CTL Model Checking

Wishnu Prasetya

wishnu@cs.uu.nl
www.cs.uu.nl/docs/vakken/pv

Background

- Example: verification of web applications → e.g. to prove existence of a path from page A to page B.

  Use of **CTL** is popular → another variant of "temporal logic" → different way of model checking.

- Model checker for verifying CTL: **SMV**. Also uses a technique called "symbolic" model checking.

  - In contrast, SPIN model checking is called "explicit state".

  - We'll show you how this symbolic MC works, but first we'll take a look at CTL, and the web application case study.

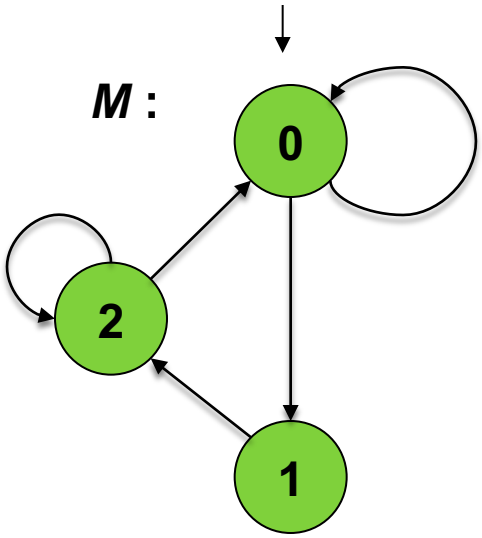
Overview

- CTL
  - CTL
  - Model checking
- Symbolic model checking
- BDD
  - Definition
  - Reducing BDD
  - Operations on BDD

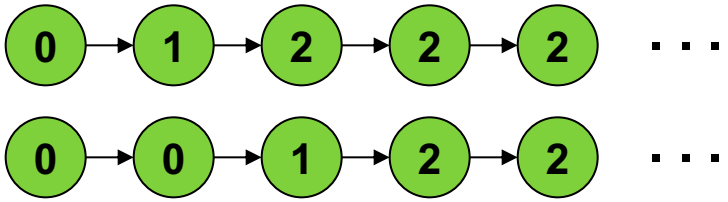
- Acknowledgement: some slides are taken and adapted from various presentations by Randal Bryant (CMU), Marsha Chechik (Toronto)

CTL

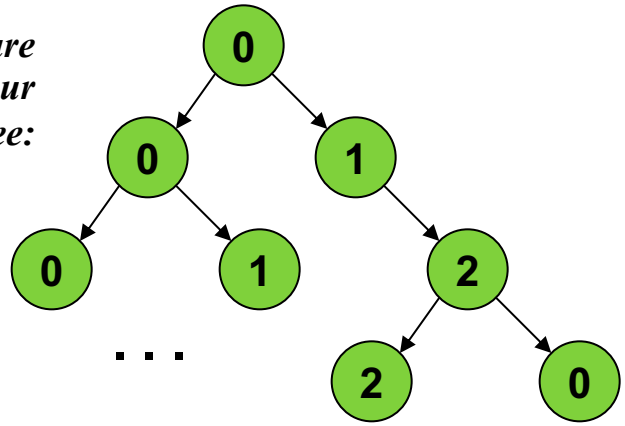
- Stands for *Computation Tree Logic*
- Consider this Kripke structure (labeling omitted) :



In LTL, properties are defined over "executions", which are <u>sequences</u> :



In CTL properties are defined in terms of your computation tree:



CTL

- Informally, CTL is interpreted over computation trees.

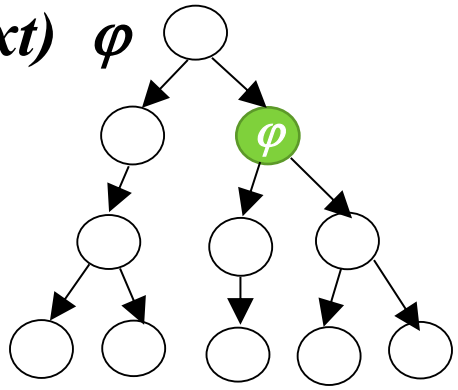
$M \models \varphi$ = M's computation trees satisfies φ

- We have <u>path quantifiers</u> :
  - **A** ... : holds for all path (starting at the tree's root)
  - **E** ... : holds for some path

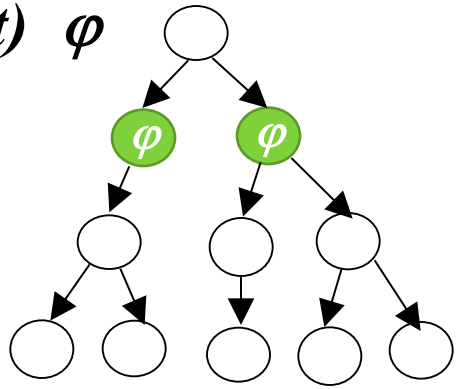
- Temporal operators :
  - **X** ... : holds next time
  - **F** ... : holds in the future
  - **G** ... : always hold
  - **U** : until

Intuition of CTL operators

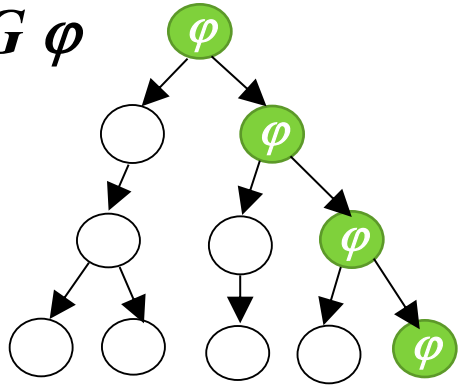
EX (exists next) φ



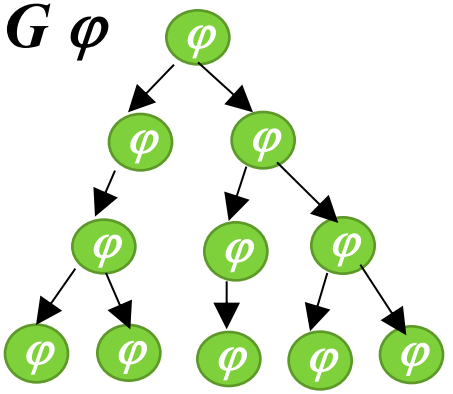
AX (all next) φ



EG φ

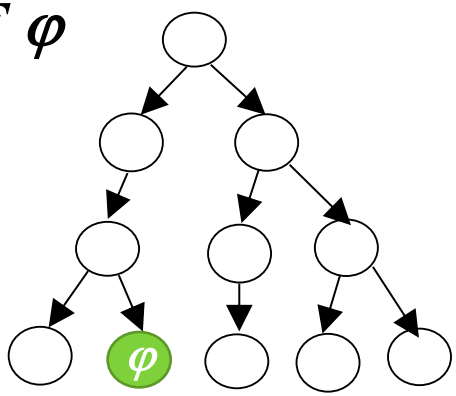


AG φ

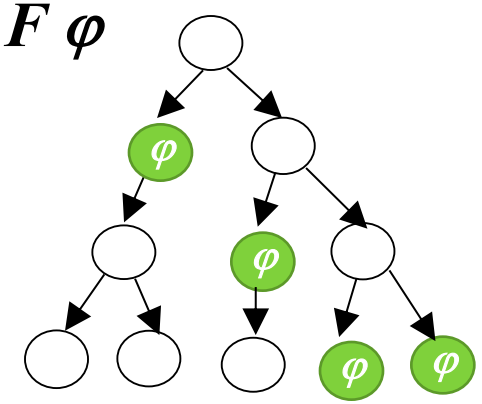


Intuition of CTL operators

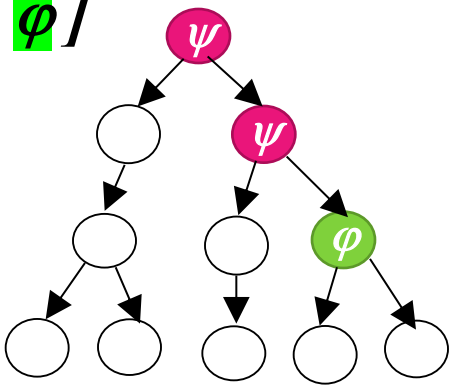
$EF\ \varphi$



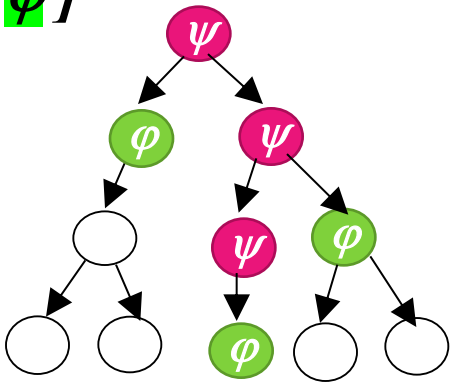
$AF\ \varphi$



$E[\ \psi\ U\ \varphi\ ]$



$A[\ \psi\ U\ \varphi\ ]$



Syntax

φ ::= *p* // atomic (state) proposition

| $\neg\varphi$ | $\varphi_1 \wedge \varphi_2$

| EX φ | AX φ

| E[φ_1 U φ_2] | A[φ_1 U φ_2]

Semantics

$R : S \to \{S\}$: transition relation
$V : S \to \{Prop\}$: observations

- Let $M = (S, \{s_0\}, R, V)$ be a Kripke structure ☺
- $M,t \models \varphi$ φ holds on the comp. tree t
- $M \models \varphi$ is defined as M, **tree**$(s_0) \models \varphi$
- **$M,t \models p$** = $p \in V($**root**$(t))$
- **$M,t \models \neg\varphi$** = not ($M,t \models \varphi$)
- **$M,t \models \varphi \wedge \psi$** = $M,t \models \varphi$ and $M,t \models \psi$

Semantic of "X"

- $M,t \models \mathbf{EX}\varphi \quad = \quad (\ \exists v \in R(\mathbf{root}(t)) :: \quad M,\mathbf{tree}(v) \models \varphi\ )$
- $M,t \models \mathbf{AX}\varphi \quad = \quad (\ \forall v \in R(\mathbf{root}(t)) :: \quad M,\mathbf{tree}(v) \models \varphi\ )$

This definition of the A-quantifier is a bit problematic if you have a terminal state t (state with no successor), because then you get $t \models AX\ \varphi$ for free, for any φ (the above $\forall$-quantification would quantify over an empty domain). This can be patched; but we'll just assume that your M contains no terminal state (all executions are infinite).

