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

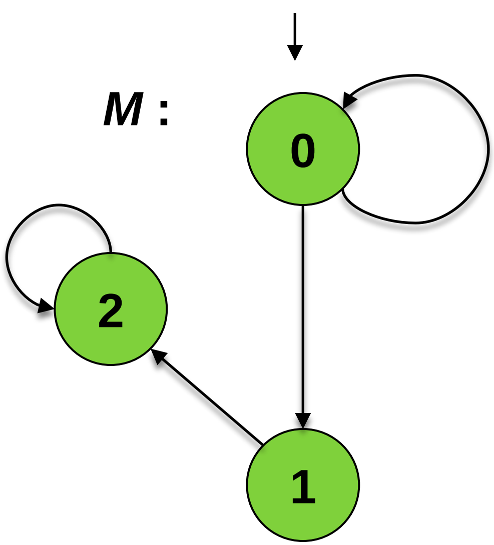
- CTL
  - CTL
  - Model checking
- Symbolic model checking
- BDD
  - Definition
  - Reducing BDD
  - Operations on BDD

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

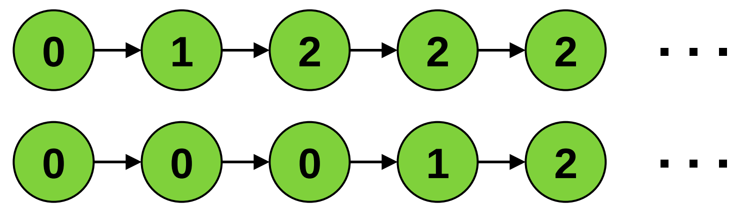
CTL

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

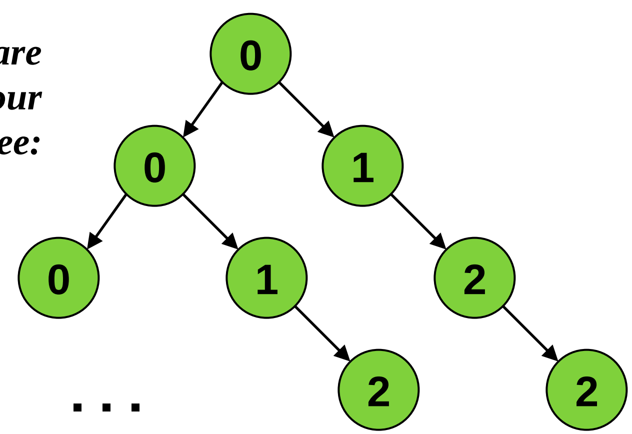
***M* :**



In LTL, properties ate 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.

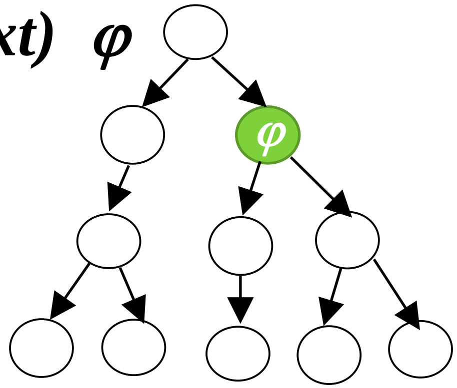
$$M \mid\!== \varphi \quad = \quad M\text{'s computation trees satisfies } \varphi$$

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

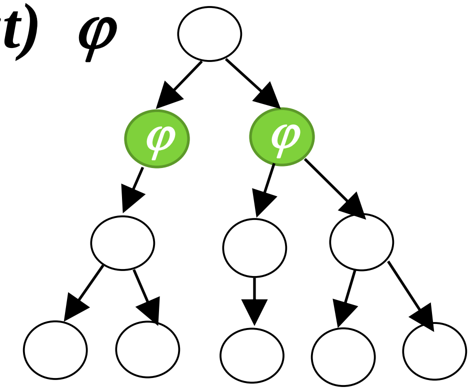
- Temporal operators :
  - **X** ... : holds next time
  - **F** ... : holds in the future
  - **G** ... : always hold
  - **U** : until

