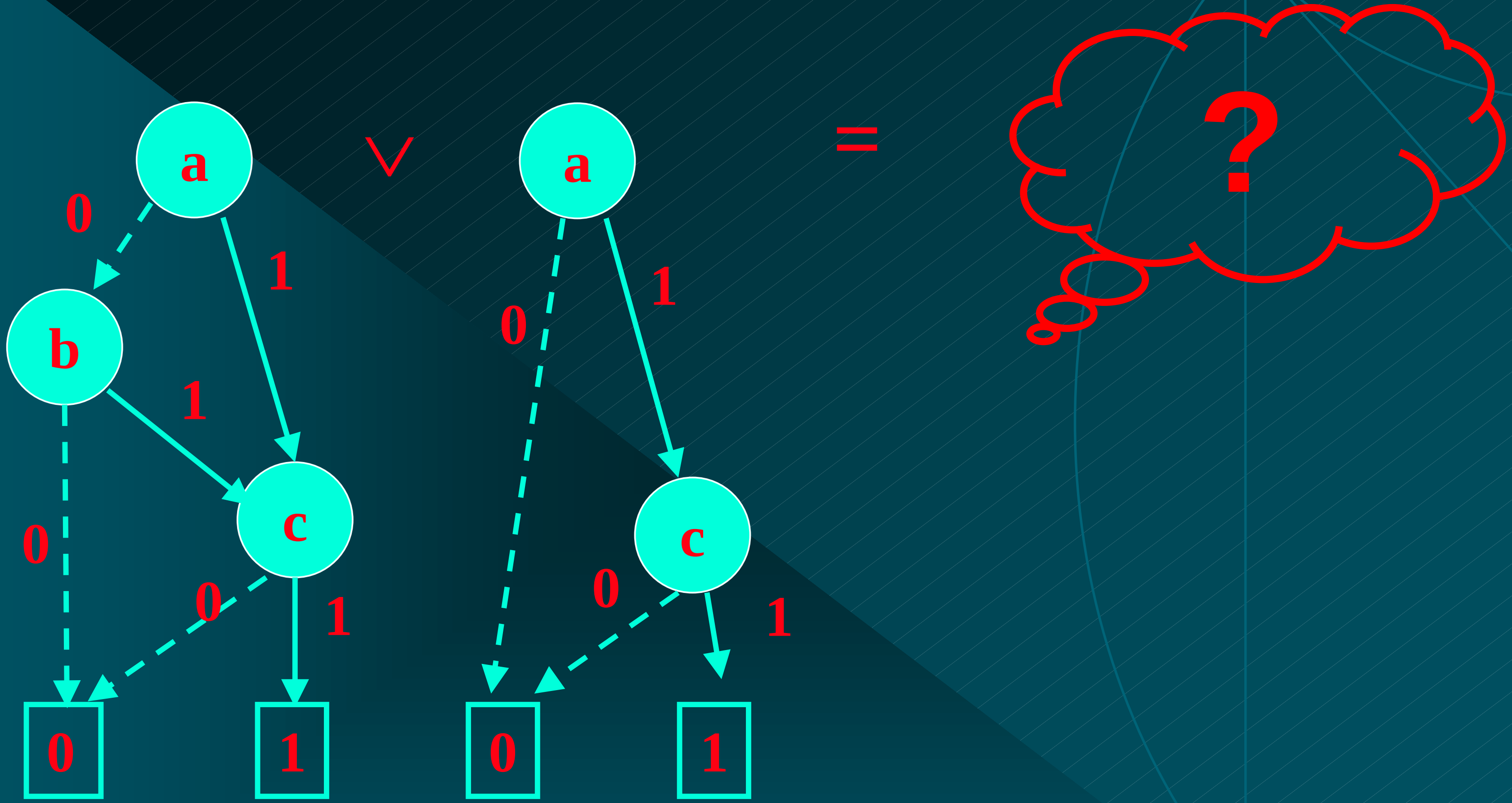


With Reduction

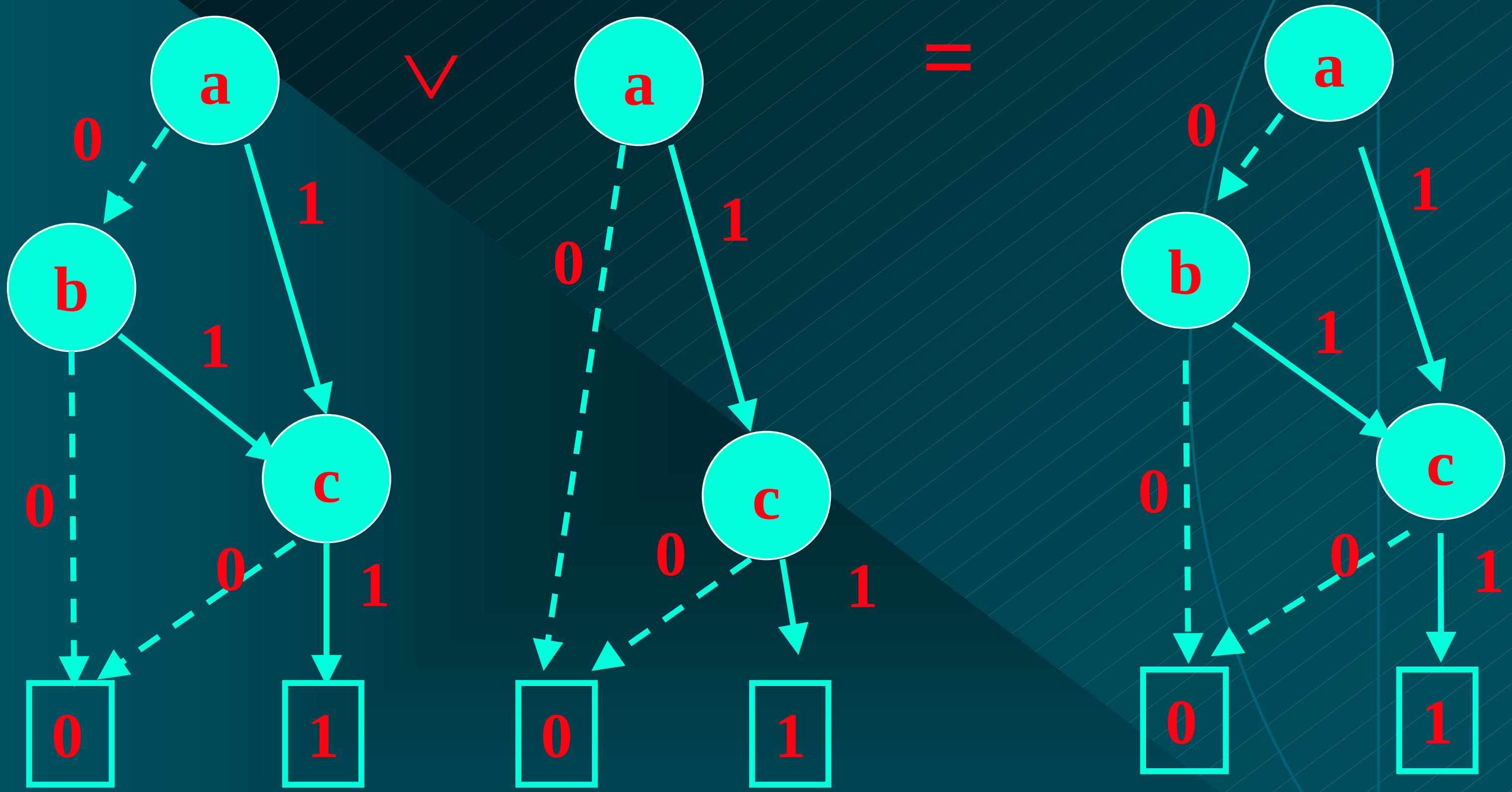


- Recursive calling structure implicitly defines unreduced BDD
- Apply reduction rules bottom-up as return from recursive calls
- Do not create new result node if both brances equal (return that result) or if equivalent node already exists in reduce table. (The apply function is also memoized.)

Example



Example



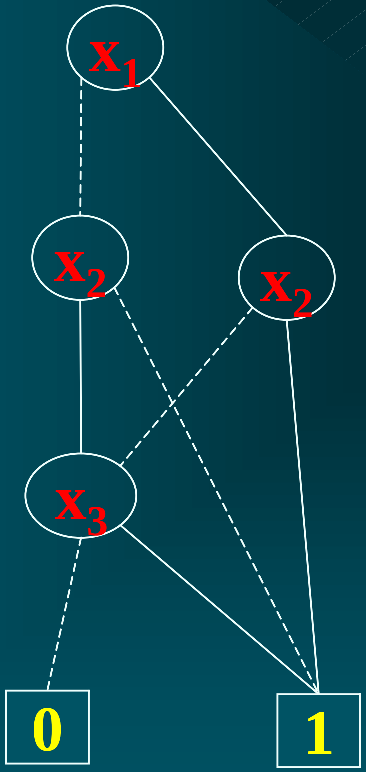
The Restrict operation

- *Problem*: Given a (partial) truth assignment $x_1=b_1,\ldots,x_k=b_k$ (where $b_j=0$ or $b_j=1$), and a ROBDD t^u, compute the restriction of t^u under the assignment.
- E.G.: if $f(x_1,x_2,x_3) = ((x_1 \Leftrightarrow x_2) \vee x_3)$ we want to compute $f(x_1,x_2,x_3)[0/x_2] = f(x_1,0,x_3)$

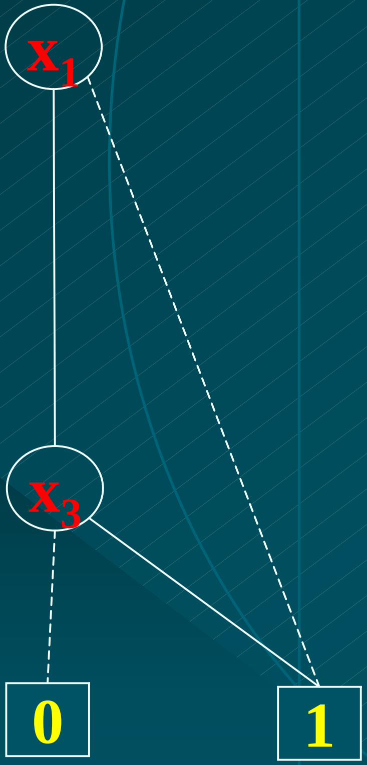
  i.e.: $f(x_1,0,x_3)= \neg x_1 \vee x_3$

Restrict Operation: example

$f(x_1,x_2,x_3) = ((x_1 \Leftrightarrow x_2) \vee x_3)$



$f(x_1,x_2,x_3)[0/x_2] = \neg x_1 \vee x_3$



Restrict Operation

- Let x be the root of G_f
- To compute $G_f|_{y=b}$ we consider:

  CASE1: x = y

  - $f|_{y=b} = \mathrm{low}(G_f)$ if b=0
  - $f|_{y=b} = \mathrm{high}(G_f)$ if b=1

Restrict Operation

- Let x be the root of G_f
- To compute $G_f|_{y=b}$ we consider:
  - CASE2: $x > y$
    - $f|_{y=b} = f$

Restrict Operation

- Let x be the root of G_f
- To compute $G_f|_{y=b}$ we consider:

  CASE2: $x < y$

  - $f|_{y=b} = (\neg x \text{ a } (f|_{x=0})|_{y=b}) \vee (x \text{ a } (f|_{x=1})|_{y=b})$

- We have to solve now two smaller problems!

Algorithm for Restrict

```
Algorithm Restrict(u,i,b)
  Function Res(u)
     if var(u) > i then return u
     else if var(u) < i then
          return mk(var(u),Res(low(u)),Res(high(u)))
     else /* var(u) = i */
          if b = 0 then
             return Res(low(u))
          else /* var(u) = i and b = 1 */
             return Res(high(u))
return Res(u)
```

running time = $\mathbf{O(2^n)}$. Why?

Efficient algorithm for Restrict

```
Algorithm Restrict(u,i,b)
    init(G)
  Function Res(u)
      if  G(u) ≠ empty then return G(u)
      if var(u) > i then return u
      else if var(u) < i then
          r = mk(var(u),Res(low(u)),Res(high(u)))
      else /* var(u) = var(v) */
          if b = 0 then
              r = Res(low(u))
          else /* var(u) = var(v) and b = 1 */
              r = Res(high(u))
      G(u) = r
      return r
return Res(u)
```

running time = **$O(|G_u|)$**. Why?

Quantification

- Extend the boolean language with

∃x.t | ∀x.t

- They can be defined in terms of ROBDD operations:

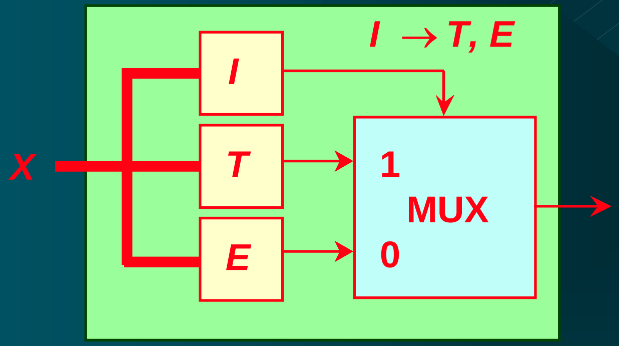
∃x.t = t[0/x] b t[1/x]

∀x.t = t[0/x] a t[1/x]

We can use an appropriate combination of *Restrict* and *Apply*

If-Then-Else Decomposition

- All operators can be expressed in terms of ITE
- Used to build BDD from logic network or formula



Arguments ***I*, *T*, *E***

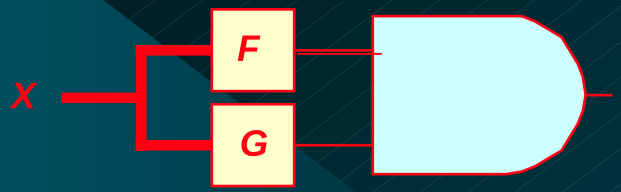
- Functions over variables *X*
- Represented as BDDs

Result

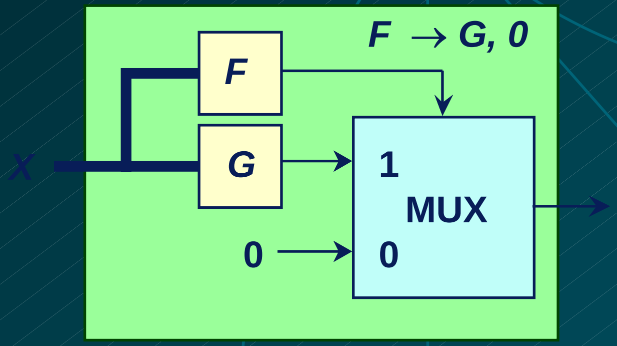
- ITE (I, T, E) = $(I \wedge T) \vee (\neg I \wedge E)$
- Represented as a BDD

❖ All operators can be expressed using ITE

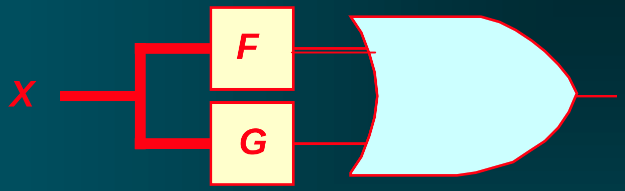
And(*F*, *G*)



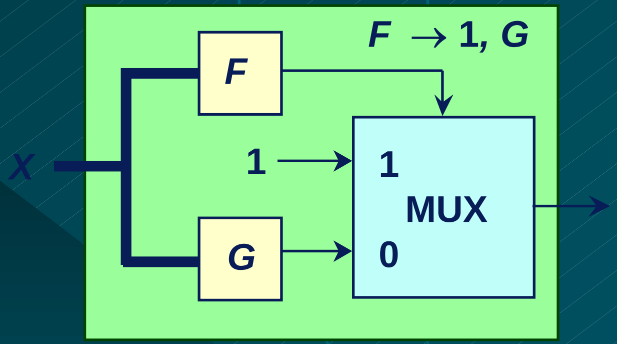
If-Then-Else(*F*, *G*, 0)



Or(*F*, *G*)



If-Then-Else(*F*, 1, *G*)



- ¬ x ➔ ITE (x, 0, 1)
- x == y ➔ ITE (x, ITE (y, 1, 0), ITE (y, 0, 1))
- ...

- **Boole's (Shannon) Decomposition**
  - $F \rightarrow \text{ITE}(x, F|_{x}, F|_{\neg x})$
  - $F = (x \wedge F|_{x}) \vee (\neg x \wedge F|_{\neg x}) = x \cdot F|_{x=1} + \neg x \cdot F|_{x=0}$
- **BDD from Boole's Decomposition**
  1. Form decomposition one variable at a time
  2. Proceed until terminal (0-1) values

This gives an "Ordered Decision Tree"

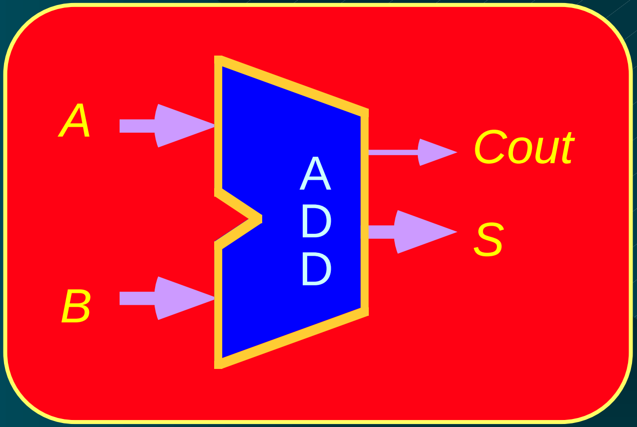
To sum up ...

- A BDD (ROBDD)
  - Is a directed acyclic graph (DAG)
    - one root node, two terminals 0, 1
    - each node, two children, and a variable
  - It uses a Shannon co-factoring tree, except that it is
    - Reduced
    - Ordered
  - Reduced
    - any node with two identical children is removed
    - two nodes with isomorphic BDD's are merged
  - Ordered
    - Co-factoring variables (splitting variables) always follow the same order along all paths

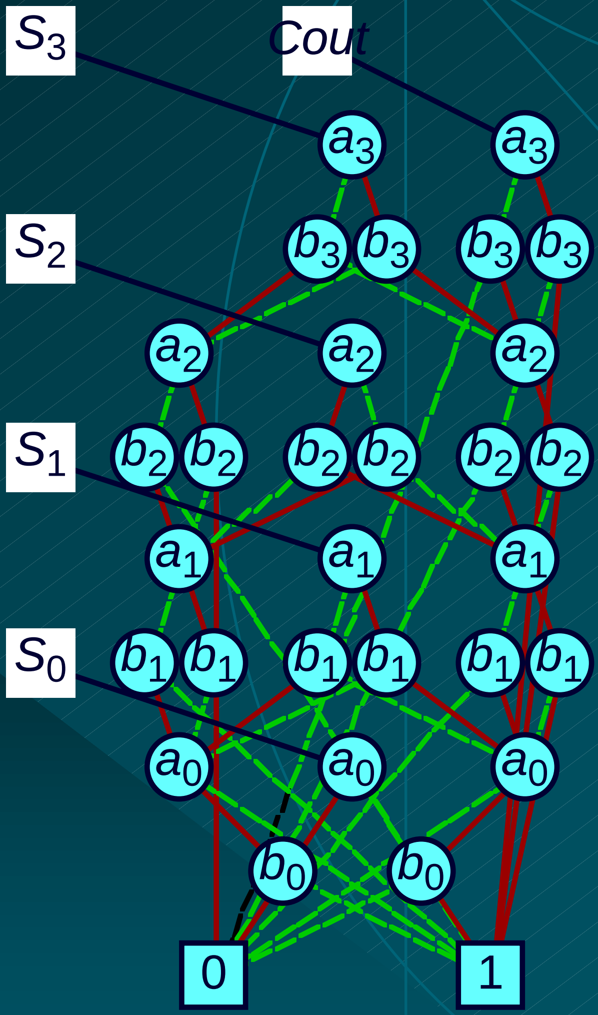
$$x_{i_1} < x_{i_2} < x_{i_3} < \ldots < x_{i_n}$$

Representing Circuit Functions

- **Functions**
  - **All outputs of 4-bit adder**
  - **Functions of data inputs**



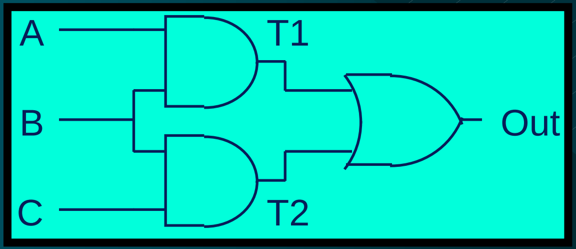
- **Shared Representation**
  - **Graph with multiple roots**
  - **31 nodes for 4-bit adder**
  - **571 nodes for 64-bit adder**
  - ***Linear growth***



Generating OBDD from Network

Task: Represent output functions of gate network as OBDDs.

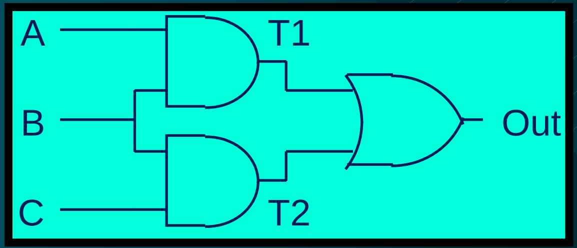
Network



Generating OBDD from Network

Task: Represent output functions of gate network as OBDDs.

Network



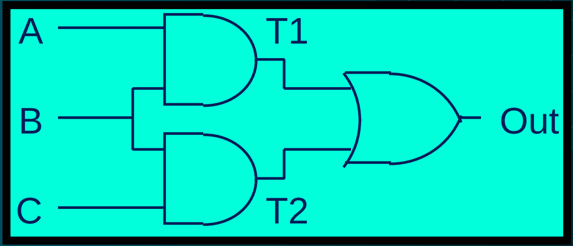
Evaluation

```
A    ←  new_var ("a");
B    ←  new_var ("b");
C    ←  new_var ("c");
T1   ←  And (A, B);
T2   ←  And (B, C);
Out  ←  Or (T1, T2);
```

Generating OBDD from Network

Task: Represent output functions of gate network as OBDDs.

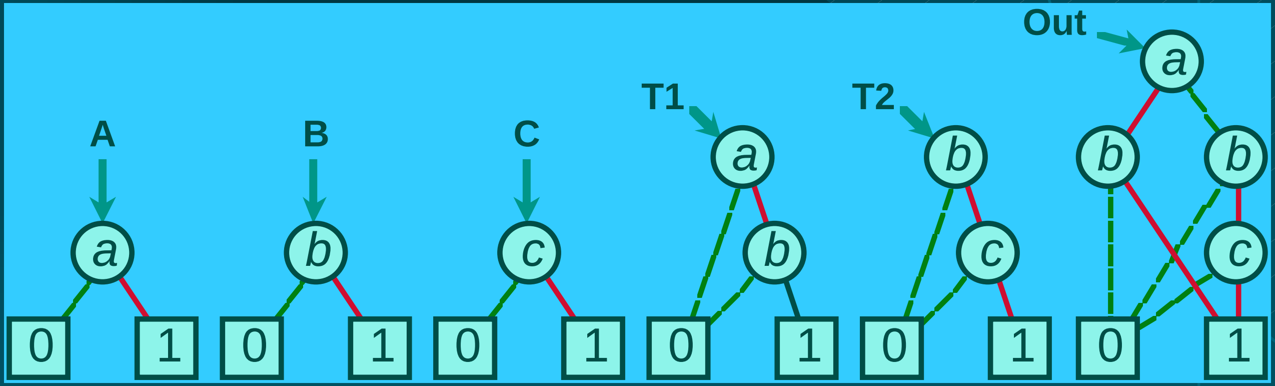
Network

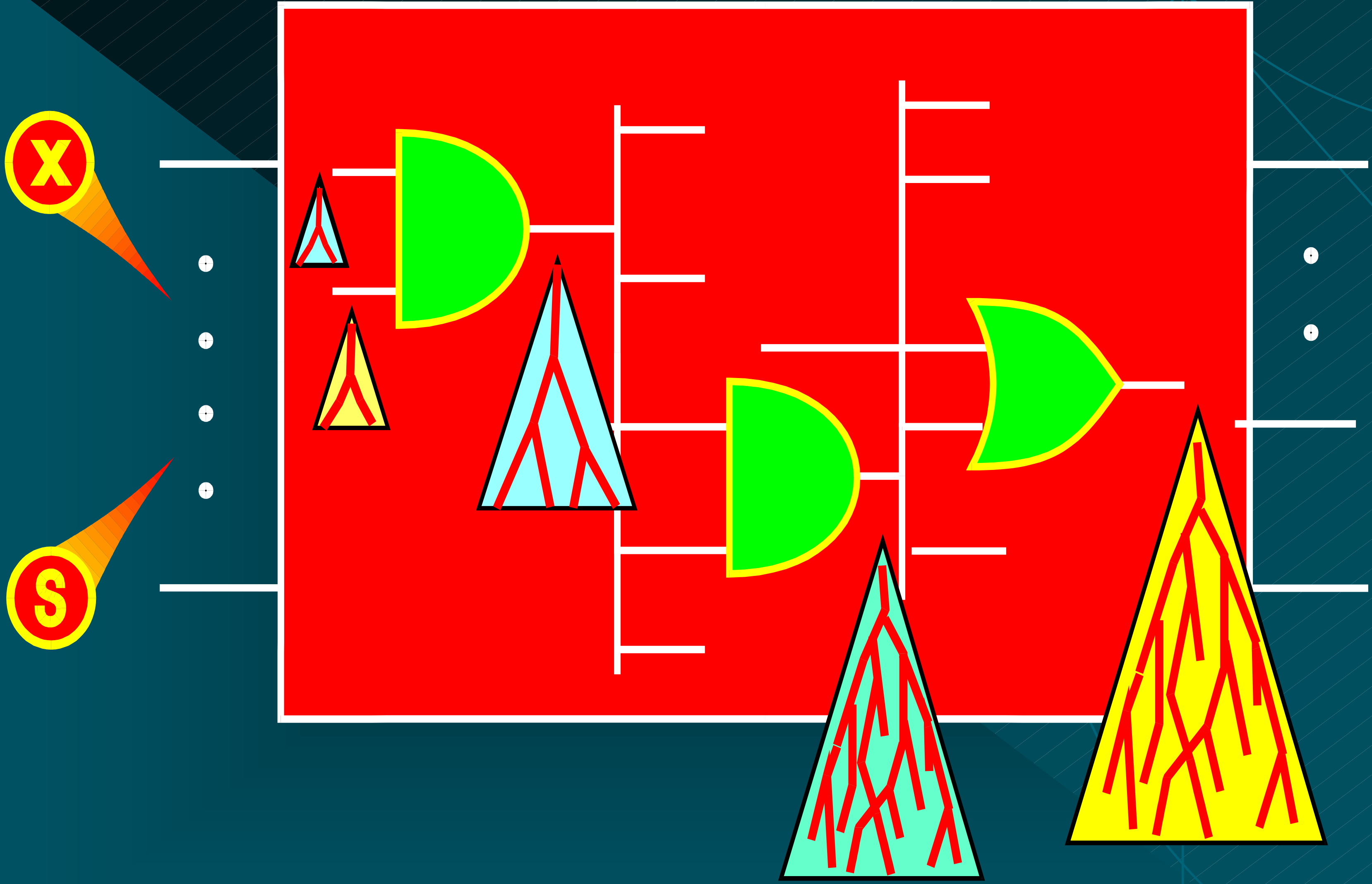


Evaluation

```
A    ←  new_var ("a");
B    ←  new_var ("b");
C    ←  new_var ("c");
T1   ←  And (A, B);
T2   ←  And (B, C);
Out  ←  Or (T1, T2);
```

Resulting Graphs




x
S

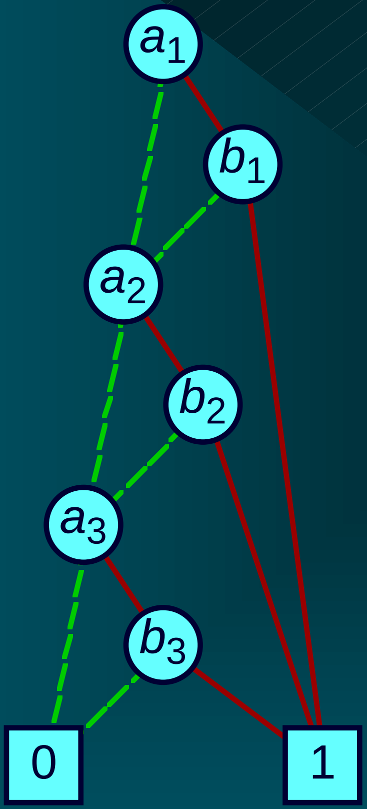
- Strategy
  - Represent data as set of OBDDs
    - Identical variable orderings
  - Express solution method as sequence of symbolic operations
    - Sequence of constructor & query operations
    - Similar style to on-line algorithm
  - Implement each operation by OBDD manipulation
    - Do all the work in the constructor operations
- Key Algorithmic Properties
  - Arguments are OBDDs with identical variable orderings
  - Result is OBDD with same ordering
  - Each step polynomial complexity

Effect of Variable Ordering

$F(a_1, a_2, a_3, b_1, b_2, b_3) = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$

Effect of Variable Ordering

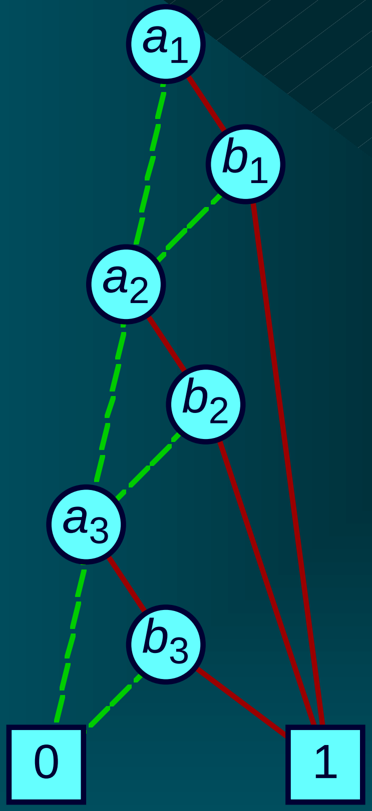
$F(a_1, a_2, a_3, b_1, b_2, b_3) = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$



Effect of Variable Ordering

$F\ (a_1, a_2, a_3, b_1, b_2, b_3) = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$

Good Ordering

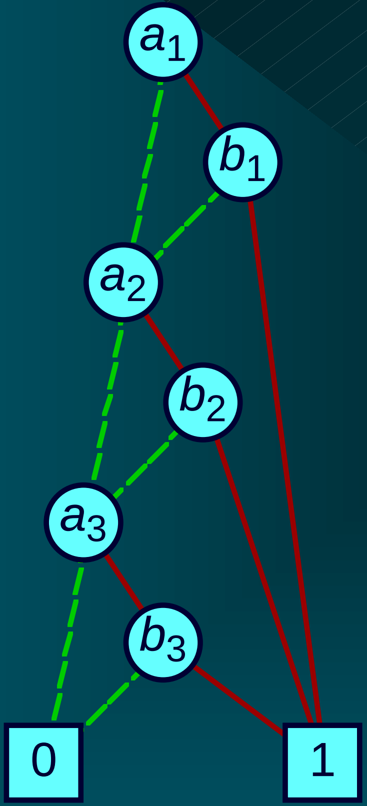


Linear Growth

Effect of Variable Ordering

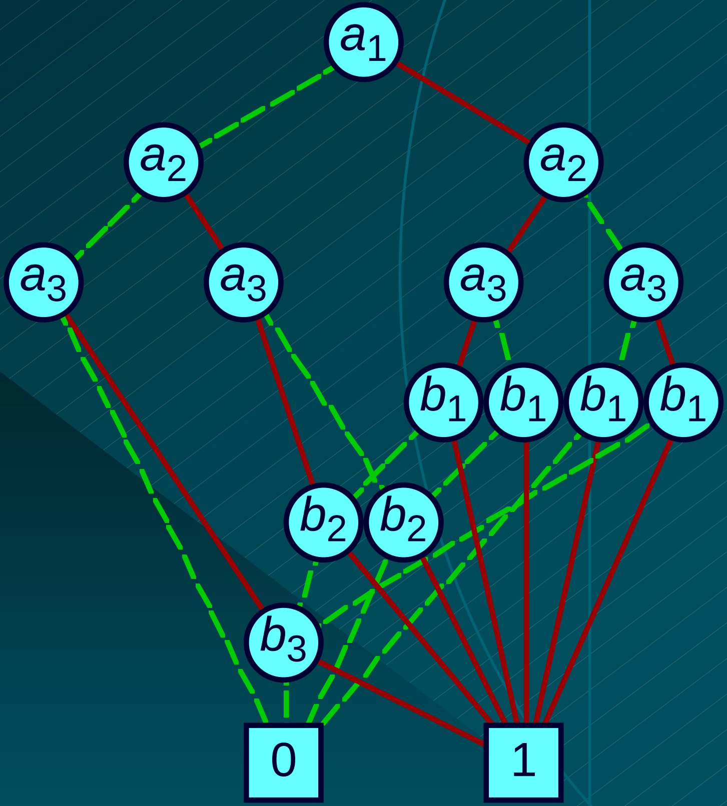
$F(a_1, a_2, a_3, b_1, b_2, b_3) = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$

Good Ordering



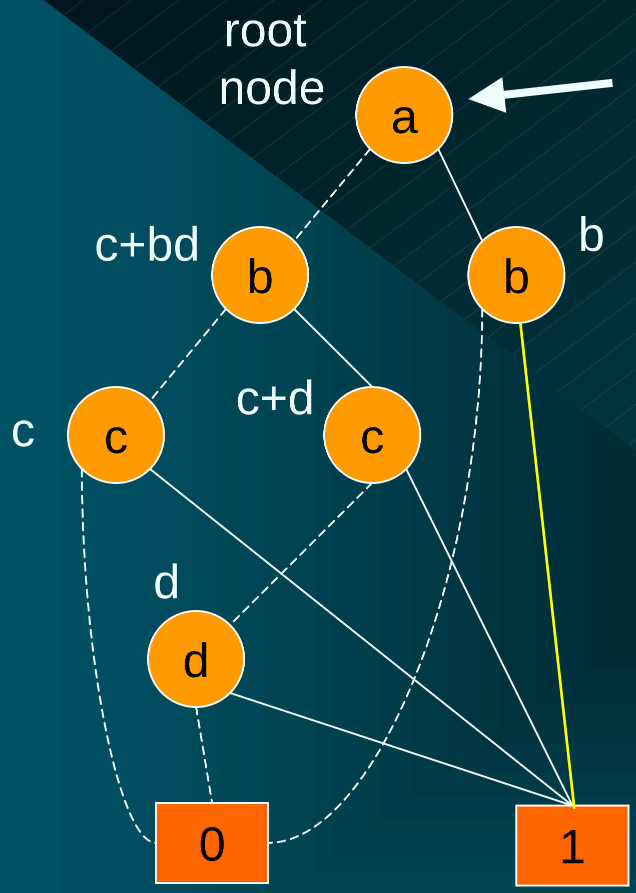
Linear Growth

Bad Ordering

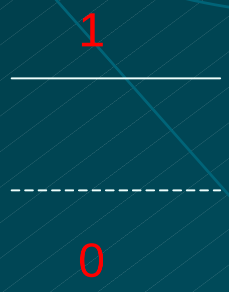


Exponential Growth

Exercise



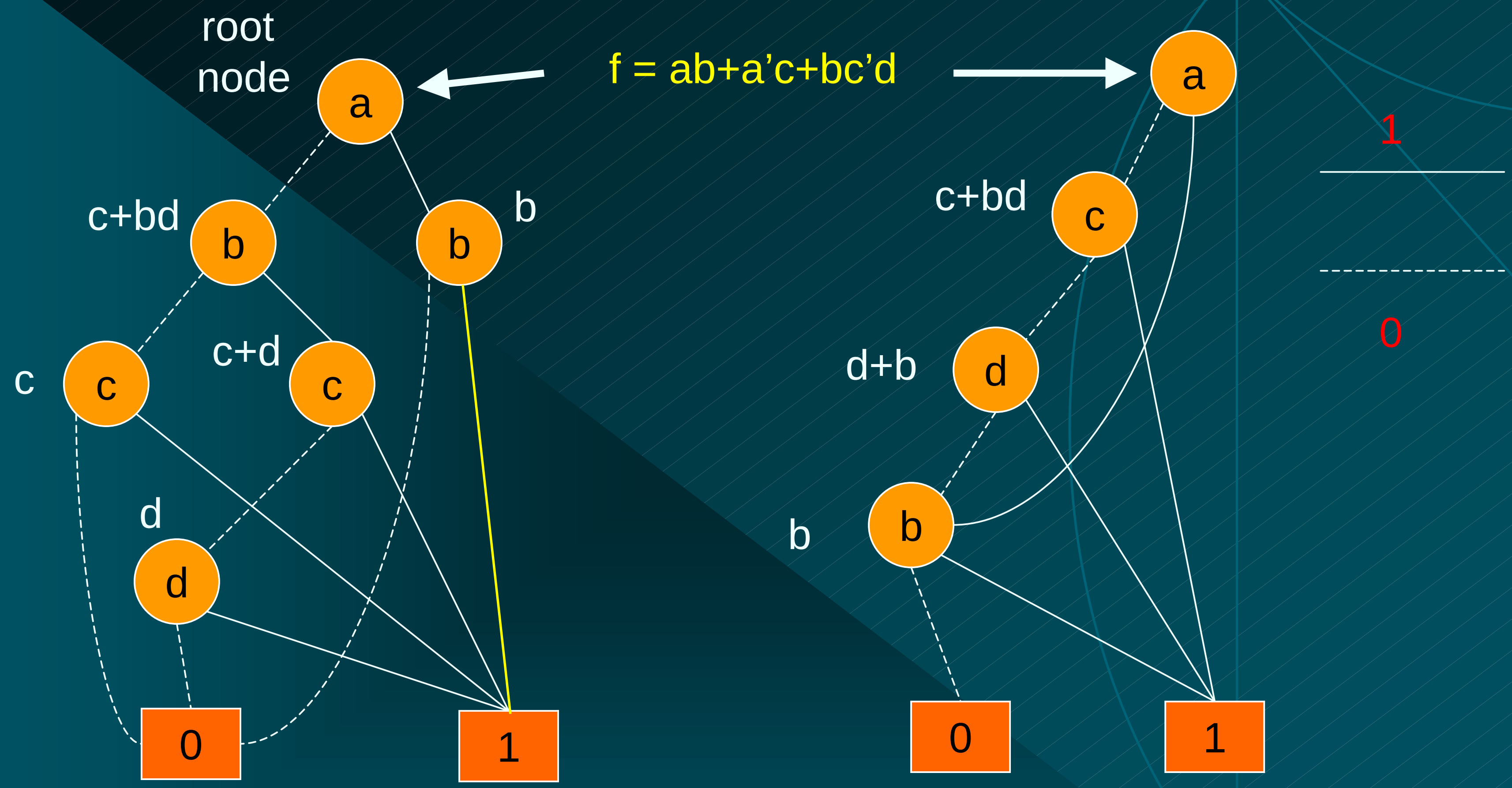
f = ab+a’c+bc’d



Given the BDD with variable order a, b, c, d

Represents it with the order a, c, d, b.