Semantic of "U"

- ***M,t*** $\models$ **E[ψ U φ]** =

  There is a path σ in M, starting in **root**(*t*) such that:

  - For some $i \geq 0$, M,**tree**(σ_i) $\models \varphi$
  - For all previous j, $0 \leq j < i$, M,**tree**(σ_j) $\models \psi$

- ***M,s*** $\models$ **A[ψ U φ]** =

  For <u>all</u> path σ in *M*, starting in **root**(*t*), these hold:

Derived operators

- $\psi \vee \varphi \quad = \quad \neg(\neg\varphi \wedge \neg\psi)$
- $\psi \rightarrow \varphi \quad = \quad \neg\psi \vee \varphi$

- $EF\ \varphi \ = \ E[\ true\ \ \mathbf{U}\ \varphi\ ]$
- $AF\ \varphi \ = \ A[\ true\ \ \mathbf{U}\ \varphi\ ]$
- $EG\ \varphi \ = \ \neg\ AF\ \neg\varphi$
- $AG\ \varphi \ = \ \neg\ EF\ \neg\varphi$

LTL vs CTL

- They are not the same.
- Some properties can be expressed in both:

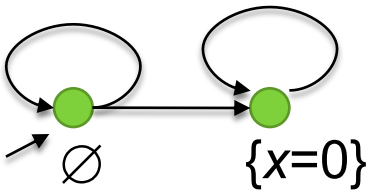
AG (*x*=0) = □(*x*=0)

AF (*x*=0) = ◇ (*x*=0)

A[*x*=0 U *y*=0] = *x*=0 U *y*=0

- Some CTL properties can't be expressed in LTL, e.g:

EF (*x* = 0)

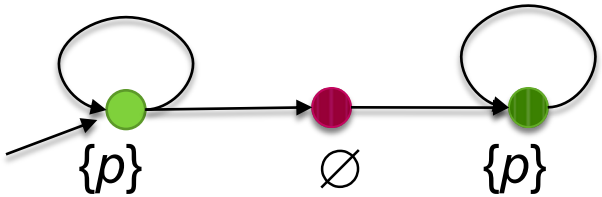


Prop = { *x*= 0 }

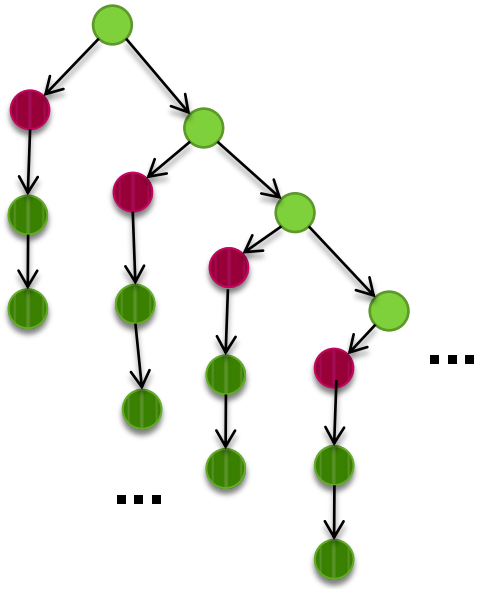
LTL vs CTL

- Some LTL properties cannot be expressed in CTL, e.g.

$\Diamond\Box p$



Prop = { *p* }



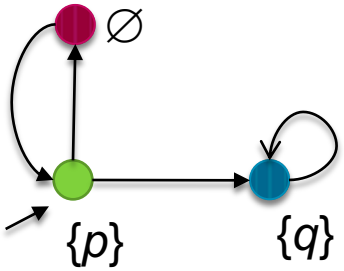
E.g. AF AG *p* does not express the property; the above Kripke does not satisfy it.

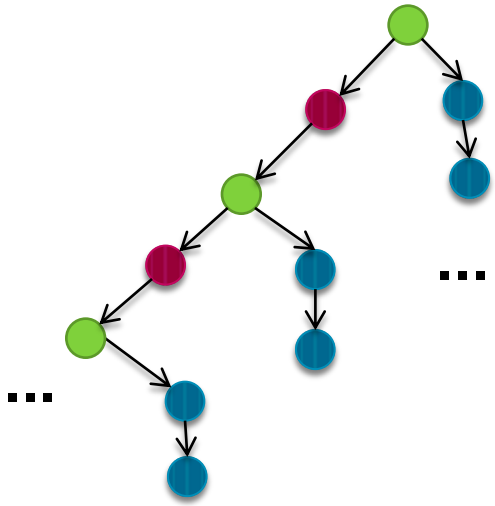
LTL vs CTL

- Another example, fairness restriction:

$$(\Box\Diamond p \rightarrow \Diamond q) \rightarrow \Diamond q$$

$$= \Box\Diamond p \vee \Diamond q$$



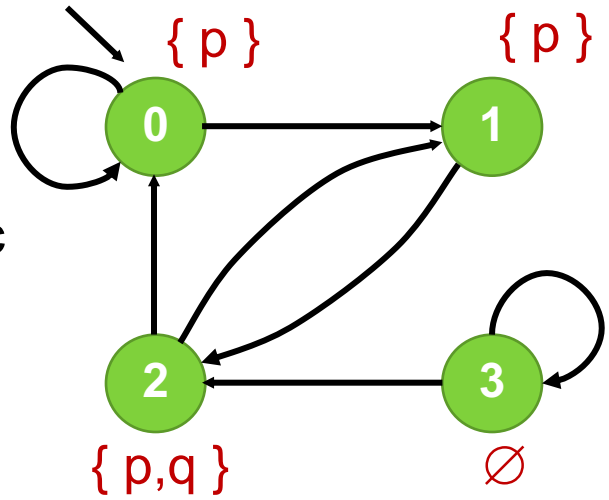


e.g. AGAF $p \vee$ AF q does not hold on the tree.

$(a \rightarrow b) \rightarrow b = \sim(\sim a \vee b) \vee b = (a \wedge \sim b) \vee b = a \vee b$

Model checking CTL formulas

- Kripke M = (S, {s_0}, R, V)
- We want to verify M |== φ
- Assume φ is expressed in CTL's (chosen) basic operators.
- The verification algorithm works by systematic-ally labeling M's states with subformulas of φ; *bottom up*.
- For a sub-formula *f* ; we inspect every state *s*:



If tree(s) |= f , we label s with f (and otherwise we don't label it)

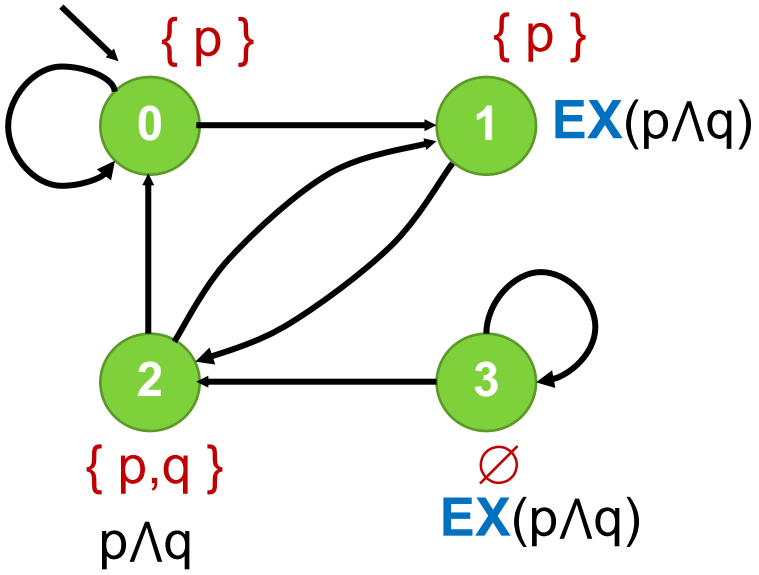
- Eventually, when we are done with the labeling of the root formula φ :

$$M \models \varphi \quad iff \quad s_0 \text{ is labeled with } \varphi$$

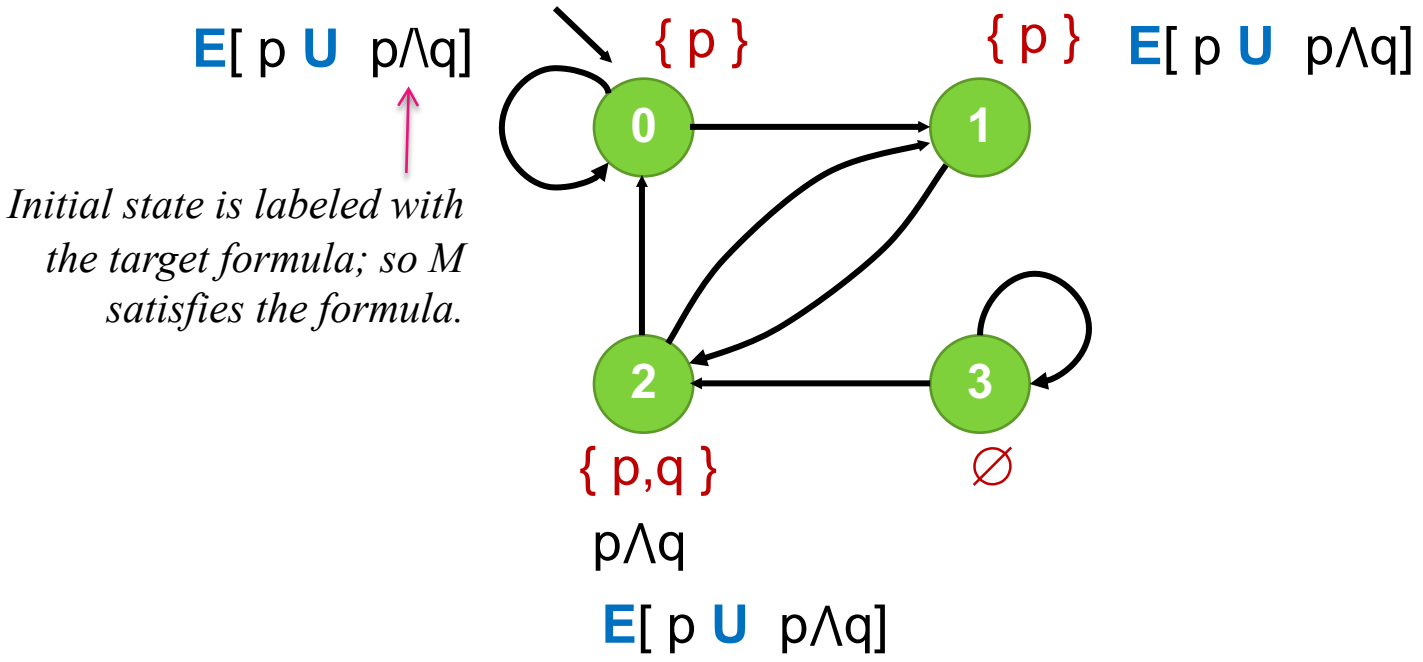
Example, checking **EX**(p∧q)

Prop = {p,q}

Initial state is not labeled with the target formula; so the formula is not valid.