Intuition of CTL operators

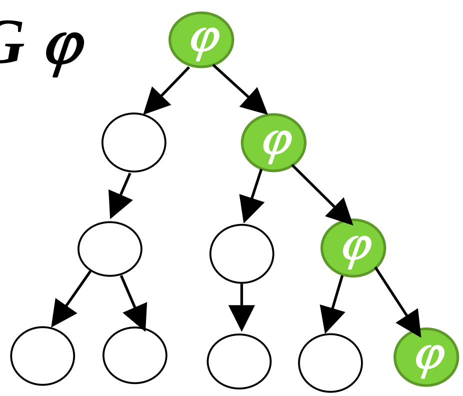
EX (exists next) φ



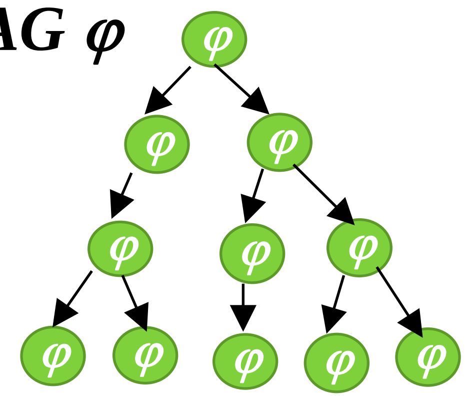
AX (all next) φ



EG φ

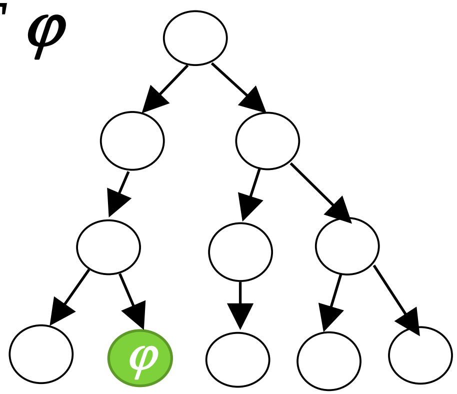


AG φ

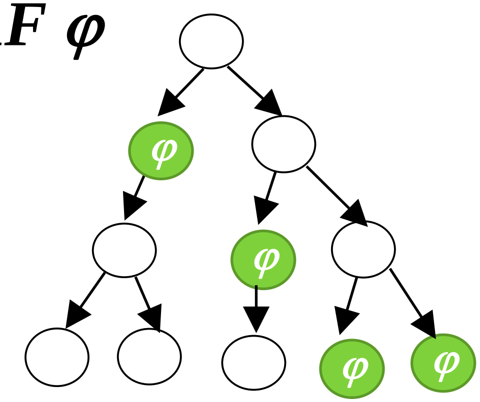


Intuition of CTL operators

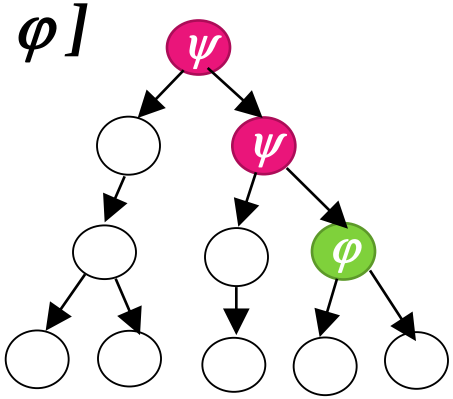
EF φ



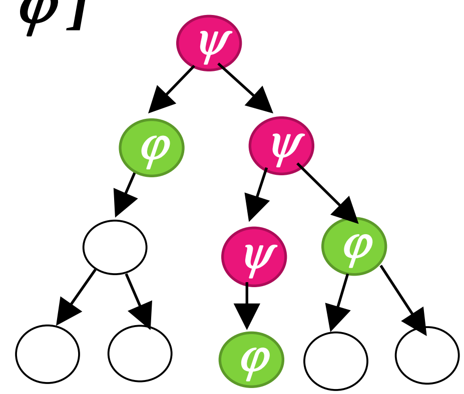
AF φ



E[ψ U φ]



A[ψ U φ]



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]