Exercise



f = ab+a'c+bc'd

Given the BDD with variable order a, b, c, d

Represents it with the order a, c, d, b.

Sample Function Classes

Function Class	Best	Worst	Ordering Sensitivity
ALU (Add/Sub)	linear	exponential	High
Symmetric	linear	quadratic	None
Multiplication	exponential	exponential	Low

- General Experience
  - Many tasks have reasonable OBDD representations
  - Algorithms remain practical for up to 5,000,000 node OBDDs
  - Heuristic ordering methods generally satisfactory

Consideration on Variable Ordering

- Variable order is fixed

  For each path from root to terminal node the order of ”input” variables is exactly the same

- Strong dependency of the BDD size (terms of nodes) and variable ordering

- Ordering algorithm:
  - Co-NP complete problem - heuristic approaches
  - Static Variable Ordering Heuristic
  - Dynamic Variable Ordering Heuristic
  - ROBDDs - Reduced Ordered Binary DDs (BDDs!)

Static Variable Ordering

- Different heuristic introduced over the years
- Usually based on the circuit structure
  - E.g., depth-first visit from the outputs
- Sufficient for “static problems”
- Insufficient for “dynamic requirements”

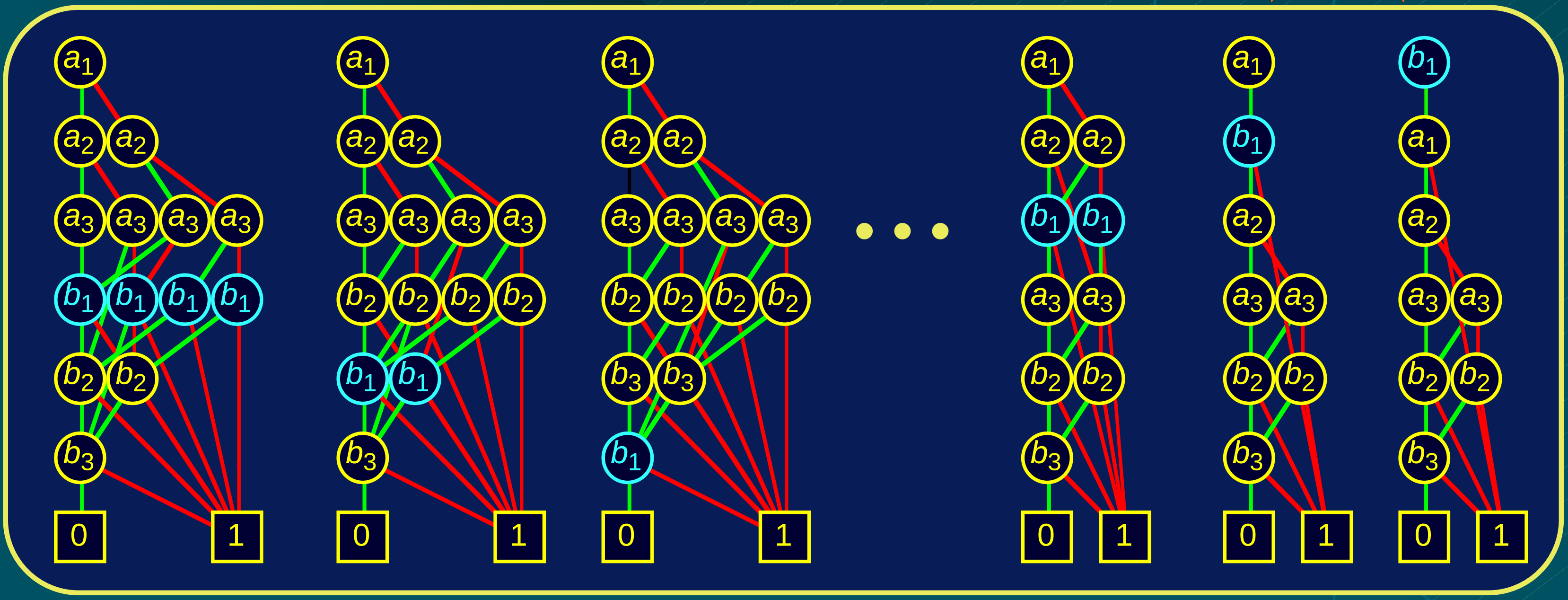
Dynamic Variable Reordering

- First Introduced by Richard Rudell, Synopsys, 1991
- Periodically Attempt to Improve Ordering for All BDDs
  - Part of garbage collection
  - Move each variable through ordering to find its best location
- Has Proved Very Successful
  - Time consuming but effective
  - Especially for sequential circuit analysis

Dynamic Reordering By Sifting

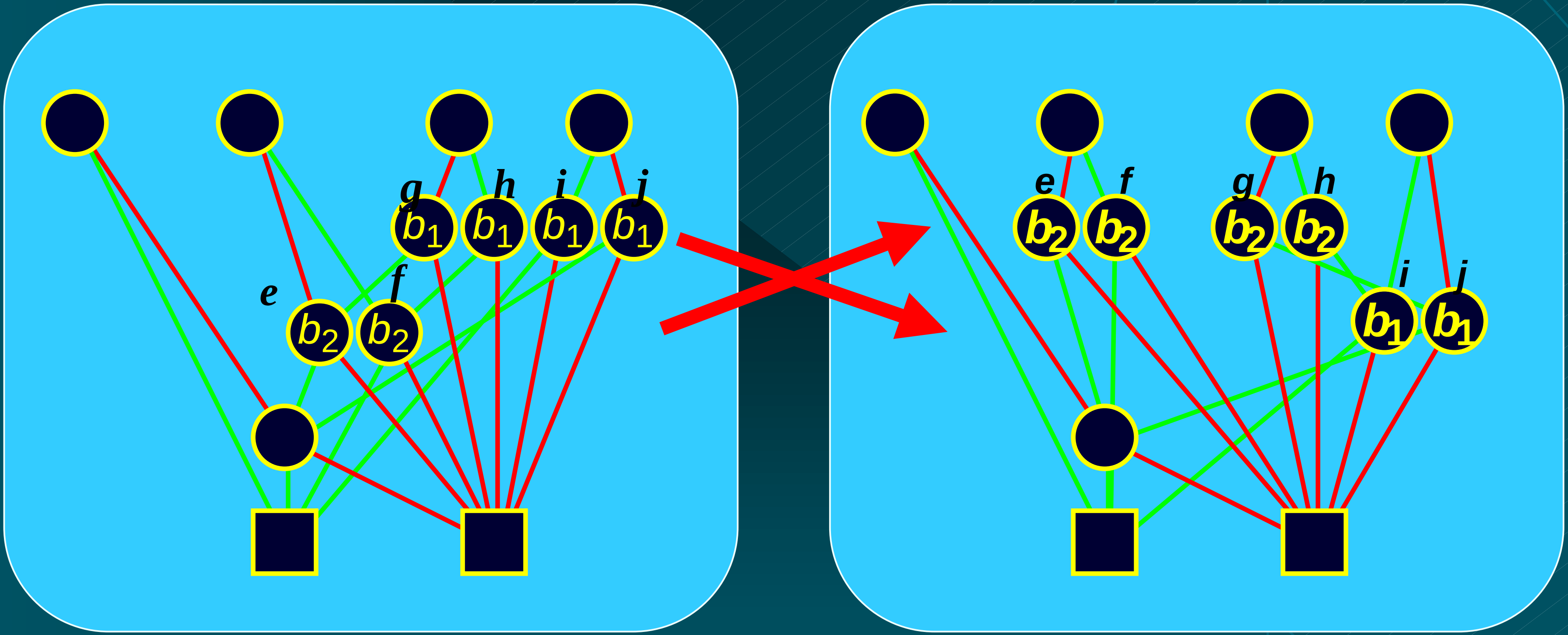
- Choose candidate variable
- Try all positions in variable ordering
  - Repeatedly swap with adjacent variable
- Move to best position found

Best Choices



Swapping Adjacent Variables

- Localized Effect
  - Add / delete / alter only nodes labeled by swapping variables
  - Do not change any incoming pointers

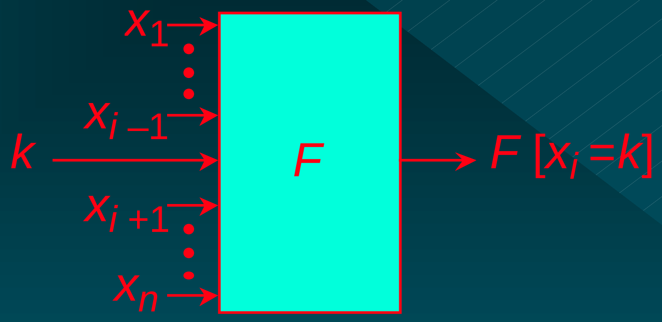


Restriction

- Concept
  - Effect of setting function argument x_i to constant k (0 or 1).
  - Also called Cofactor operation (UCB)

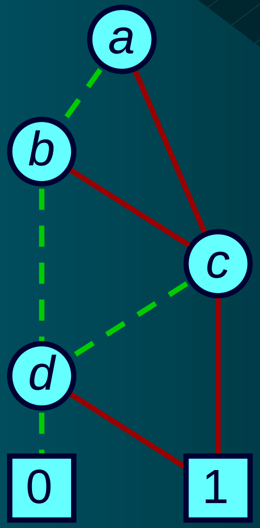
    F_x equivalent to $F\,[x=1]$

    $F_{\neg x}$ equivalent to $F\,[x=0]$

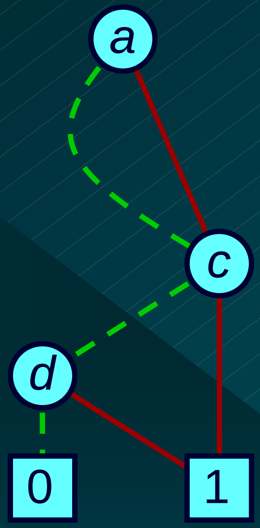


Restriction Execution Example

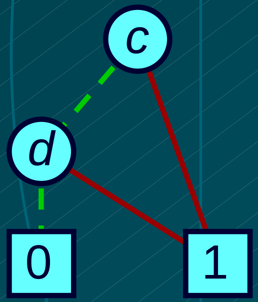
Argument *F*



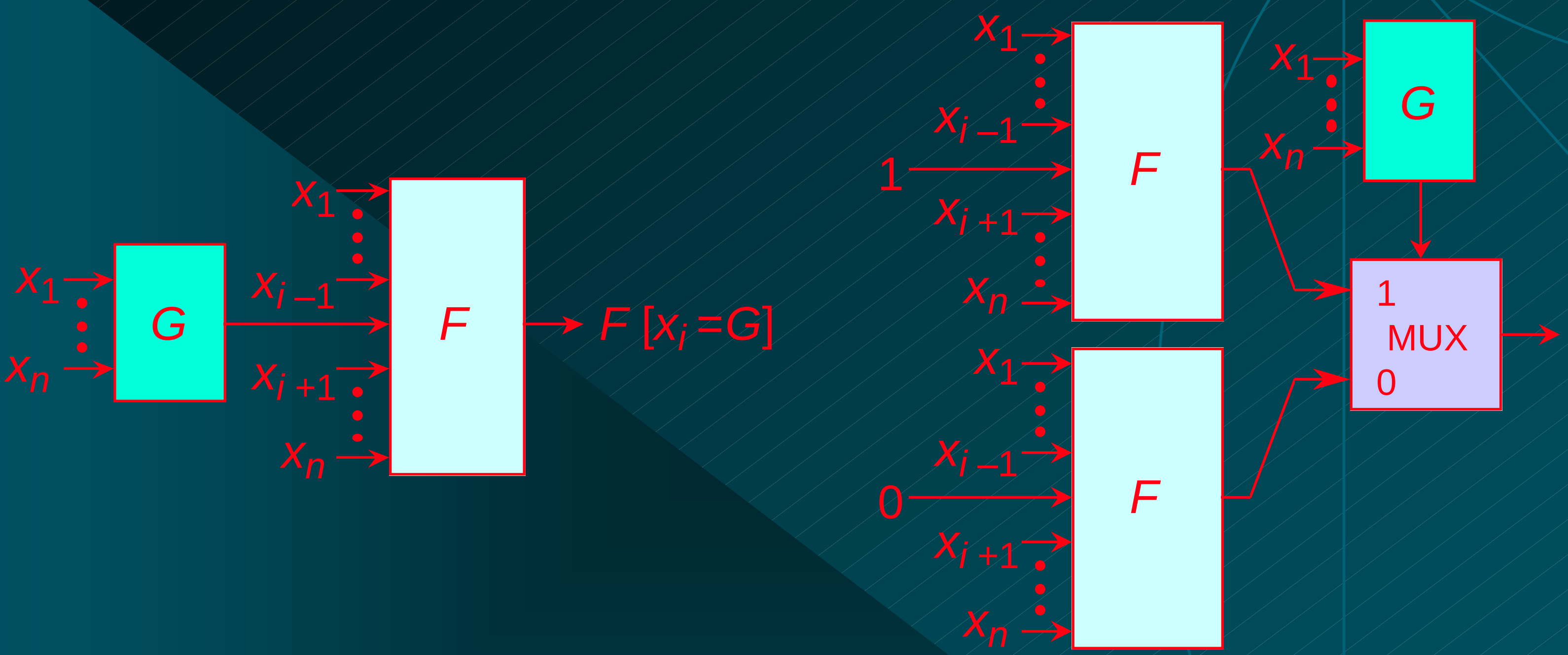
Restriction *F*[*b*=1]



Reduced Result



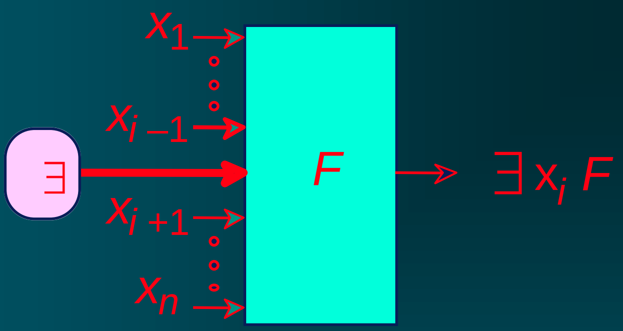
Functional Composition

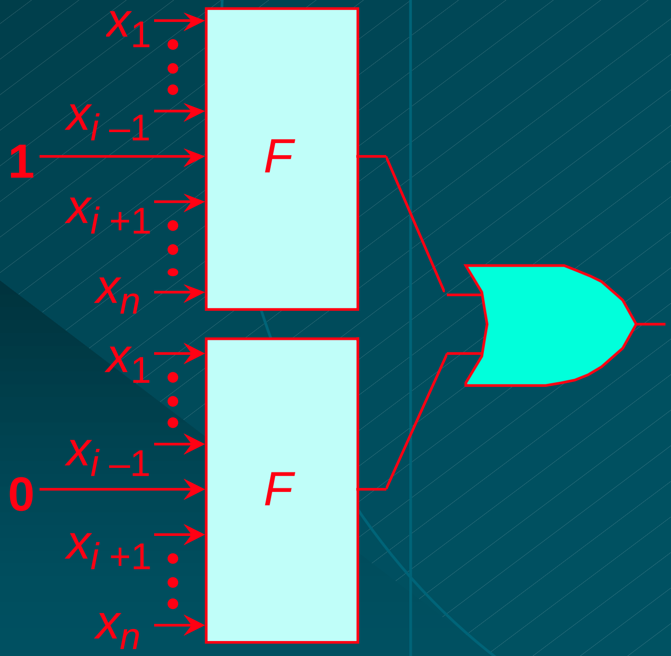


- Create new function by composing functions F and G.
- Useful for composing hierarchical modules.

Existential Variable Quantification

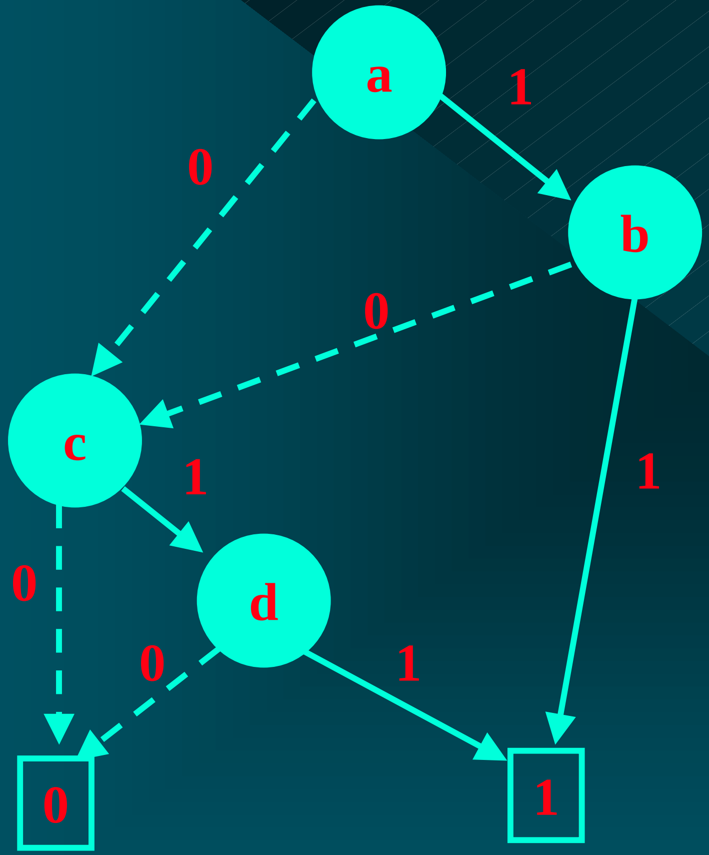
- $\exists_b f = f|_{b=0} \vee f|_{b=1}$
  - Eliminate dependency on some argument
  - Efficient algorithm for quantifying over a set of variables





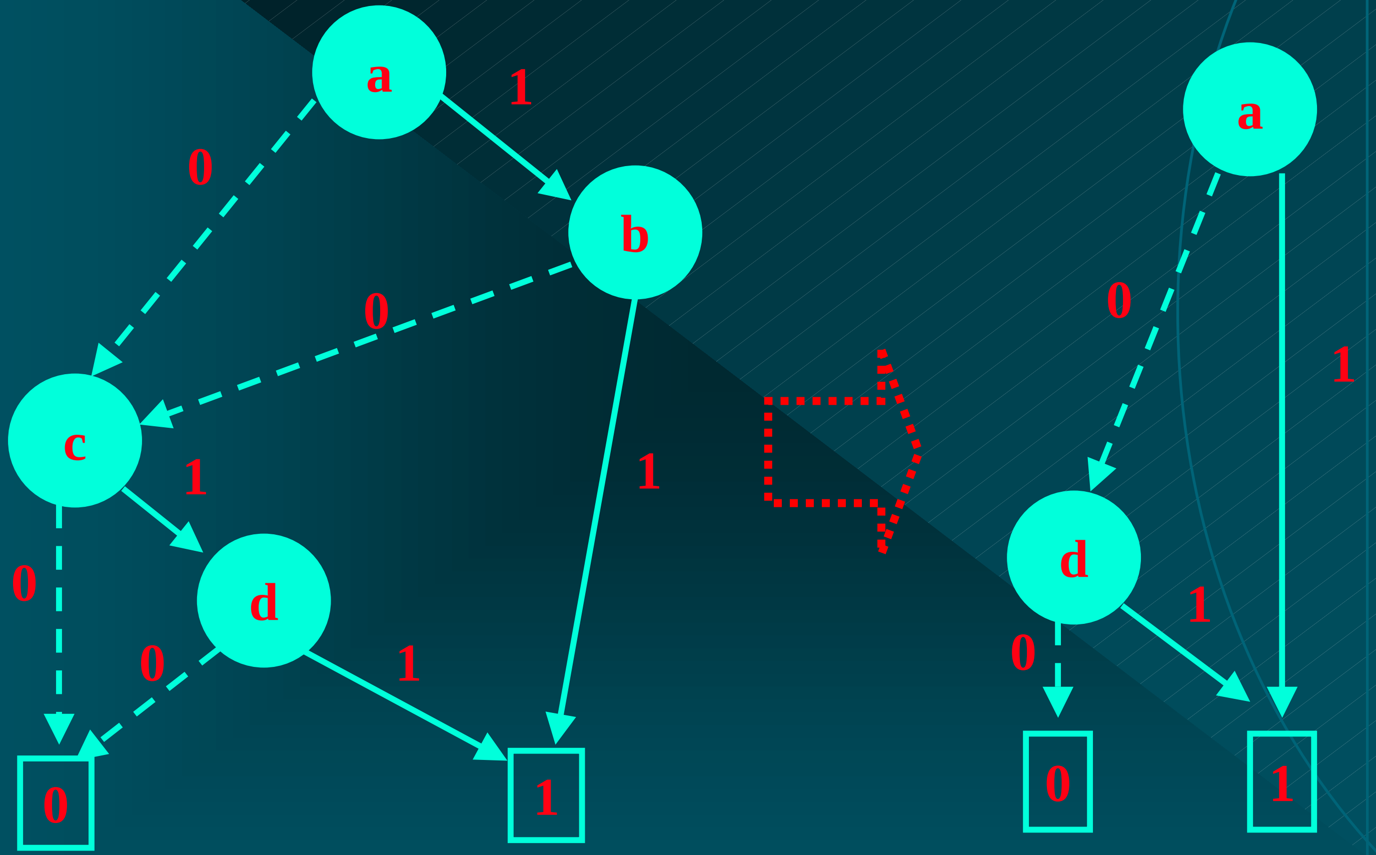
Example

$\exists_{(b,c)}.\ ((a \wedge b) \vee (c \wedge d))\ =\ ?$



Example

$\exists_{(b,c)}.\ ((a \wedge b) \vee (c \wedge d)) = a \vee d$



Universal Variable Quantification

- $\forall_b f = f|_{b=0} \wedge f|_{b=1}$
  - Obtained with cofactor combined with AND

What’s good about BDDs?

- Powerful Operations
  - Creating, manipulating, testing
  - Each step polynomial complexity
    - Graceful degradation
- Generally Stay Small Enough
  - Especially for digital circuit applications
  - Given good choice of variable ordering
- Extremely useful in practice
- (Until late 90s) Weak Competition
  - No other method close in overall strength
  - Especially with quantification operations

What’s bad about BDDs?

- **Some formulas do not have small representation! (e.g., multipliers)**
- **BDD representation of a function can vary exponentially in size depending on variable ordering; users may need to play with variable orderings (less automatic)**
- **Size limitations: a big problem**
- **(Last years) Competitive Approach: CNF representation + SATisfiability solvers**

A few BDD Packages

- **Brace, Rudell, Bryant:** KBDD
  - Carnegie Mellon, 1990
  - Synopsys, 1993 on
  - Digital, Compaq, Intel, 1993 on
- **Long:** KBDD
  - Carnegie Mellon, 1993
  - AT&T, 1995 on
- **Armin Biere**: ABCD
  - Carnegie Mellon / Universität Karlsruhe
- **Olivier Coudert**: TiGeR
  - Synopsys / Monterey Design Systems
- **Geert Janssen**: EHV
  - Eindhoven University of Technology

- **Geert Janssen**: EHV
  - Eindhoven University of Technology
- **Rajeev K. Ranjan**: CAL
  - UCB, Synopsys
- **Bwolen Yang**: PBF
  - Carnegie Mellon
- **Stefan Horeth**: TUDD
  - University TU Darmstadt
  - **http://marple.rs.e-technik.tu-darmastadt.de/~sth**
- **Fabio Somenzi**: CUDD
  - University of Colorado
  - **http://vlsi.colorado.edu/~fabio**

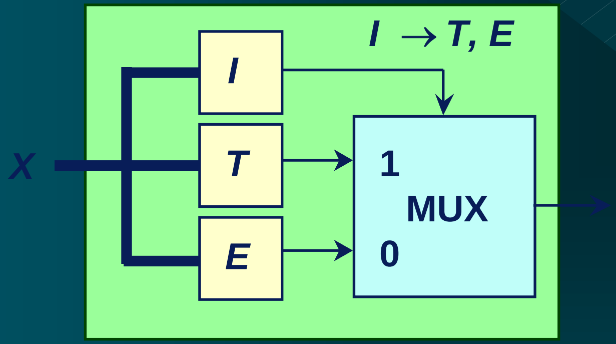
Symbolic manipulation with OBDDs

- Strategy
  - Represent data as set of OBDDs
    - Identical variable orderings
  - Express solution method as sequence of symbolic operations
  - Implement each operation by OBDD manipulation

- Key Algorithmic Properties
  - Arguments: OBDDs with identical variable orders
  - Result is OBDD with same ordering
  - Each step polynomial complexity

If-Then-Else operation

- **Concept**
  - **Basic technique for building OBDD from logic network or formula.**



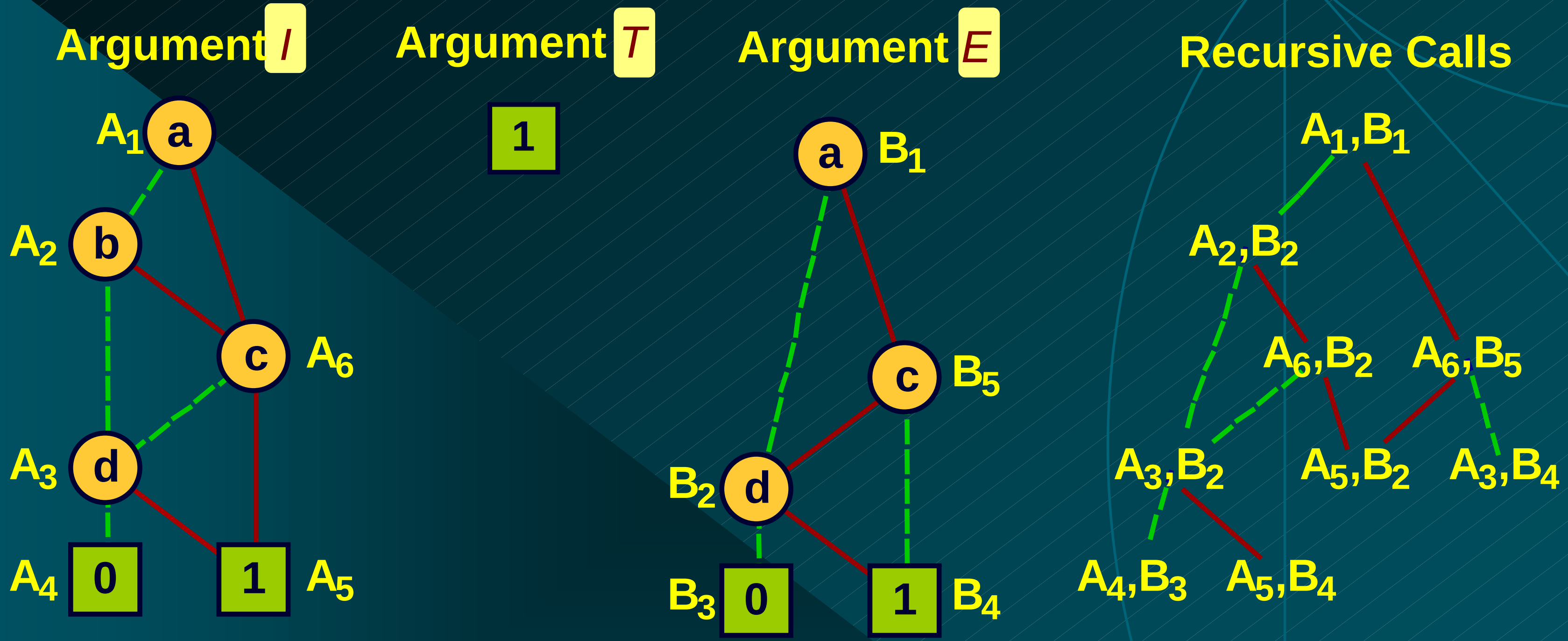
Arguments *I*, *T*, *E*

- Functions over variables *X*
- Represented as OBDDs

Result

- OBDD representing composite function
- $(I \wedge T) \vee (\neg I \wedge E)$

If-Then-Else execution example

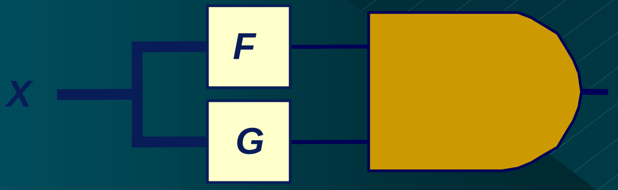


- **Optimizations**
  - **Dynamic programming**
  - **Early termination rules**
  - **Apply reduction rules bottom-up as return from recursive calls**
    - ***(Recursive calling structure implicitly defines unreduced BDD)***

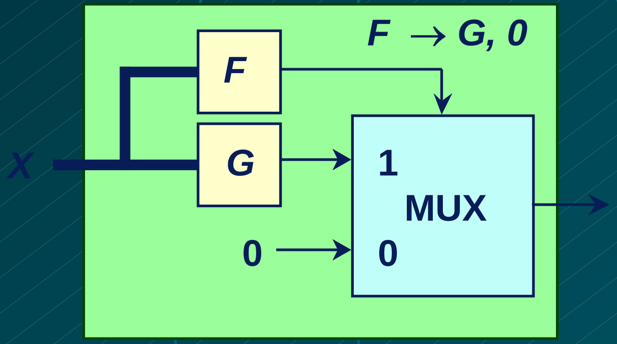
Derived algebraic operations

- Other operations can be expressed in terms of If-Then-Else

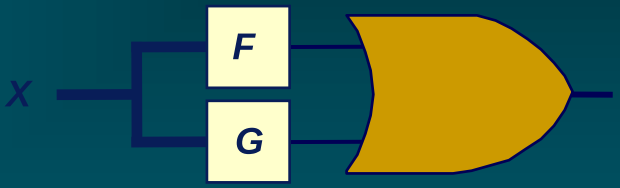
And(*F*, *G*)



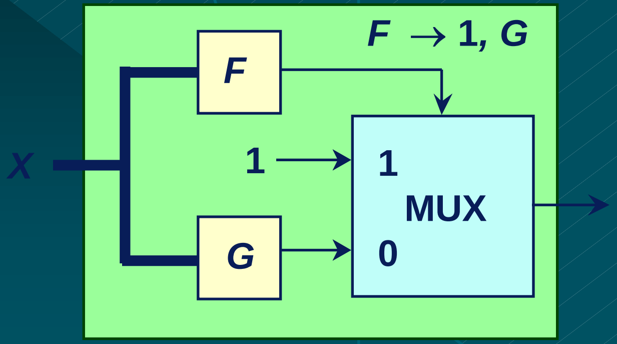
If-Then-Else(*F*, *G*, 0)



Or(*F*, *G*)

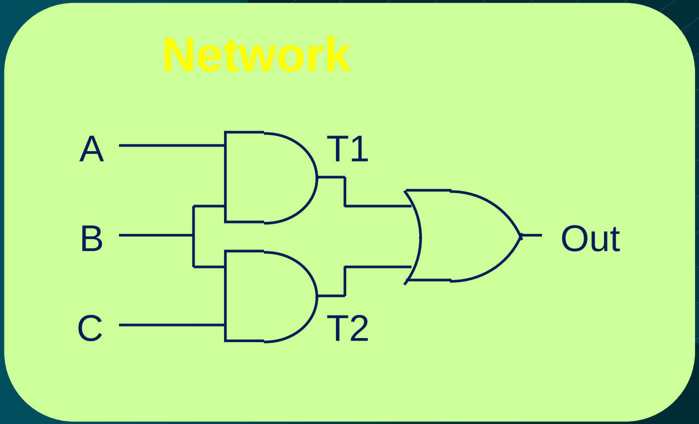


If-Then-Else(*F*, 1, *G*)



Generating OBDD from network

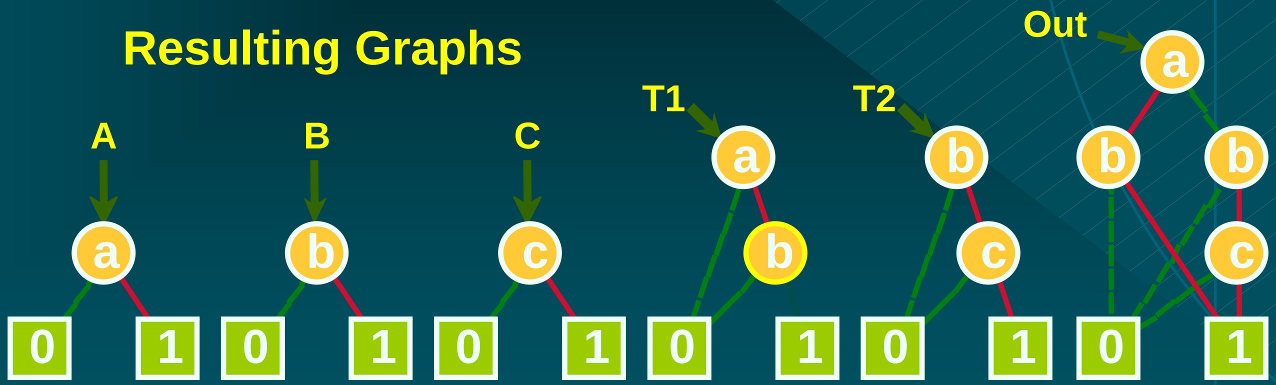
Task: Represent output functions of gate network as OBDDs.