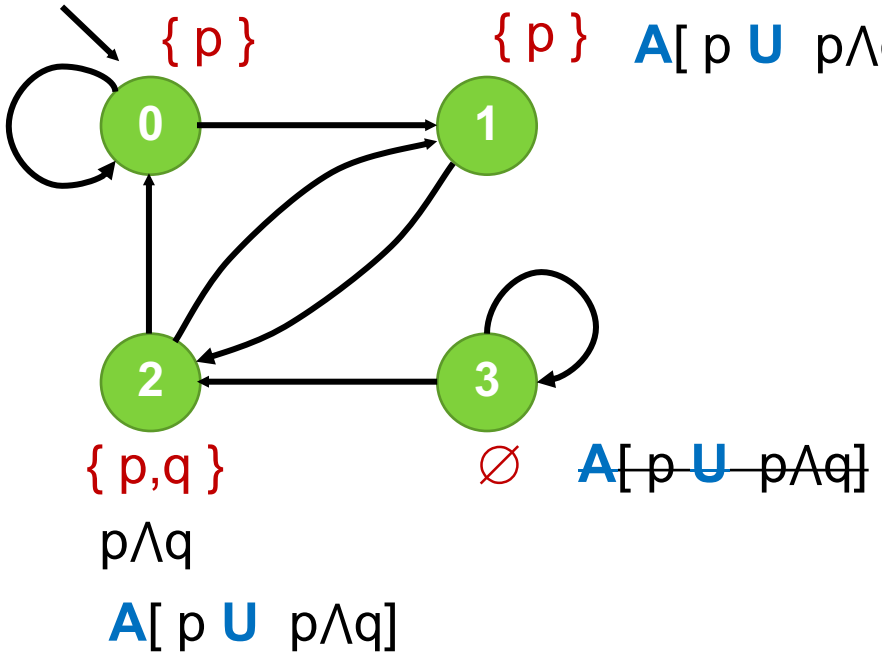


Example, checking: **E**[p **U** (p∧q)]



Example, checking A[p U (p∧q)]

At the end, initial state is <u>not</u> labeled with the target formula; so the formula is not valid

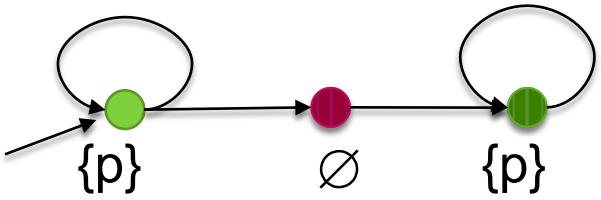


Can we apply this to LTL ?

- Consider ◇□**p**

Prop = { p }

- Applying labeling :



we can't label this with □p;
thus also not with ◇□p

◇□p

□p

◇□p

CTL*

- CTL formulas are also CTL* formulas. LTL formulas can also be expressed in CTL*. Syntax:

  (State formula)

  φ :: p // p is atomic proposition
  | $\neg\varphi$ | $\varphi_1 \wedge \varphi_2$
  | **E** f | **A** f // f is a path formula

  (Path formula)

  f :: φ
  | $\neg f$ | $f \wedge g$ | **X**f | **F**f | **G**f | f_1 **U** f_2

- An example of CTL* formulas that is not CTL: EFG(x=0).

CTL*

- The meaning of state formulas:

  $t \models p$ = p holds at **root**(t)

  $t \models \neg\varphi$ = not $t \models \varphi$

  $t \models \varphi_1 \wedge \varphi_2$ = $t \models \varphi_1$ and $t \models \varphi_2$

  $t \models \mathbf{A}\, f$ = for **all** paths σ starting in **root**(t), $\sigma,0 \models f$

- **E***f* can be defines as $\neg\mathbf{A}\neg f$.

- The meaning of path formulas:

  $\sigma,i \models \varphi$ = **tree**(σ_i) $\models \varphi$

  $\sigma,i \models \neg f$ = not $\sigma,i \models f$

  $\sigma,i \models \mathbf{X}f$ = $\sigma,i+1 \models f$

  the defs. of other operators as in LTL.

Checking CTL*

- Let M/s be the same Kripke structure as M, but s as the initial state.
- **Idea**: we again perform labelling, such that a state $s \in M$ is labelled by some formula ψ if and only if $M/s \models \psi$.
- After the labelling process is finished, $M \models \varphi$ holds if and only if the initial state s_0 is labelled by the original formula φ.

Checking CTL*

- Can we label s with φ? Yes, if M/s $\models \varphi$. In other words, if **tree**(s) $\models \varphi$.

- Case 1: φ is an atomic proposition p: **tree**(s) $\models p$ iff *p* holds at *s*.
- Case-2: labelling with $\neg\varphi$ and $\varphi \wedge \psi$ ... well these should be obvious. We label s with $\neg\varphi$ if it cannot be labelled with φ. We label s with $\varphi \wedge \psi$ if it can be labelled with both φ and ψ.
- Case-3: a single path quantifier followed by a formula *f* that does not contain any path quantifier.
  - Case-3a: **tree**(s) $\models$ **A** *f*
    e.g. **tree**(s) $\models$ **AGF***q* → by checking M/s $\models \Box\Diamond$q in LTL.

  - Case-3b: **tree**(s) $\models$ **E** *f*
    Well, **E** *f* = $\neg$**A** $\neg f$ ... see case-2.

Checking CTL*

- Case-4: a formula φ that contains a subformula f that starts with a path quantifier.

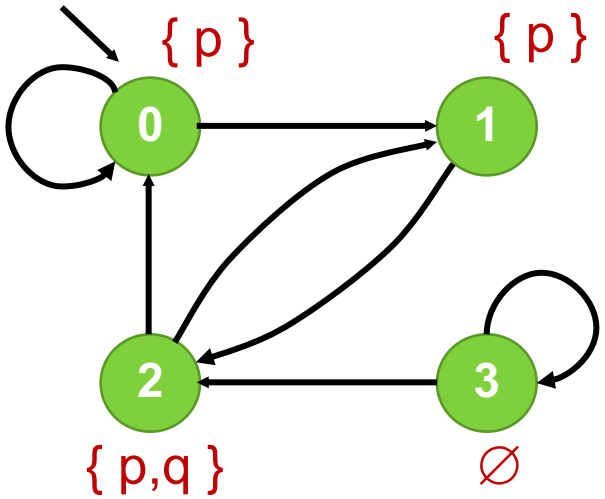
  e.g. tree(s) ⊨ **AGF**(**EF**q)

  1. First recurcively label the states of M with **EF**q.
  2. Pretend **EF**q is a fresh state-predicate r; replace the labelling with EFq with r.
  3. Now check t ⊨ **AGF** *r*.

Symbolic representation

- You need the full state space to do the labeling!
- Idea:
  - Use formulas to encode sets of states (e.g. to express the set of states labeled by something)
  - A small formula can express a large set of states → suggest a potential of space reduction.

Example



4 states, can be encoded by 2 boolean variables x and y.

St-0 ¬x¬y
St-1 ¬xy
St-2 x¬y
St-3 xy

E.g. the set of states where q holds is encoded by the formula:

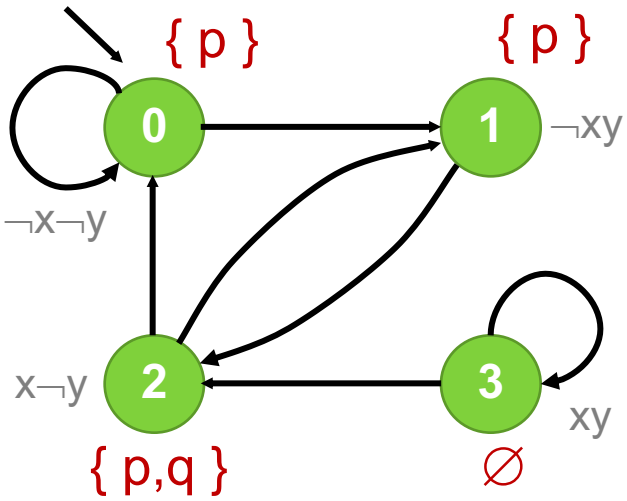
x¬y

Similarly, the set of states where p holds : {0,1,2}, can be encoded by formula:

¬(xy)

More generally, a boolean formula *f* represents the set of states, formed by value-combinations that satisfies *f.*

Example



We can also describe this more program-like:

if state∈{0,2} → goto {0,1}
[] state∈{1,3} → goto 2
[] state=3 → goto {2,3}
fi

N.D.

which can be encoded with this boolean formula:

¬y¬x' ∨ yx'¬y' ∨ xyx'

States encoding:

St-0 ¬x¬y
St-1 ¬xy
St-2 x¬y
St-3 xy

Example

```
byte x ;  // unspecified initial value

if x≠255 →  x=0 ;
```

The automaton has 256 states, with 256 arrows.

- Bit matrix : 8.2 Kbyte
- List of arrows: 510 bytes

With boolean formula:

$$\neg(x_0..x_7) \wedge \neg x'_0 ... \neg x'_7$$

Model checking

- When we label states with a formula f, we are basically calculating the set of states (of M) that satisfy f.

- Introduce this notation:

W_f = the set of states (whose comp. trees) satisfy f

= { s | $s \in S$, M, tree(s) |== f }

- We now encode W_f as as a boolean "*formula*"

M |= f if and only if W_f evaluated on s_0 returns true

Labeling

- If p is an atomic formula:

W_p = boolean formula representing the set of states where p holds.

- For conjunction: $W_{f/\backslash g} = W_f \wedge W_g$
- Negation: $W_{\neg f} = \neg W_f$

- For EX: $W_{EXf} = \exists x',y' :: R \wedge W_f[x',y'/x,y]$

(The relation R is assumed to be defined in terms of x,y and x',y')

- AX f = $\neg EX \neg f$ So: $W_{AXf} = \neg W_{EX\neg f}$

Restricting the arrows over the destinations

States encoding:

St-0 $\neg x \neg y$
St-1 $\neg x y$
St-2 $x \neg y$
St-3 $x y$

Suppose we have these arrows, R = $\{1,3\} \rightarrow \{2\}$, $\{3\} \rightarrow \{1,3\}$

$$y\, x' \neg y' \;\vee\; x y y'$$

The set U of all states that has **at least one** outgoing arrow to one of {2,3}

$$\{\, s \mid (\exists\, t ::\; t \in R(s) \wedge t \in \{2,3\}) \,\}$$

Encoding U in Boolean formula:

$$(\exists x', y' ::\; (\, y\, x' \neg y' \;\vee\; x y y' \,) \wedge x')$$

Restricting the arrows over the destinations

The set U of all states whose **all** outgoing arrows go to one of {2,3} :

$$\{ s \mid (\forall t:: t \in R(s) \Rightarrow t \in \{2,3\}) \}$$

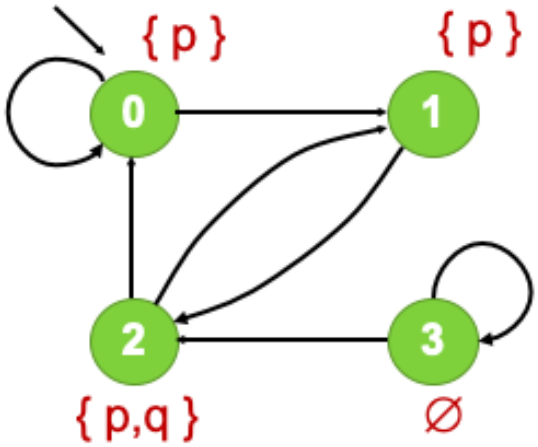
Encoding U in Boolean formula :

$$(\forall x', y' :: R(x,y,x',y') \Rightarrow x')$$

Note:

- Some state-encoding may be invalid (e.g. what if state 0 does not exist?) Some care need to be taken, not to quantify over invalid states.
- In the $\forall$ example, all terminal states in M will automatically be included in the set U … weird, but we discussed this before. We assumed M does not contain terminals.