Derived operators

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

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

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 |== φ φ holds on the comp. tree t
- M |== φ is defined as M, **tree**(s_0) |== φ
- ***M,t* |== *p*** = $p \in V($**root**$(t))$
- ***M,t* |== ¬φ** = not (M,t |== φ)
- ***M,t* |== φ/\ψ** = M,t |== φ and M,t |== ψ

Semantic of "X"

- $\boldsymbol{M,t}$ **|== EX**φ = $(\exists v \in R(\mathbf{root}(t)) :: \ M,\mathbf{tree}(v) \ |\!== \varphi)$
- $\boldsymbol{M,t}$ **|== AX**φ = $(\forall v \in R(\mathbf{root}(t)) :: \ M,\mathbf{tree}(v) \ |\!== \varphi)$

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 |== AX φ 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 |- 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) |== φ
  - For all previous j, $0 \leq j < i$, M,**tree**(σ_j) |== ψ

- M,s |- A[ψ U φ] =

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

LTL vs CTL

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

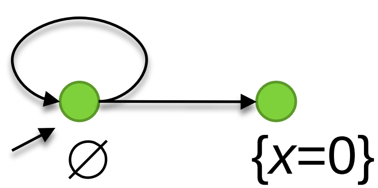
$$\mathbf{AG}\ (x=0) \quad = \quad [\,]\ (x=0)$$

$$\mathbf{AF}\ (x=0) \quad = \quad <>(x=0)$$

$$\mathbf{A}[x=0\ \ \mathbf{U}\ \ y=0] \quad = \quad x=0\ \ \mathbf{U}\ \ y=0$$

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

$$\mathbf{EF}\ (x = 0)$$

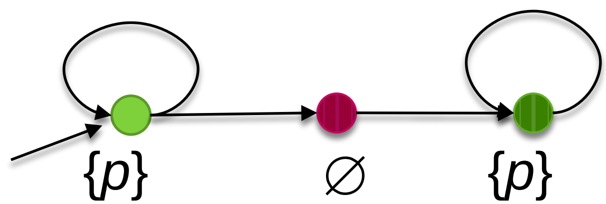


Prop = { *x*= 0 }

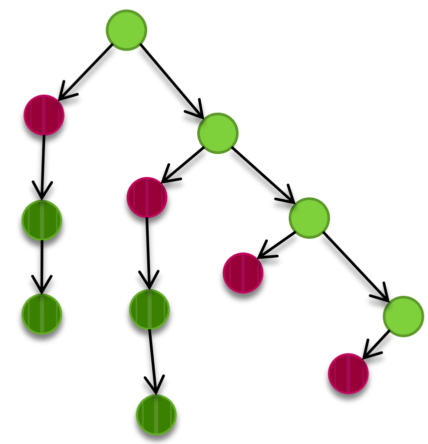
LTL vs CTL

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

<>[] *p*



Prop = { *p* }



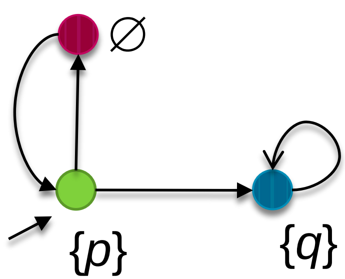
E.g. AF AG *p* does not express the property; the above Kripke does not satisfy it.

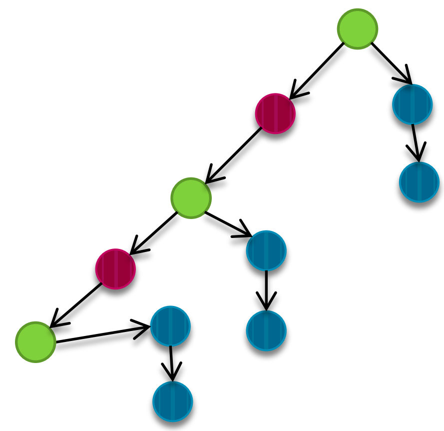
LTL vs CTL

- Another example, fairness restriction:

$$([]<> p \rightarrow <>q) \rightarrow <>q$$

$$= \quad []<>p \;\; \vee \;\; <>q$$





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

CTL*

- Allows more combinations of path and temporal quantifiers.
- A CTL* formula is a "state formula", syntax:

(State formula)

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

(Path formula)

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

We can express all CTL formulas in CTL*, but e.g. this is also possible in CTL* :

AFG (x=0)

Example: web application

- Based on:

  A Model Checking-based Method for Verifying Web Application Design, Donini et al, in Int. Workshop on Web Lang. and Formal Methods (WLFM), 2005.