Evaluation

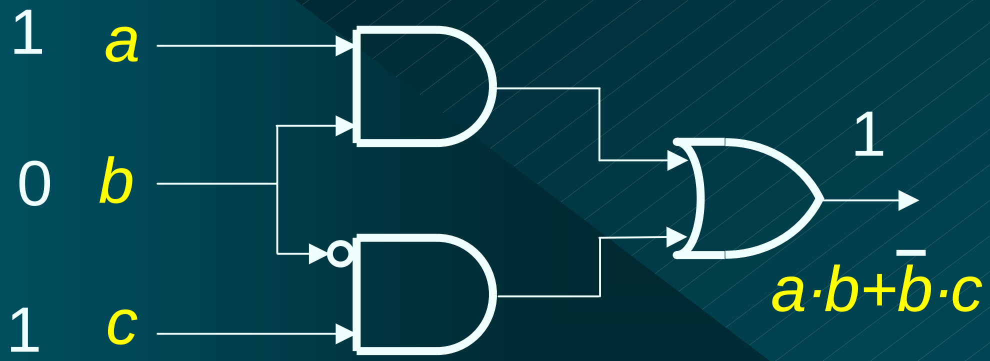
```
A    ←  new_var ("a");
B    ←  new_var ("b");
C    ←  new_var ("c");
T1   ←  And (A, 0, B);
T2   ←  And (B, C);
Out  ←  Or (T1, T2);
```

Resulting Graphs



Symbolic simulation

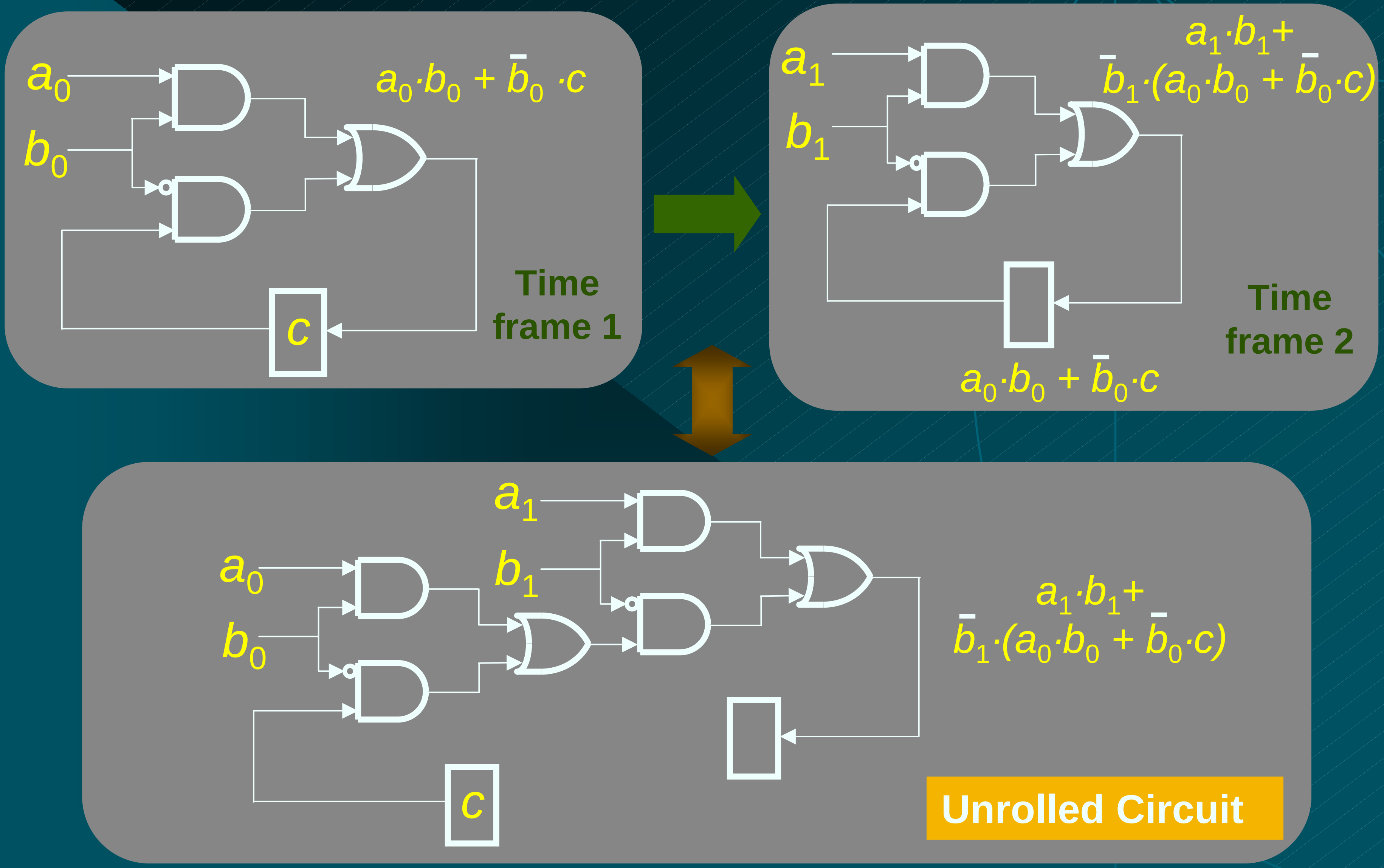
- **Conventional simulation:**
  - **Input & outputs are constants (0,1,X,…)**



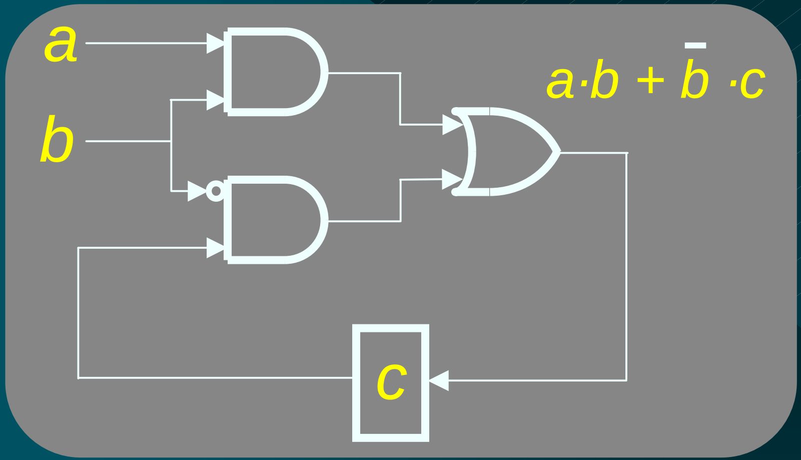
Problem: Too many constant input combinations to simulate !!

- **Symbolic simulation:**
  - **Symbolic expressions used for inputs**
  - **Expressions propagated to compute outputs**
  - **Equivalent to multiple constant simulations !!**

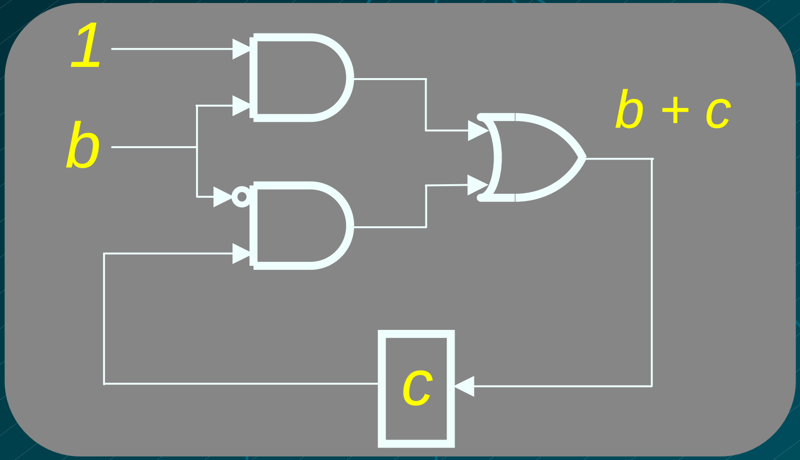
Symbolic simulation (sequential case)


a_0
b_0
$a_0 \cdot b_0 + \bar{b}_0 \cdot c$
c
Time frame 1
a_1
b_1
$a_1 \cdot b_1 +$
$\bar{b}_1 \cdot (a_0 \cdot b_0 + \bar{b}_0 \cdot c)$
$a_0 \cdot b_0 + \bar{b}_0 \cdot c$
Time frame 2
a_1
a_0
b_1
b_0
$a_1 \cdot b_1 +$
$\bar{b}_1 \cdot (a_0 \cdot b_0 + \bar{b}_0 \cdot c)$
c
Unrolled Circuit

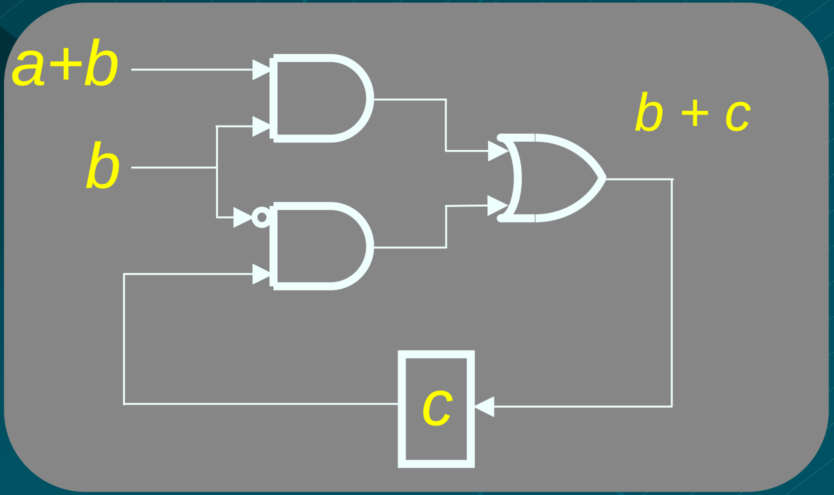
Symbolic simulation



Inputs can be constants



Input expressions can be related



- **Simulate certain set of patterns**
- **Model signal correlations**
- **Can result in simpler output expressions**

Symbolic simulation

- Use BDDs as the symbolic representation
- Work at gate and MOS transistor level
- Can exploit abstraction capabilities of ’X’ value
  - Can be used to model unknown/don’t care values
  - Common use in representing uninitialized state variables
  - Boolean functions extended to work with {0,1,X}
  - Two BDD (binary) variables used to represent each symbolic variable

Symbolic simulation

Advantages

- Can handle larger designs than model checking
- Can use a large variety of circuit models
- Possibly more natural for non-formalists.
- Amenable to partial verification.

Disadvantages

- Not good with state machines (possibly better with data paths).
- Does not support temporal logic
  - Requires ingenuity to prove properties.

Practical deployment

- Systems:
  - COSMOS [bryant et al], Voss[Seger et al Intel]
  - Magellan [Synopsys]
  - Innologic
- Exploiting hierarchy
  - Symbolically encode circuit structure
    - Based on hierarchy in circuit description
  - Simulator operates directly on encoded circuit
    - Use symbolic variables to encode both data values & circuit structure
  - Implemented by Innologic, Synopsys (DAC ‘02)
  - Greatest success in memory verification (Innologic)

High-level symbolic simulation

- **Data Types: Boolean, bitvectors, int, reals, arrays**
- **Operations: logical, arithmetic, equality, uninterpreted functions**
- **Final expression contains variables and operators**
- **Coupled with Decision procedures to check correctness of final expression**
- **Final expressions can also be manually checked for unexpected terms/variables, flagging errors   e.g. in JEM1 verification [Greve '98]**

High-level symbolic simulation

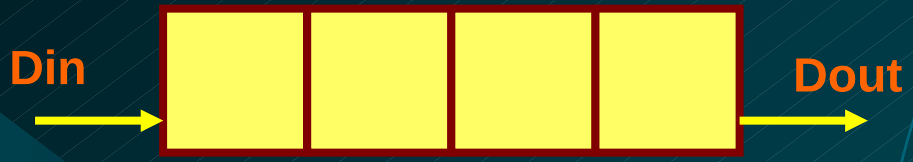
- Manipulation of symbolic expressions done with
  - Rewrite systems like in PVS
  - Boolean and algebraic simplifiers along with theories of linear inequalities, equalities and uninterpreted functions
- Extensively used along with decision procedures in microprocessor verification
  - Pipelined processors: DLX
  - Superscalar processors: Torch (Stanford)
  - Retirement logic of Pentium Pro
  - Processors with out of order executions

Symbolic trajectory evaluation (STE)

- Trajectory : Sequence of values of system variables
  - Example: c = AND (a, b) and delay is 1
  - A possible trajectory : (a,b,c) = (0, 1, X), (1, 1, 0), (1, 0, 1), (X, X, 0), (X, X, X),…
- Express behavior of system model as a set of trajectories I and desired property as a set of trajectories S
- Determine if I is inconsistent with S
  - Inconsistent: I says 0 but S says 1 for a signal at time t
  - Consistent: I says 0 but S says X or 0 for a signal at time t

STE: An example

4-Bit Shift Register



Din

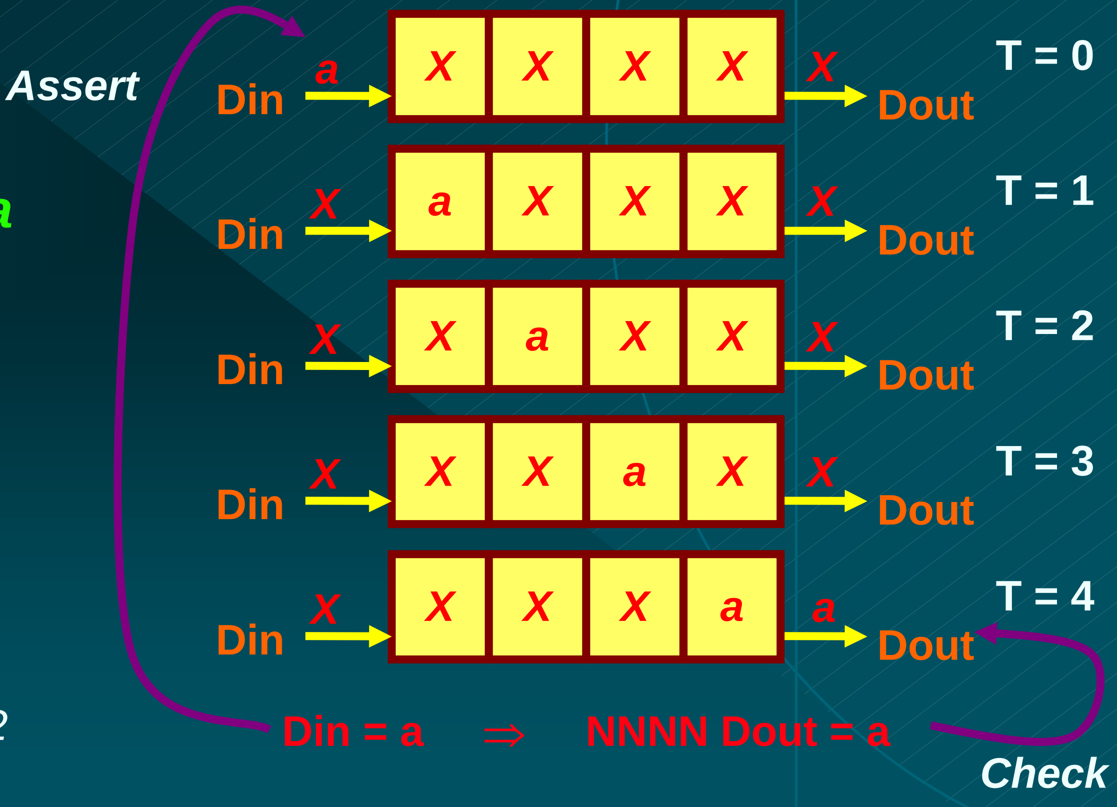
Dout

Specification

Din = *a* ⇒ NNNN Dout = *a*

If apply input a then 4 cycles later will get output a

Ref: Prof. Randal Bryant, 2002



Assert

Din						Dout	
a	X	X	X	X	X		T = 0
X	*a*	X	X	X	X		T = 1
X	X	*a*	X	X	X		T = 2
X	X	X	*a*	X	X		T = 3
X	X	X	X	*a*	*a*		T = 4

Din = a ⇒ NNNN Dout = a

Check

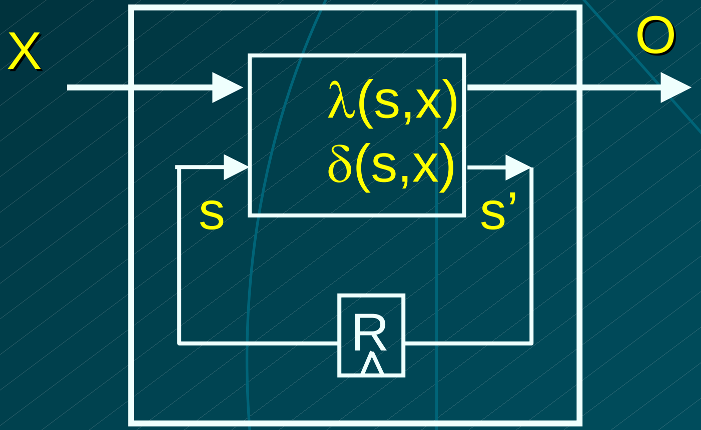
STE: Pros, cons & u

- Advantage: Higher capacity than symbolic model checking
- Disadvantage: Properties checkable not as expressive as CTL
- Practical success of STE
  - Verification of arrays (memories, TLBs etc.) in Power PC architecture
  - x86 Instruction length decoder for Intel processor
  - Intel FP adder
  - Microprocessor verification

Symbolic

Reachability Analysis

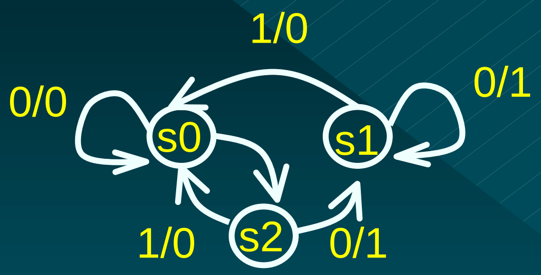
Finite State Machines (FSM)



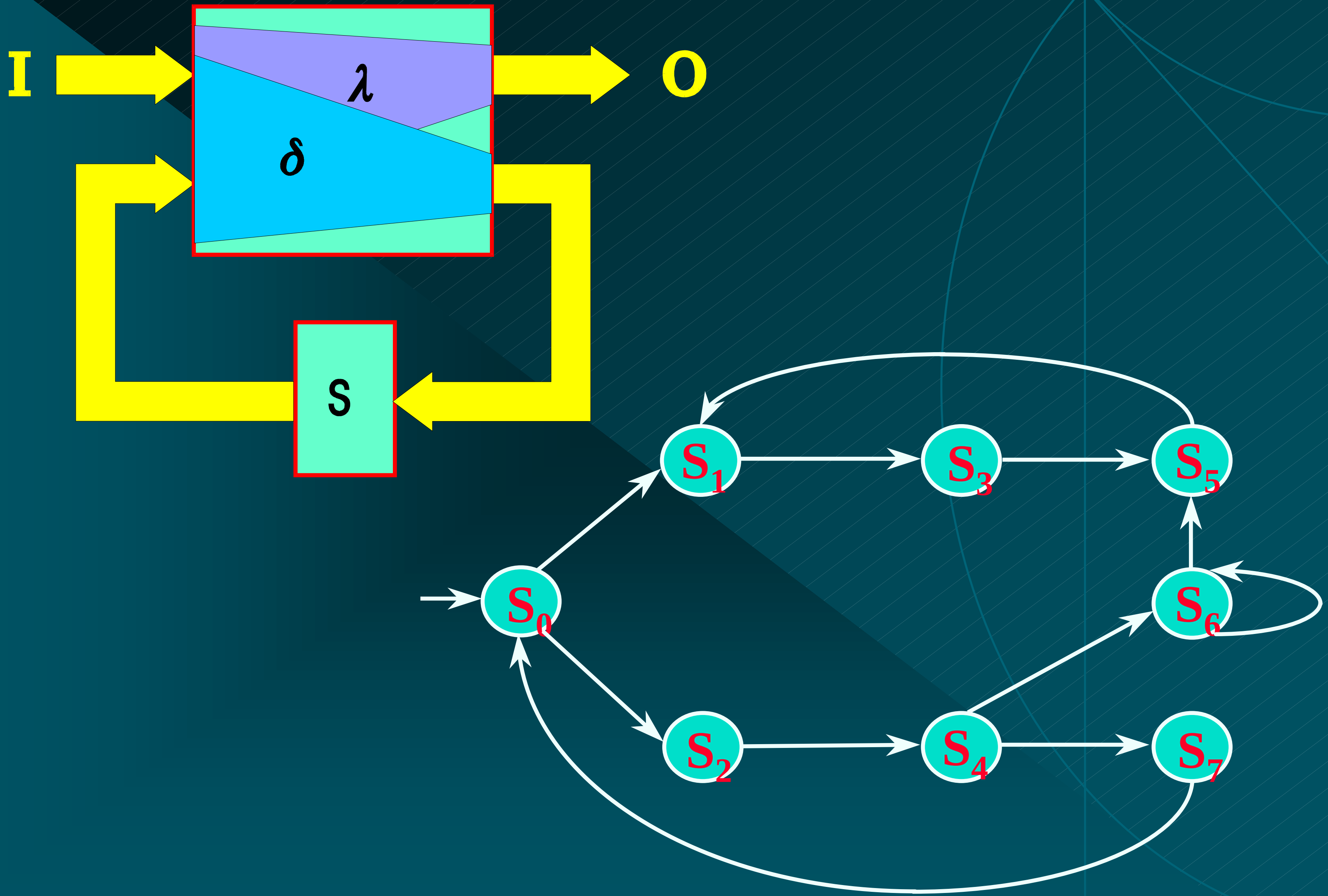
- FSM M(X,S, δ, λ,O)
  - Inputs: X
  - Outputs: O
  - States: S
  - Next state function, δ(s,x) : S × X → S
  - Output function, λ(s,x) : S × X → O

FSM Traversal

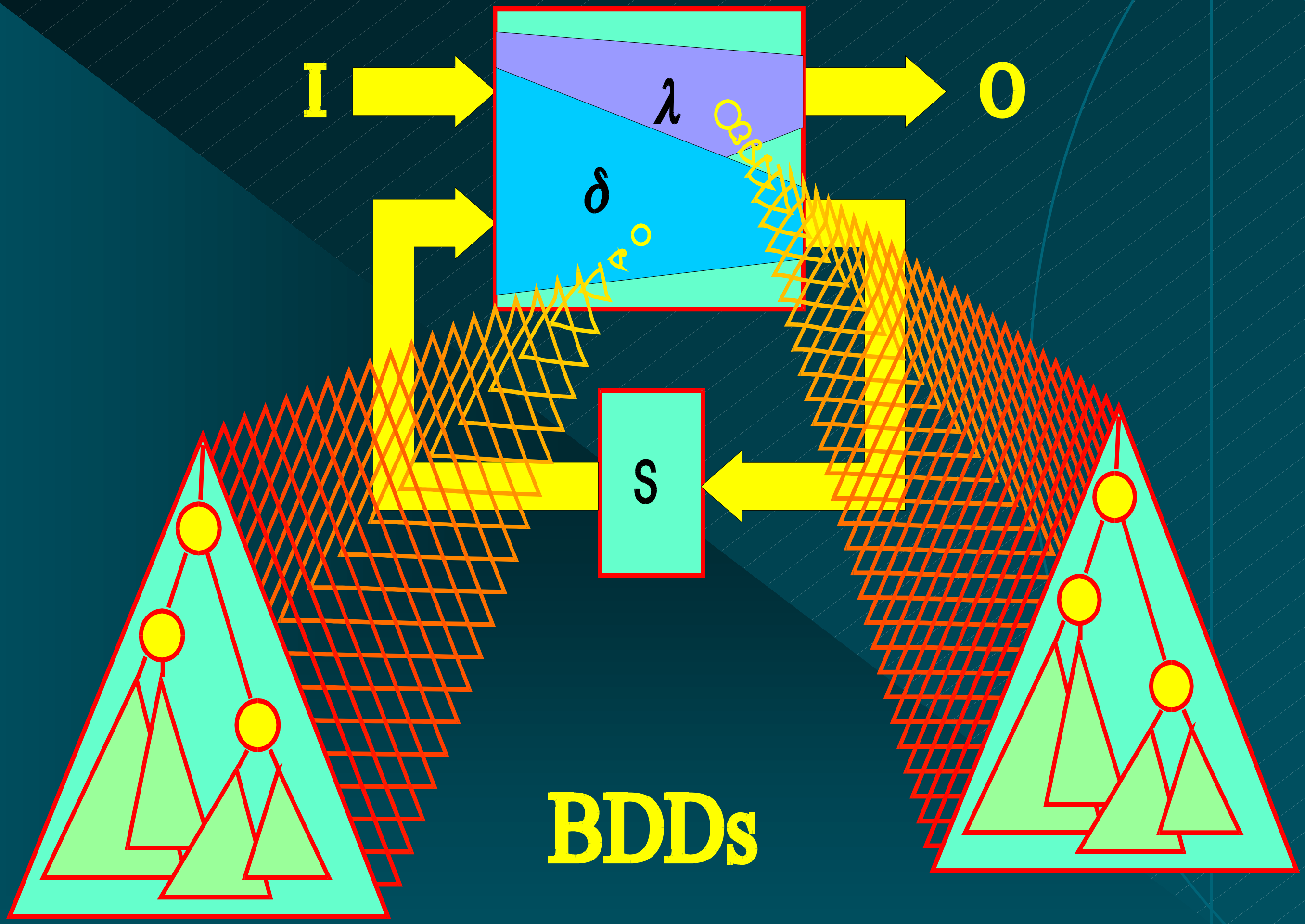
- State Transition Graphs
  - directed graphs with labeled nodes and arcs (transitions)
  - symbolic state traversal methods
    - important for symbolic verification, state reachability analysis, FSM traversal, etc.



Symbolic FSM representation



Function Representation



I

O

λ

δ

S

BDDs

Set Representation

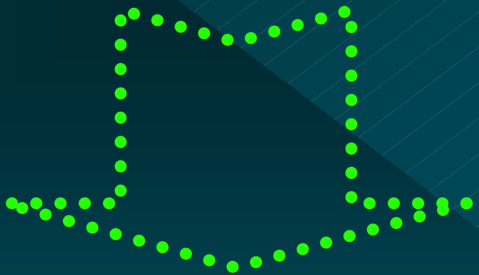
- **Idea**

  A formula can represent a set of states (its models)

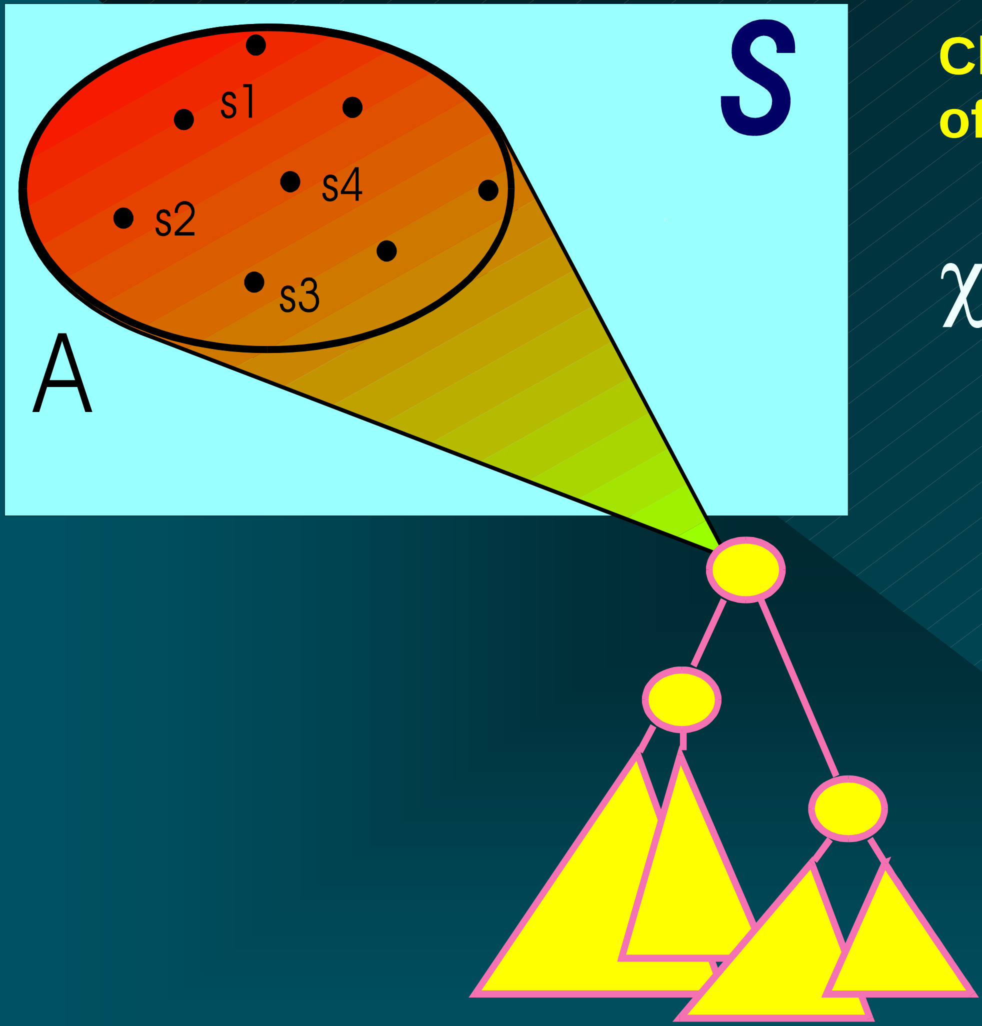
- **Example**

  (x ⊕ y) ⊕ z

  represents {100,010,110,111}



Characteristic Function

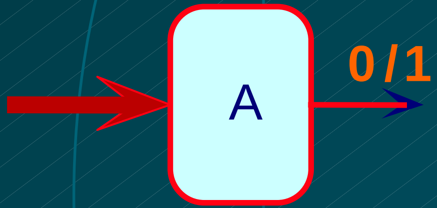


Characteristic Function of set A:

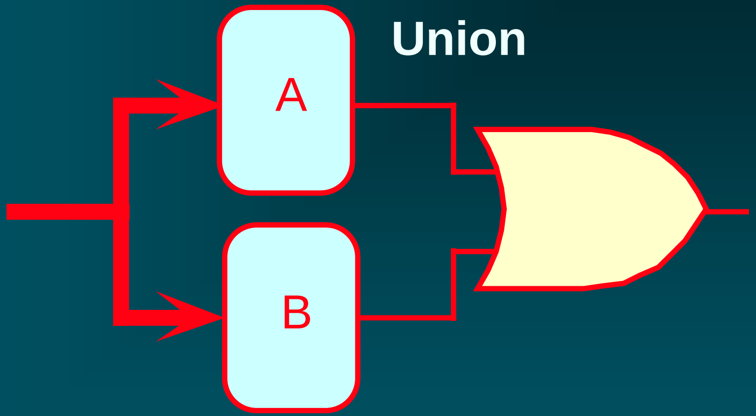
$$\chi_A(s) \quad = 1 \ \text{ IFF } s \in A$$
$$\quad\quad\quad = 0 \ \text{ IFF } s \notin A$$

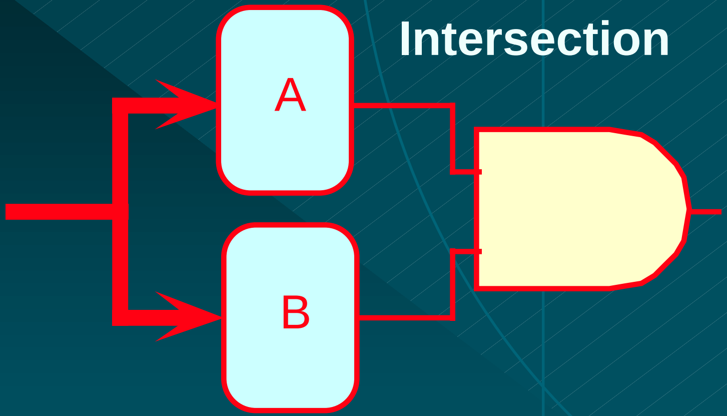
Characteristic functions

- $A \subseteq \{0,1\}^n$
  (Set of bit vectors of length *n*)
- Represent set *A* as Boolean function A of *n* variables
  $X \in A$ if and only if A(X) = 1



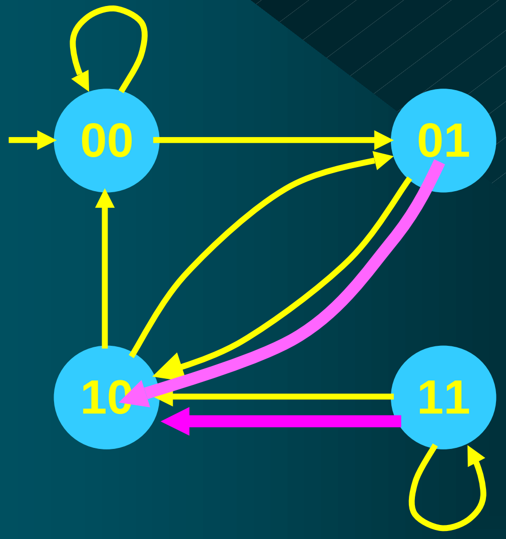
Set Operations



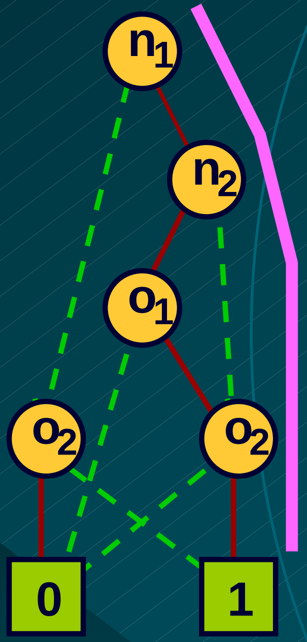


Symbolic FSM representation

Nondeterministic FSM



Symbolic Representation: Transition relation



o_1,o_2 encoded old state

n_1, n_2 encoded new state

- Represent set of transitions as function **tr(*Old*, *New*)**
  - Yields 1 if can have transition from state ***Old*** to state ***New***
- Represent as Boolean function
  - Over variables encoding states

The Transition Relation (deterministic FSM)

$$TR\ (s, x, y) = \prod_{i=1}^{n} (y_i \equiv \delta_i\ (s, x))$$

The Transition Relation espresses

present-state, primary input ⇒ next state correspondence.

$$TR(s,x,y) =$$

$$= \prod_{i=1}^{n}(y_i \equiv \delta_i\,(s,x))$$

$TR(s,x,y) =$

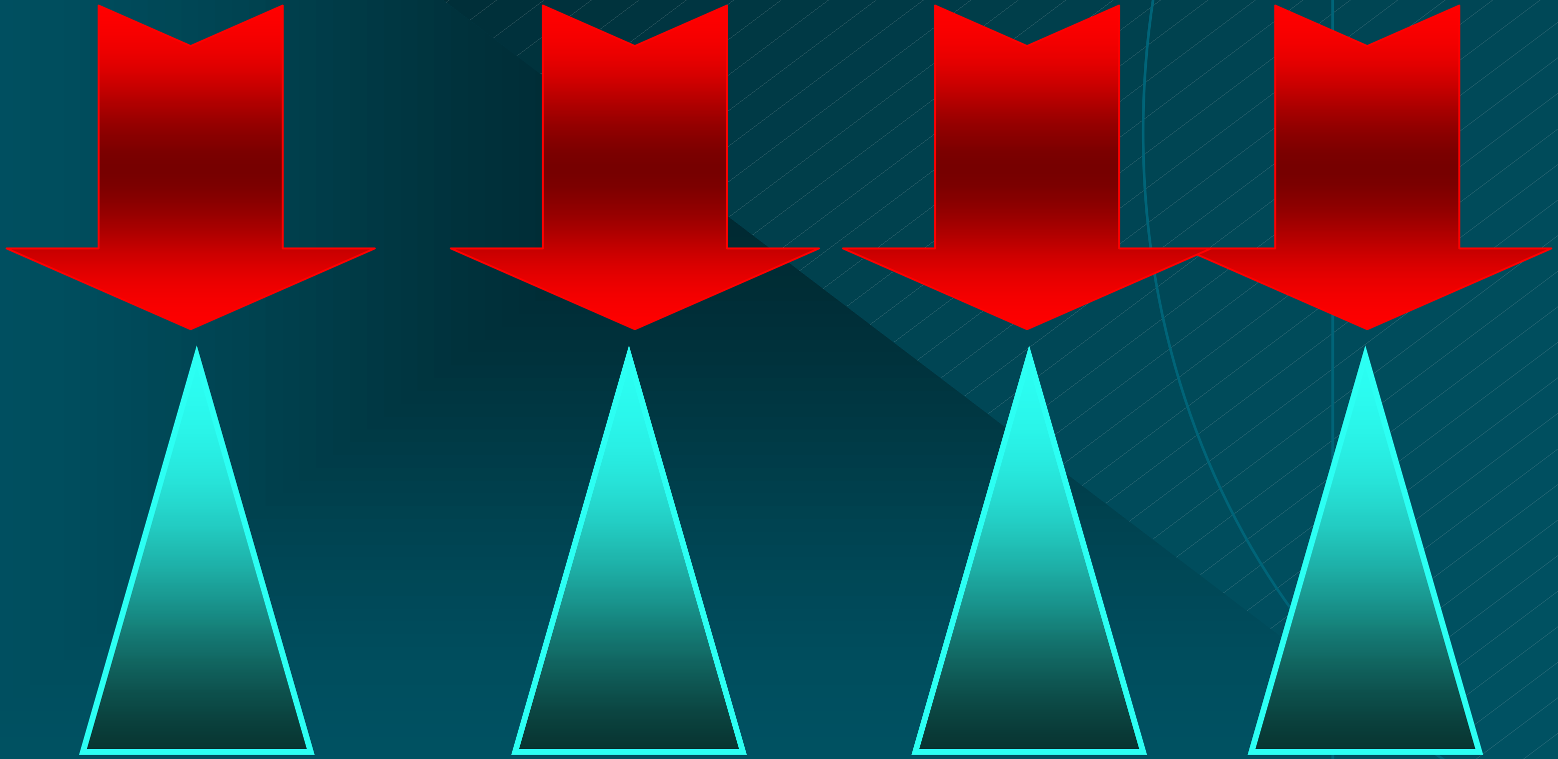
$$= \Pi_{i=1}^{n}(y_i \equiv \delta_i (s,x))$$

$$= [\ (y_1 \equiv \delta_1 (s,x)) \ \cdot \ (y_2 \equiv \delta_2 (s,x)) \ \cdot \ \ldots \ \cdot \ (y_n \equiv \delta_n (s,x))\ ]$$

$TR(s,x,y) =$

$$= \Pi_{i=1}^{n}(y_i \equiv \delta_i\,(s,x))$$

$$= [\,(y_1 \equiv \delta_1\,(s,x)) \;\cdot\; (y_2 \equiv \delta_2\,(s,x)) \;\cdot\; \ldots \;\cdot\; (y_n \equiv \delta_n\,(s,x))\,]$$

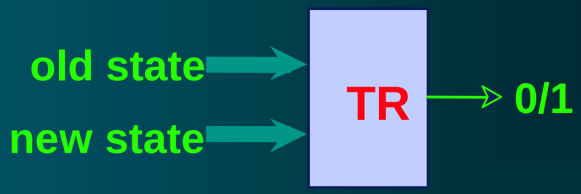


Reachability analysis

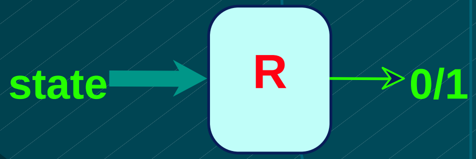
Task

- Compute set of states reachable from initial state Q_0
- Represent as Boolean function $R(S)$
- Never enumerate states explicitly