Example



States encoding:

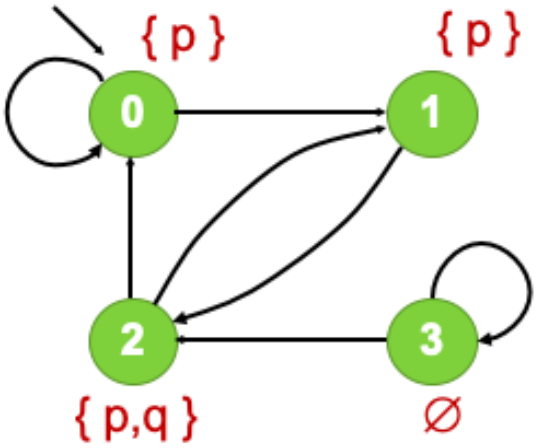
St-0	$\neg x \neg y$
St-1	$\neg x y$
St-2	$x \neg y$
St-3	$x y$

- $W_p = \neg(xy)$
- $W_q = x\neg y$
- $W_{p/\backslash q} = W_p \,/\backslash\, W_q = \neg(xy) \,/\backslash\, x\neg y$

$$R$$

- $W_{\mathbf{EX}(p/\backslash q)} = \exists x',y' :: \neg y\neg x' \;\backslash/\; yx'\neg y' \;\backslash/\; xyx' \;/\backslash\; W_{p/\backslash q}[x',y'/x,y]$
  $= \exists x',y' :: \neg y\neg x' \;\backslash/\; yx'\neg y' \;\backslash/\; xyx' \;/\backslash\; \neg(x'y') \,/\backslash\, x'\neg y'$

Another Example, **AX**p/\¬q



States encoding:

St-0	¬x¬y
St-1	¬xy
St-2	x¬y
St-3	xy

$W_{p\wedge\neg q} = \neg x$

$W_{\mathbf{AX}p\wedge\neg q}$

$= \forall x',y' :: R \Rightarrow W_{p\wedge\neg q}[x',y' / x,y]$

$= \forall x',y' :: (\neg y\neg x' \vee yx'\neg y' \vee xyx') \Rightarrow \neg x'$

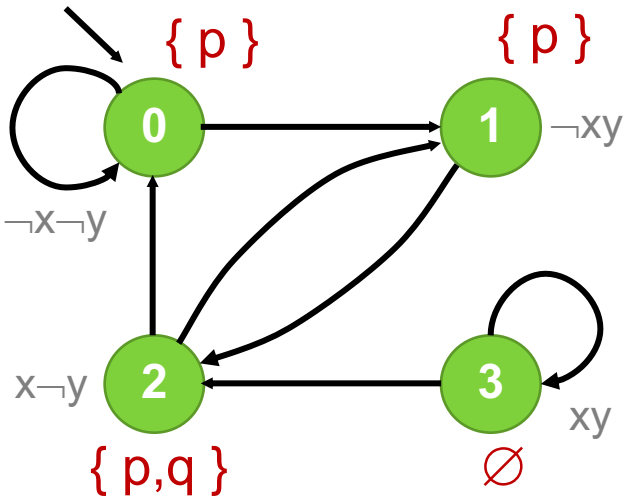
Labeling

- E.g. the states satisfying **E**[f **U** g] can be computed by:
  - Let $Z_1 = W_{\mathbf{g}}$
  - Iteratively construct Z_i , for $i \geq 1$:

$$Z_{i+1} = Z_i \vee (W_{\mathbf{f}} \wedge (\exists x', y' :: R \wedge Z_i[x', y'/x, y]))$$

  - Stop when $Z_{i+1} = Z_i$ (we mean semantical equality!); then $W_{\mathbf{E[p\ U\ q]}} = Z_i$

Example, **E**[p **U** q]



$Z_1 = W_q = x\neg y$

$Z_2 = Z_1 \vee (W_p \wedge (\exists x',y' :: R \wedge Z_1[x',y'/x,y]))$

$x\neg y \vee (\neg(xy) \wedge (\exists x',y' :: \ldots \wedge x'\neg y'))$

States encoding:

St-0 ¬x¬y
St-1 ¬xy
St-2 x¬y
St-3 xy

- $Z_3 = \ldots$

Till fix point.

But how to efficiently do this fix point iteration ?

- Firstly, we need a way to efficiently check the equivalence of two boolean formulas: $f \leftrightarrow g$
  So, we can decide when to we have reached a fix-point.
- In general this is an NP-hard problem.
- We can use a SAT-solver to check if $\neg(f \leftrightarrow g)$ is unsatisfiable.

- But **in addition** to that, your formulas grow as we iterate over the fix point! We also need a **space efficient** way represent them.
- We'll discusss BDD approach

Canonical representation

- = standard form.
- Here, a canonical representation C_f of a formula f is a representation such that:

$$f \leftrightarrow g \quad \text{iff} \quad C_f = C_g$$

- Gives us a way to check equivalence.
- Only useful if the cost of constructing C_f, C_g + checking $C_f = C_g$ is cheaper than directly checking $f \leftrightarrow g$.
- Some possibilities:
  - Truth table → exponentially large.
  - DNF/CNF → can also be exponentially large.

BDD

- *Binary Decision Diagram*; a compact, and canonical representation of a boolean formula.
- Can be constructed and combined efficiently.
- Invented by Bryant:

  "Graph-Based Algorithms for Boolean Function Manipulation". Bryant, in IEEE Transactions on Computers, C-35(8),1986.

Decision Tree

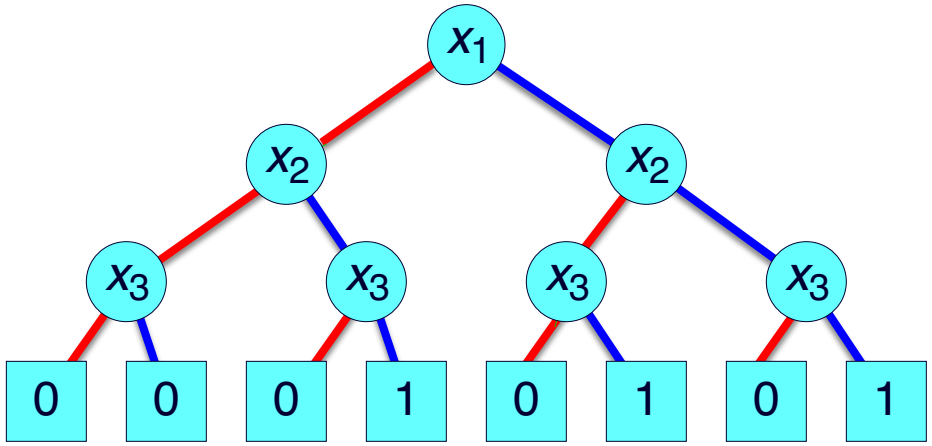
$$\neg x_1\, x_2\, x_3 \;\vee\; x_1 \neg x_2\, x_3 \;\vee\; x_1\, x_2\, x_3$$

with truth table :

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

TT is canonical if we fix the order of the columns.

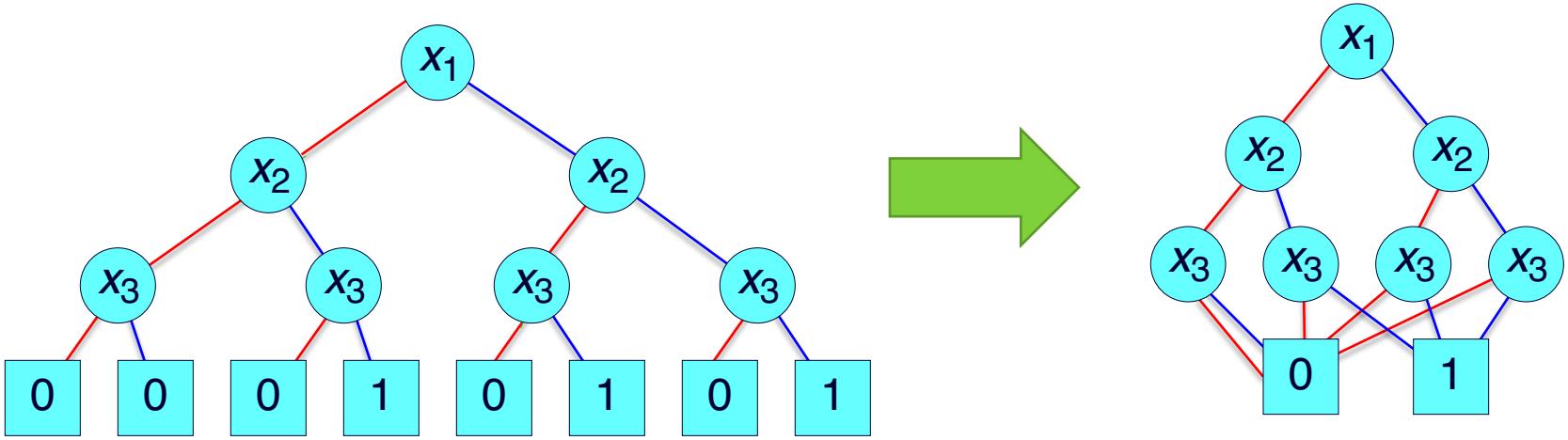
Or representing the table with a (binary decision) tree :



- Each node x_i represents a decision:
  - **Blue** out-edge from x_i → assigning 1 to x_i
  - **Red** out-edge from x_i → assigning 0 to x_i
- Function value is determined by leaf value.