- In their approach, models are obtained from UML design of the web application.
- Other possibilities:
  - By crawling a web site
  - By analyzing log

WAG

- Model web application as a graph (N,C), where

$$N = W \cup P \cup L \cup A$$

W	set of windows
P	set of pages
L	set of links
A	set of actions

each component is disjoint.

$C : N \rightarrow 2^N$ defines the arrows in the graph, and such that:

- A window can only be connected to pages
- A page can only be connected to links or actions
- A link or an action can only be connected to windows

- Called “Web Application Graph” (WAG)

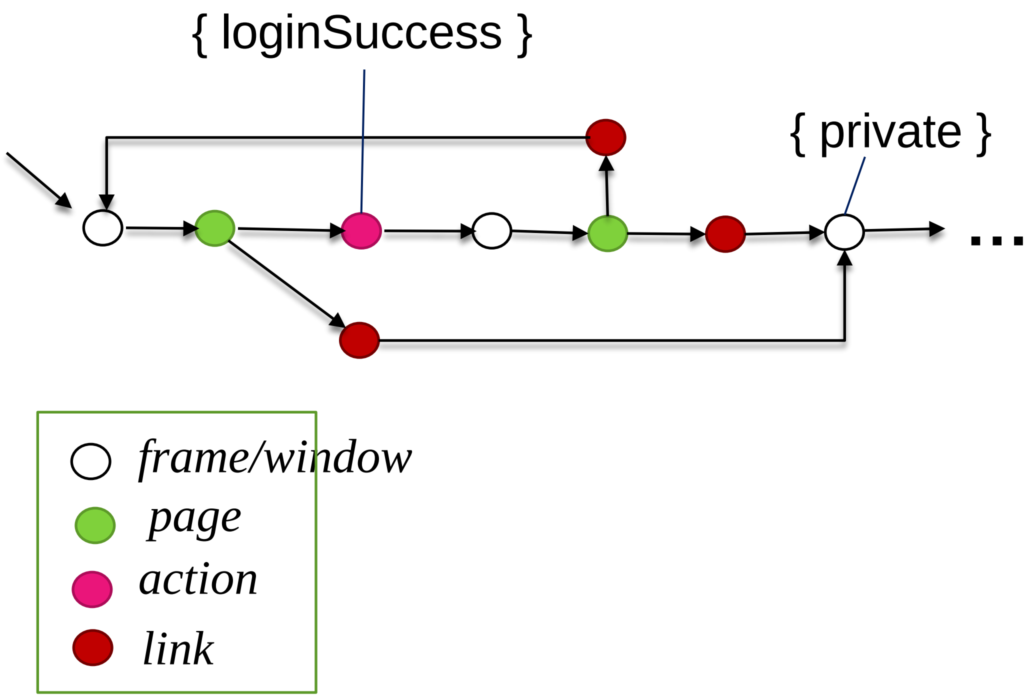
WAG as Kripke

- See a WAG as a Kripke structure, e.g. each node in the WAG is a state in the Kripke structure.
- Label each state with propositions w,p,l,a to express whether it is a window, or a page etc.
- Introduce other propositions of interest, e.g.

• login, logout	To mark a login/logout action
• private	To mark states considered "private"
• error	To mark "error page".

- Label the states with these propositions.

Example



Now properties like these are well defined…

- **A** ($\neg$private **W** $\neg$private /\ loginSuccess)

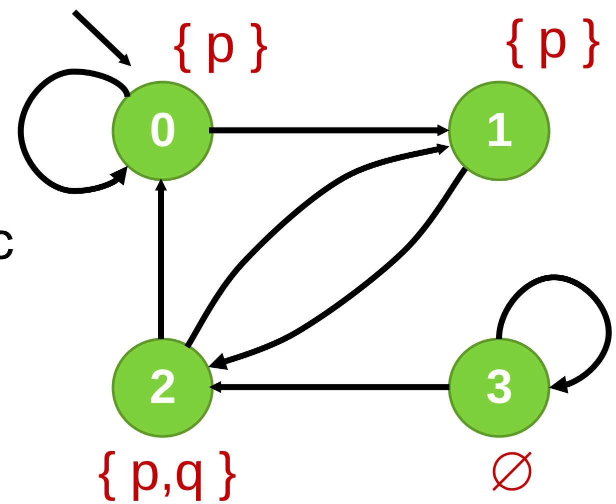
  You cannot get to the private part without logging in….