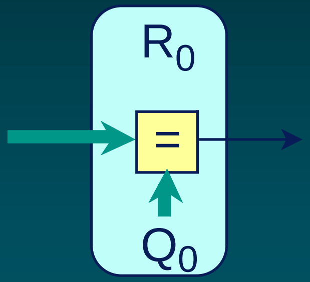
Given

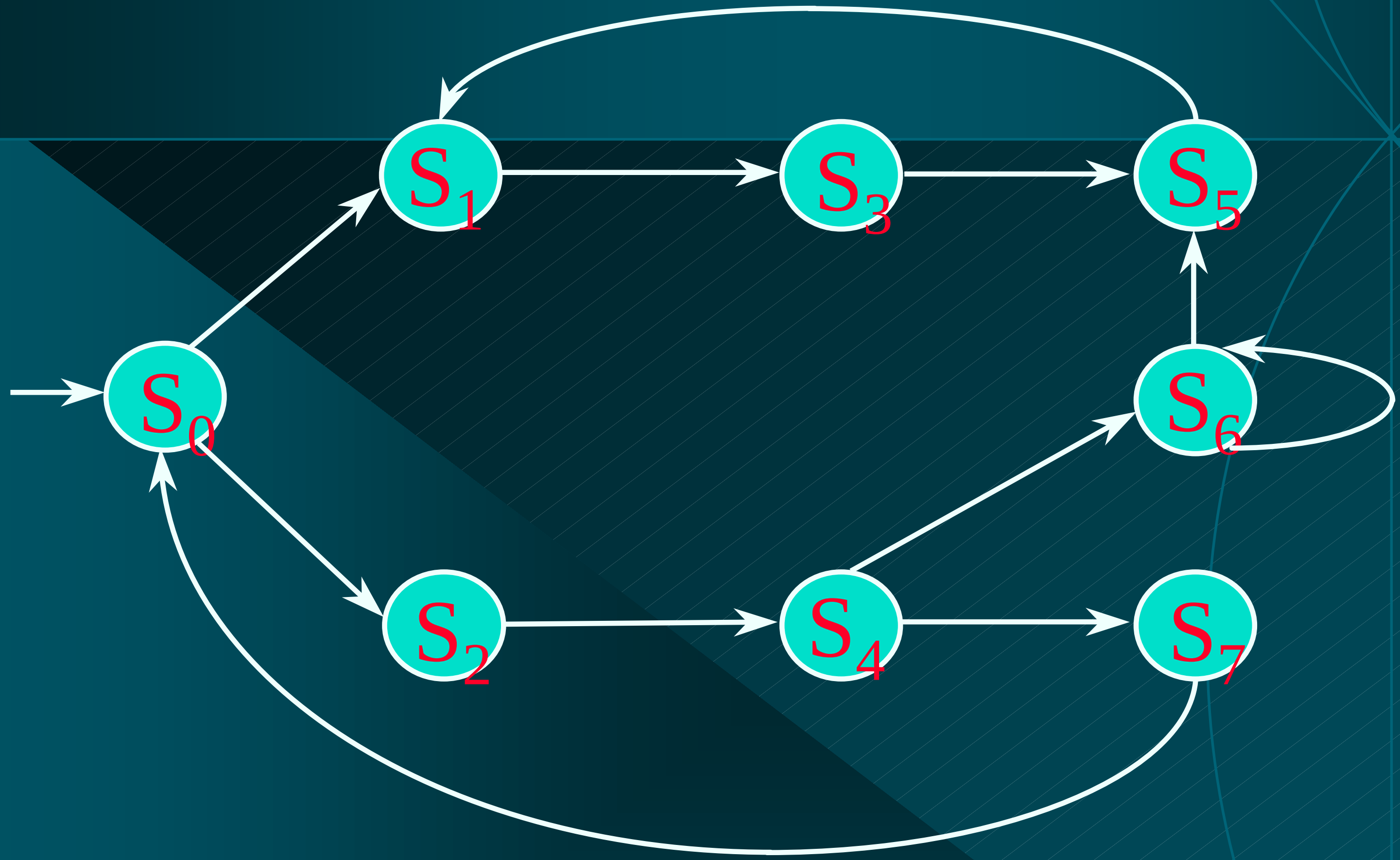


Compute

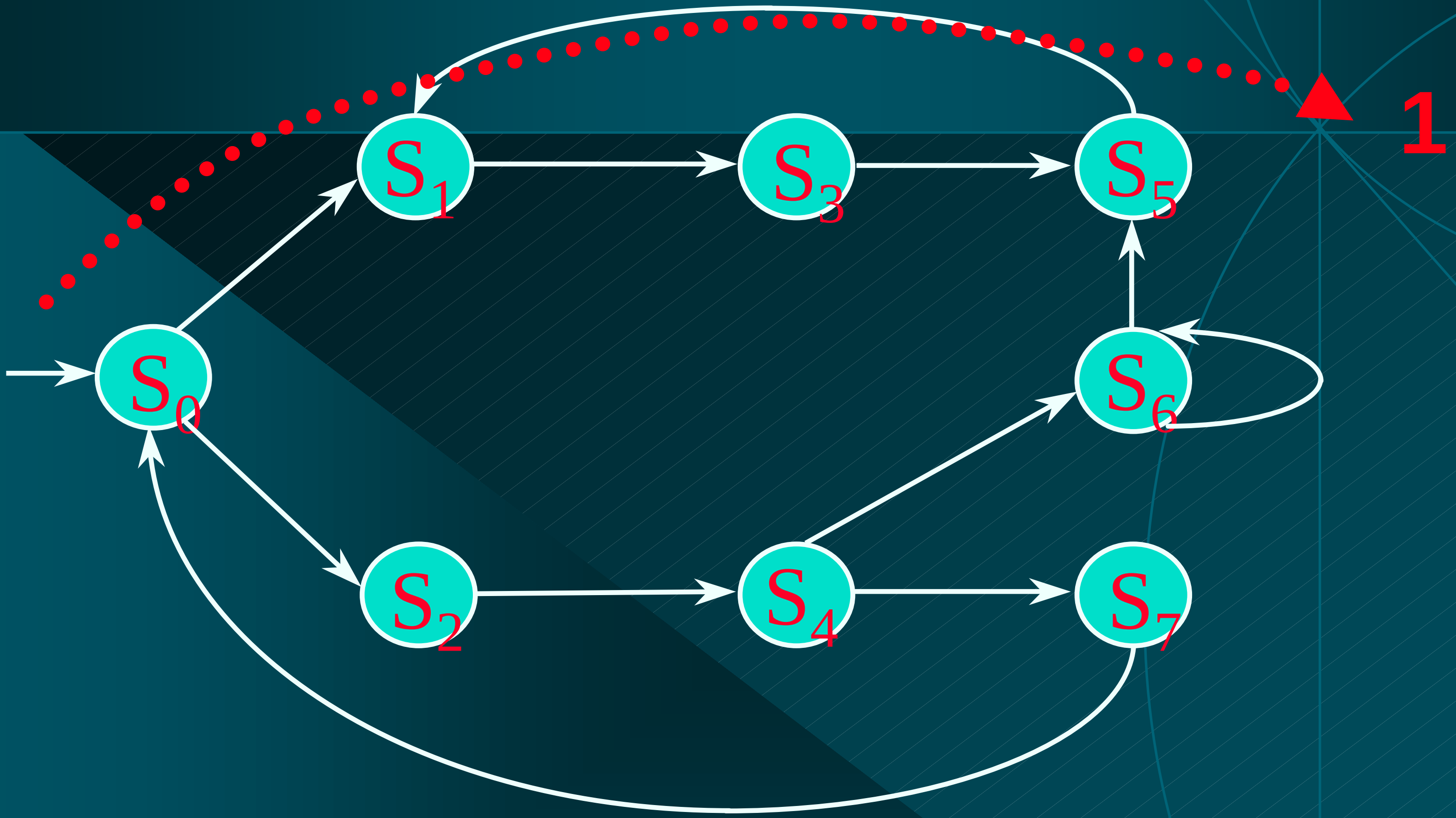


Initial

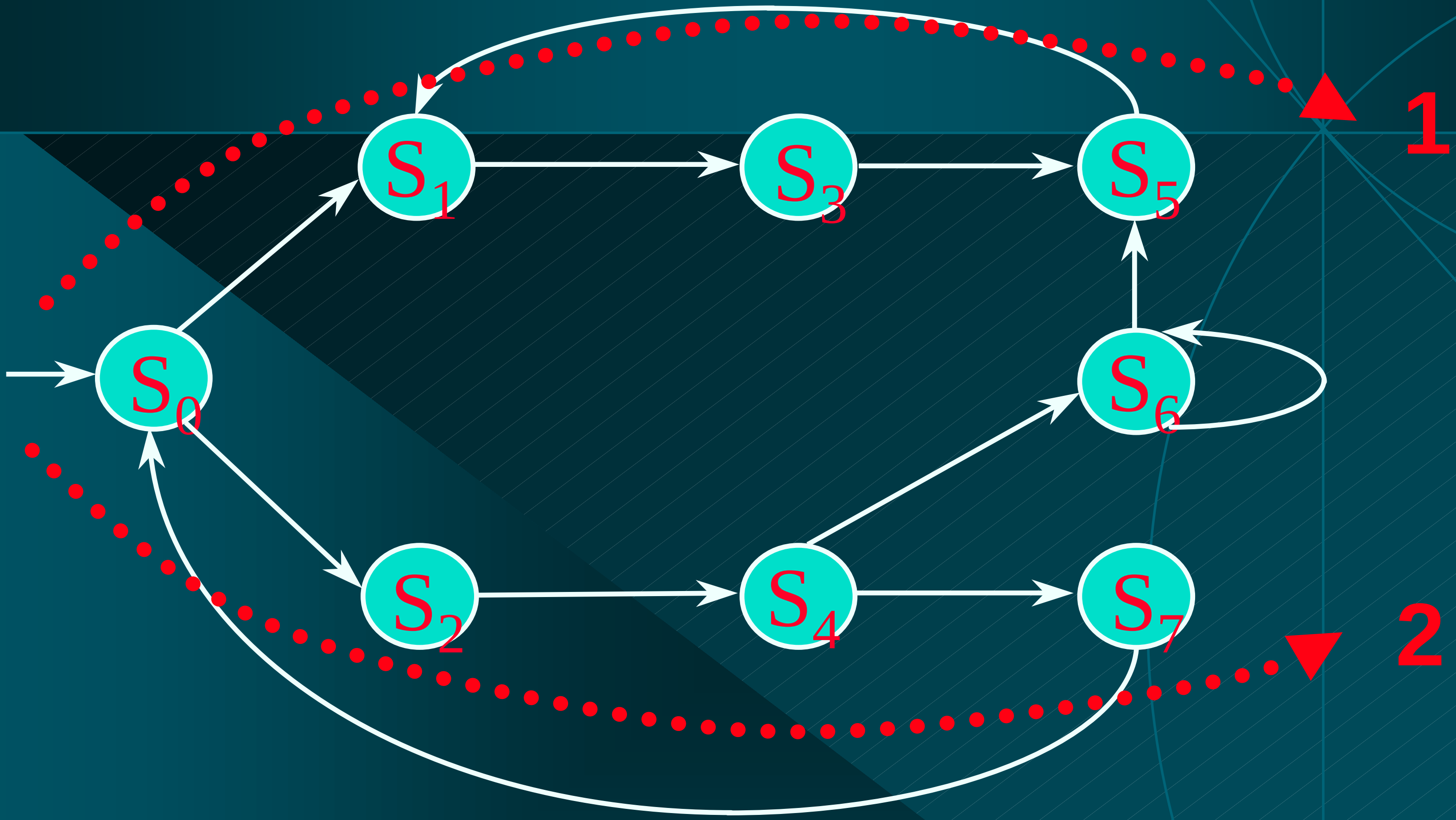




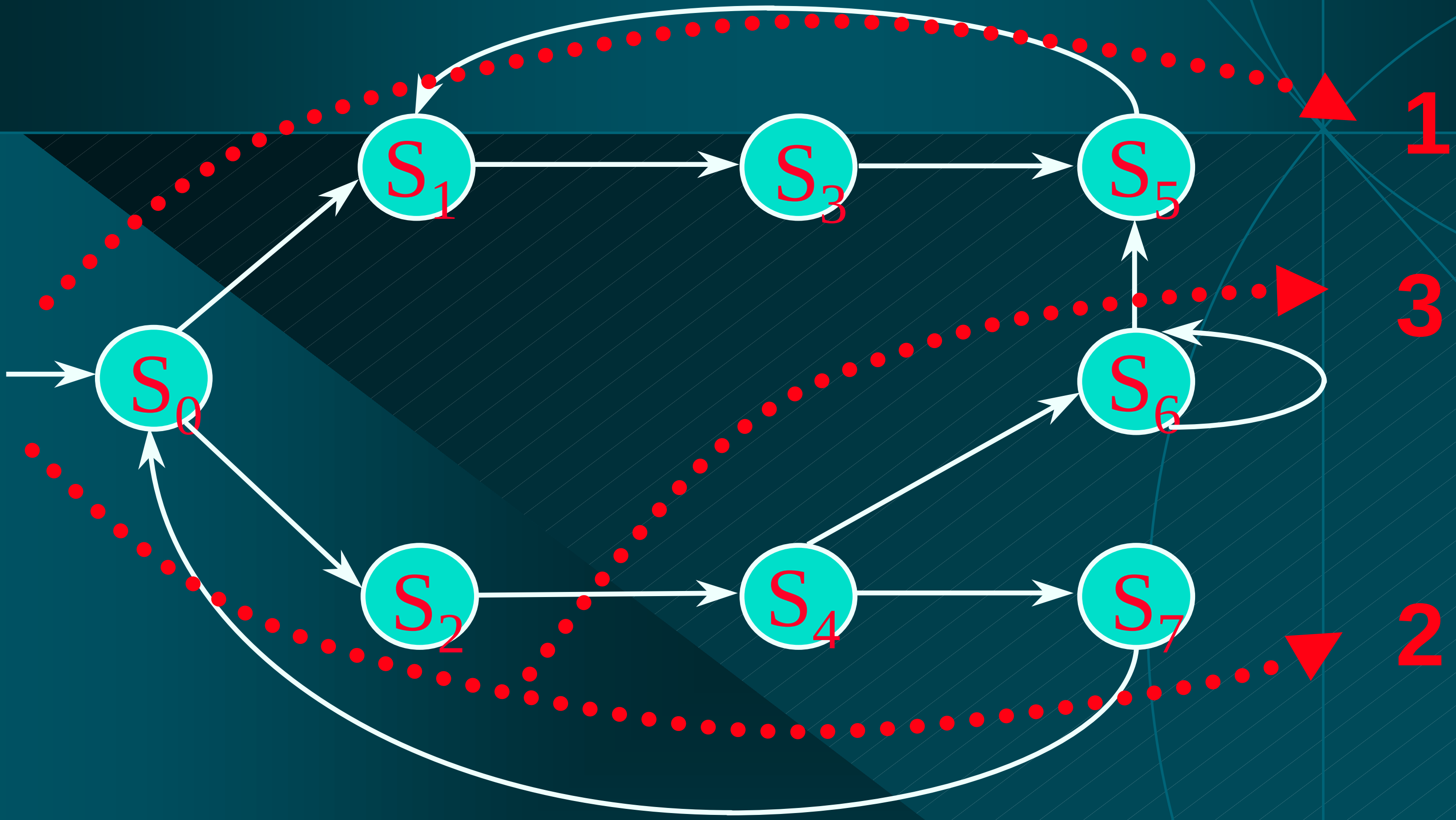
- Depth-First Visit



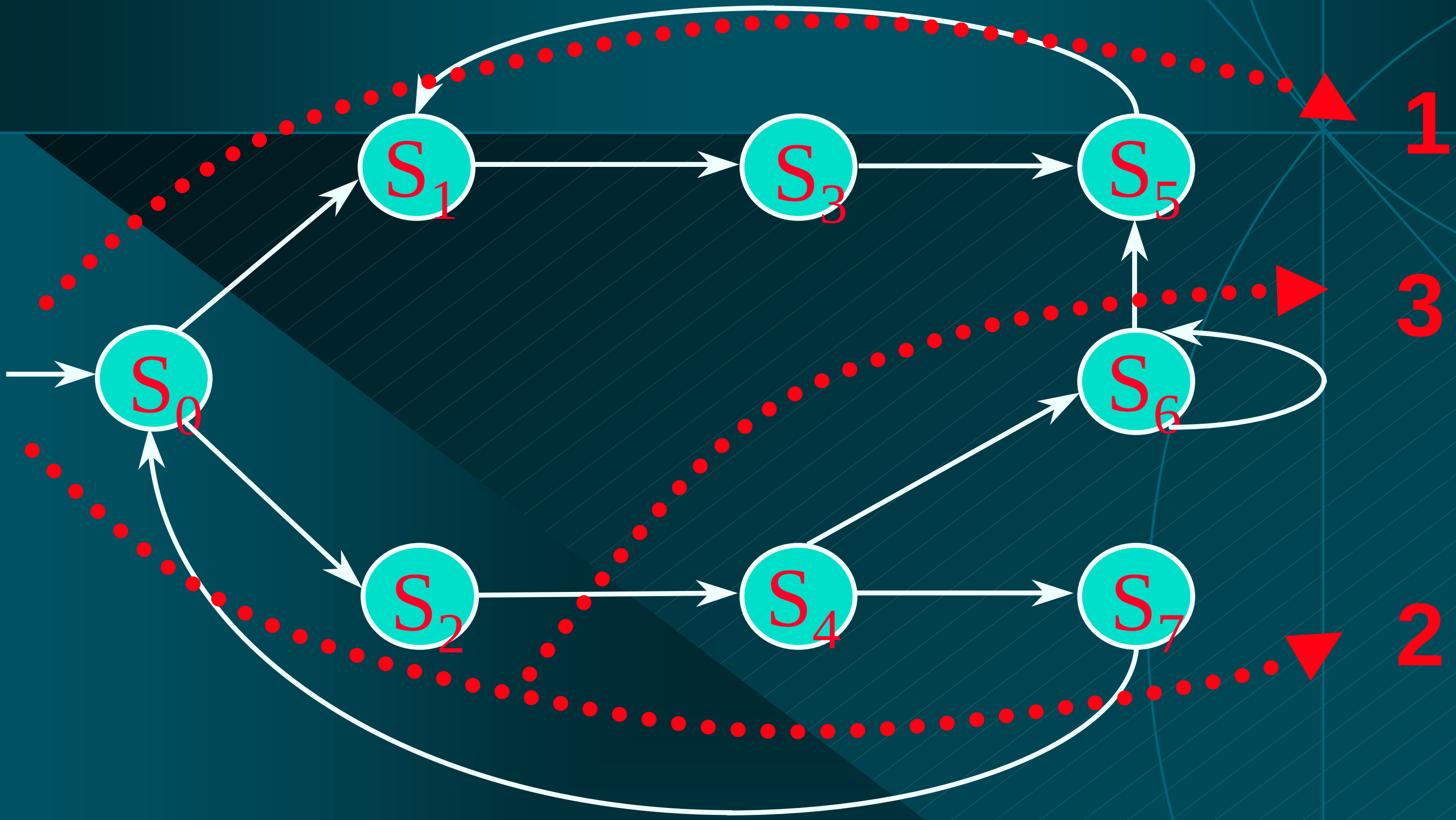
- Depth-First Visit



- Depth-First Visit

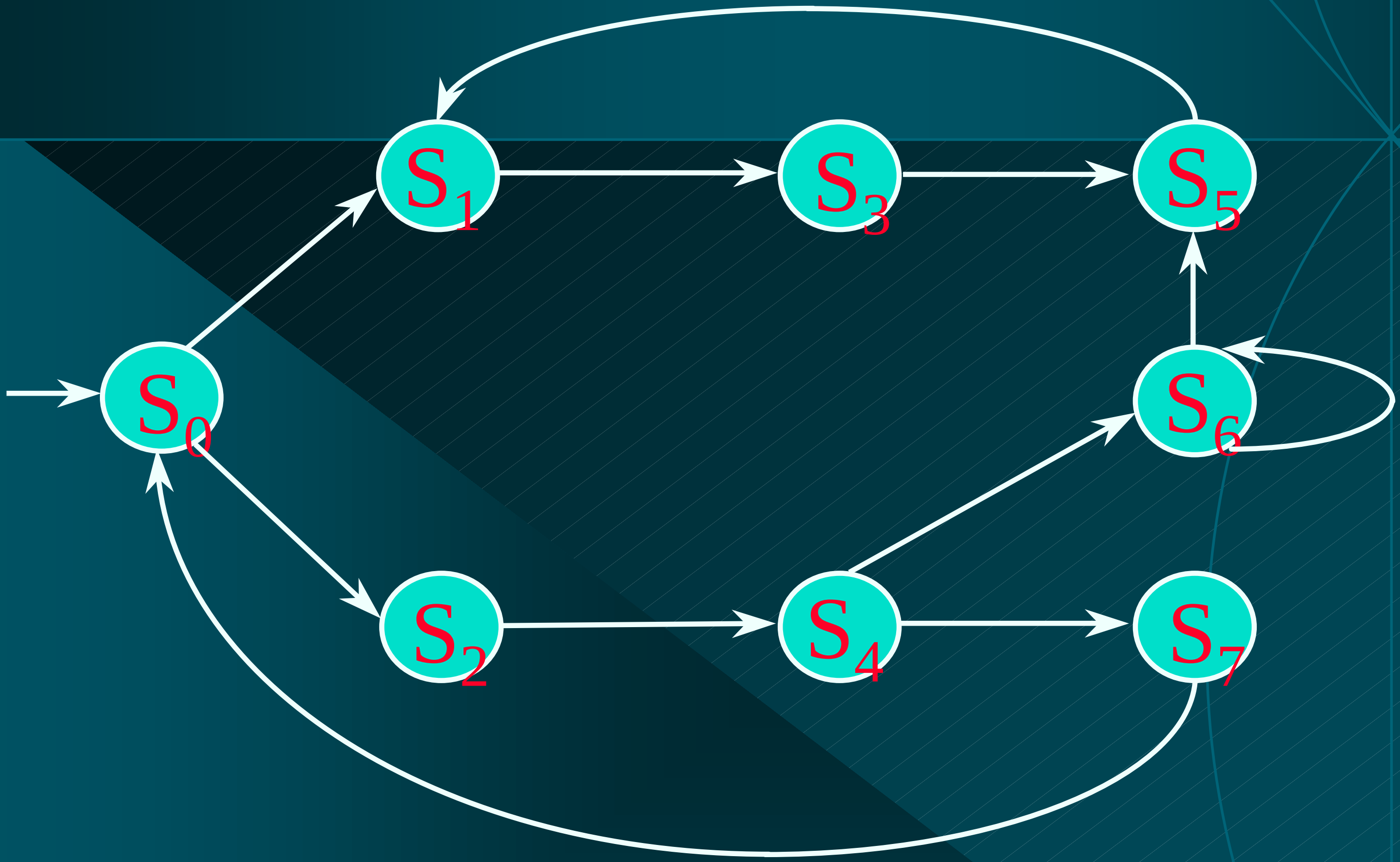


- Depth-First Visit

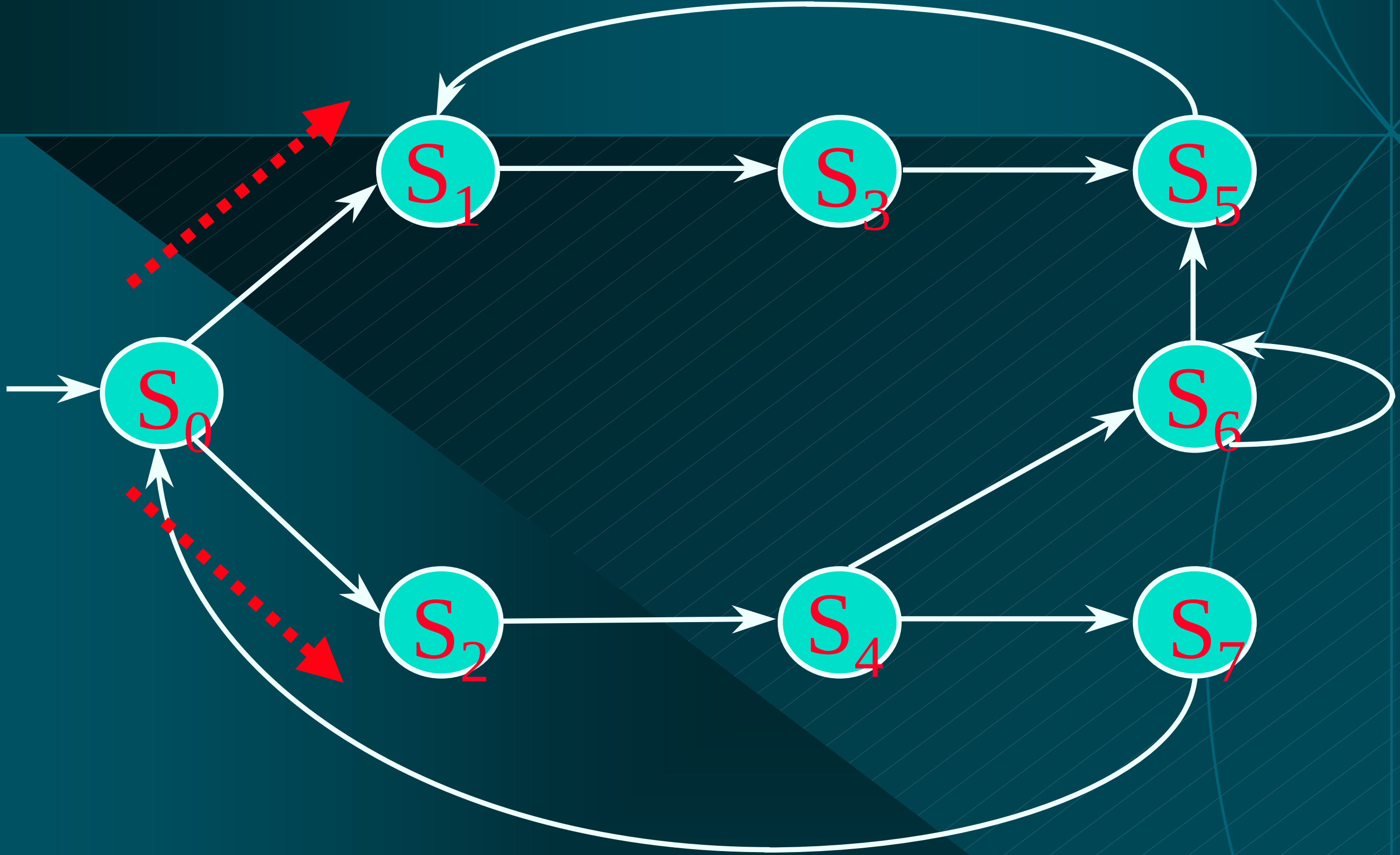


- **Depth-First Visit**

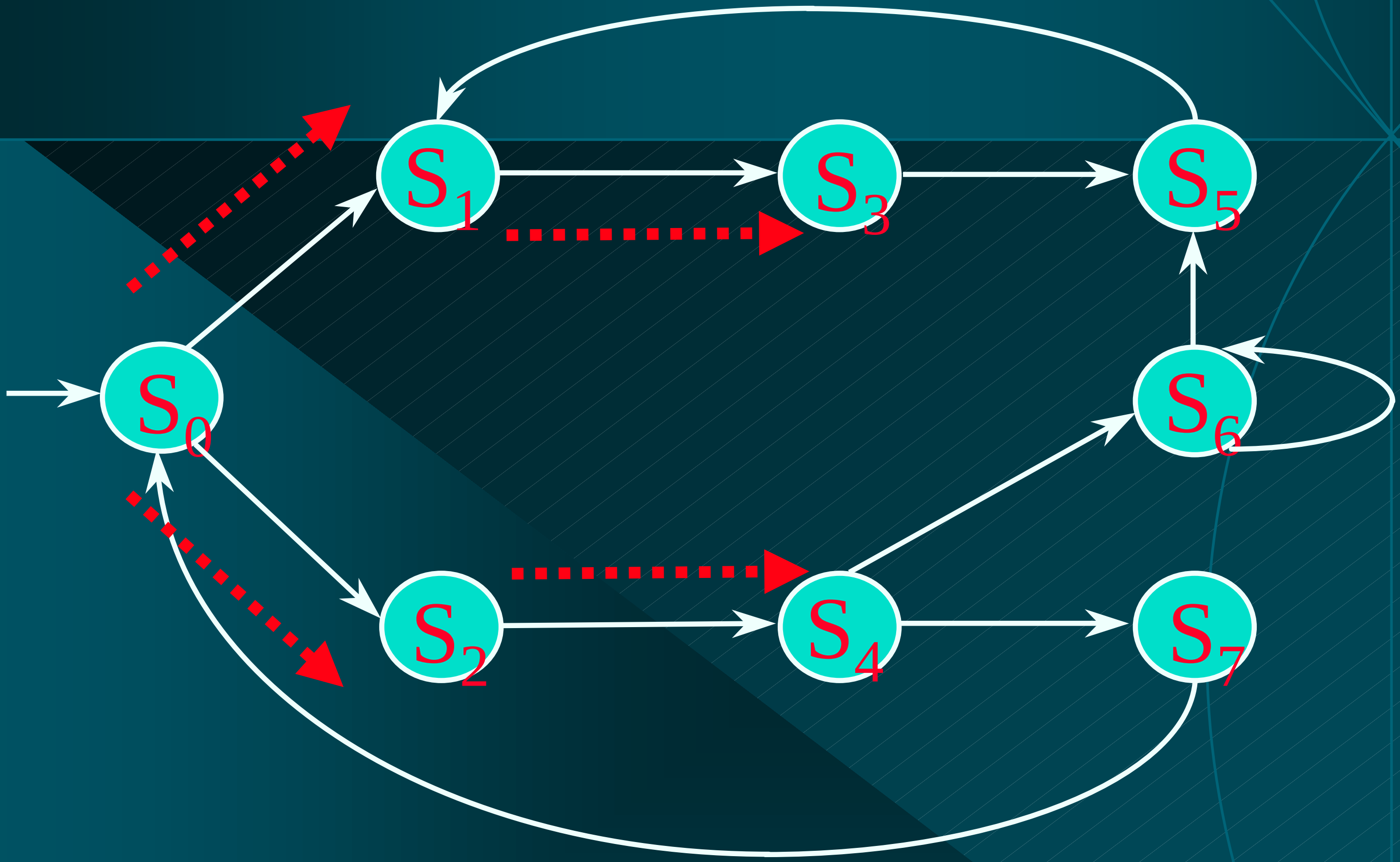
INEFFICIENT (it deals one state at a time)