But we can compact the tree…

E.g. by merging the duplicate leaves:



We can compact this further by merging duplicate subgraphs …

Results

Word Size	Gates	Patterns	CPU Minutes	*A=B* Graph
4	52	1.6×10^{4}	1.1	197
8	123	4.2×10^{6}	2.3	377
16	227	2.7×10^{11}	6.3	737
32	473	1.2×10^{21}	22.8	1457
64	927	2.2×10^{40}	95.8	2897

Table 2.ALU Verification Examples

Note: this is from Bryant's paper in 1986. They use their version of MC at that time, running it on an DEC VAX 11/780, with about 1 MIP speed ☺

Boolean formula

- A boolean formula (proposition logic formula) e.g. `x . y \/ z` can be seen as a function :

f(x,y,z) = x.y \/ z

- In Bryant's paper this is called a : boolean function.

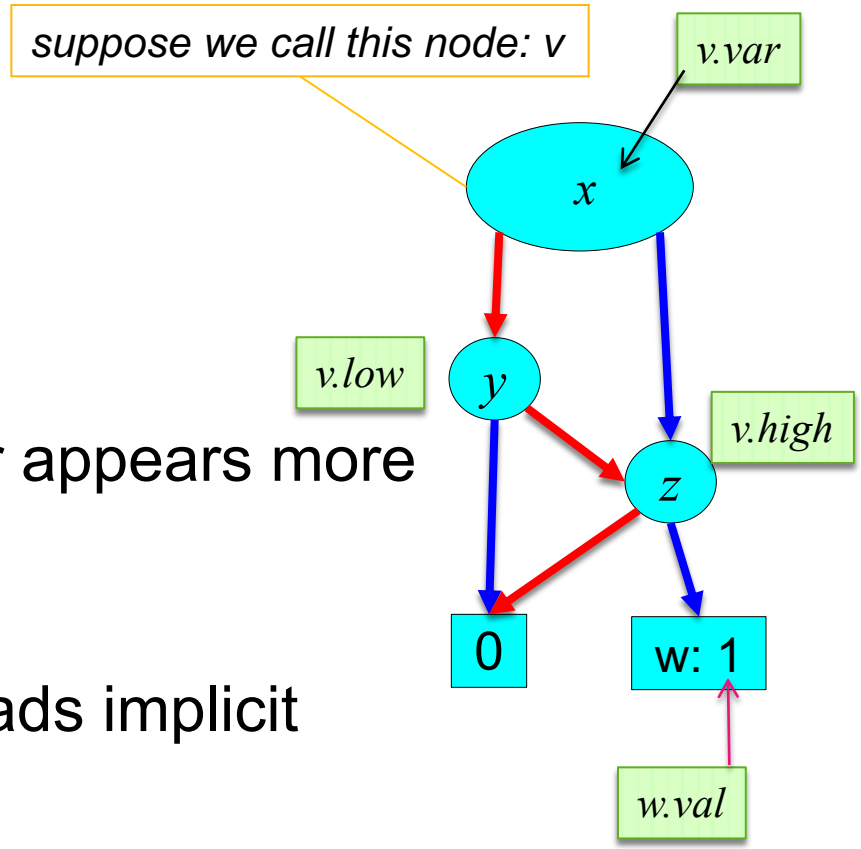
- E.g. 'composing' functions as in

"f(x, y, g(x,y,z))"

is the same as the corresponding substitution.

Binary Decision Diagram

- A BDD is a directed acyclic graph, with
  - a single root
  - two 'leaves' → 0/1
  - non-leaf node
    - labeled with 'varname'
    - has 2 children
- Along every path, no var appears more than 1x
- We'll keep the arrow-heads implicit
  - always from top to bottom

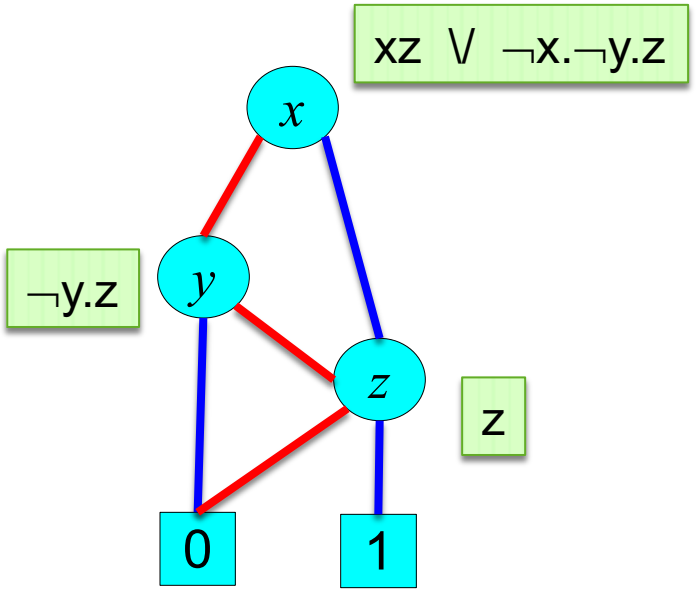


func(G)

x = var(v)

- func(v) = ¬x . func(v.low) ∨ x . f(v.high)

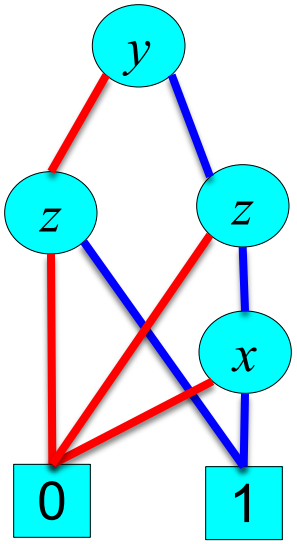
func(G) = func(root)



func(0) = 0, func(1) = 1

Ordered BDD

- OBDD → fix an ordering on the variables
  - let index(v) → the order of v in this ordering ☺
  - *index(v) < index(v.low)*
  - *same with v.high*



satisfies ordering [y,z,x] but not [y,x,z]

Reduced BDD

- Two BDDS F ang G are *isomorphic* if you can obtain G from F by renaming F's nodes, vice versa.

  But you are not allowed to rename v.var nor v.val !

  then: func(F) = func(G)

- A BDD G is *reduced* if:
  - for any non-leaf node v, v.low ≠ v.high.
  - for any distinct nodes u and v, the sub-BDDs rooted at them are not isomorphic.

otherwise G can be reduced!

Reduced OBDD

- Reduced OBDD is canonical:

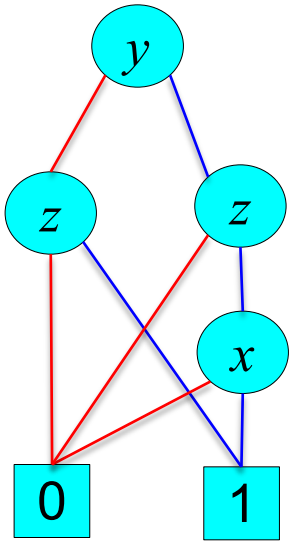
If we fix the variable ordering, every boolean function is uniquely represented by a reduced OBDD (up to isomorphism).

- Same idea as in truth tables: canonical if you fix the order of the columns.

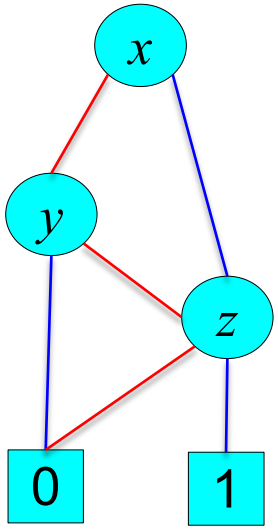
- However, the chosen ordering may influence the size of the OBDD.

Effect of ordering

Consider: xyz ∨ ¬yz



Order: y,z,x

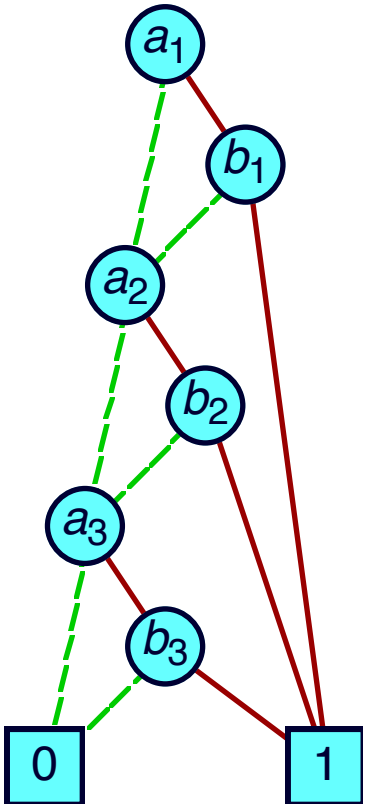


Order: x,y,z

xyz ∨ ~yz = (xy ∨ ~y)z = (x ∨ ~y)z = (x ∨ ~x~y)z = xz ∨ ~x~yz

The difference can be huge…

consider: $a_1b_1 \vee a_2b_2 \vee a_3b_3$

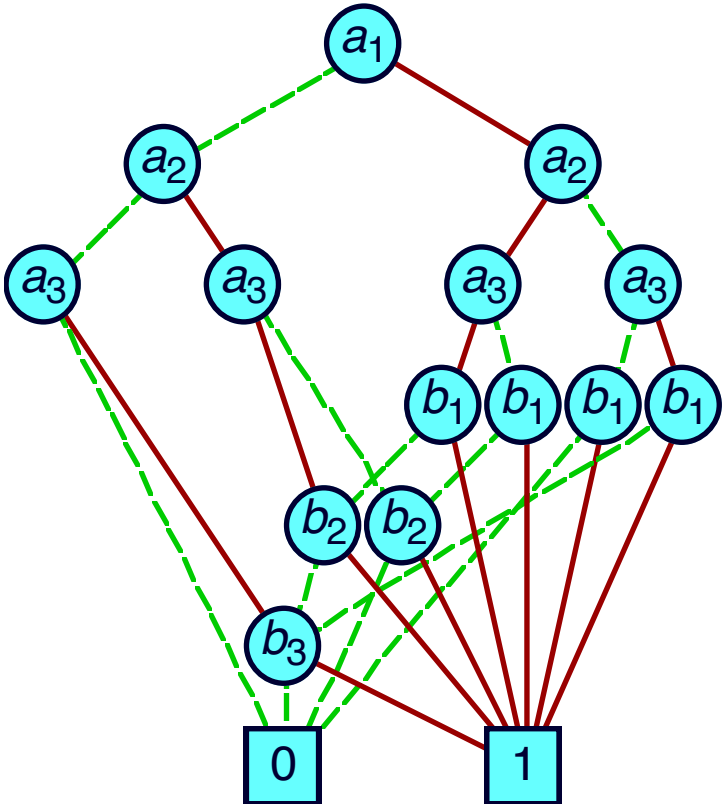


Linear Growth

In these pictures:

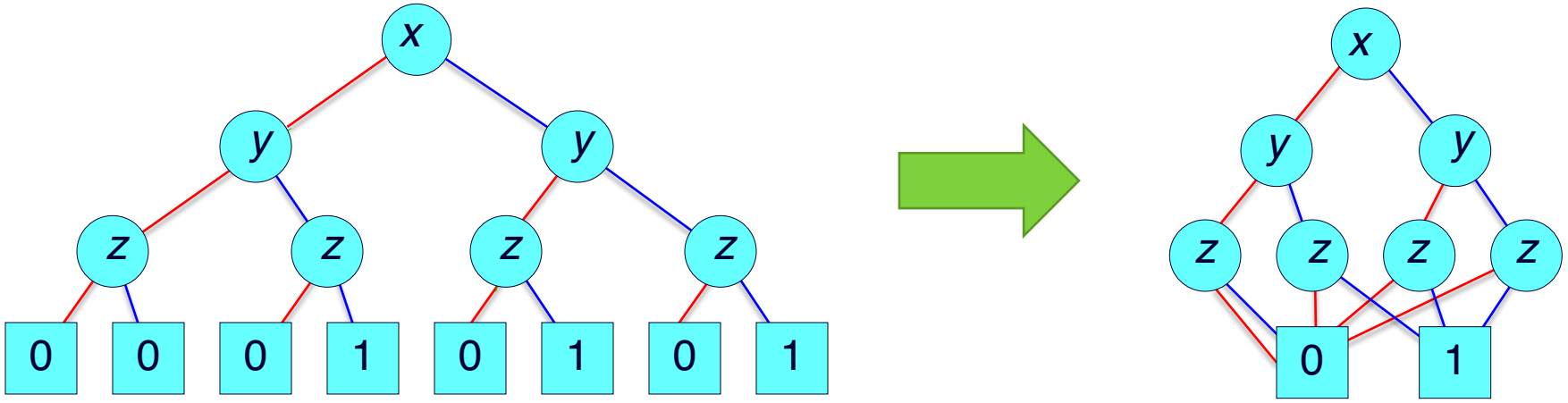
(red) : true/1

(green) : false/0



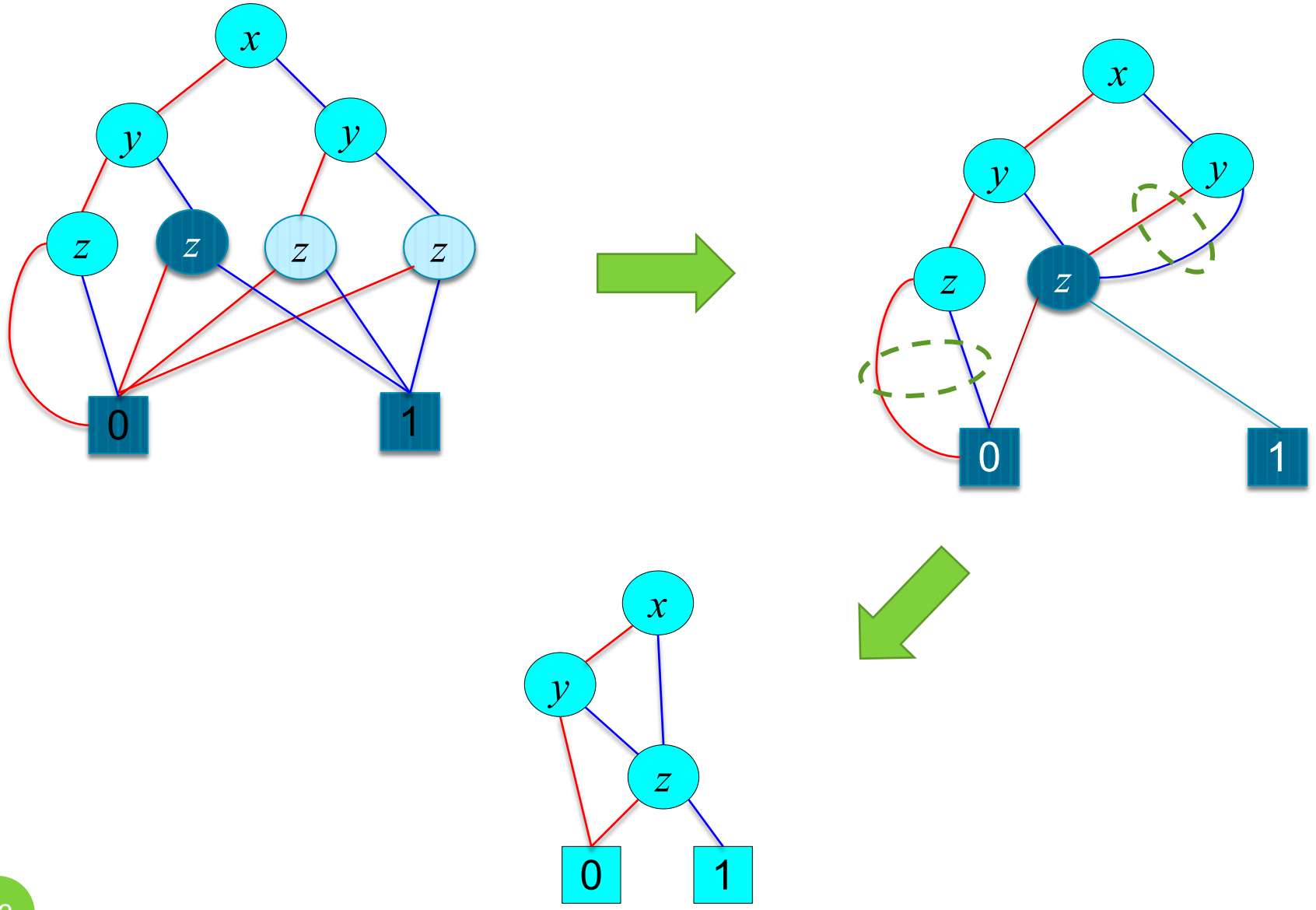
Exponential Growth

Reducing BDD



By sharing leaves…

Reducing BDD



The reduction algorithm

- Introduce a “class” Node to represent the nodes in a BDD.
- If v is an instance of Node and it has these fields
  - v.var (non-leaf)
  - v.low , v.high : Node (non-leaf)
  - v.value (leaf)

- A BDD can be represented by a single instance of Node, which is to be the root of the BDD.

The reduction algorithm

- Introduce ***id***, mapping/function Node → Node

  Use it to keep track which nodes actually represent the same formula.

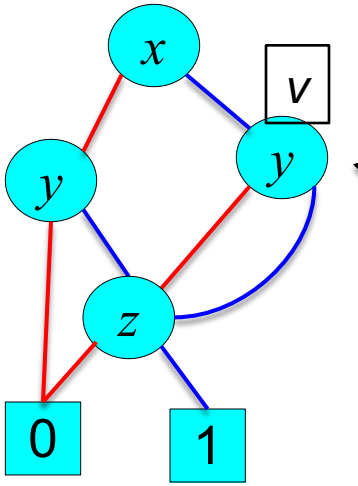
  Iterate/recurse and maintain this invariant:

  $$func(u) = func(id(u))$$

- Initially, $id(u) = u$, for all u.
- So, we can remove u from the graph, and re-route arrows to it, to go to id(u) instead.
- Work bottom up, and such that a node decorated with x is processed after all nodes whose decorations come later in the var-ordering are processed first.

The reduction algorithm

- We'll work recursively, bottom-up.
- Keep track of V = set of visited/processed nodes.
- Now suppose we are at a node v labeled with y, index(y)=i.
- Note that this implies that we must have done the work for all non-leaves labeled with z with index(z)>i.
  - **Case-1**, id(v.low) = id(v.high) ; suppose v.var = "y"



func(v) = ¬y . func(v.low) \/ y . func(v.high)

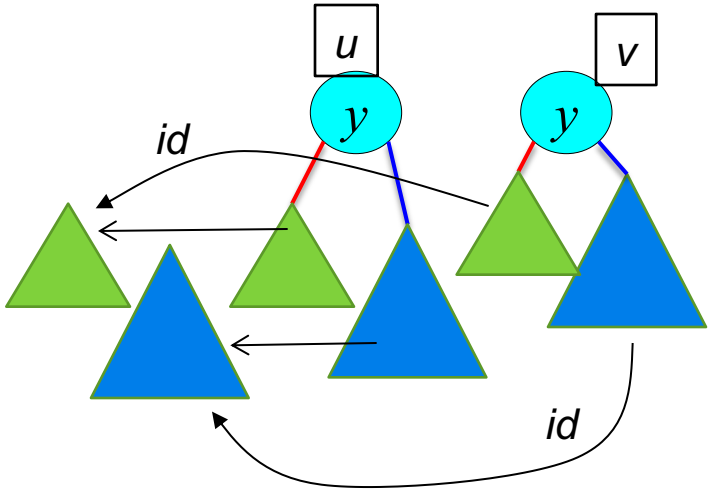
= ¬y . func(id(v.low)) \/ y . func(id(v.high))
= ¬y . func(id(v.low)) \/ y . func(id(v.low))
= func(id(v.low))

• *update:* ***id(v)*** *:= id(v.low)*

The reduction algorithm

- **Case-2:** there is another non-leaf u∈V (u has been processed) such that:

  1. u.var = v.var ; suppose this is “y”
  2. id(u.low) = id(v.low)
  3. id(u.high) = id(v.high)




```
func(v)  =  ¬y . func(u.low)  \/  y . func(v.high)
         =  ¬y . func(u.low)  \/  y . func(u.high)   // by inv
         =   func(u)
         =   func(id(u))
```

- *update:* ***id(v)*** *:= id(u)*

The reduction algorithm

- **Case 3** : otherwise we can't reduce v , if v is not a leaf, update :
  - v.high := id(v.high)
  - v.low := id(v.low)
- We are done if we have processed the root Node; return the graph induced by id(v), rooted at id(root), as the new graph.
- Orphan nodes can be garabage-collected.

Building a BDD

- So far: we can reduce a BDD.
- Recall in CTL model checking, e.g. to the set of states satisfying **EX** p is calculated by constructing this formula:

$$\exists x', y' :: \; R \; /\backslash \; W_{\mathbf{p}} [x', y' / x, y]$$

Since formulas are now represented as BDDs, this implies the need to combine BDDs.

- The combinators should be efficient!

Basic operations to combine BDDs

- *Apply* f_1 <op> f_2
- *Restrict* $f|_{x=b}$ *// b is constant*
- *Compose* $f_1|_{x=f2}$ *// f2 is another function*

- *Satisfy-one*

Return a single combination of the variables of f that would make it true, else return nothing.

Quantification

- With restriction we can encodes boolean quantifications :

$$(\exists y::\ f(x,y)\ ) \quad = \quad f(x,y)\mid_{y=0} \ \vee \ f(x,y)\mid_{y=1}$$

$$(\forall y::\ f(x,y)\ ) \quad = \quad \neg\ (\exists\ y::\ \neg\ f(x,y))$$

(Recall that we need this in the MC algorithm).