- **AG** (loginSucess $\rightarrow$ **EF** private)

  Once logged in, it should be possible to get to the private part

-

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

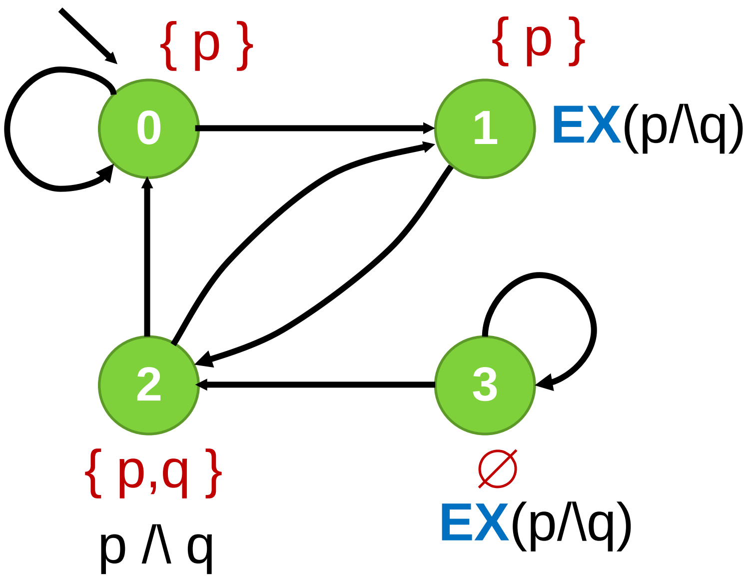
- Eventually, when we are done with the labeling of the root formula φ :

> $M \models \varphi$ *iff* s_0 *is labeled with* φ

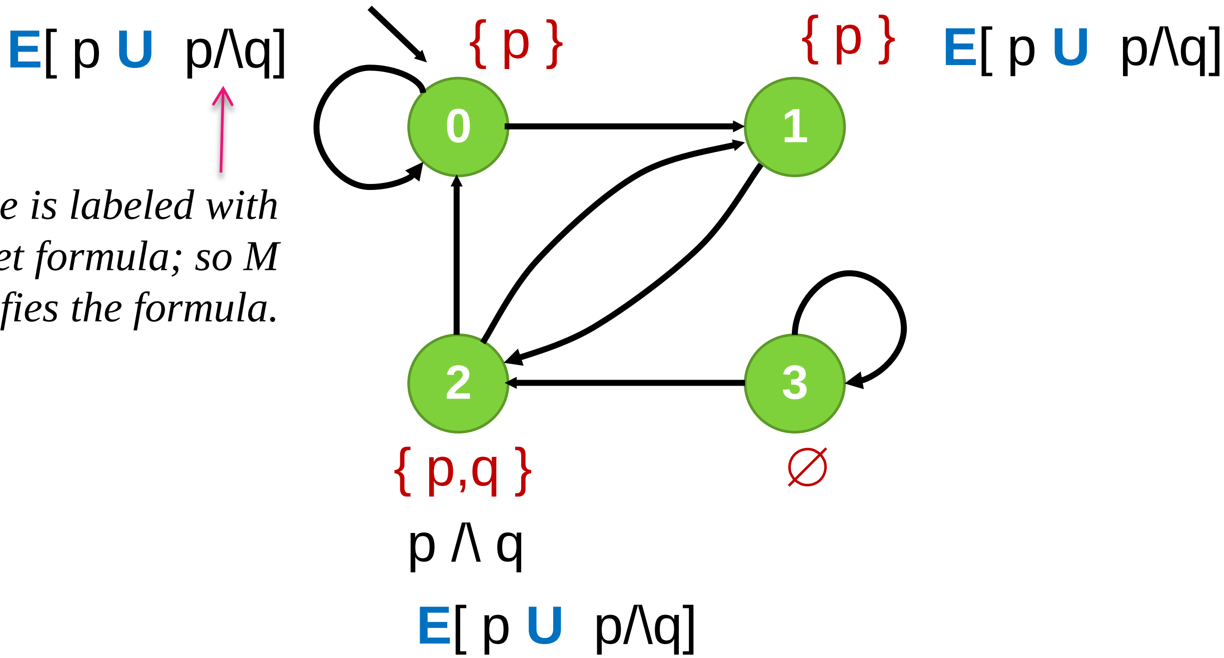
Example, checking **EX**(p/\q)