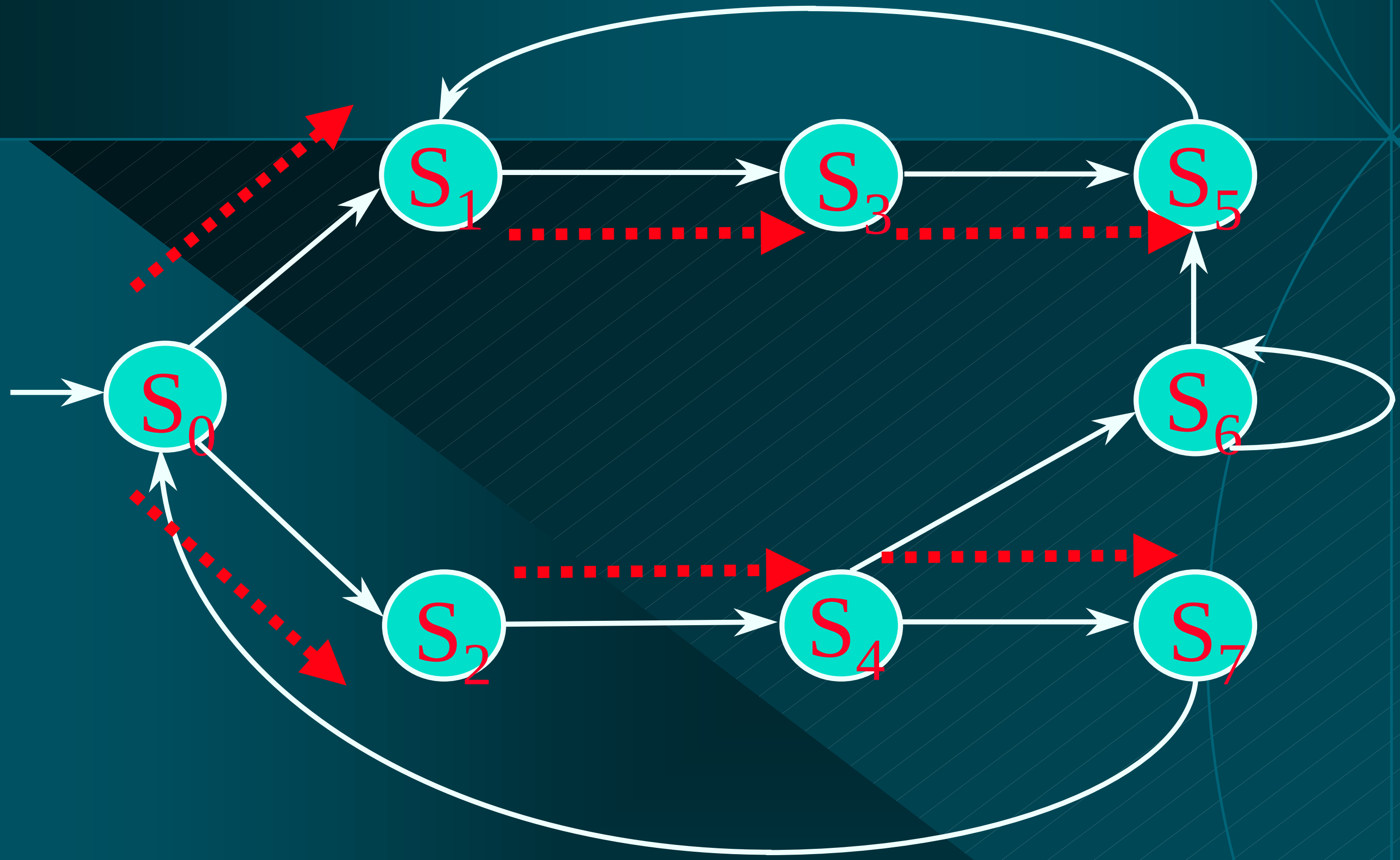
- **Breadth-First Visit**



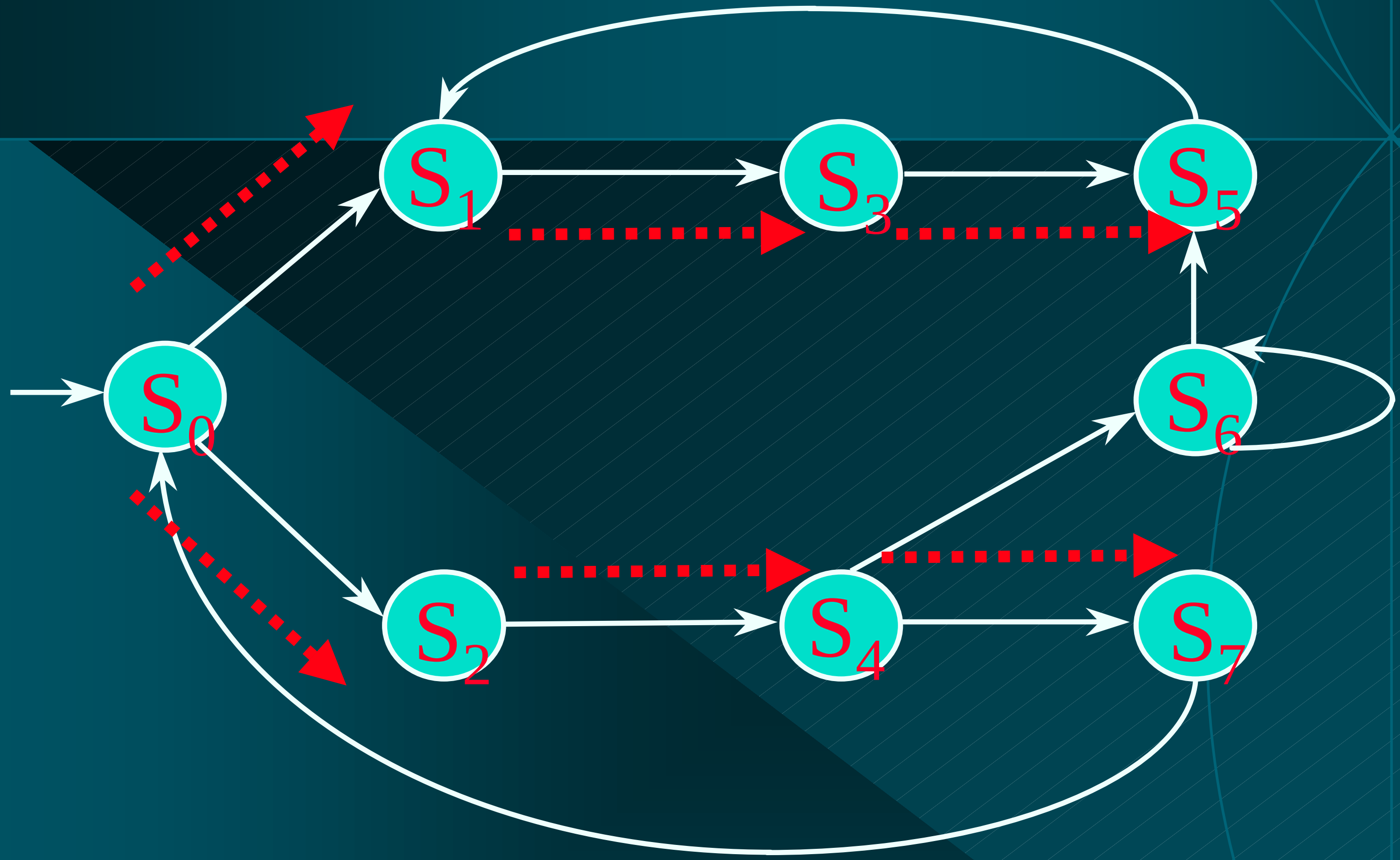
- **Breadth-First Visit**



- **Breadth-First Visit**

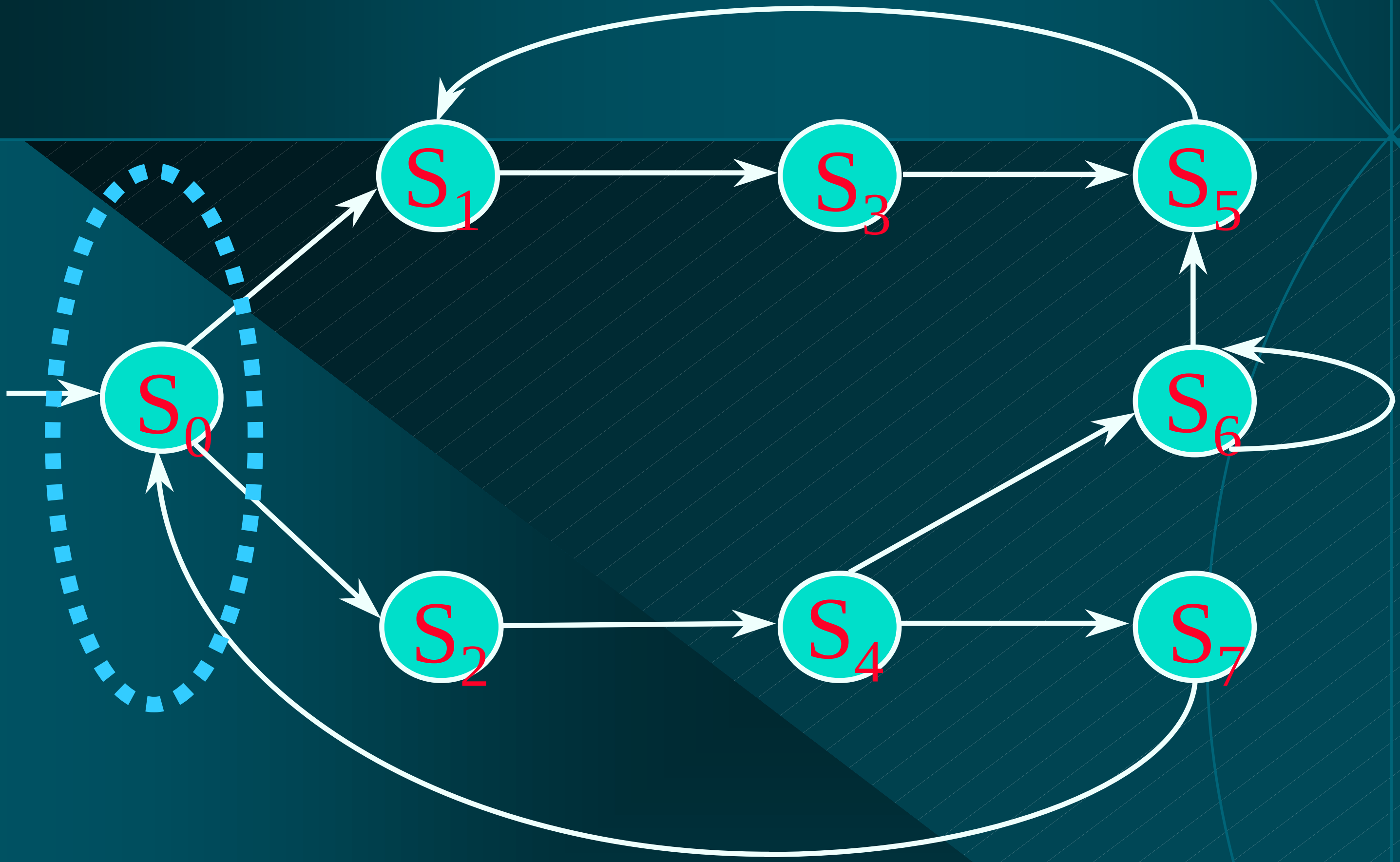


- Breadth-First Visit



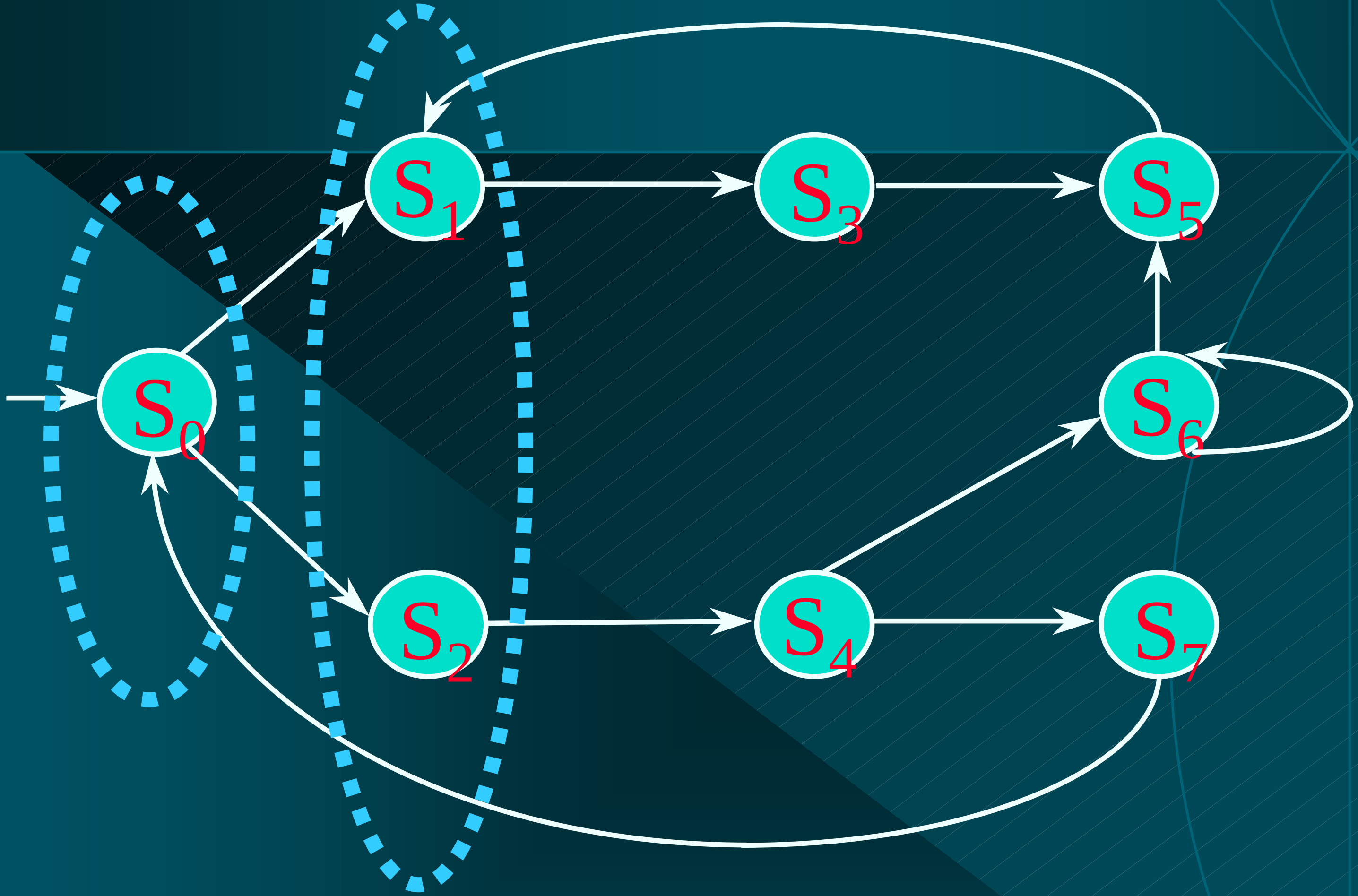
- **Breadth-First Visit**

EFFICIENT IFF we can deal with multiple states (sets of states)



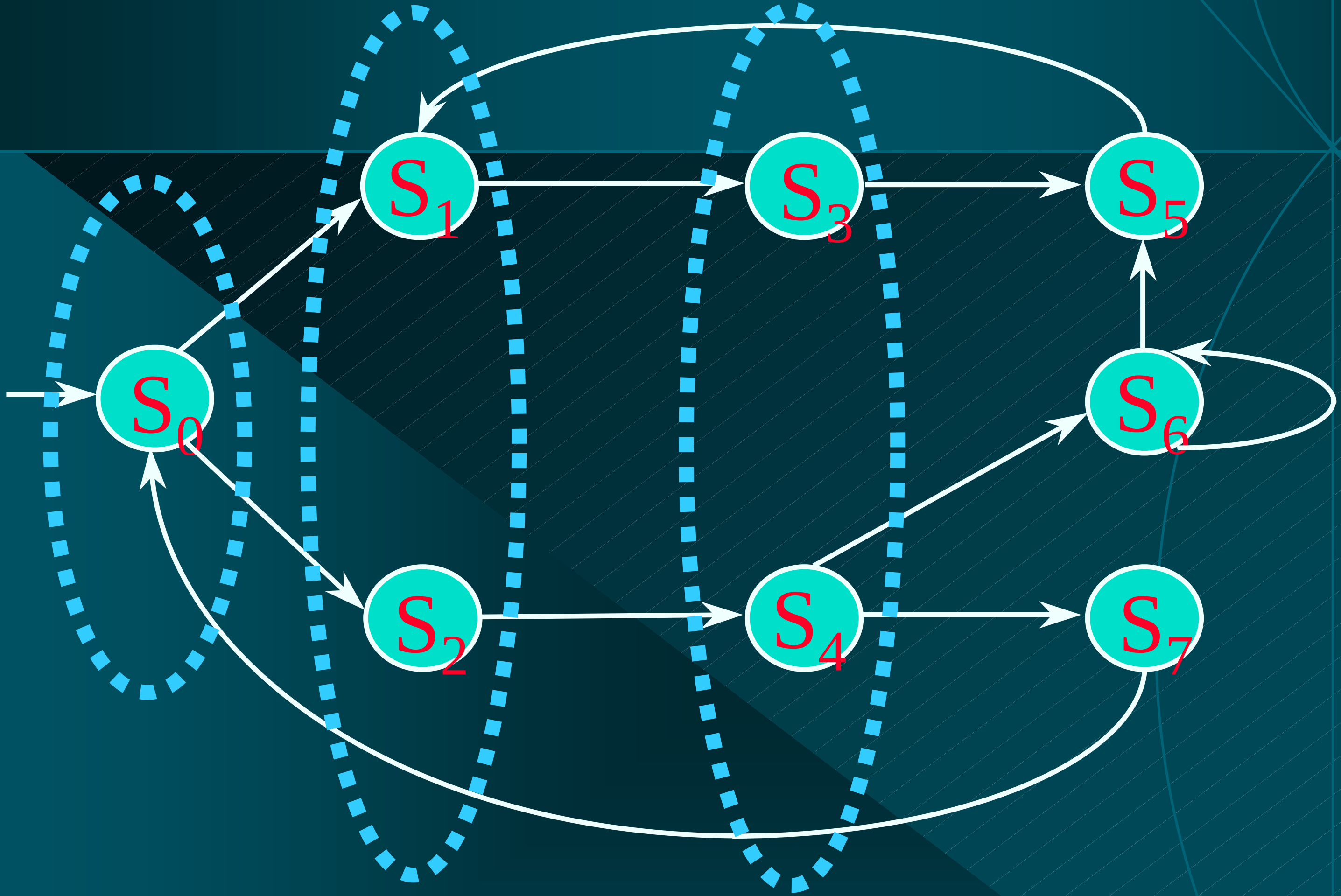
- **Breadth-First Visit**

EFFICIENT IFF we can deal with multiple states (sets of states)



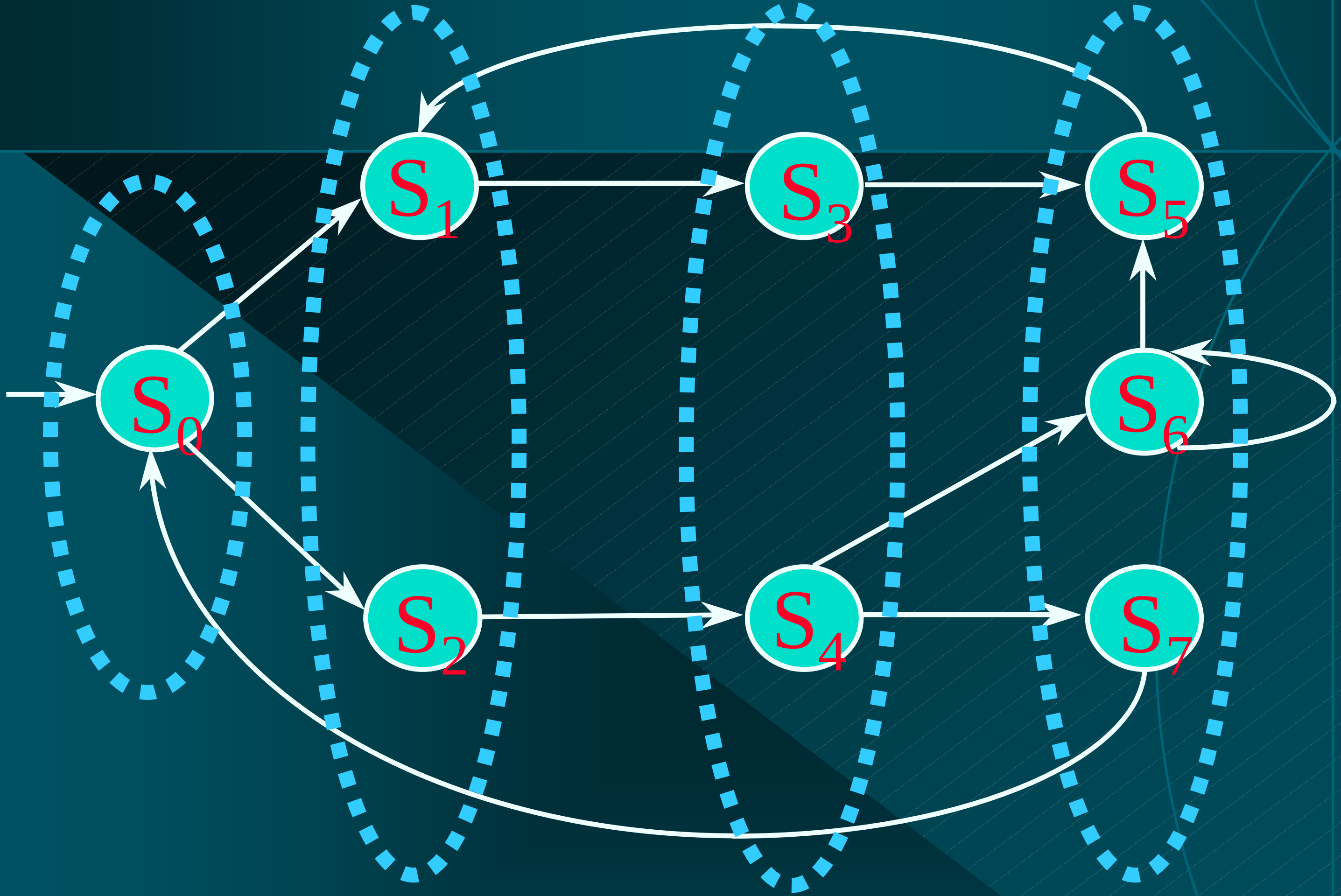
- **Breadth-First Visit**

EFFICIENT IFF we can deal with multiple states (sets of states)



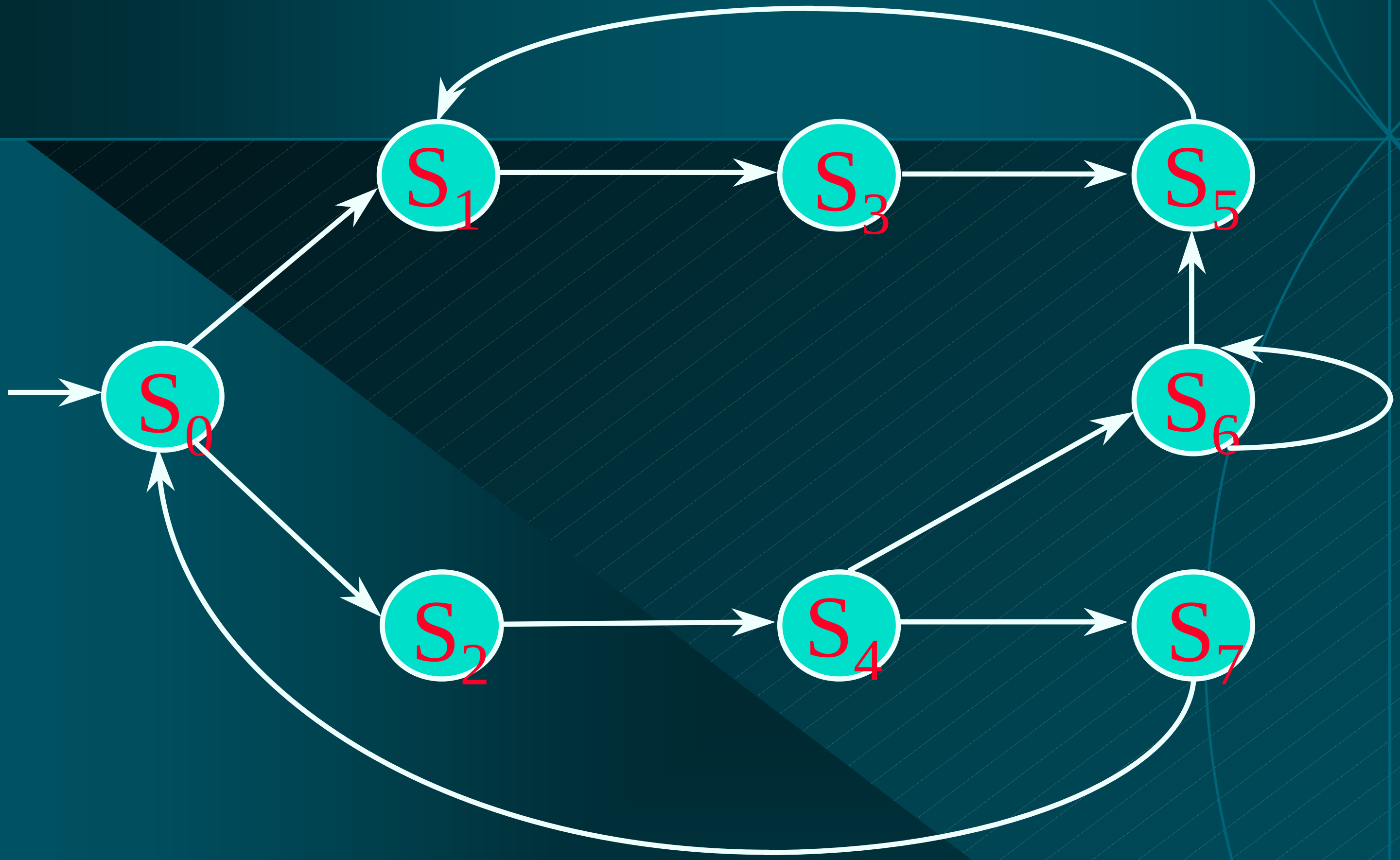
- **Breadth-First Visit**

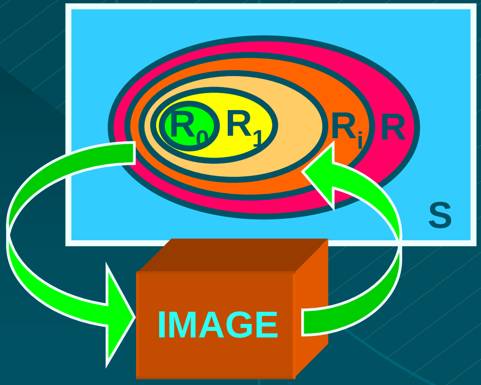
EFFICIENT IFF we can deal with multiple states (sets of states)



- **Breadth-First Visit**

EFFICIENT IFF we can deal with multiple states (sets of states)

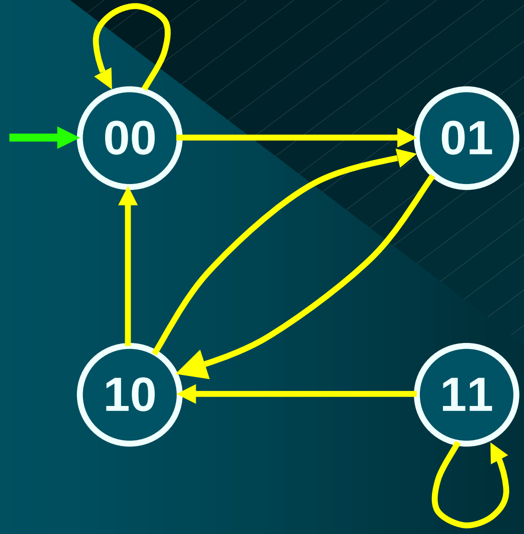

S_0
S_1
S_2
S_3
S_4
S_5
S_6
S_7

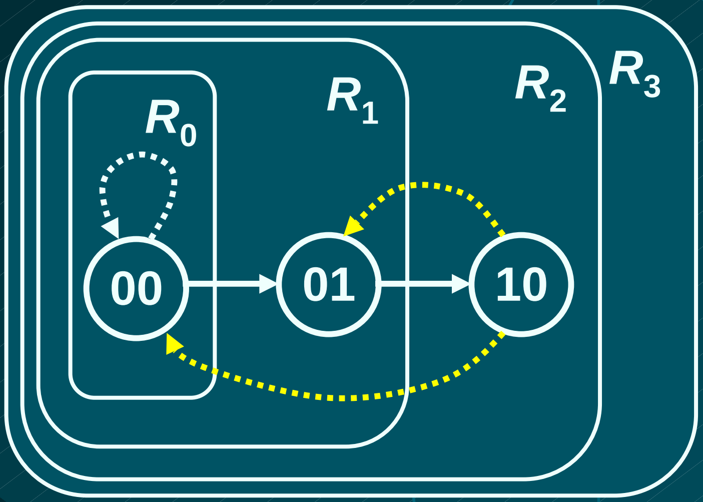

R_0
R_1
R_i
R
S
IMAGE

Representations

- Explicit reachability analysis
  - Represent states explicitly (e.g. as bit string) => limited capacity
  - Use hashtable to find quickly whether state was reached before
  - Image operation: simple simulation
  - Preimage operation: SAT run
- Symbolic reachability analysis
  - Represent states and transition relation symbolically
    - E.g. BDDs, circuits, DNF, etc.
  - Use BDD operations to perform image and preimage operation (simple AND or AND_EXIST)
  - Lots of heuristic improvements to keep BDD size under control

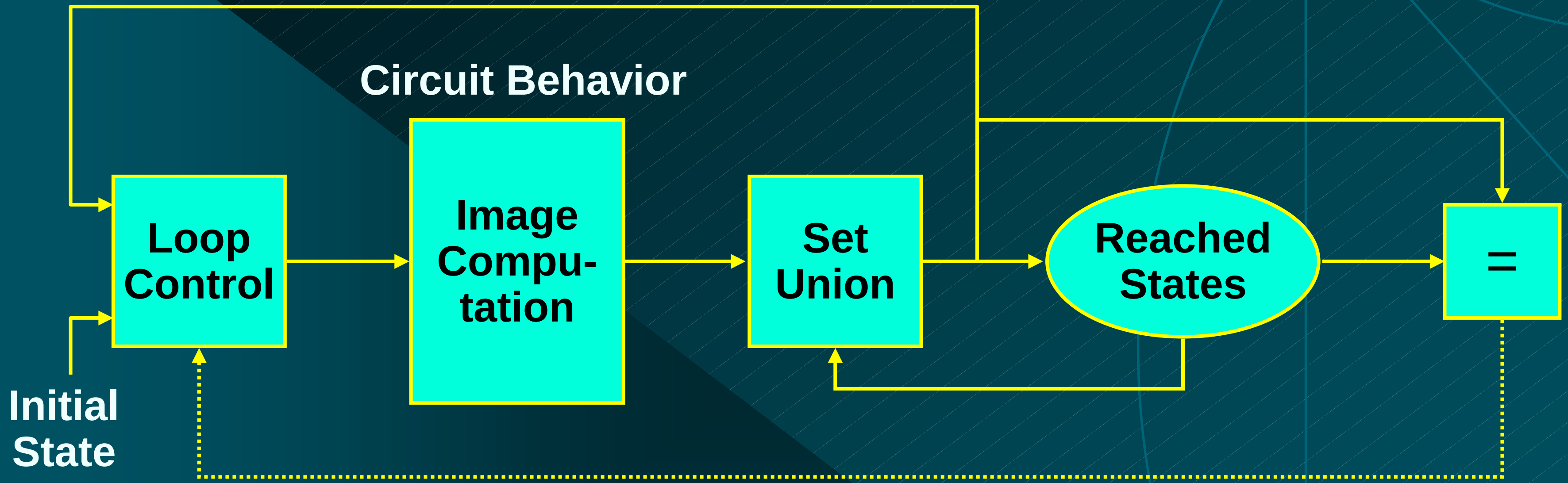
Breadth-First reachability analysis





- R_i – set of states that can be reached in i transitions
- Reach fixed point when $R_n = R_{n+1}$
  - Guaranteed since finite state

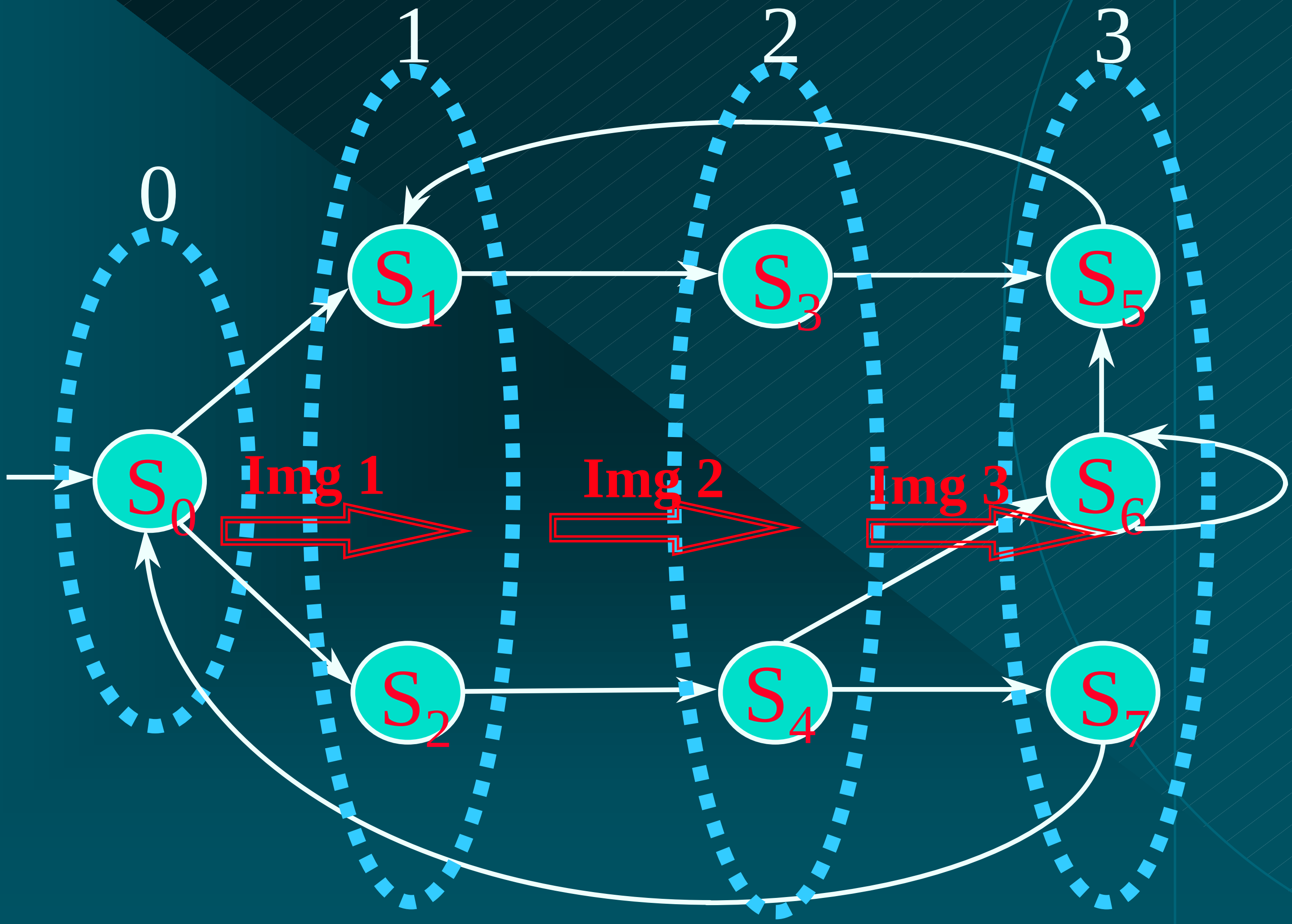
Breadth-First reachability analysis



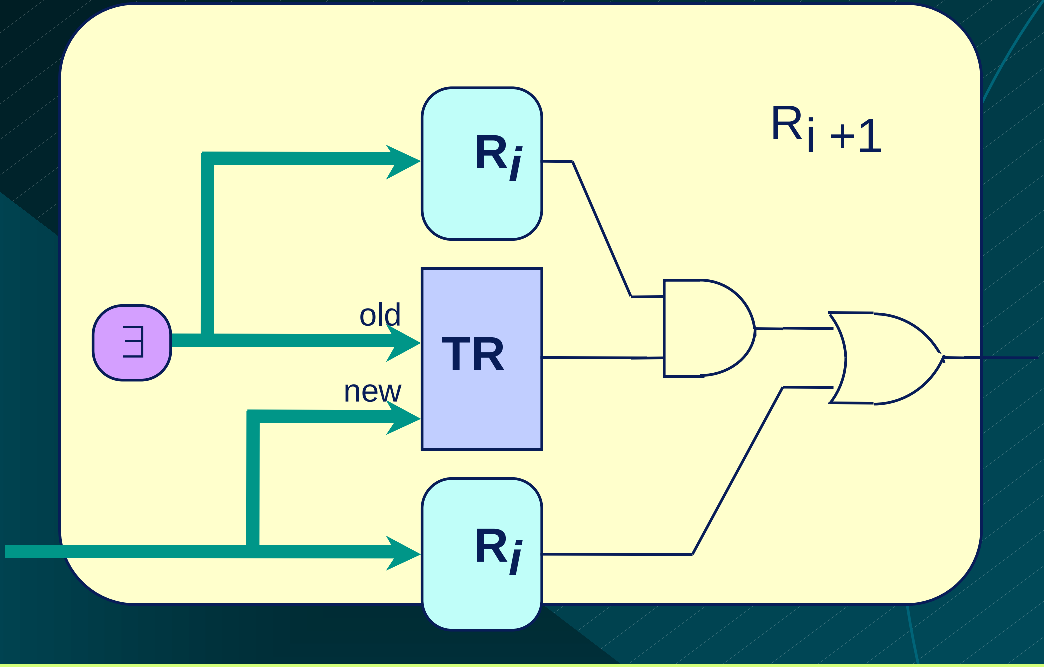
- Determine set of all reachable states of circuit
- Key step in model checking
  - Many (but not all) properties can be checked by some form of reachability computation

Forward Reachability Analysis (Forward Traversal)

Sequence of image computations … until fix-point …



Iterative computation

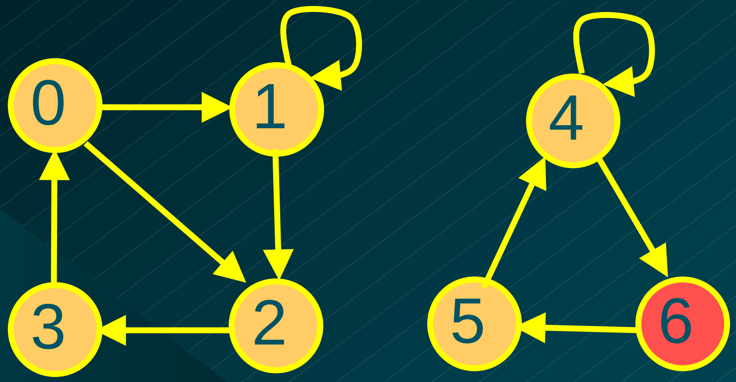


```
$R_0 = Q_0$
do
    $R_{i+1}(s) = R_i(s) \vee \exists_{s'}[R_i(s') \wedge \delta(s', s)]$
    $i \leftarrow i + 1$
until  $R_i = R_{i-1}$
```

ALGORITHM

```
FwdTraversal (TR, $S_0$)
    Reached = From = New = $S_0$ (s)
    while ( New $\neq \phi$ )
        To = Img (TR, From)
        To $|_{y \rightarrow s}$
        New = To $\wedge \neg$Reached
        Reached = Reached $\vee$ New
        From = Best_BDD (New, Reached)
    return (Reached (s))
```

- **Example**

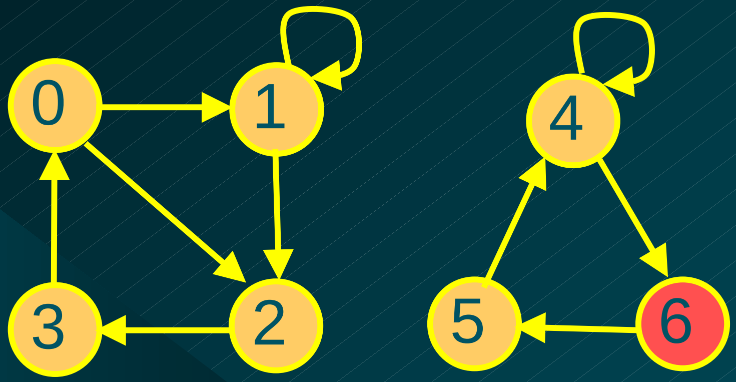


Iteration:	1	2	3
From:	{0}	{1,2}	{1,2,3}
To:	{1,2}	{1,2,3}	{0,1,2,3}
Reached:	{0}	{0,1,2}	{0,1,2,3}

Backward State Traversal

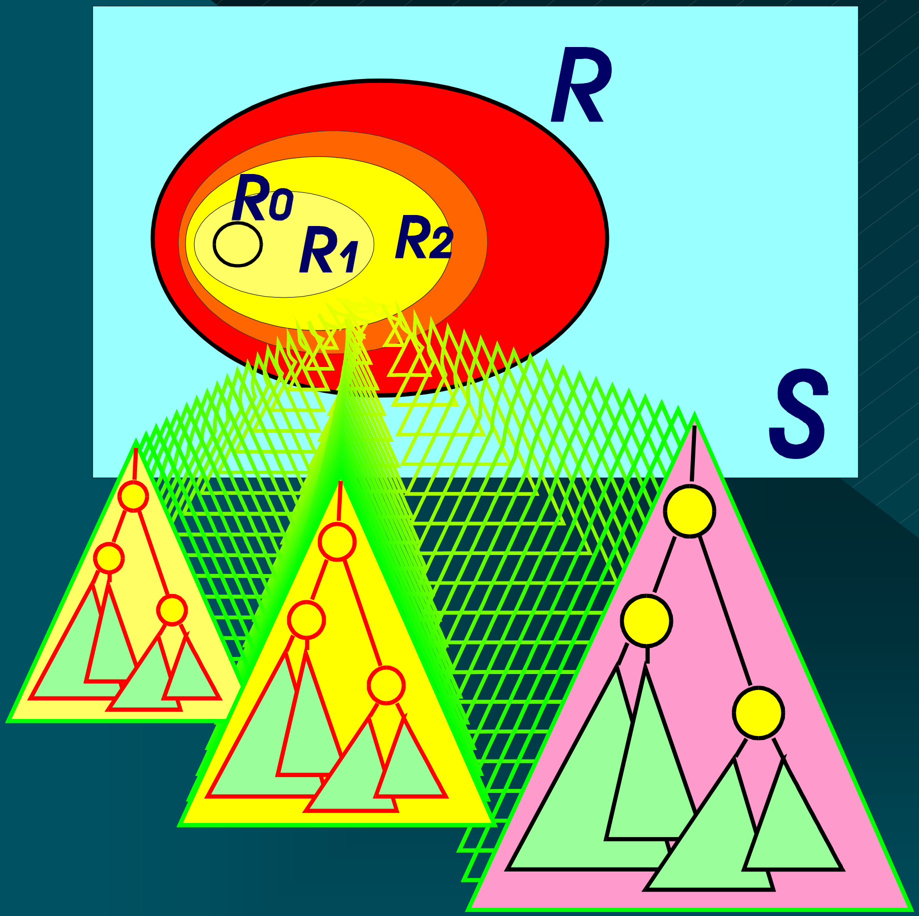
```
BwdTraversal (TR, $S_0$)
    Reached = From = New = $S_0$ (s)
    while ( New $\neq \phi$ )
        To = PreImg (TR, From)
        To | $_{y \rightarrow s}$
        New =  To $\wedge \neg$Reached
        Reached = Reached $\vee$ New
        From = Best_BDD (New, Reached)
    return (Reached (s))
```

- **Example**



Iteration:	1	2	3
From (current):	{6}	{4}	{4,5}
To (previous):	{4}	{4,5}	{4,5,6}
Reached:	{6}	{4,6}	{4,5,6}

To sum up



Forward Traversal

R_0 = *Initial State Set*

R_{i+1} = R_i + Img (TR, R_i)

Backward Traversal

R_0 = *Initial State Set*

R_{i+1} = R_i + PreImg (TR, R_i)

Image and inverse image

$$\mathrm{Img}\,(f, X) = f\,(X) = \{\, y \in B^m \mid x \in X \wedge y = f\,(x) \,\}$$