Restriction

- $f(x,y,z)\,|_{y=c}$ how to construct the BDD of the new function??

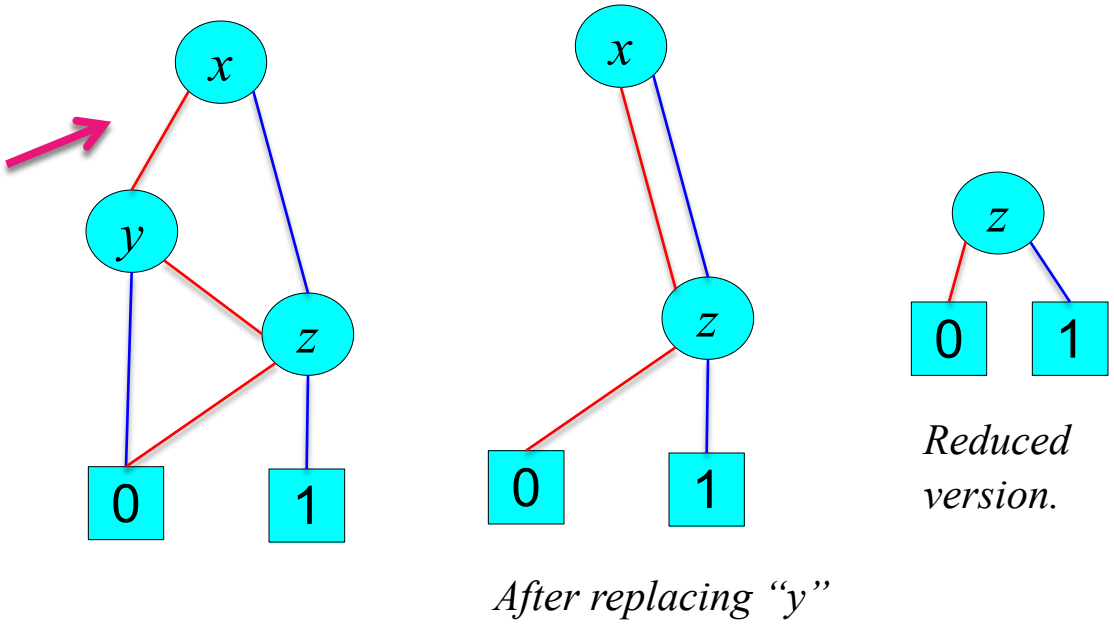
$f(x,y,z)\,|_{y=0}$ → replace all y nodes by low-sub-tree

$f(x,y,z)\,|_{y=1}$ → replace all y nodes by high-sub-tree

Example:

$f(x,y.z) = xz \vee \neg x \neg yz$

So, $f(x,y,z)\,|_{y=0} = z$



Apply

- "Apply", denoted by **f <op> g** , means the boolean function obtained by applying op to f and g.

  E.g. assuming they take x,y as parameters, f <and> g means the function that maps x,y to f(x,y) /\ g(x,y).

  - A single algorithm to implement /\, \/, xor
  - We can even implement ¬**f** , namely as f <xor> 1

Apply

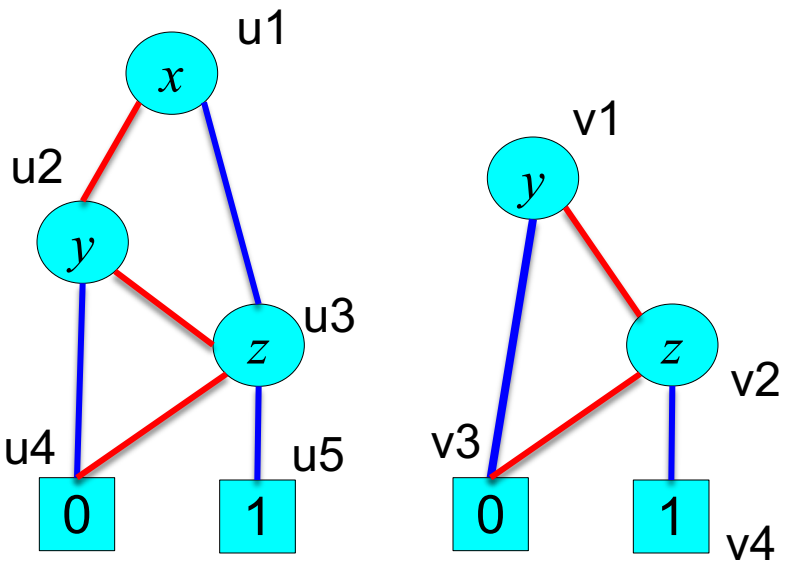
- So, given the BDDs of f and g, how to construct the BDD of f <op> g ?
- There is this '*Shannon expansion*' :

$$f \langle op \rangle g = \neg x \,.\, (f|_{x=0} \langle op \rangle g|_{x=0}) \;\vee\; x \,.\, (f|_{x=1} \langle op \rangle g|_{x=1})$$

- This tells us how to implement "apply" recursively !

  Detail, see LN.

Example

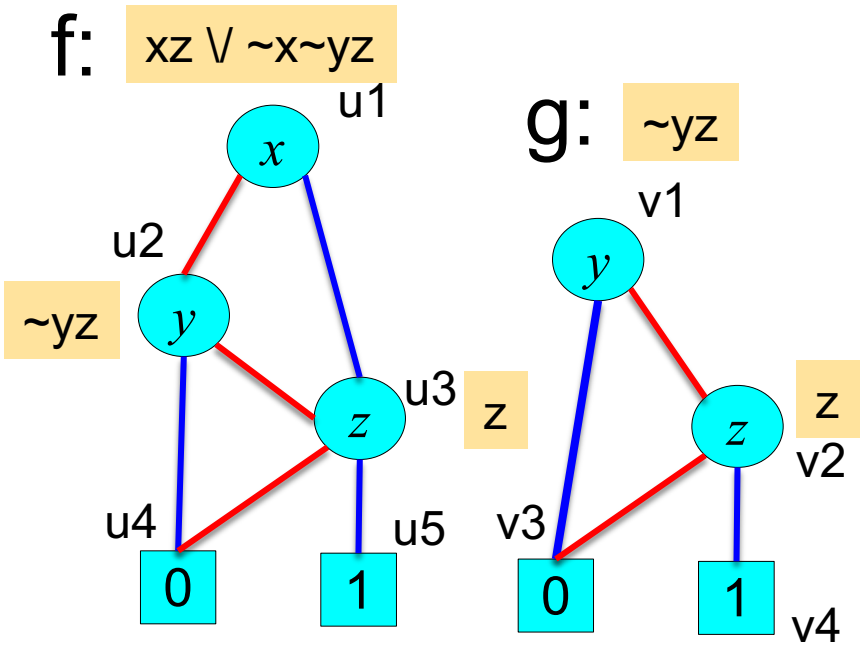


We'll do this by hand.

We name the nodes, just so that we can refer to them.

$$f \langle and \rangle g = \neg x \,.\, (f|_{x=0} \langle and \rangle g|_{x=0}) \;\vee\; x \,.\, (f|_{x=1} \langle and \rangle g|_{x=1})$$

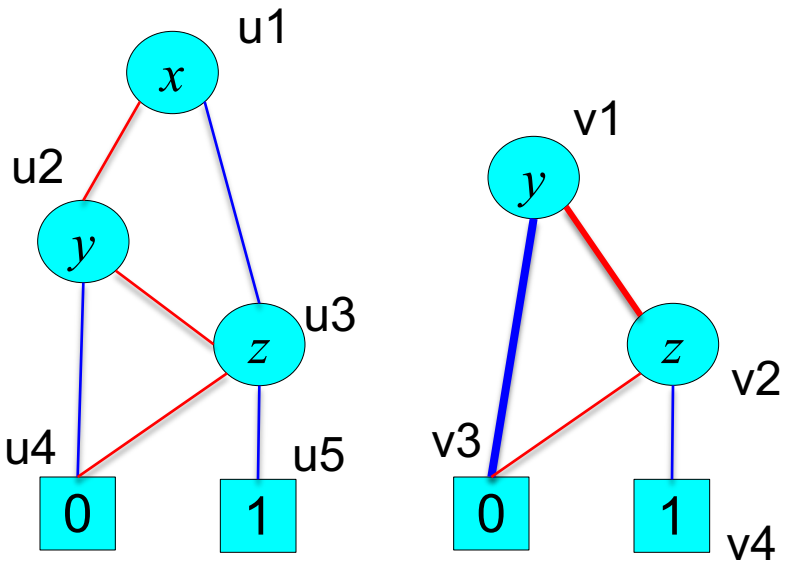
Example

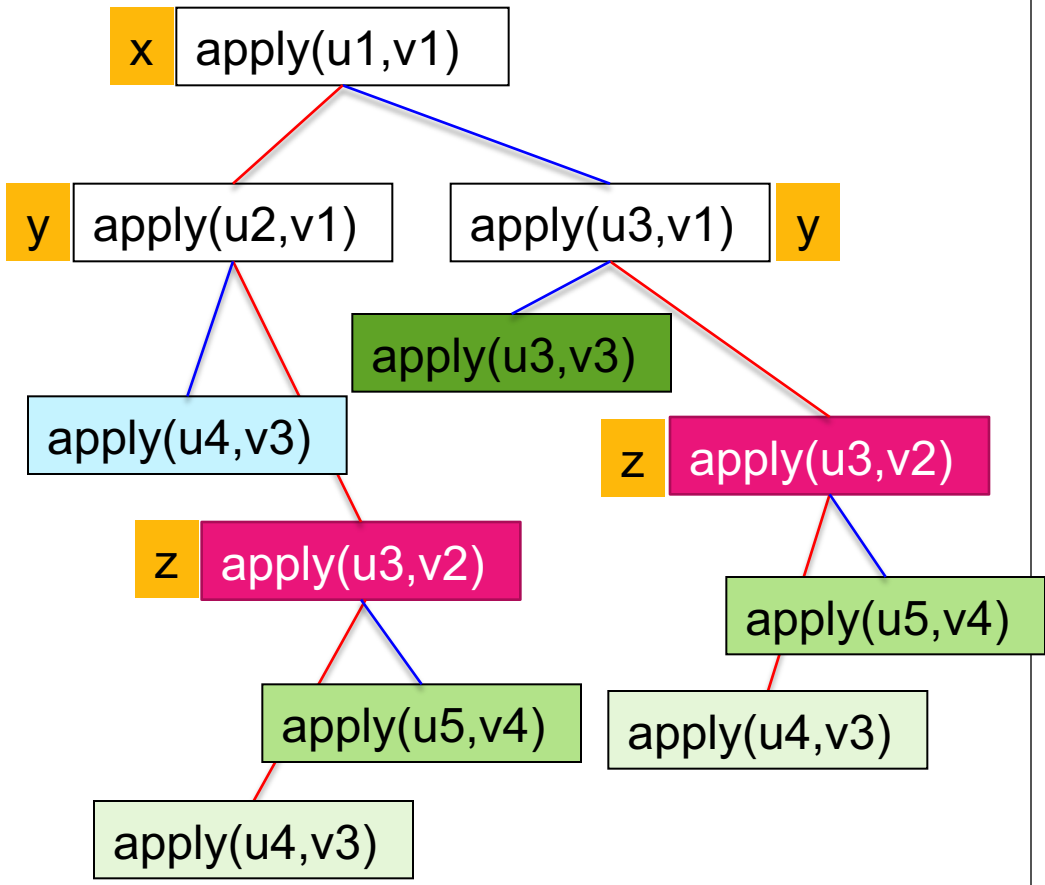


f /\ g
=
¬x . (f |x=0 /\ g |x=0) \/ x . (f |x=1 /\ g |x=1)
=
¬x (~yz /\ g) \/ x(z /\ g)
=
....

f <and> g
=
¬x . (f |x=0 <and> g |x=0) \/ x . (f |x=1 <and> g |x=1)

Example





Repeated call in recursion! To avoid this, maintain a table to keep track of already computed results.