Prop = {p,q}

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



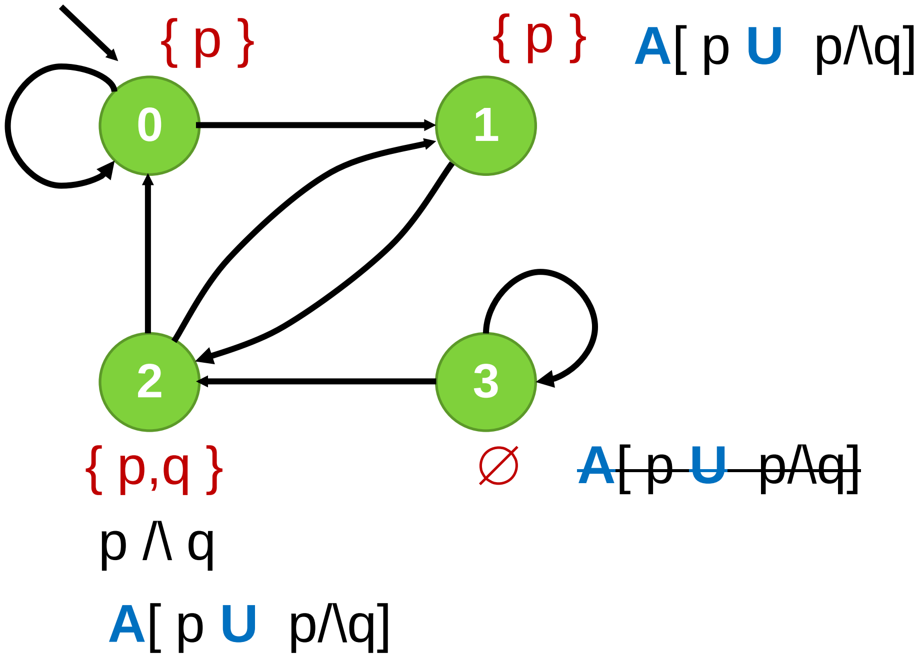
Example, checking: **E**[p **U** (p/\q)]



Initial state is labeled with the target formula; so M satisfies the formula.

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

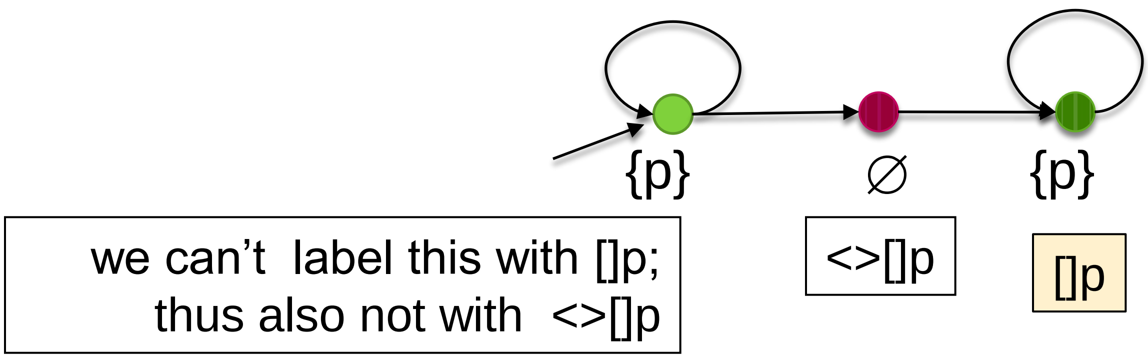


Can we apply this to LTL ?