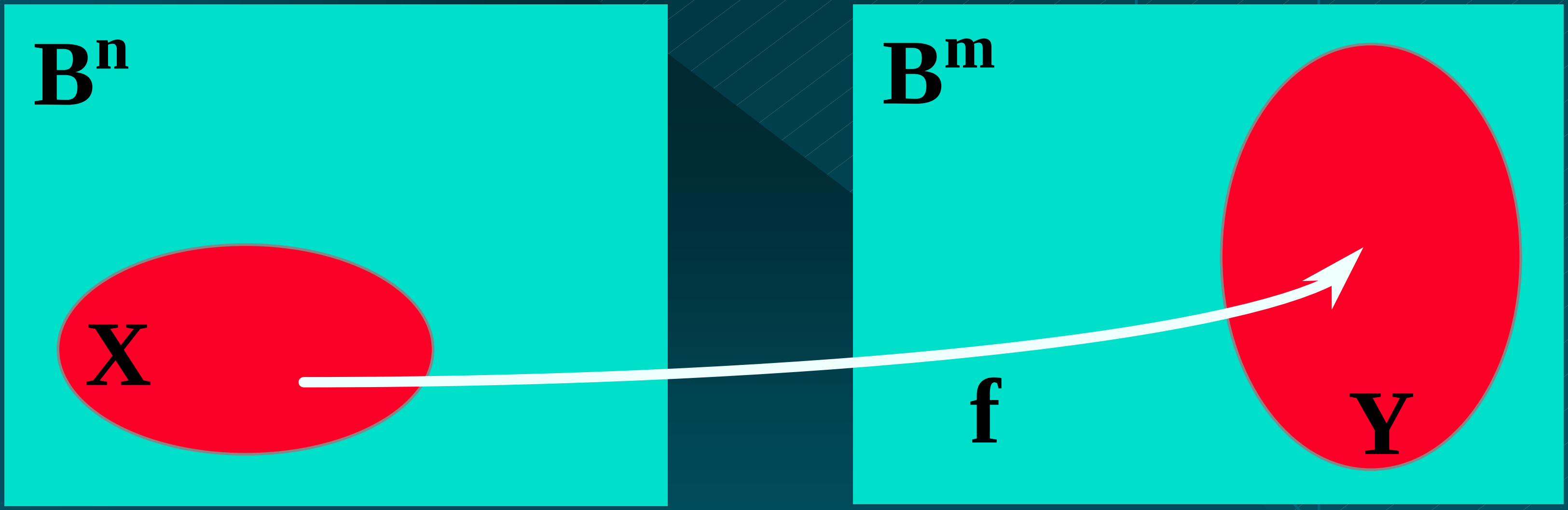


Image and inverse image

$$\mathrm{Img}\,(f, X) = f\,(X) = \{\, y \in B^m \mid x \in X \wedge y = f\,(x) \,\}$$

$$\mathrm{PreImg}\,(f, Y) = f^{-1}\,(Y) = \{\, x \in B^n \mid y \in Y \wedge y = f\,(x) \,\}$$

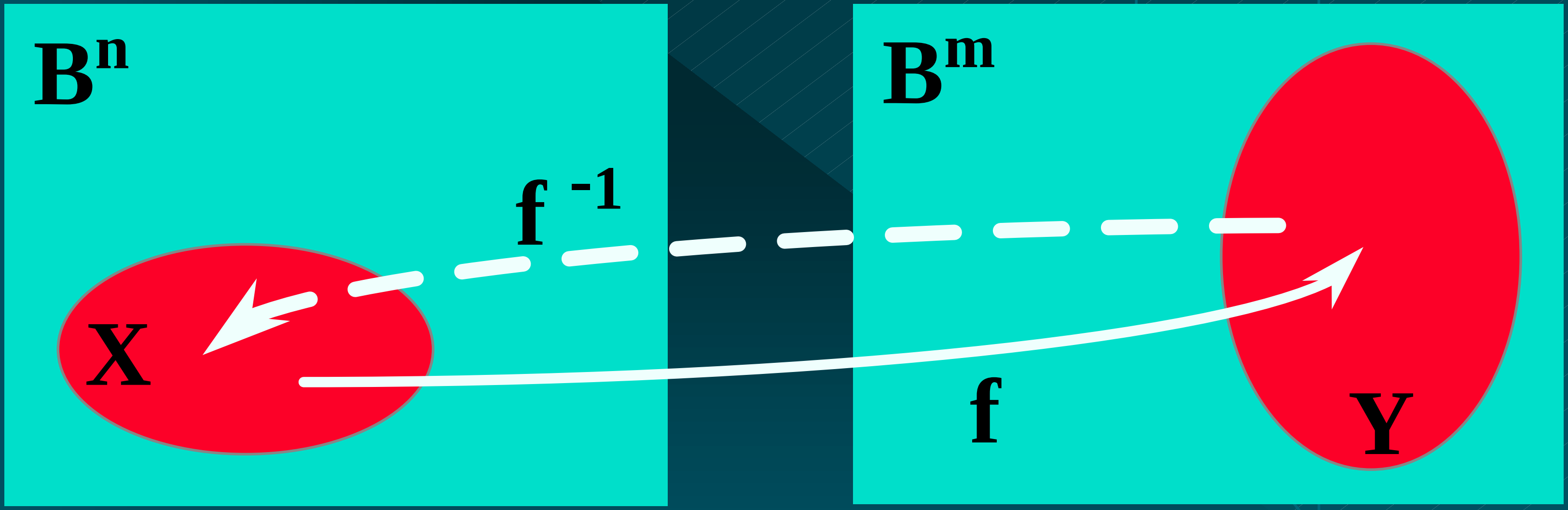
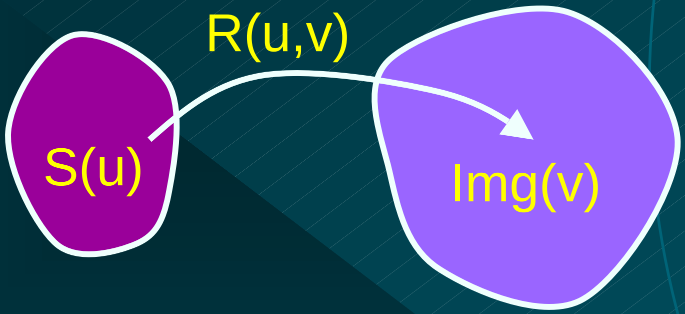


Image Computation

- **Computing *set* of next states from a given initial state (or set of states)**

$$\mathbf{Img(\ S,R\ ) = \exists_u\ S(u) \bullet R(u,v)}$$



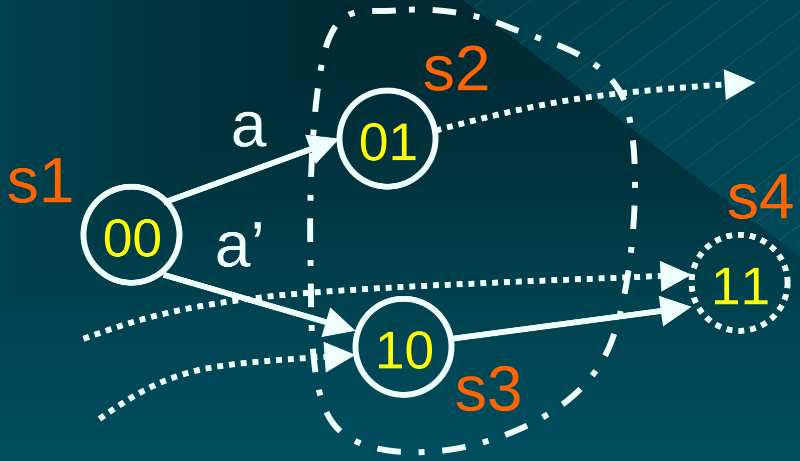
- FSM: when transitions are labeled with input predicates *x,* quantify w.r.to all inputs (primary inputs and state var)

$$Img(\ S,R\ ) = \exists_{\mathbf{x}} \exists_u\ S(u) \bullet R(x,u,v)$$

Image Computation - example

Compute a *set* of next states from state s1

- **Encode the states: s1=00, s2=01, s3=10, s4=11**
- **Write transition relations for the *encoded* states: *R* = (*ax’y’X’Y* + *a’x’y’XY’* + *xy’XY* + ….)**



a	xy	XY
1	00	01
0	00	10
-	10	11
....		

Example - cont'd

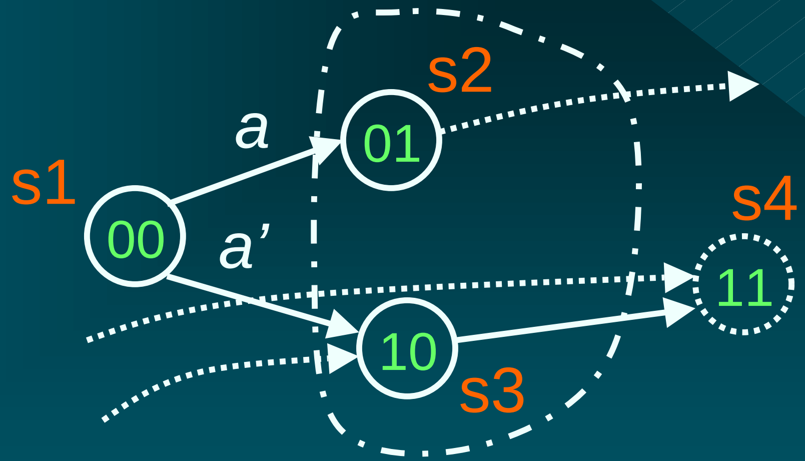
- **Compute Image from s1 under R**

$$\mathbf{Img(\,s1,R\,)} = \exists_a \exists_{xy}\, \mathbf{s1}(x,y) \bullet \mathbf{R}(a,x,y,X,Y)$$

$$= \exists_a \exists_{xy} (x'y') \bullet (ax'y'X'Y + a'x'y'XY' + xy'XY + \ldots .)$$

$$= \exists_{axy} (ax'y'X'Y + a'x'y'XY') = (X'Y + XY')$$

$$= \{01, 10\} = \{s2, s3\}$$



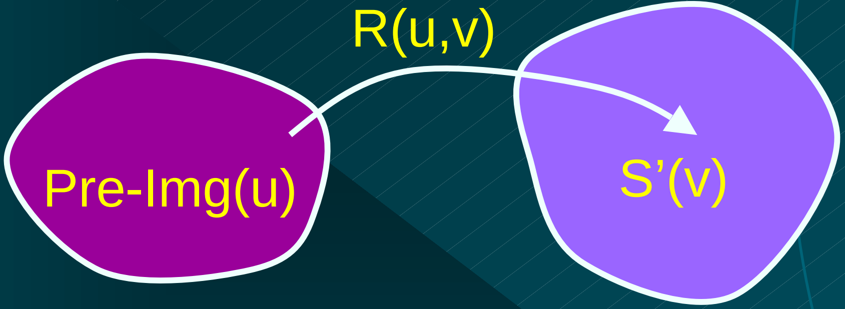
Result: a set of next states for *all* inputs

s1 → {s2, s3}

Pre-Image Computation

- **Computing a *set* of present states from a given next state (or set of states)**

$$\textbf{Pre-Img(S',R) = } \exists_v \textbf{ R(u,v))• S'(v)}$$



- Similar to Image computation, except that quantification is done w.r.to *next state* variables
- The result: a set of states *backward* reachable from state set S', expressed in present state variables *u*
- Useful in computing CTL formulas: AF, EF

Existential Quantification

- **Existential quantification (abstraction)**

$$\exists_x f = f|_{x=0} + f|_{x=1}$$

- **Example:**

$$\exists_x (x\, y + z) = y + z$$

- **Note: $\exists_x f$ does not depend on *x* (smoothing)**

- **Useful in symbolic image computation (*sets* of states)**

Existential Quantification - cont'd

- Function can be existentially quantified w.r.to a vector: $X = x_1 x_2 \ldots$

$$\exists_X f = \exists_{x1x2\ldots} f = \exists_{x1} \exists_{x2} \exists_{\ldots} f$$

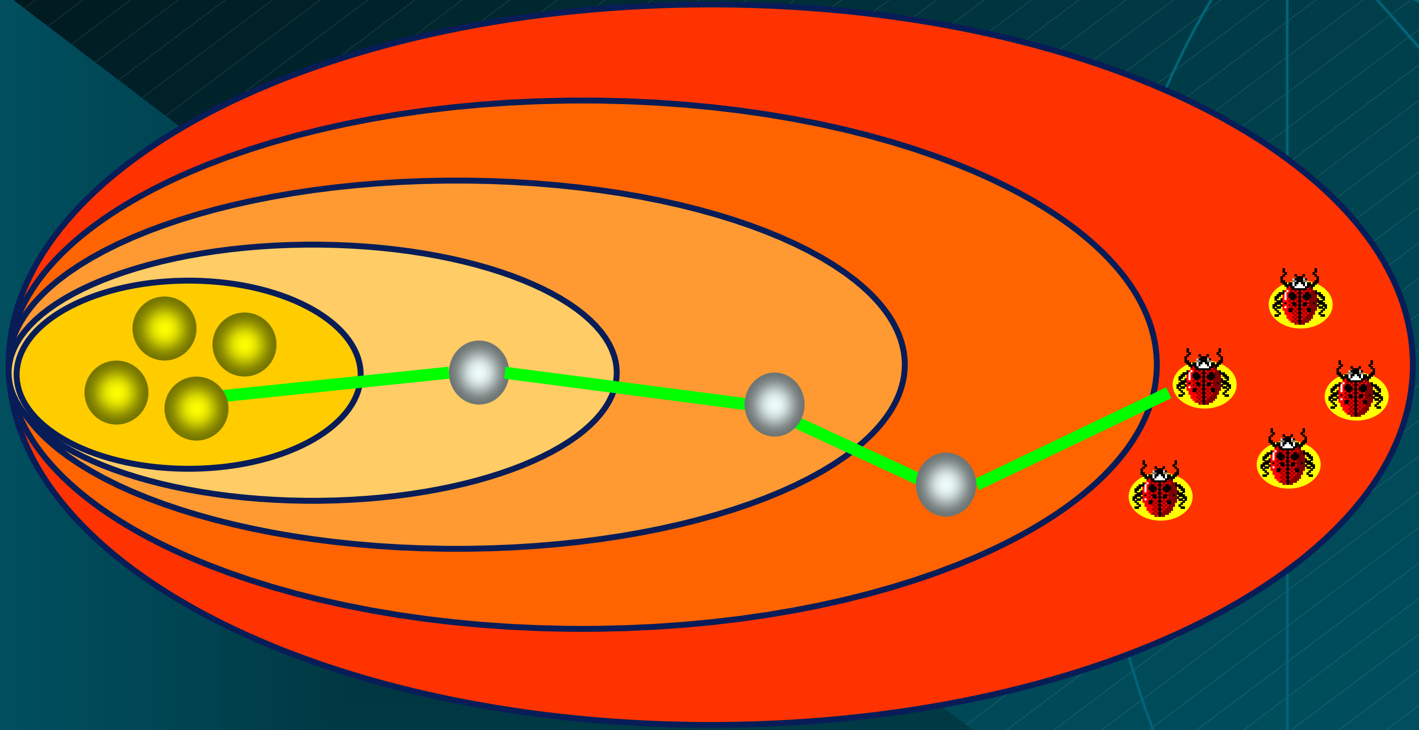
- Can be done efficiently directly on a BDD
- Very useful in computing *sets* of states
  - Image computation: *next* states
  - Pre-Image computation: *previous* states

from a given *set* of initial states

State Traversal Techniques

- Forward Traversal
  - Start from initial state(s)
  - Traverse forward to check whether “bad”
  - State(s) is reachable
- Backward Traversal
  - Start from bad state(s)
  - Traverse backward to check whether intial
  - State(s) can reach them
- Combines Forward/Backward traversal
  - compute over-approximation of reachable
  - states by forward traversal
  - for all bad states in over-approximation, start backward traversal to see whether intial state can reach them

BFS traversal (forward)

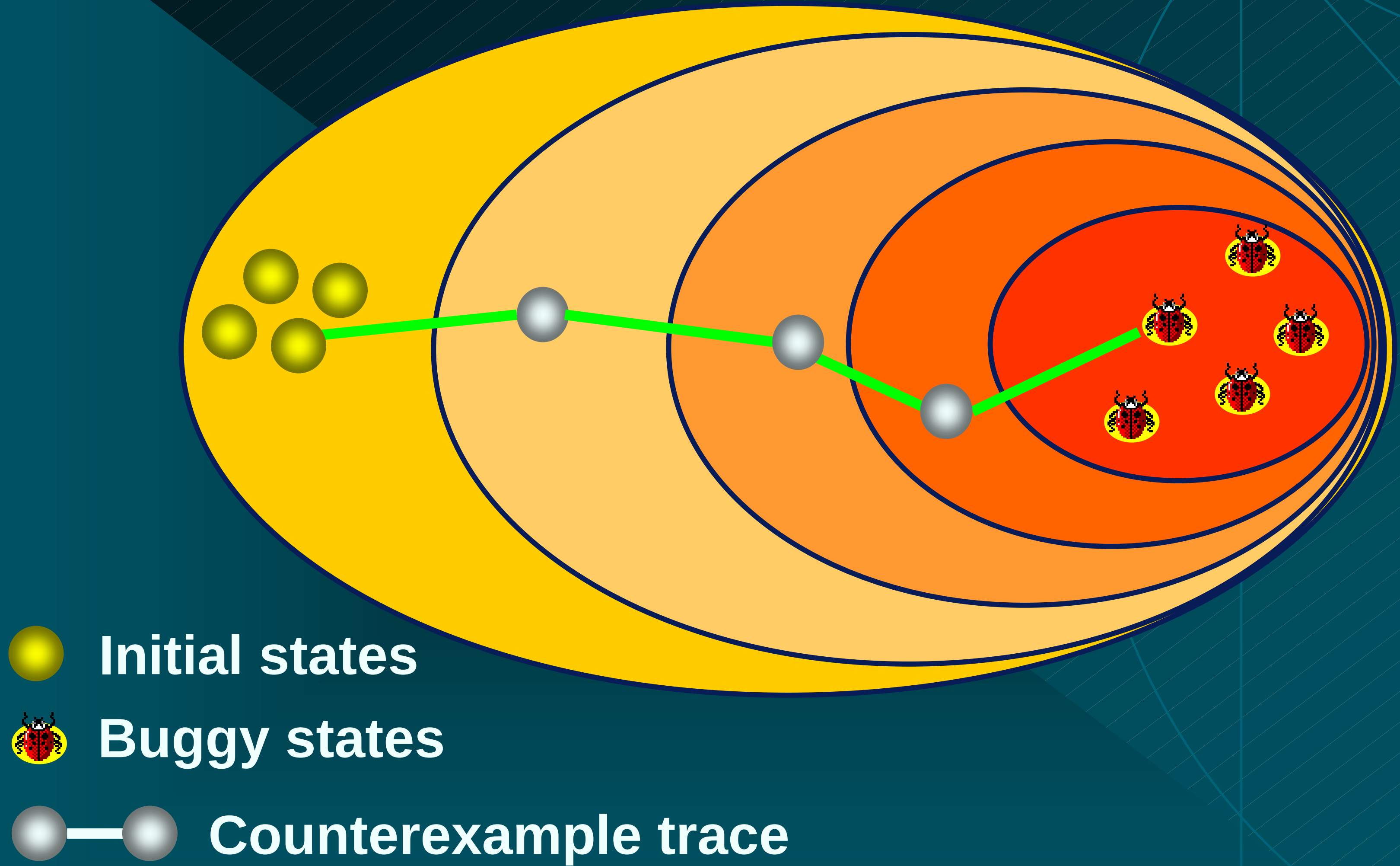


● Initial states

🐞 Buggy states

●—● Counterexample trace

BFS traversal (backward)



● Initial states

🐞 Buggy states

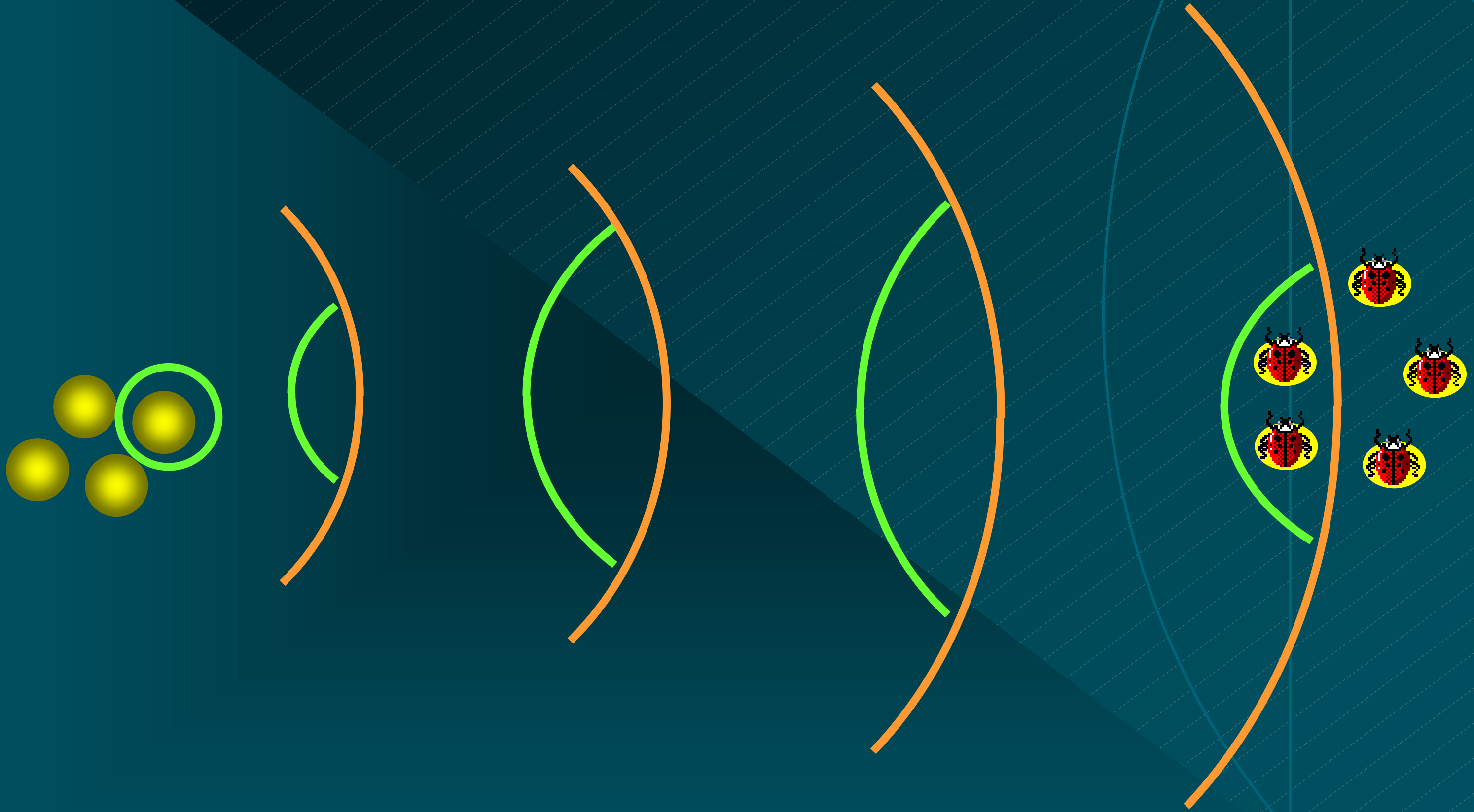
○—○ Counterexample trace

BFV ModelCheck (invariant check)

```
BfvMC (TR, S, T)
    Reached = from = new = S
    for (i=0; new $\neq \phi$ ; i++)
        if (from $\cap$ T $\neq \phi$)
          return (counterEx (TR, frontier, T))
        to = Img (TR, from)
        new =  $\text{frontier}_i$ = to $\cap \neg$ Reached
        Reached = Reached $\cup$ new
        from = new
    return (OK)
```

Forward-Backward BMC

- BFV focused/guided by combination of forward and backward approx/exact traversals



FSM Analysis Impact

- Systems Represented as Finite State Machines
  - Sequential circuits
  - Communication protocols
  - Synchronization programs
- Analysis Tasks
  - State reachability
  - State machine comparison
  - Temporal logic model checking
- Traditional Methods Impractical for Large Machines
  - Polynomial in number of states
  - Number of states exponential in number of state variables
  - Example: single 32-bit register has 4,294,967,296 states!

BDD-based MC: Current status

- Symbolic model checkers can analyze sequential circuits with ~200- 400 flip flops
  - For specific circuit types, larger state spaces have been analyzed
- Challenges
  - Memory/runtime bottlenecks
  - Adoption of TLs for property specification
- Frontier constantly being pushed
  - Abstraction & approximation techniques
  - Symmetry reduction
  - Compositional reasoning
  - Advances in BDD technology …

Performance bottleneck: Memory blow-up within Image

$$\text{Img}(TR, From) = \exists_{s,x} [TR(s, x, y) \cdot From(s)]$$

Image is computed through:

a conjunction-abstraction operation between present state set and transition relation.

Generally existential quantification reduces BDD size. BUT BDD can blow-up while computing:

$TR(s,x,y) = \Pi_i(y_i \equiv \delta_i (s,x))$

$TR (s, x, y) \cdot From(s)$

Image Computation

$$\mathrm{Img}\ (TR, From) = \exists_{s,x}\ [TR\ (s, x, y) \cdot From(s)]$$

From(s)

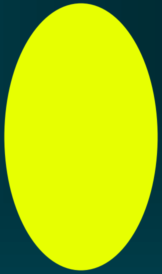


Image Computation

$$\text{Img (TR, From)} = \exists_{s,x}\ [\text{TR}\ (s, x, y) \cdot \text{From}(s)]$$

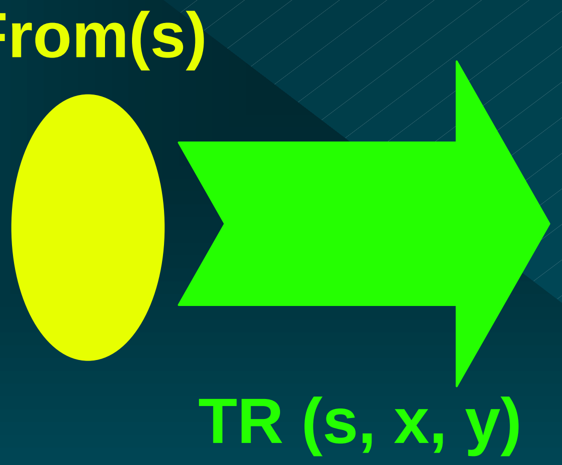


Image Computation

$$\text{Img}\ (TR, From) = \exists_{s,x}\ [TR\ (s, x, y) \cdot From(s)]$$

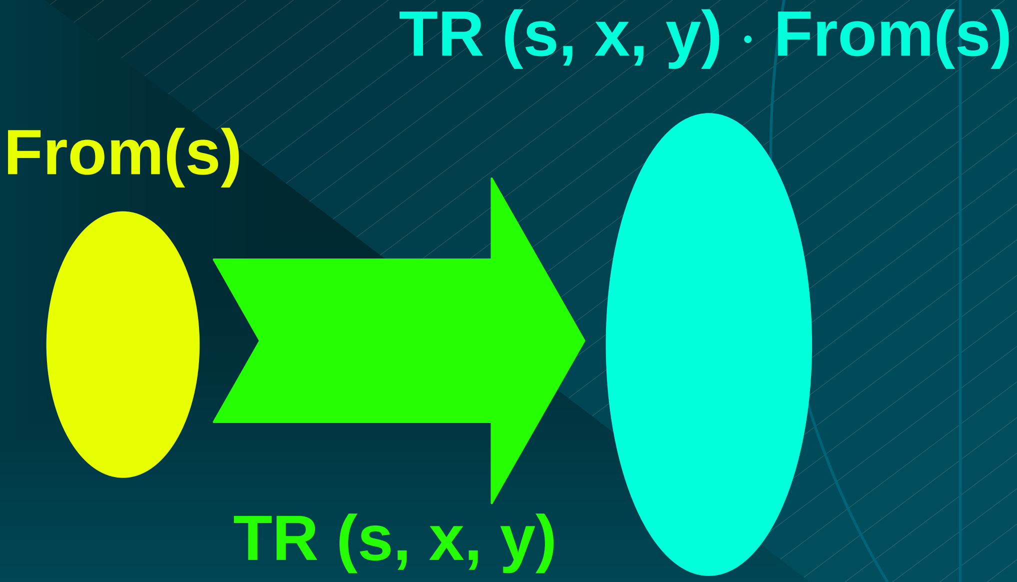
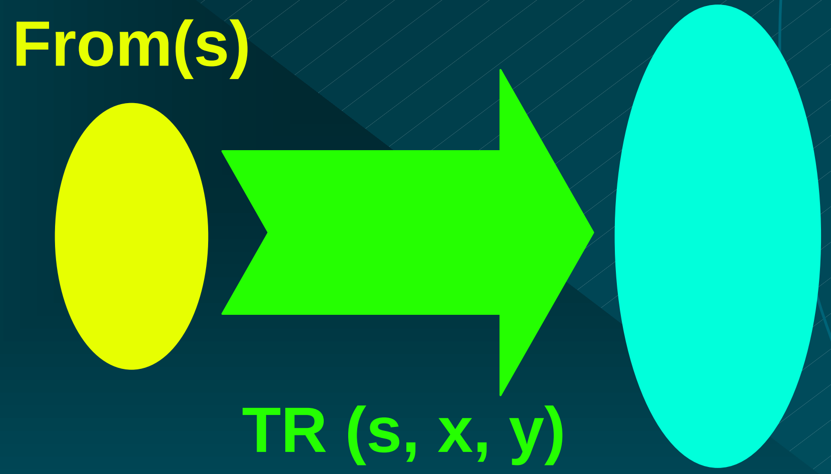


Image Computation

$$\text{Img}(TR, From) = \exists_{s,x} [TR(s, x, y) \cdot From(s)]$$



$\exists s,x\ [TR(s, x, y) \cdot From(s)]$

From(s)

TR (s, x, y)

Image Computation

$$\mathrm{Img}\ (TR, From) = \exists_{s,x}\ [TR\ (s, x, y) \cdot From(s)]$$

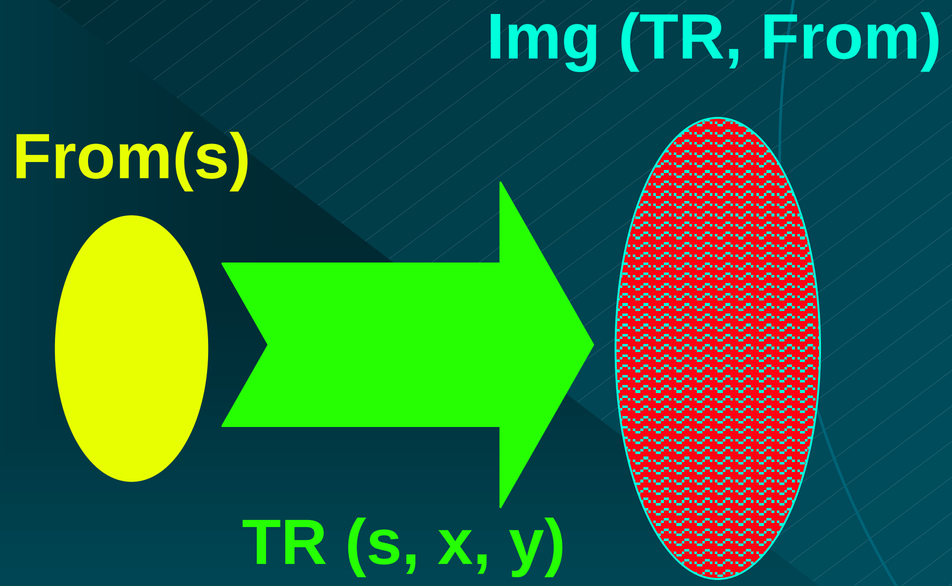
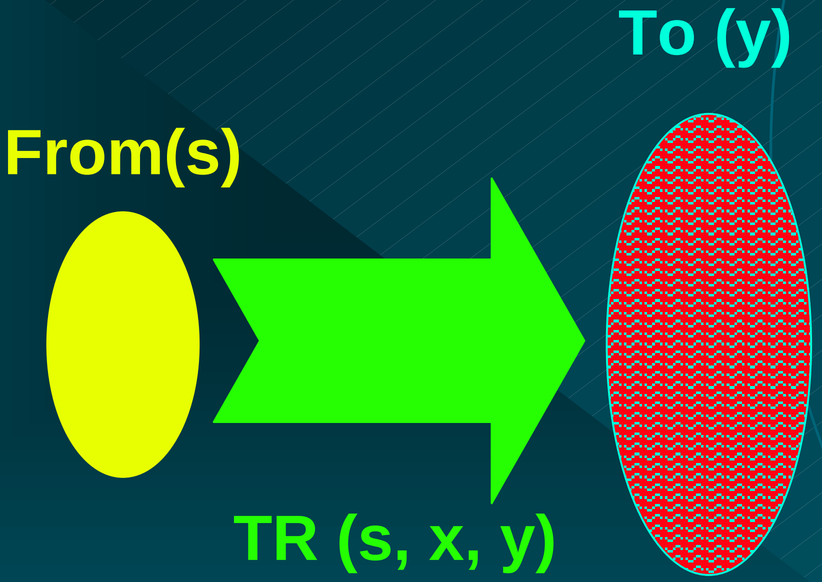


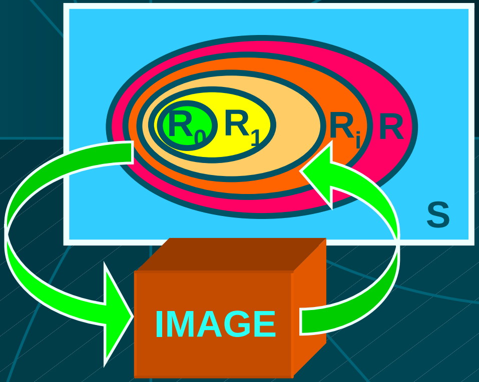
Image Computation

$$To\ (y) = Img\ (TR, From) = \exists_{s,x}\ [TR\ (s, x, y) \cdot From(s)]$$



$$To\ (y) =$$

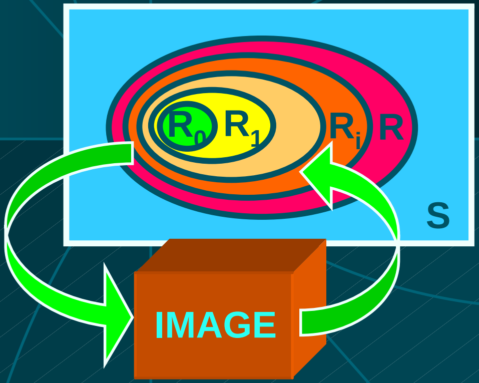
$$= \exists_{sx}[\ TR\ (s,x,y) \cdot From\ (s)\ ]$$

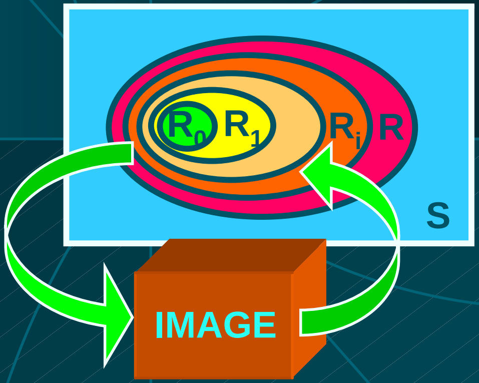


$$\text{To}(y) =$$

$$= \exists_{sx}[\ \text{TR}(s,x,y) \cdot \text{From}(s)\ ]$$

$$= \exists_{sx}[\ (y_1 \equiv \delta_1) \cdot (y_2 \equiv \delta_2) \cdot \ \ldots\ \cdot (y_n \equiv \delta_n) \cdot \text{From}(s)\ ]$$