Satisfy and Compose

- Compose, constructed through :

  $f_1|_{x=f2} = f_2 \cdot f_1|_{x=1} \vee \neg f_2 \cdot f_1|_{x=0}$

- In a reduced graph of a satisfiable formula, every non-terminal node must have both leaf-0 and leaf-1 as decendants.

  It follows that satisfy-one can be implemented in O(n) time (e.g. run a DFS from root to find a path leading to 1)

And substitution…

- Recall in CTL model checking, e.g. to the set of states satisfying **EX** p is calculated by constructing this formula:

$$\exists x', y' ::\ R\ \wedge\ W_p[x', y'/x, y]$$

So, how to we construct the BDD representing e.g. f[x',y'/x,y] ?

- Just replace x,y in the BDD with x',y', assuming this does not violate the BDD's ordering constraint (e.g. if x<y but x'>y'). Else use compose.

The cost of various operations

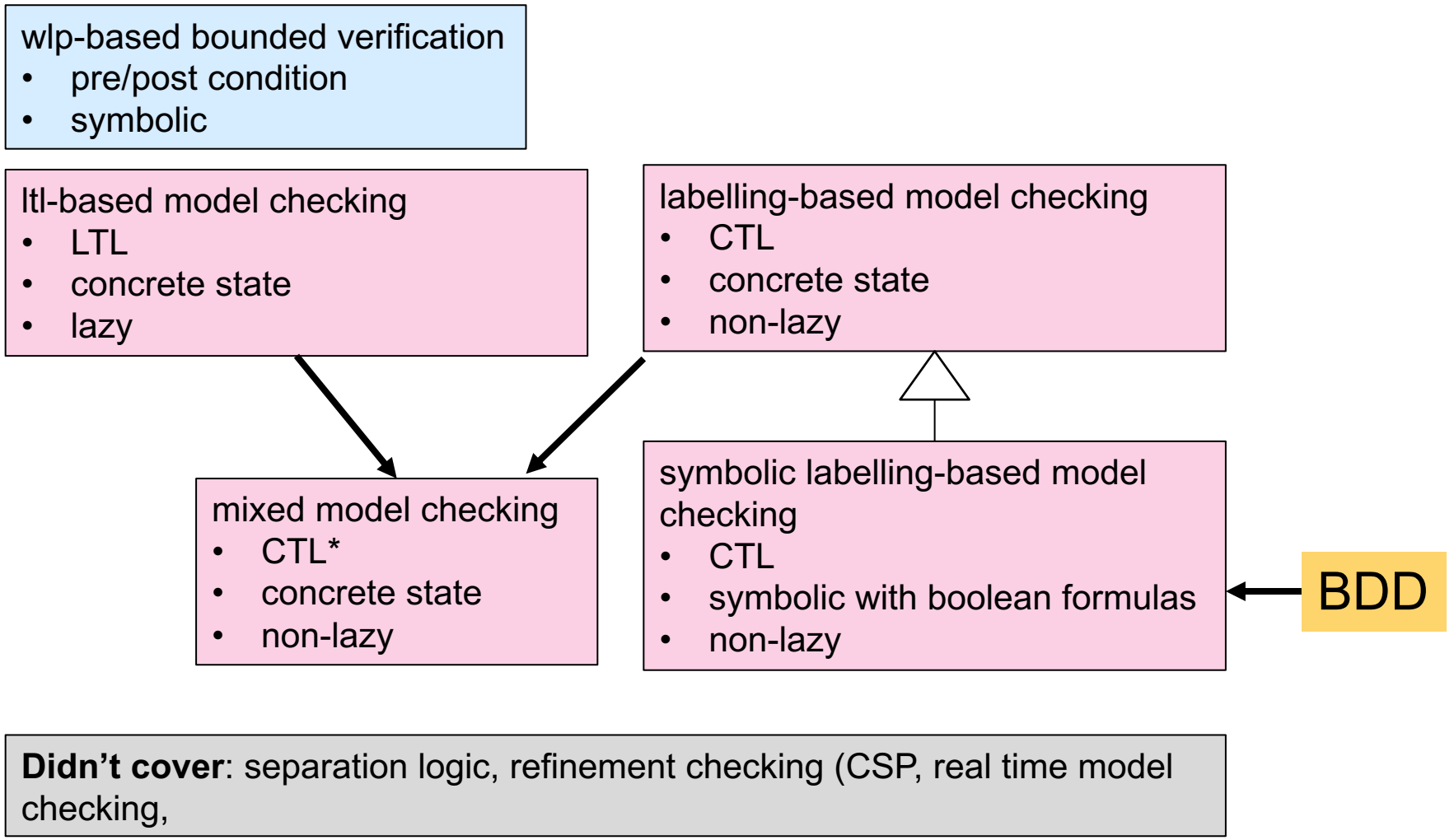
- *Reduce f* — $O(|G| \times \log|G|)$

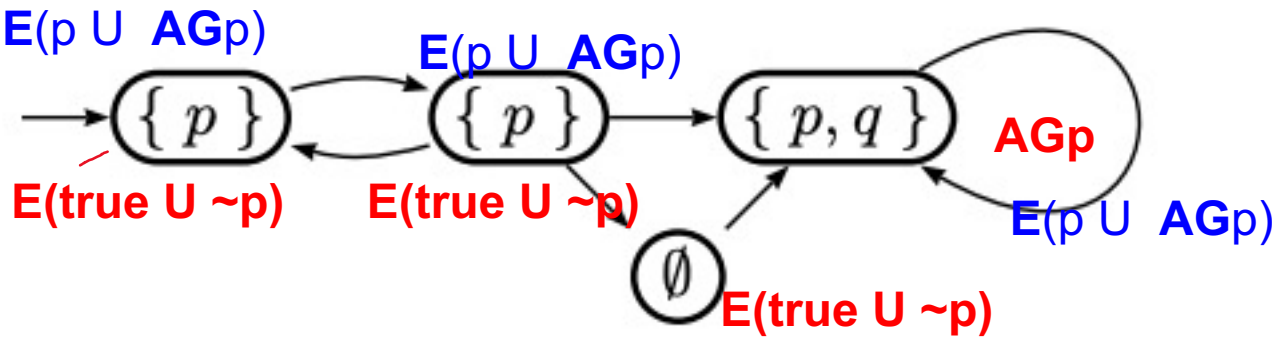
  where G is the graph of f's BDD.

- *Apply* f_1 <op> f_2 — $O(|G1| \times |G2|)$
- *Restrict* $f|_{x=b}$ — $O(|G| \times \log|G|)$
- *Compose* $f_1|_{x=f2}$ — $O(|G1|^2 \times |G2|)$

- *Satisfy-one* — $O(|G|)$

Overview of all verification algorithms we learned





Prop = {p, q}

(a) Do LTL model checking to verify the LTL property $\Diamond\Box p$.

(b) Can the above property be expressed in CTL? How about in CTL*?

(c) Do CTL model checking to verify $\mathbf{EF}(p \wedge q)$.

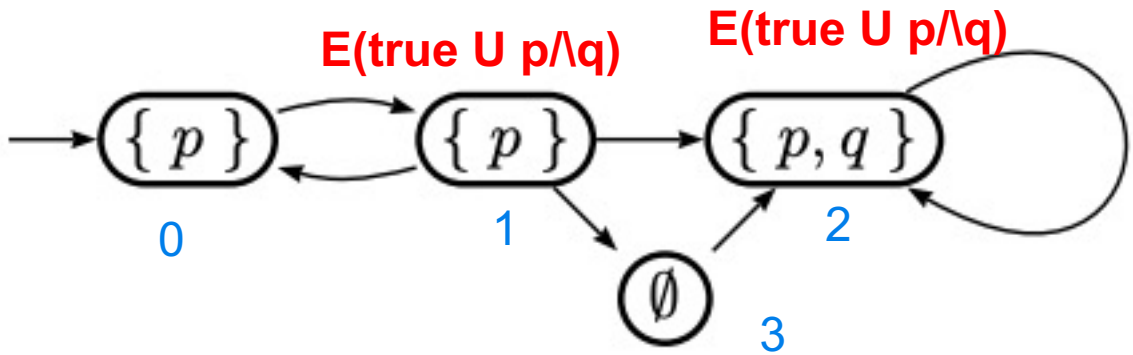
(d) Ok, now try these properties:

- $\mathbf{EF}\ \neg p$
- $\mathbf{AG}\ p$
- $\mathbf{E}[p\ \mathbf{U}\ (\mathbf{AG}\ p)]$
- $\mathbf{A}[p\ \mathbf{U}\ (\mathbf{AG}\ p)]$
- $\mathbf{AFAG}\ p$

E(p U AGp)

AGp
=
~EF~p
=
~E(true U ~p)

0: ~x~y
1: ~xy
2: x~y
3: xy



R(x,y,x',y') =
~xy . ~(~x'y')
\/ x x'~y'
\/ ~x~y ~x'y'

EF(p /\ q) = E(true U p/\q)
$W_{p/\q}$ = x~y
$W_{\mathbf{EF}(p /\ q)}$ = ??
Z1 = $W_{p/\q}$
Z2 = Z1 \/ (true /\ (Exists x',y' :: R(x,y,x',y) /\ Z1))
Z3 = Z2 \/ (true /\ (Exists x',y' :: R(x,y,x',y) /\ Z2))
....
Z_k+1 = Z_k

Suppose L(0) = {p,q}
W_(p/\q) = { 0, 2 } = ~x~y \/ x~y = ~y