- Consider **<>[] p**

Prop = { p }

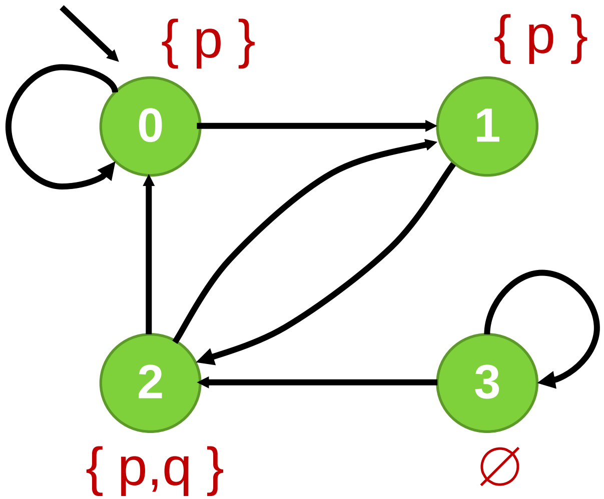
- Applying labeling :



Symbolic representation

- You need the full statespace 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	$\neg x \neg y$
St-1	$\neg x y$
St-2	$x \neg y$
St-3	xy

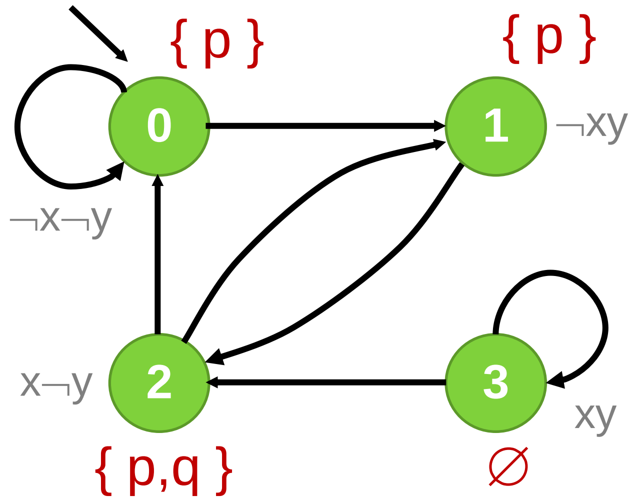
E.g. the set of states where q holds is encoded by the formula:

$$x \neg y$$

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

$$\neg (xy)$$

Example



States encoding:

St-0	¬x¬y
St-1	¬xy
St-2	x¬y
St-3	xy

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'

Example

```
byte x ;  // unspecified initial value

if x≠255 →  x=0 ;
```

The automaton has 256 states, with 256 arrows.

- Bit matrix : 8.3 Kbyte
- List of arrows: 512 bytes

With boolean formula:

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

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:

$W_{\mathbf{f}}$ = 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 $W_{\mathbf{f}}$ 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: $W_{f \wedge g} = W_f \wedge W_g$
- 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	xy

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

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

The set of all states that has at least <u>an outgoing arrow</u> to {0,1,2}

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

Encoding in Boolean formula:

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

Restricting the arrows over the destinations

The set of all states whose <u>all outgoing arrows</u> go to {0,1,2} :

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

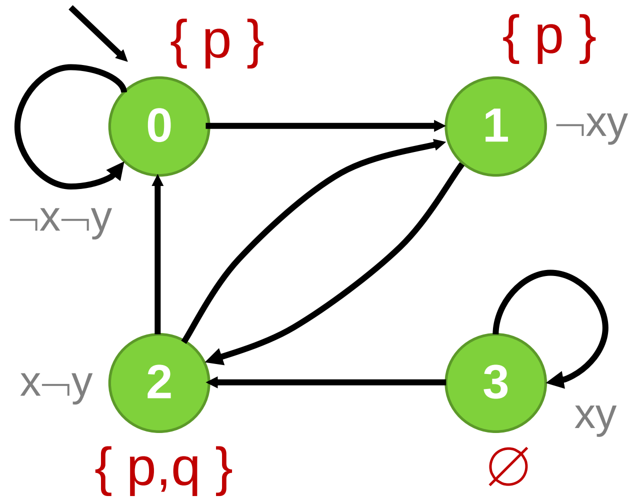
Encoding in Boolean formula :

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

Note:

- In both examples, invalid encodings (those states that were not actually in your M) are actually also quantified along as well → incorrect → add a constraint that filters your result to drop those states.
- In the ∀ example, all terminal states in M will automatically be included in the set … weird, but we discussed this before. We assumed M does not contain terminals.