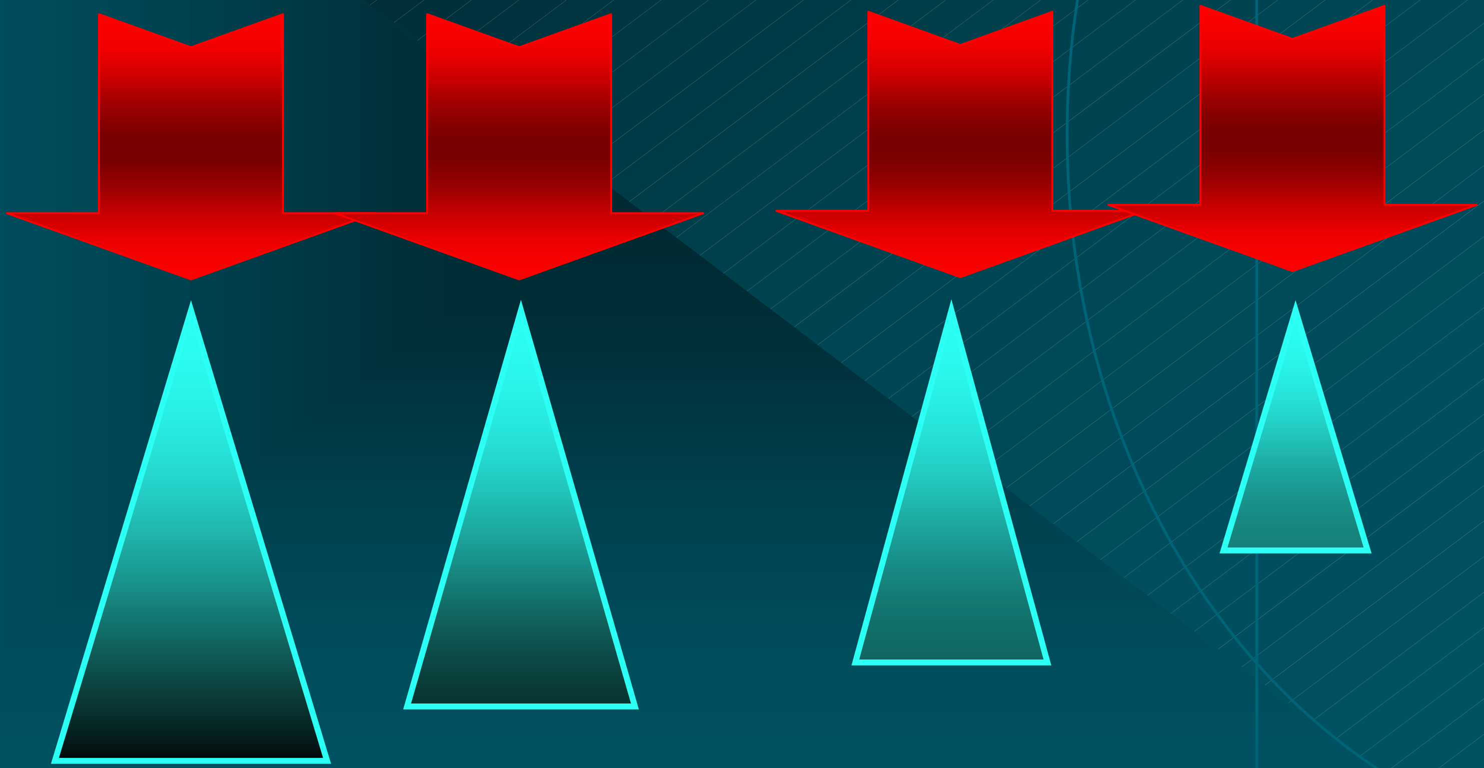


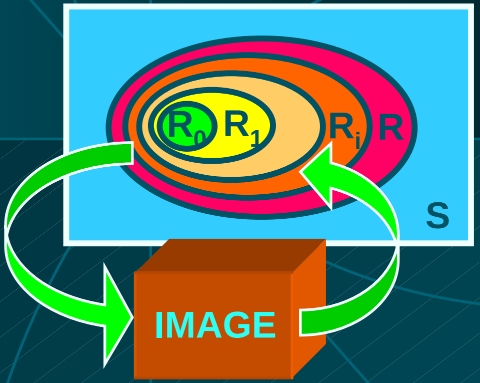


To (y) =

$$= \exists_{sx}[\ TR\ (s,x,y) \cdot From\ (s)\ ]$$

$$= \exists_{sx}[\ (y_1 \equiv \delta_1) \cdot (y_2 \equiv \delta_2) \cdot \ \ldots\ \cdot (y_n \equiv \delta_n) \cdot From\ (s)\ ]$$





To (y) =

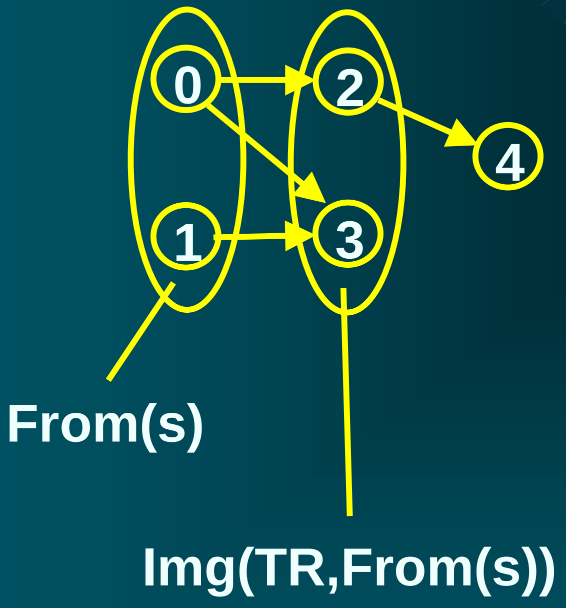
$$= \exists_{sx}[\ TR\ (s,x,y) \cdot From\ (s)\ ]$$

$$= \exists_{sx}[\ (y_1 \equiv \delta_1) \cdot (y_2 \equiv \delta_2) \cdot \ \ldots \ \cdot (y_n \equiv \delta_n) \cdot From\ (s)\ ]$$

Image and Pre-Image of States: An Example

Image of a set of states From(s)

Example:



From (s) =

$(s \equiv 0) \vee (s \equiv 1)$ $\{0,1\}$

TR (s, y) =

$(s \equiv 0) \wedge (y \equiv 2) \vee$ $\{(0,2),$

$(s \equiv 0) \wedge (y \equiv 3) \vee$ $(0,3),$

$(s \equiv 1) \wedge (y \equiv 3) \vee$ $(1,3),$

$(s \equiv 2) \wedge (y \equiv 4)$ $(2,4)\}$

TR (s, y) $\wedge$ From (s) =

$(s \equiv 0) \wedge (y \equiv 2) \vee$ $\{(0,2),$

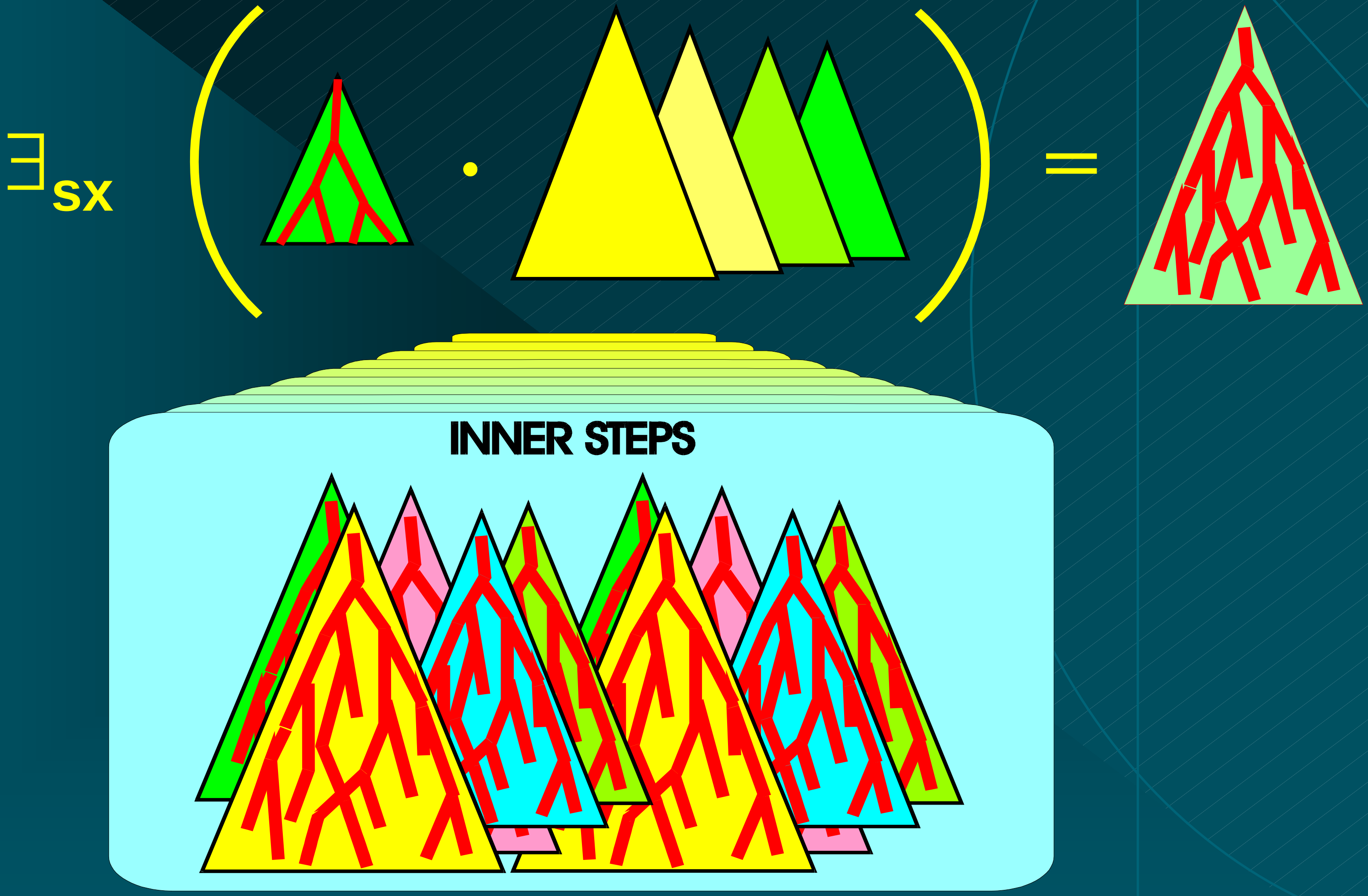
$(s \equiv 0) \wedge (y \equiv 3) \vee$ $(0,3),$

$(s \equiv 1) \wedge (y \equiv 3)$ $(1,3)\}$

To (y) = $\exists$s (TR $\wedge$ From) =

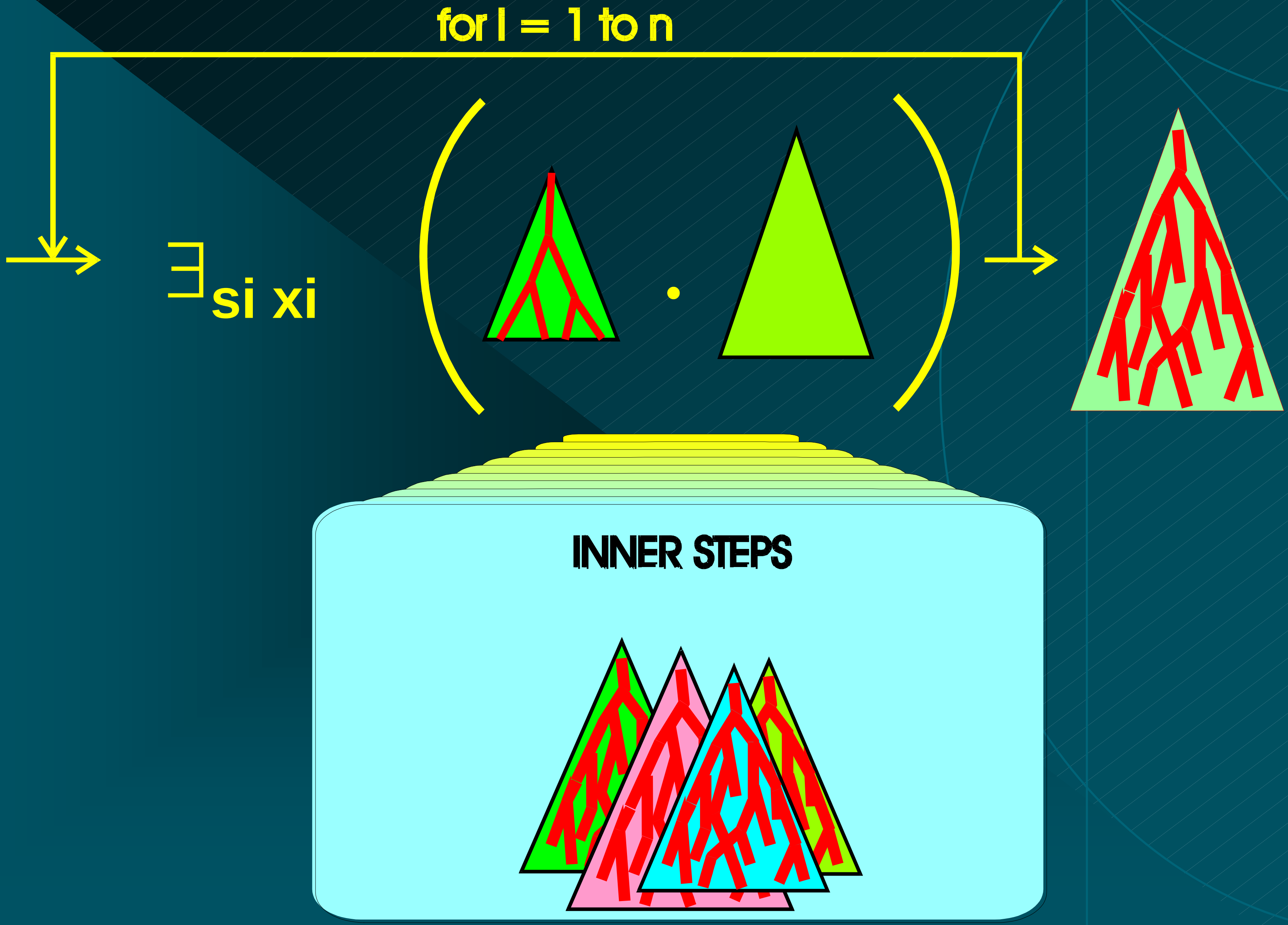
$(y \equiv 2) \vee (y \equiv 3)$ $\{(2,3)\}$

Image Computation: conjunctive partitioning



INNER STEPS

Conjunctive partitioning with early quantification

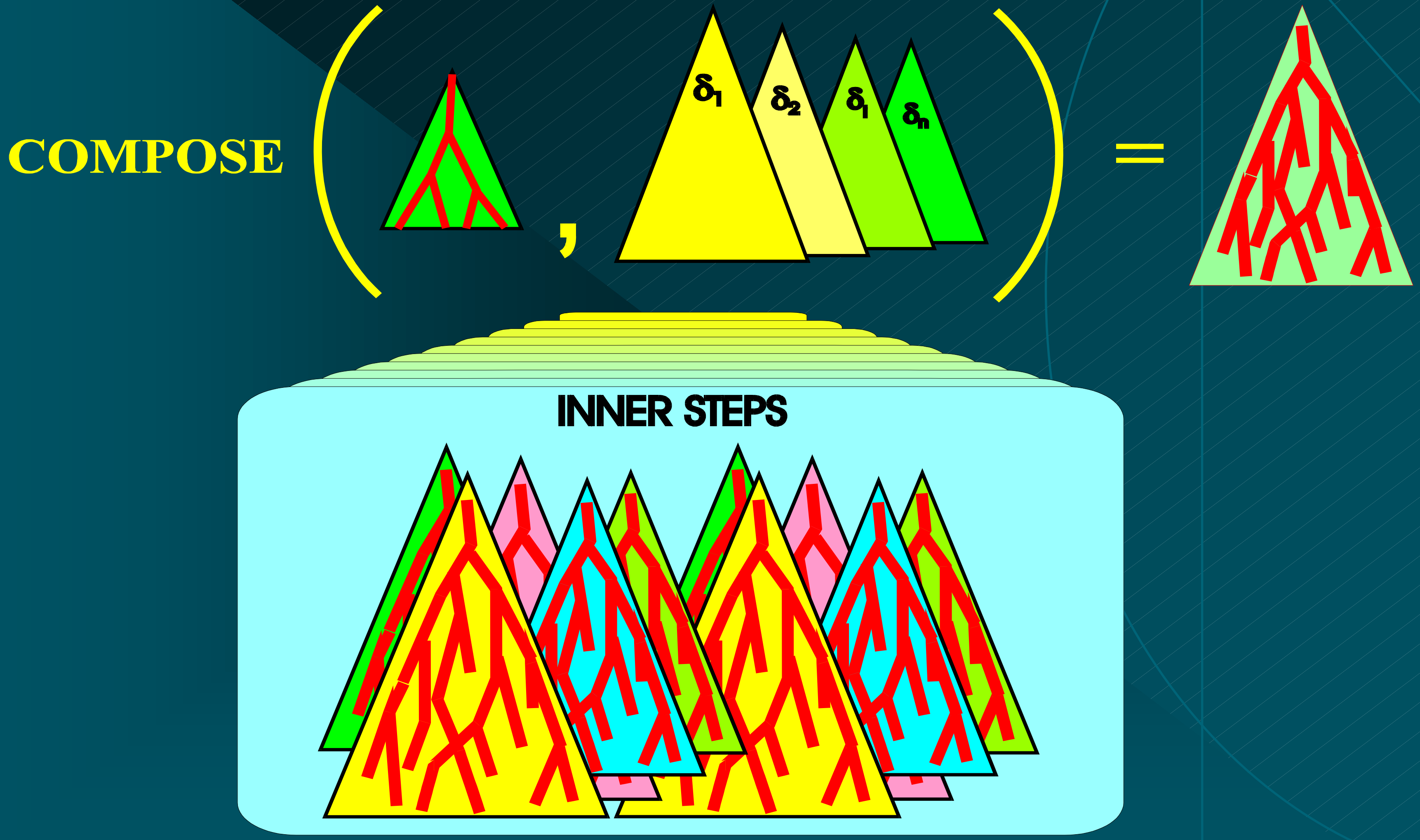


for I = 1 to n

$\exists_{si\ xi}$

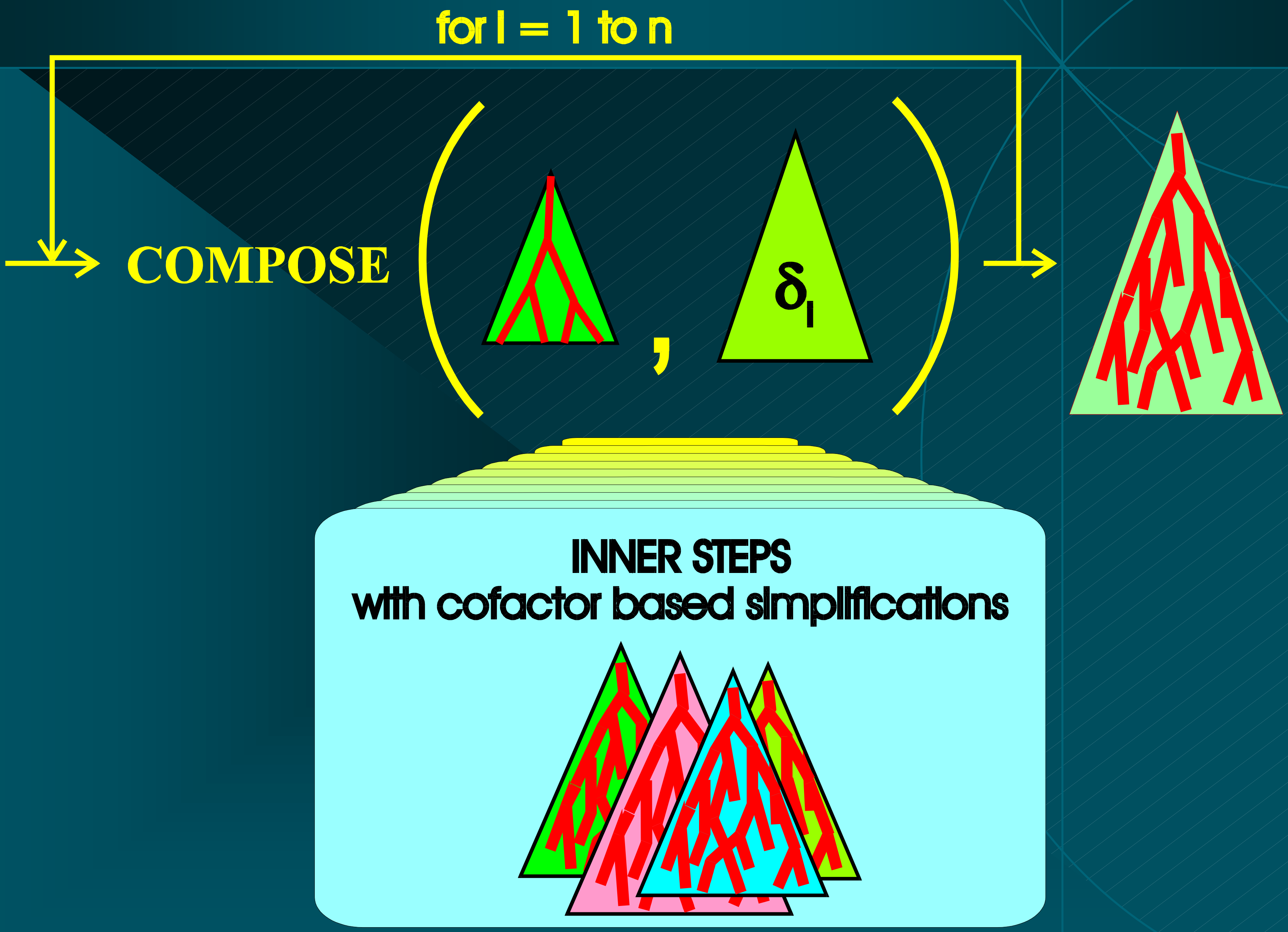
INNER STEPS

Pre-Image Computation



COMPOSE

INNER STEPS



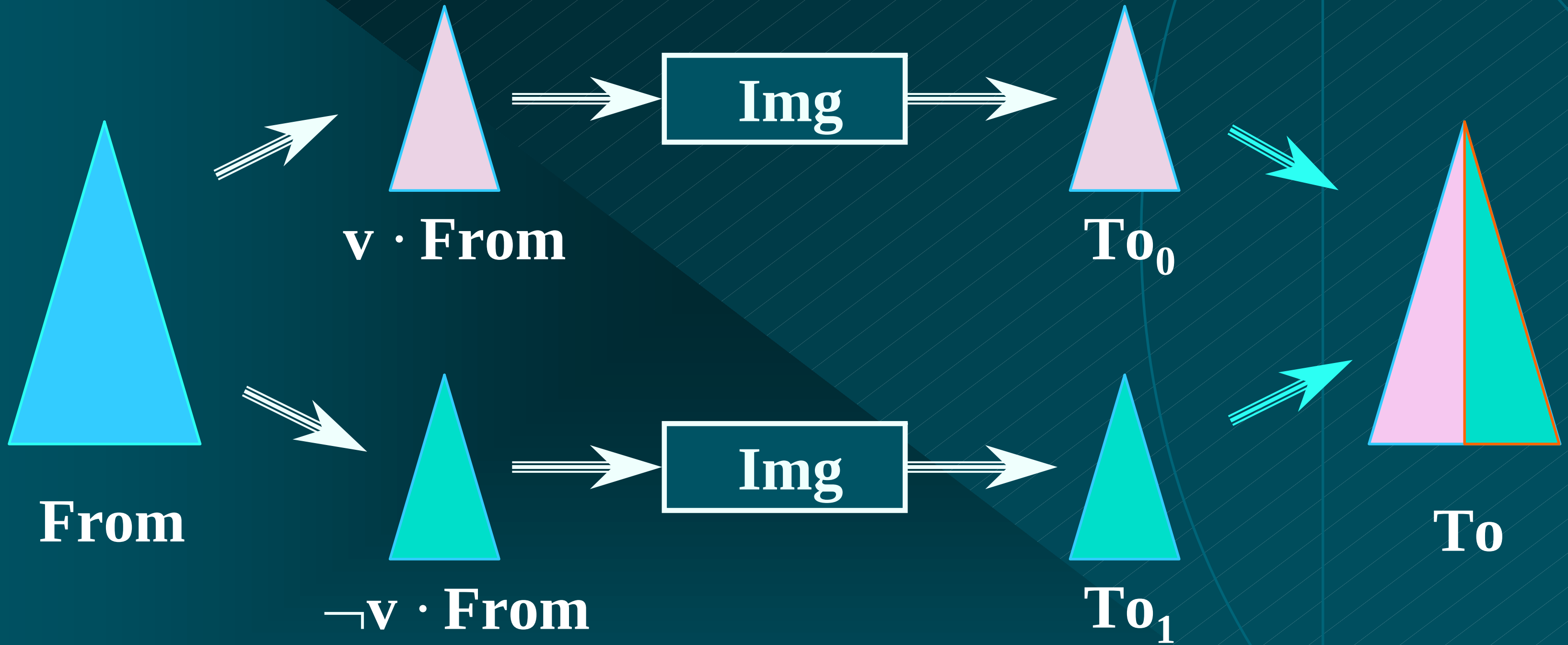
for i = 1 to n

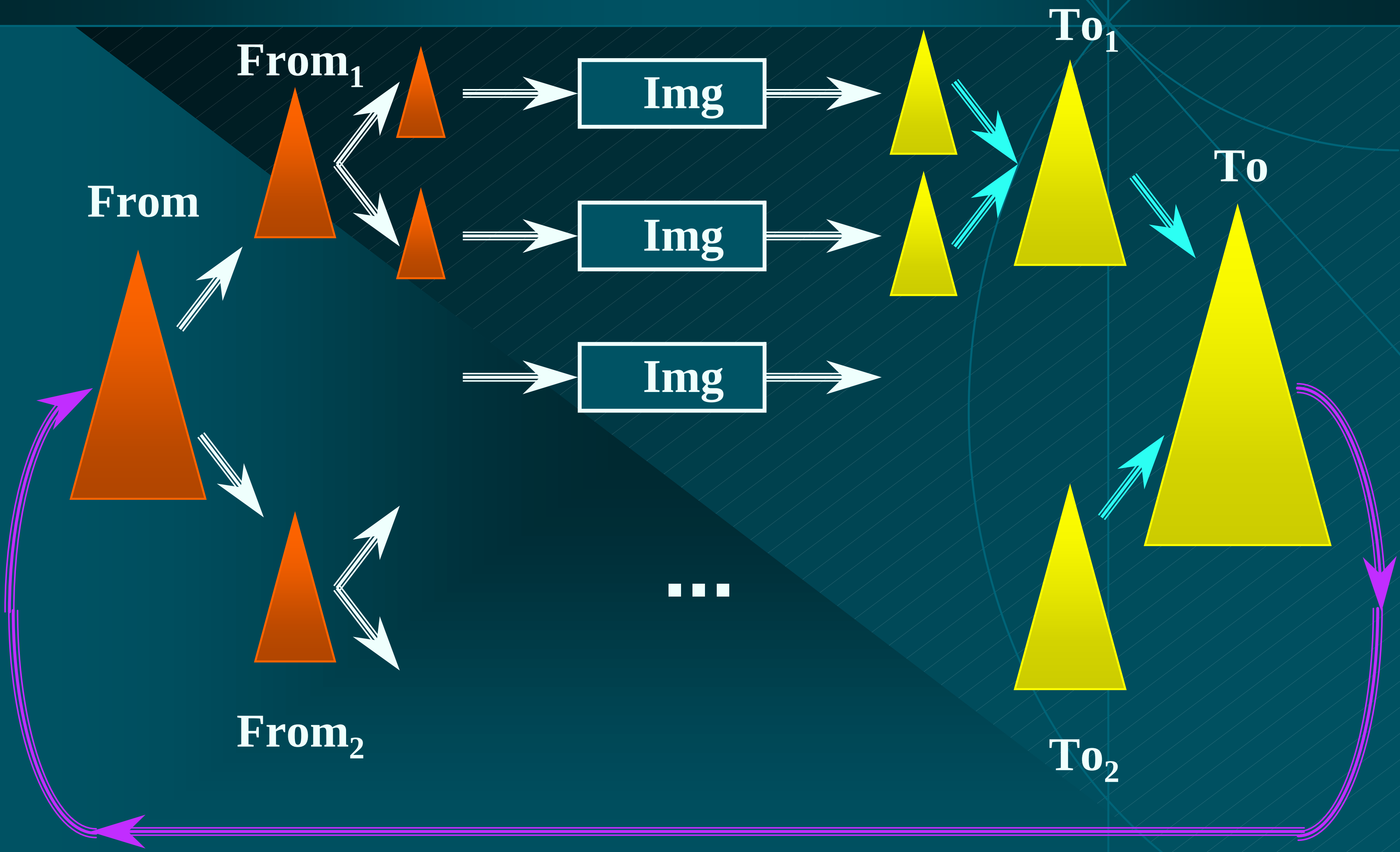
COMPOSE

$\left(\; , \; \delta_i \right)$

INNER STEPS
with cofactor based simplifications

Disjunctively Partitioned IMAGE




From$_1$
From
Img
Img
Img
To$_1$
To
...
From$_2$
To$_2$

```
Partitioned_Traversal ($\delta$, $S_0$ , th) {
     $R_p$ =  $F_p$ =  $N_p$ = $S_0$;
     while ( $N_p \neq \phi$ ) {
       $T_p = \phi$;
       foreach $f \in F_p$ {
         $T_p$ = ($T_p$,  Img ($\delta$, f));
         $N_p$ =  $F_p$ =  Set_Diff ($T_p$,  $R_p$); $R_p$ =  Set_Union ($N_p$,  $R_p$);
         $F_p$ =  Re_Partition ($F_p$, th);  $R_p$ =  Re_Partition ($R_p$, th);
       }
    return ( $R_p$);
}
```

Maximum Size at intermediate steps [Ravi & Somenzi, ICCAD'95]

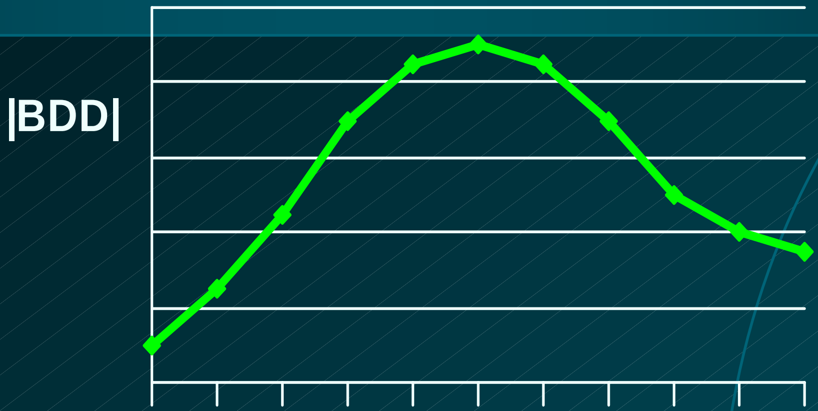
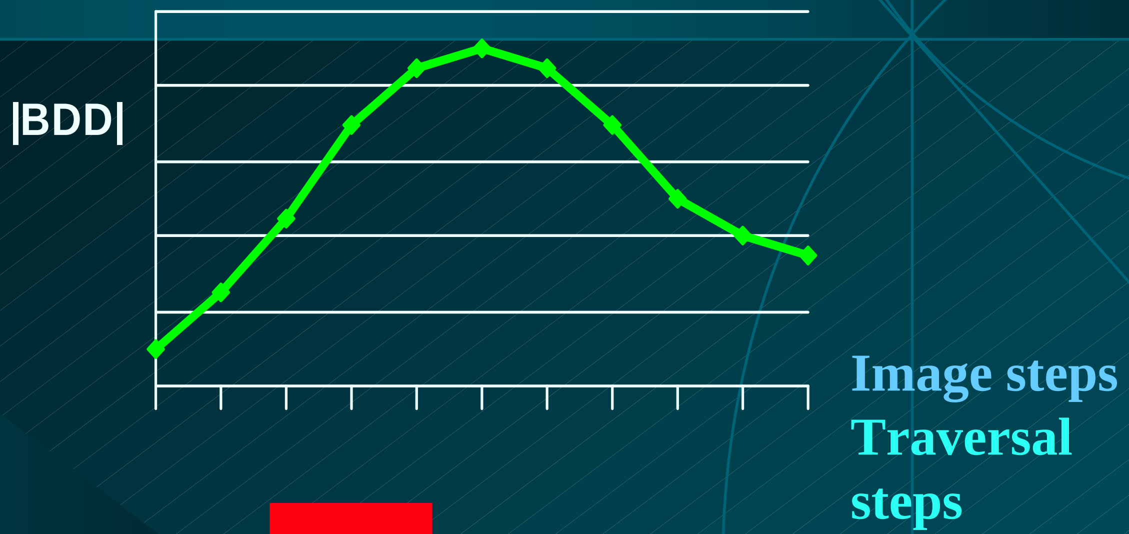
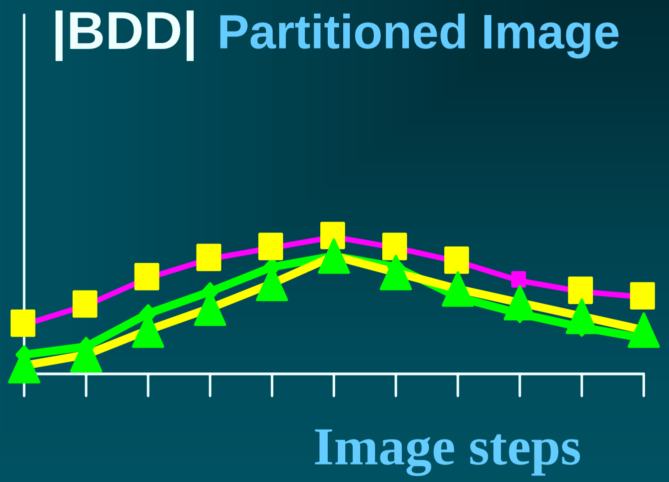


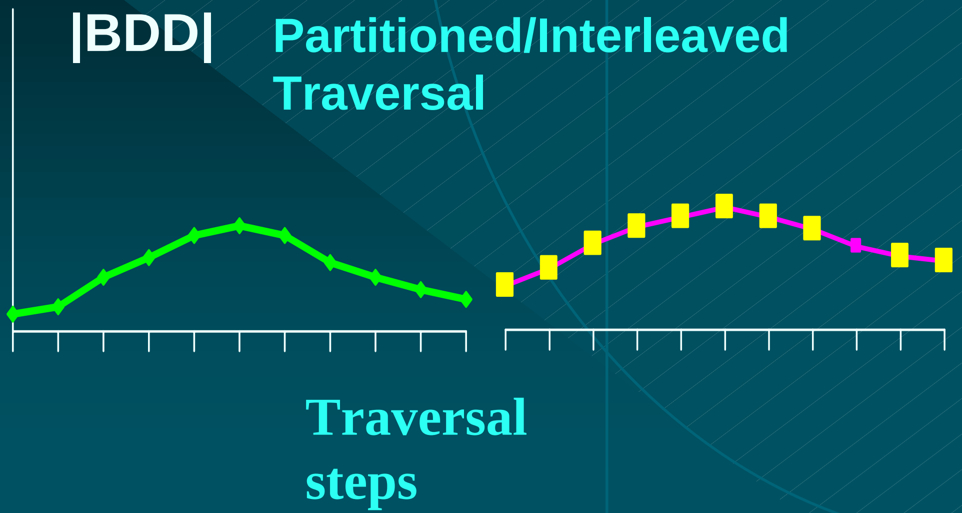
Image steps

Traversal steps

Maximum Size at intermediate steps [Ravi & Somenzi, ICCAD’95]







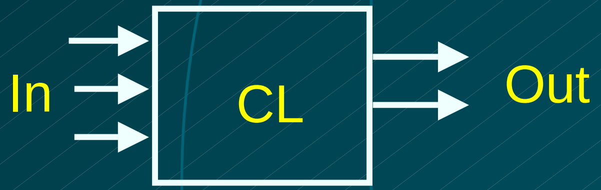
- Conjunctively pertitioned IMG (sort – clustering – Exist)
- Disj part img
- Disj part trav

EQUIVALENCE CHECKING

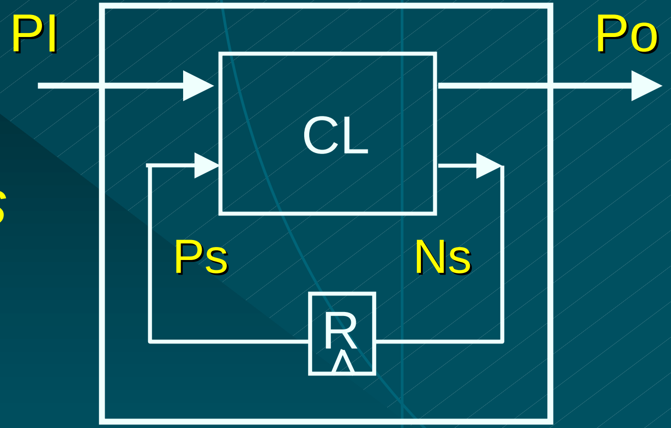
Equivalence Checking

- Two circuits are *functionally* equivalent if they exhibit the same behavior

- Combinational circuits
  - for all possible input *values*



- Sequential circuits
  - – for all possible input *sequences*



Combinational Equivalence Checking

- Functional Approach
  - transform output functions of combinational circuits into a unique (*canonical*) representation
  - two circuits are equivalent if their representations are identical
  - efficient canonical representation: BDD

- Structural
  - identify structurally *similar* internal points
  - prove internal points (cut-points) *equivalent*
  - find implications

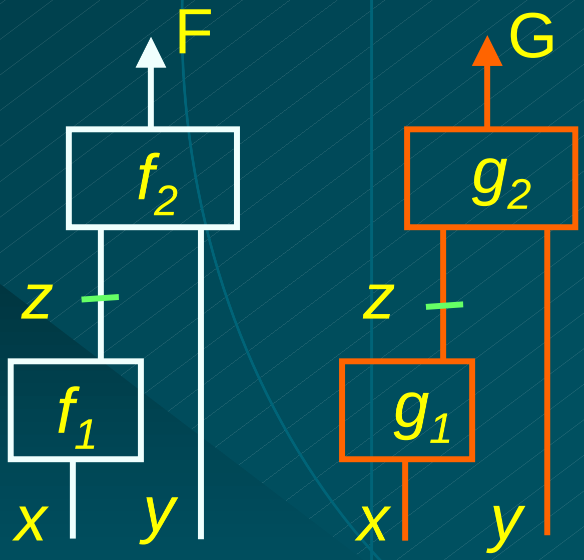
Functional Equivalence

- **If BDD can be constructed for each circuit**
  - **represent each circuit as *shared* (multi-output) BDD**
    - **use the same variable ordering !**
  - **BDDs of both circuits must be *identical***

- If BDDs are too large
  - cannot construct BDD, memory problem
  - use partitioned BDD method
    - decompose circuit into smaller pieces, each as BDD
    - check equivalence of internal points

Functional Decomposition

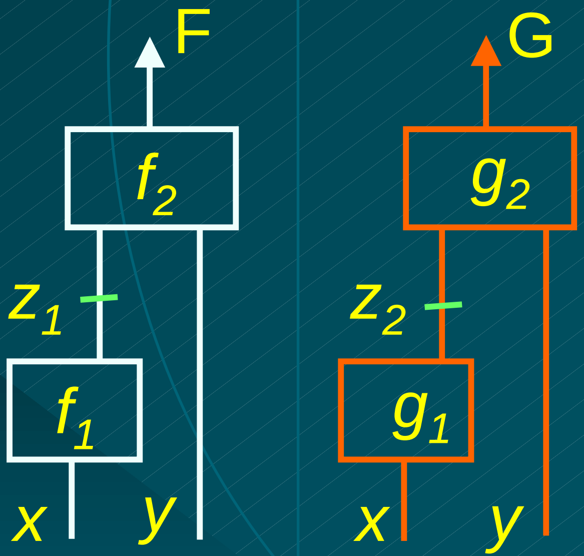
- Decompose each function into *functional* blocks
  - represent each block as a BDD (*partitioned BDD* method)
  - define *cut-points (z)*
  - verify equivalence of blocks at cut-points starting at primary inputs



Cut-Points Resolution Problem

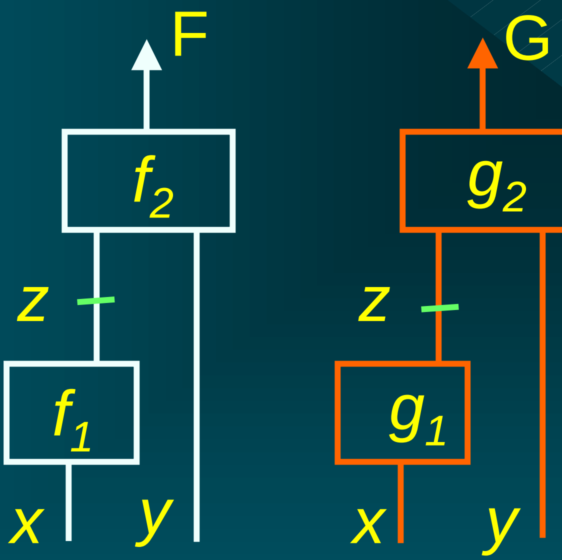
- If *all pairs* of cut-points (z_1,z_2) are equivalent
  - so are the two functions, F,G
- If *intermediate* functions (f_2,g_2) are not equivalent
  - the functions (F,G) may still be equivalent
  - this is called *false negative*

- Why do we have false negative ?
  - functions are represented in terms of *intermediate* variables
  - to prove/disprove equivalence must represent the functions in terms of *primary inputs* (BDD composition)



Cut-Point Resolution – Theory

- Let $f_1(x)=g_1(x)\ \forall x$
  - if $f_2(z,y) \equiv g_2(z,y),\ \forall z,y$ then $f_2(f_1(x),y) \equiv g_2(f_1(x),y) \Rightarrow F \equiv G$
  - if $f_2(z,y) \neq g_2(z,y),\ \forall z,y \quad \neq\Rightarrow \quad f_2(f_1(x),y) \neq g_2(f_1(x),y) \Rightarrow F \cancel{\neq} G$



We *cannot* say if $F \equiv G$ or not

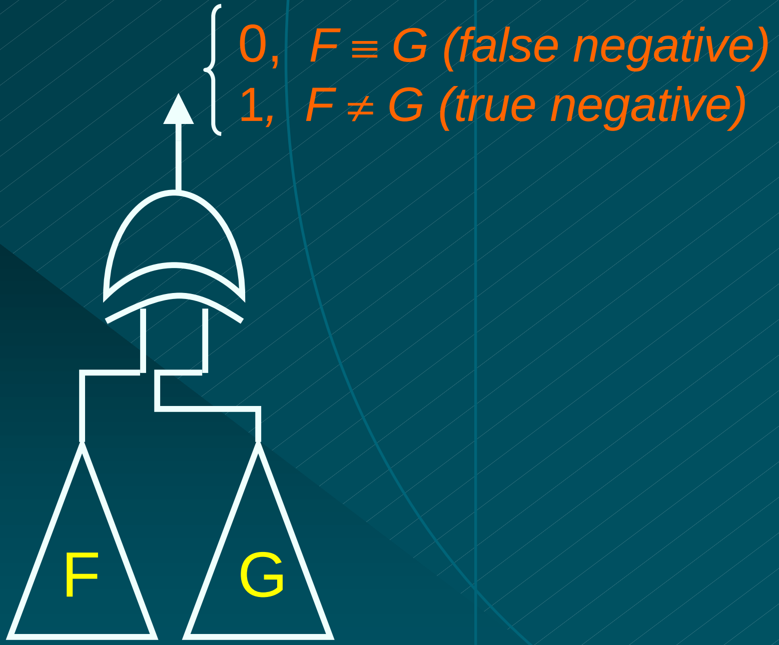
- *False negative*
  - – two functions are equivalent, but the verification algorithm declares them as different.

Cut-Point Resolution – cont'd

- How to verify if negative is *false* or *true* ?

❖ **<u>Procedure 1</u>: create a miter (XOR) between two potentially equivalent nodes/functions**

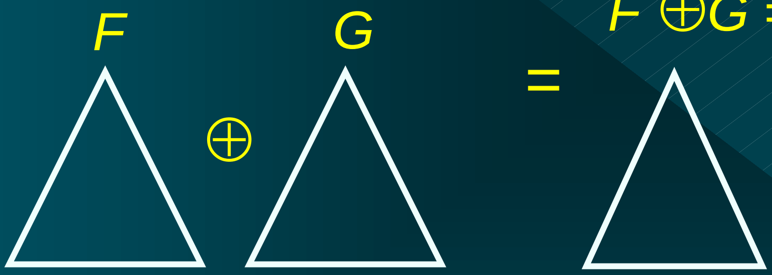
- **perform ATPG test for *stuck-at 0***
- **find test pattern to prove *F ≠ G***
- **efiicient for true negative (gives *test vector*, a proof)**
- **inefficient when there is no test**



$$\begin{cases} 0, & F \equiv G \text{ (false negative)} \\ 1, & F \neq G \text{ (true negative)} \end{cases}$$

Cut-Point Resolution – cont'd

- Procedure 2: create a BDD for $F \oplus G$
  - perform satisfiability analysis (SAT) of the BDD
    - if BDD for $F \oplus G = \emptyset$, problem is *not* satisfiable, *false* negative
    - BDD for $F \oplus G \neq \emptyset$, problem is satisfiable, *true* negative



$$F \oplus G = \begin{cases} \emptyset, & F \equiv G \text{ (false negative)} \\ \text{Non-empty}, & F \neq G \end{cases}$$

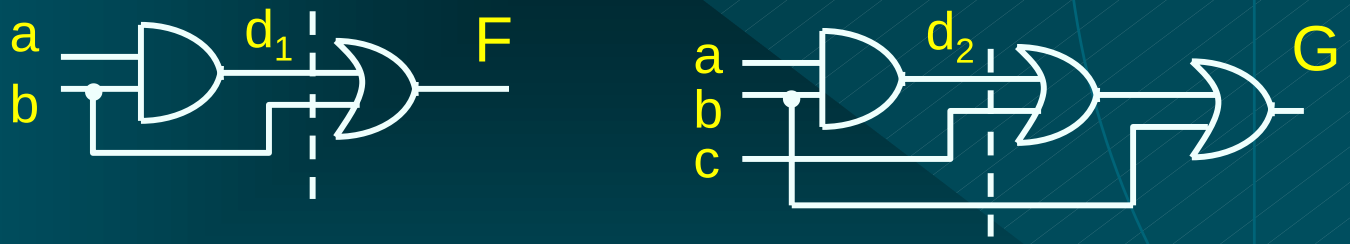
Note: must compose BDDs until they are equivalent, or expressed in terms of primary inputs

– the SAT solution, if exists, provides a *test vector* (proof of non-equivalence) – as in ATPG

– unlike the ATPG technique, it is effective for false negative (the BDD is empty!)

Structural Equivalence Check

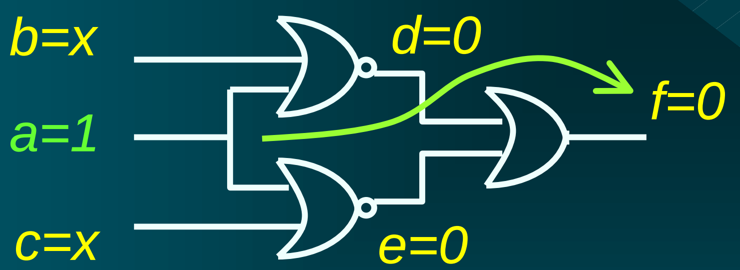
- **Given two *circuits,* each with its own structure**
  - **identify “similar” internal points, *cut sets***
  - **exploit internal equivalences**
- **False negative problem may arise**
  - **F ≡ G, but differ structurally (different local support)**
  - **verification algorithm declares F,G as different**

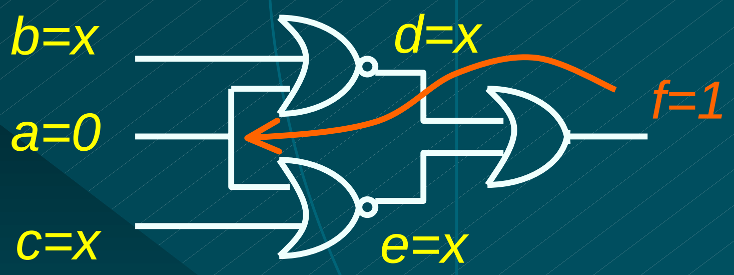


- Solution: use BDD-based or ATPG-based methods to resolve the problem. Also: *implication, learning techniques.*

Implication Techniques

- Techniques that extract and exploit internal correspondences to speed up verification
- Implications – *direct* and *indirect*



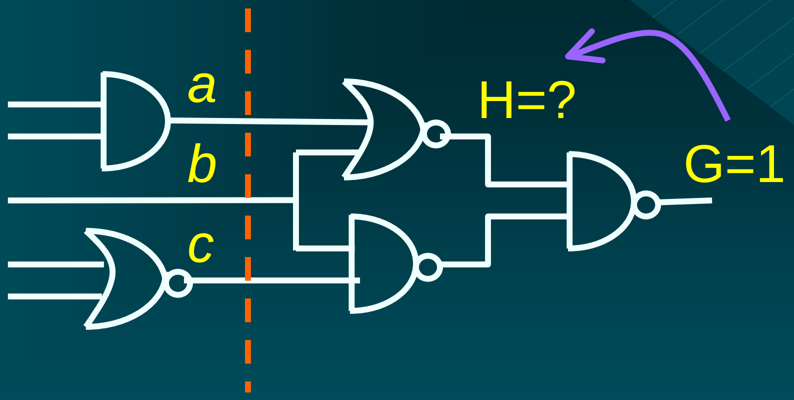


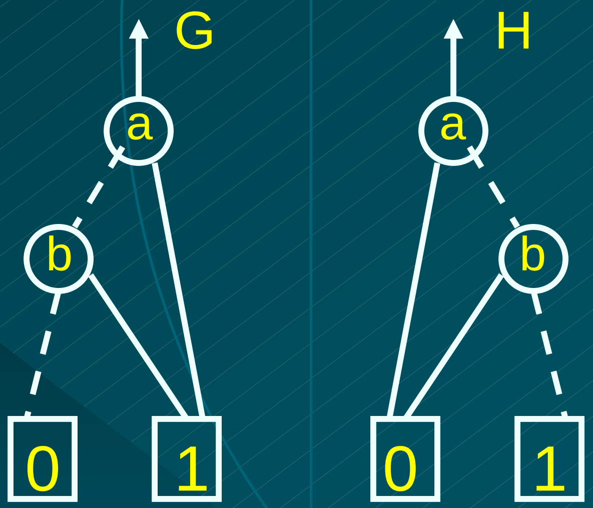
Direct: $a=1 \Rightarrow f=0$

Indirect (learning): $f=1 \Rightarrow a=0$

Learning Techniques

- Learning
  - process of deriving *indirect* implications
  - Recursive learning
    - recursively analyzes effects of each justification
  - Functional learning
    - uses BDDs to learn indirect implications

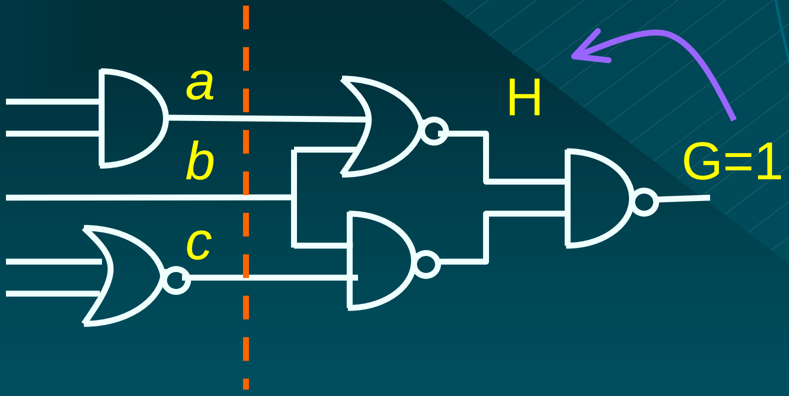




G=1 ⇒ H=0

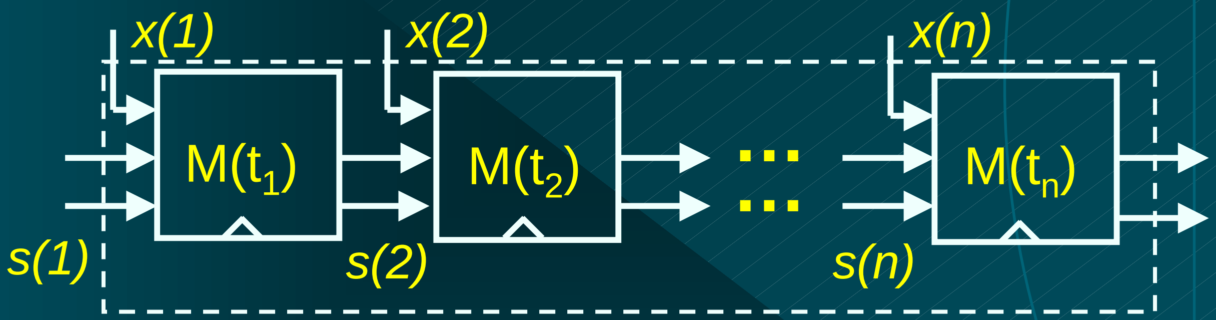
Learning Techniques – cont'd

- Other methods to check implications G=1 ⇒ H=0
  - Build a BDD for G • H'
    - If this function is satisfiable (G·H'=1), the implication holds and gives a test vector
    - Otherwise it does not hold
  - Since G=1 ⇒ H=0 ≡ (G'+H')=1, build a BDD for (G'+H')
    - The implication holds if (G'+H') ≡1 (tautology, trivial BDD)



Sequential Equivalence Checking

- Represent each sequential circuit as an FSM
  - verify if two FSMs are equivalent
- Approach 1: reduction to *combinational* circuit
  - unroll FSM over *n* time frames (flatten the design)



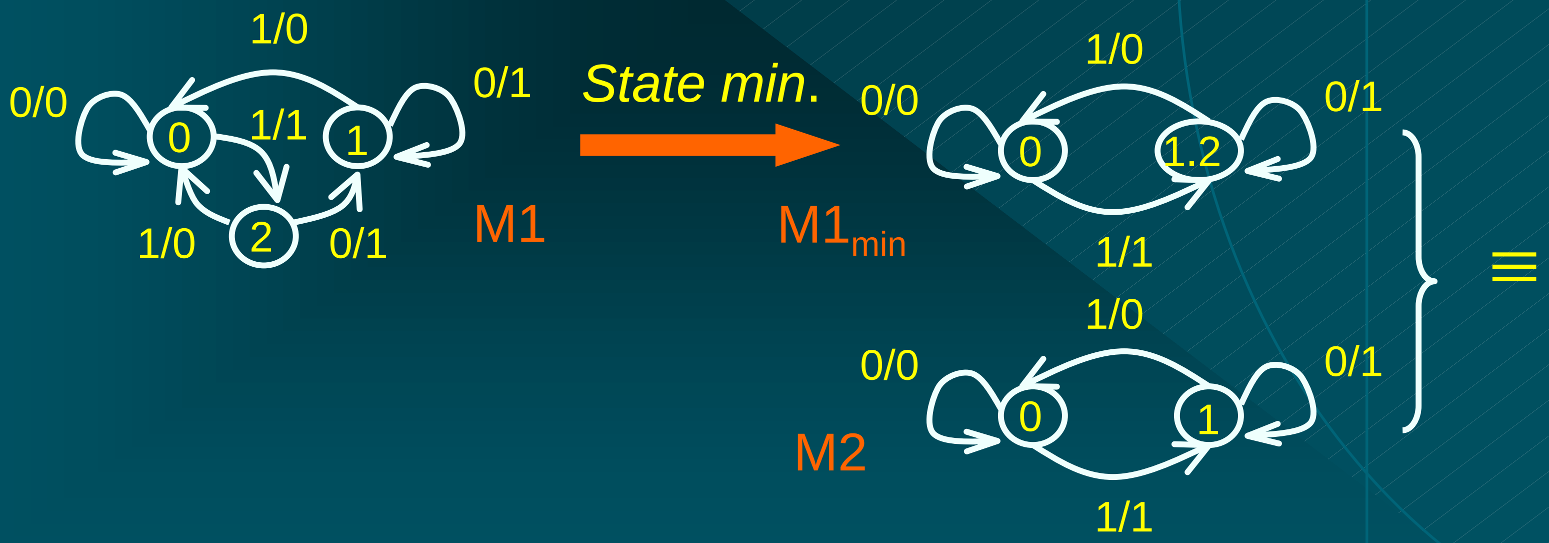
Combinational logic: *F(x(1,2, …n), s(1,2, … n))*

– check equivalence of the resulting combinational circuits

– problem: the resulting circuit can be too large too handle

Sequential Verification

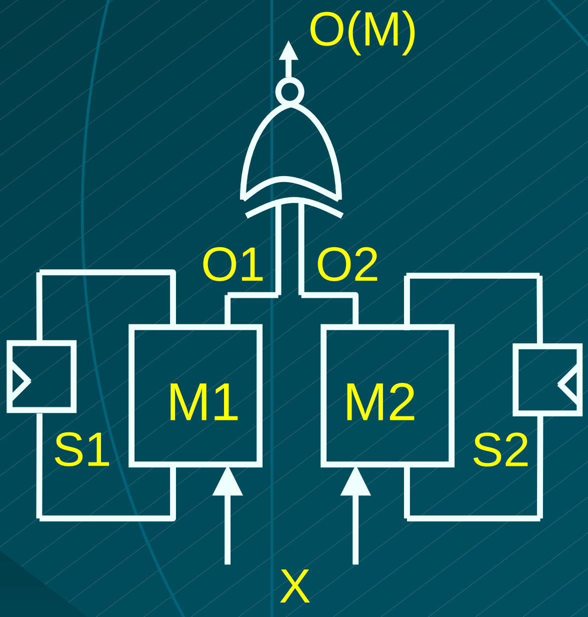
- Approach 2: based on isomorphism of state transition graphs
  - two machines M1, M2 are *equivalent* if their state transition graphs (STGs) are *isomorphic*
  - perform state minimization of each machine
  - check if STG(M1) and STG(M2) are isomorphic



Sequential Verification

❖ **Approach 3: symbolic FSM traversal of the product machine**

- Given two FSMs: $M_1(X,S_1, \delta_1, \lambda_1,O_1)$, $M_2(X,S_2, \delta_2, \lambda_2,O_2)$
- Create a product FSM: $M = M_1 \times M_2$
  - traverse the states of M and check its output for each transition
  - the output O(M) =1, if outputs $O_1 = O_2$
  - if all outputs of M are 1, M_1 and M_2 are *equivalent*
  - otherwise, an *error state* is reached
  - *error trace* is produced to show: $M_1 \neq M_2$

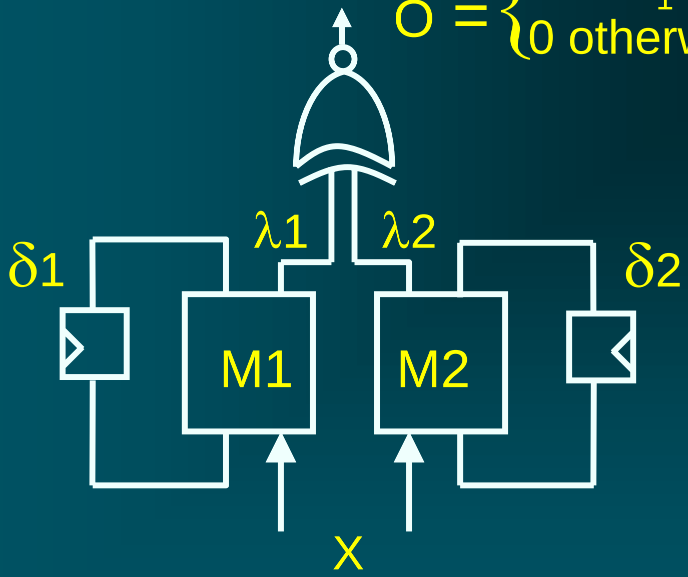


Product Machine - Construction

- **Define the product machine M(X,S, δ, λ,O)**
  - **states,** $S = S_1 \times S_2$
  - **next state function,** $\delta(s,x) : (S_1 \times S_2) \times X \rightarrow (S_1 \times S_2)$
  - **output function,** $\lambda(s,x) : (S_1 \times S_2) \times X \rightarrow \{0,1\}$

$$O = \begin{cases} 1 & \text{if } O_1 = O_2 \\ 0 & \text{otherwise} \end{cases}$$

$$\boxed{\lambda(s,x) = \lambda_1(s_1,x)\ \overline{\oplus}\ \lambda_2(s_2,x)}$$



- Error trace (*distinguishing sequence*) that leads to an error state
  - sequence of inputs which produces 1 at the output of M
  - produces a state in M for which M1 and M2 give different outputs

FSM Traversal - Algorithm

- Traverse the product machine M(X,S,δ, λ,O)
  - start at an initial state S_0
  - iteratively compute symbolic image $Img(S_0,R)$ (set of *next states*):

$$Img(\ S_0,R\ ) = \exists_x \exists_s\ S_0(s) \bullet R(x,s,t)$$

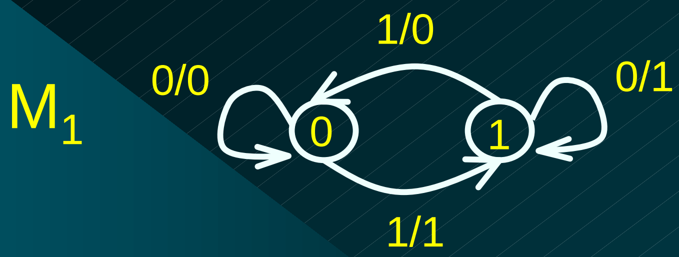
$$R = \prod_i R_i = \prod_i (t_i \equiv \delta_i(s,x))$$

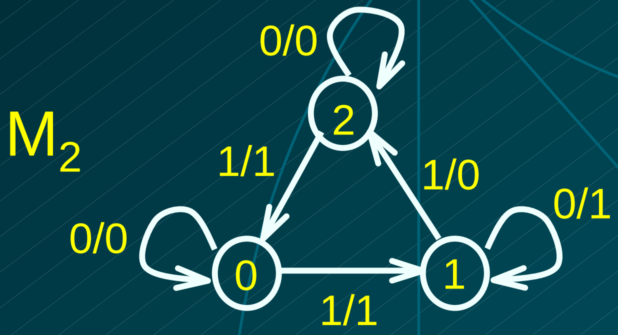
until an *error state* is reached

  - transition relation R_i for each next state variable t_i can be computed as $t_i = (t \otimes \delta(s,x))$

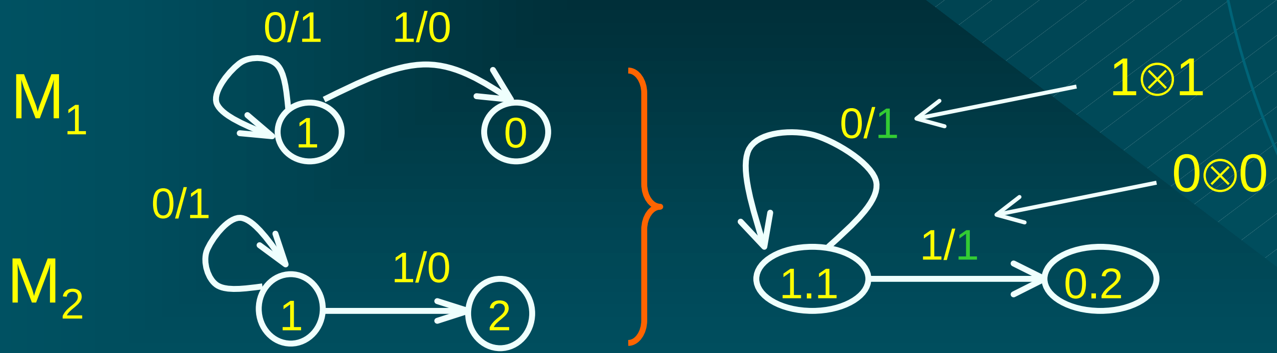
    (this is an alternative way to compute transition relation, when design is specified at gate level)

Construction of the Product FSM



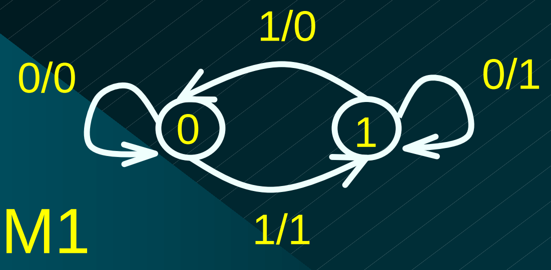


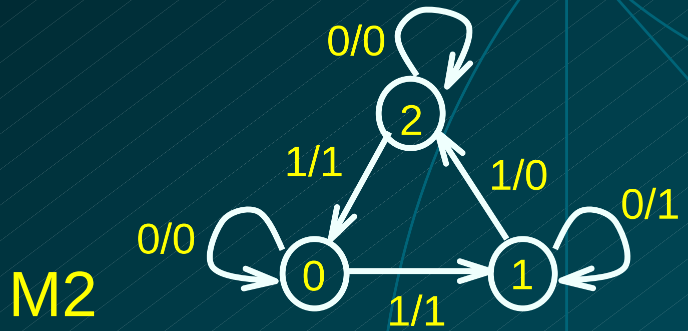
- For each pair of states, $s_1 \in M_1$, $s_2 \in M_2$
  - create a combined state $s = (s_1. s_2)$ of M
  - create transitions out of this state to other states of M
  - label the transitions (*input/output*) accordingly



$\text{Output} = \begin{cases} 1 & \text{OK} \\ 0 & \text{error} \end{cases}$

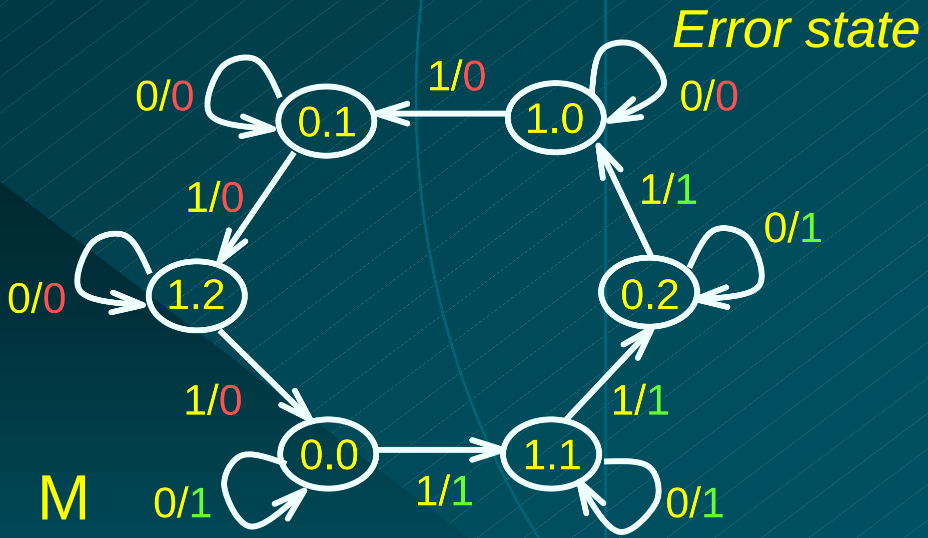
FSM Traversal in Action





Initiall states: s_1=0, s_2=0, s=(0.0)

	Out(M)	
State reached	*x=0*	*x=1*
• *New* ^{0} = (0.0)	1	1
• *New* ^{1} = (1.1)	1	1
• *New* ^{2} = (0.2)	1	1
• *New* ^{3} = (1.0)	0	0



❖ **STOP - backtrack to initial state to get *error trace: x*={1,1,1,0}**

Symbolic CTL Model Checking

- Represent the required subsets of states as boolean functions and hence as ROBDDs.
- Represent the transition relation as a boolean function and hence as a ROBDD.
- Reduce the iterative fixed point computations of the model checking process to operations on ROBDDs.
- Check for the termination of the fixpoint computation by checking ROBDD equivalence.

CTL Formulas

- Temporal logic formulas are evaluated w.r.to a state in the model

- State formulas
  - apply to a specific state

- Path formulas
  - apply to all states along a specific path

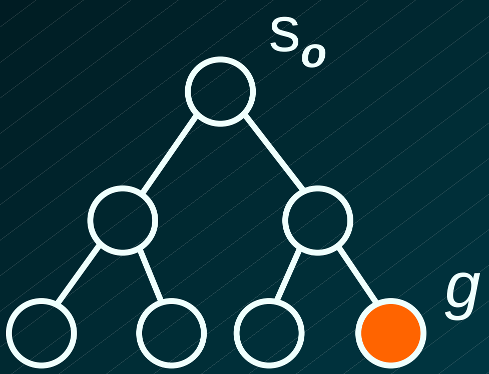
Basic CTL Formulas

- E X (f)
  - **true in state *s* if *f* is true in some successor of *s* (there *exists* a next state of s for which *f* holds)**
- A X (f)
  - **true in state *s* if *f* is true for all successors of *s (*for *all* next states of *s* *f* is true)**
- E G (f)
  - **true in *s* if *f* holds in *every* state along *some* path emanating from *s (there exists a path ….)***
- A G (f)
  - **true in *s* if *f* holds in every state along *all* paths emanating from *s* (*for all* paths ….*globally*)**

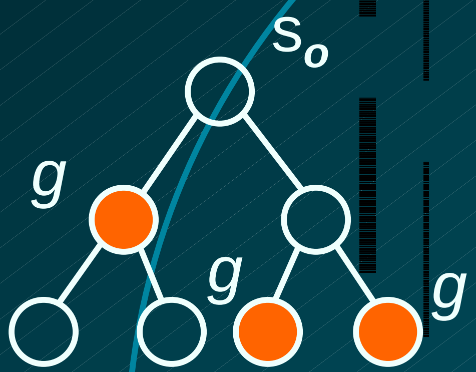
Basic CTL Formulas - cont ’d

- E F (g)
  - there *exists* a path which *eventually* contains a state in which *g* is true
- A F (g)
  - for *all* paths, eventually there is state in which *g* holds
- E F, A F are special case of E [f U g], A [f U g]
  - E F (g) = E [*true* U g], A F (g) = A [*true* U g]
- f U g (f *until* g)
  - true if there is a state in the path where *g* holds, and at every previous state *f* holds

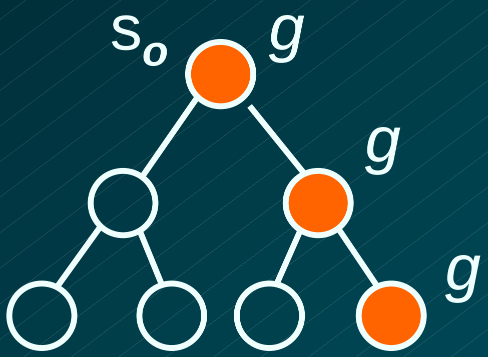
CTL Operators - examples



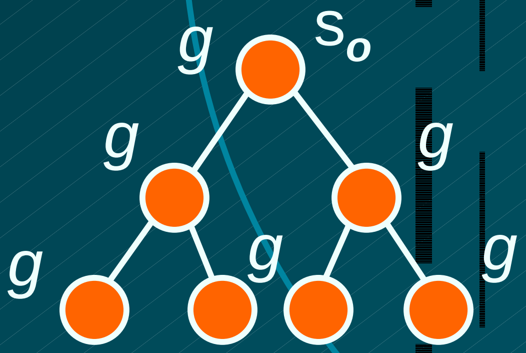
s_o |= E F g



s_o |= A F g



s_o |= E G g



s_o |= A G g

Minimal set of CTL Formulas

- Full set of operators
  - Boolean: $\neg, \wedge, \vee, \oplus, \rightarrow$
  - temporal: E, A, X, F, G, U, R

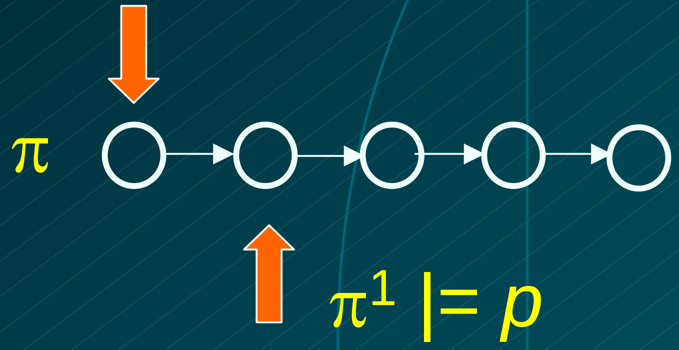
- Minimal set sufficient to express any CTL formula
  - Boolean: $\neg, \vee$
  - temporal: E, X, U

- Examples:

  $f \wedge g = \neg(\neg f \vee \neg g)$, F f = *true* U f , A (f) = $\neg$E($\neg f$)

Semantics of X and U

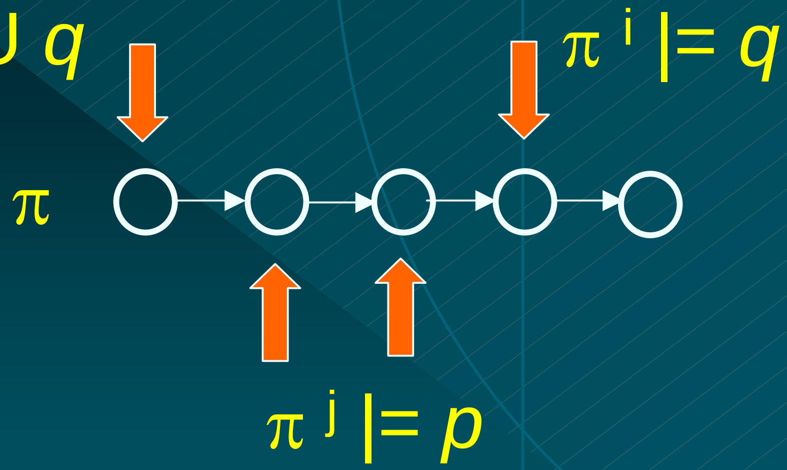
- Semantics of X: $\pi \models X\, p$



π

$\pi^1 \models p$

- Semantics of U: $\pi \models p\ U\ q$



$\pi^i \models q$

π

$\pi^j \models p$

Typical CTL Formulas

- $\mathrm{E\,F}\,(\,\mathit{start} \wedge \neg\,\mathit{ready}\,)$
  - eventually a state is reached where *start* holds and *ready* does not hold
- $\mathrm{A\,G}\,(\,\mathit{req} \rightarrow \mathrm{A\,F}\,\mathit{ack}\,)$
  - any time *request* occurs, it will be eventually *ack*nowledged
- $\mathrm{A\,G}\,(\,\mathrm{E\,F}\,\mathit{restart}\,)$
  - from any state it is possible to get to the *restart* state

CTL symbolic Model Checking

- $|[\phi]| = f_{x_i}(x_1,\ldots,x_n) = x_i$
  (the OBDD for the *boolean variable* x_i)
- $|[\neg\phi]| = \neg f_{\phi}(x_1,\ldots,x_n)$
  (apply negation to the OBDD for ϕ)
- $|[\phi \vee \psi]| = f_{\phi}(x_1,\ldots,x_n) \vee f_{\psi}(x_1,\ldots,x_n)$
  (apply $\vee$ operation to the OBDDs for ϕ and ψ)
- $|[\phi \wedge \psi]| = f_{\phi}(x_1,\ldots,x_n) \wedge f_{\psi}(x_1,\ldots,x_n)$
  (apply $\wedge$ operation to the OBDDs for ϕ and ψ)

CTL Symbolic Model Checking

- $|[\mathrm{EX}\ \phi]| =$

  $\exists x'_1,\ldots,x'_n(f_\phi(x'_1,\ldots,x'_n) \wedge TR(x_1,\ldots,x_n,x'_1,\ldots,x'_n))$

This is also the *pre-image of* $|[\phi]|$ by *TR*

- $|[\mathrm{EU}(\phi,\psi)]| =$

  $\mu Z.(f_\psi(x_1,\ldots,x_n) \vee (f_\phi(x_1,\ldots,x_n) \wedge \mathrm{EX}\ Z))$

- $|[\mathrm{EG}\ \phi]| = \nu Z.(f_\phi(x_1,\ldots,x_n) \wedge \mathrm{EX}\ Z)$