Example, **EX**p



States encoding:

St-0 ¬x¬y
St-1 ¬xy
St-2 x¬y
St-3 xy

$W_p = \neg(xy)$

$W_{EXp} = \exists x', y' :: \ R \land \neg(x'y')$

$= \exists x', y' :: ((\neg y \neg x' \ \lor \ yx' \neg y' \ \lor \ xyx') \ \land \neg(x'y'))$

$= \text{true}$

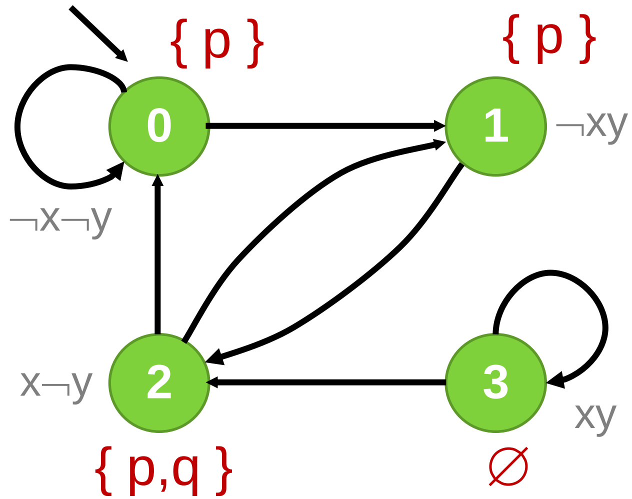
Labeling

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

$$Z_{i+2} = Z_{i+1} \lor (\exists x', y' :: R \land W_{\mathbf{f}} \land Z_{i+1}[x', y'/x, y])$$

  - Stop when $Z_{i+1} = Z_i$; then $W_{\mathbf{E[p\ U\ q]}} = Z_i$

Example, EX[p U q]



$$Z_1 = W_q = x\neg y$$

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

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

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 check fix point?

- To make this works, 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.
- Use a SAT-solver to check if $\neg(f \leftrightarrow g)$ is unsatisfiable.
- We'll discusss BDD approach

Canonical representation

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

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

- 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

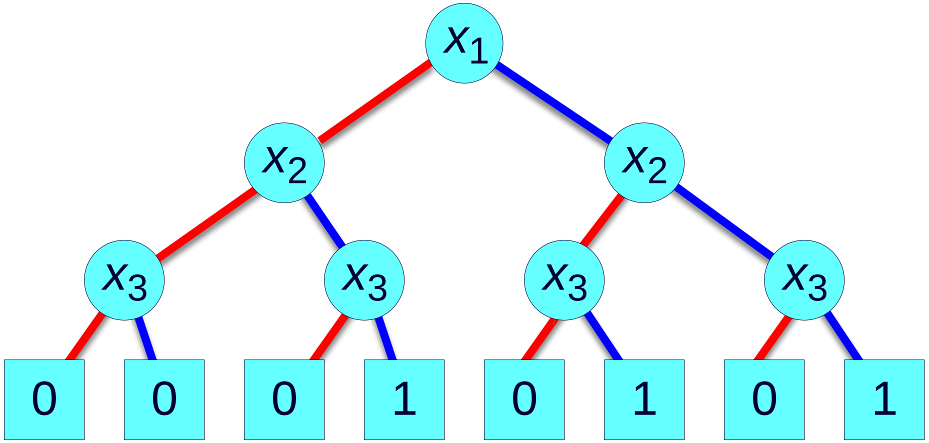
$$\neg x_1\, x_2\, x_3 \quad \vee \quad x_1 \neg x_2\, x_3 \quad \vee \quad x_1\, x_2\, x_3$$

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.

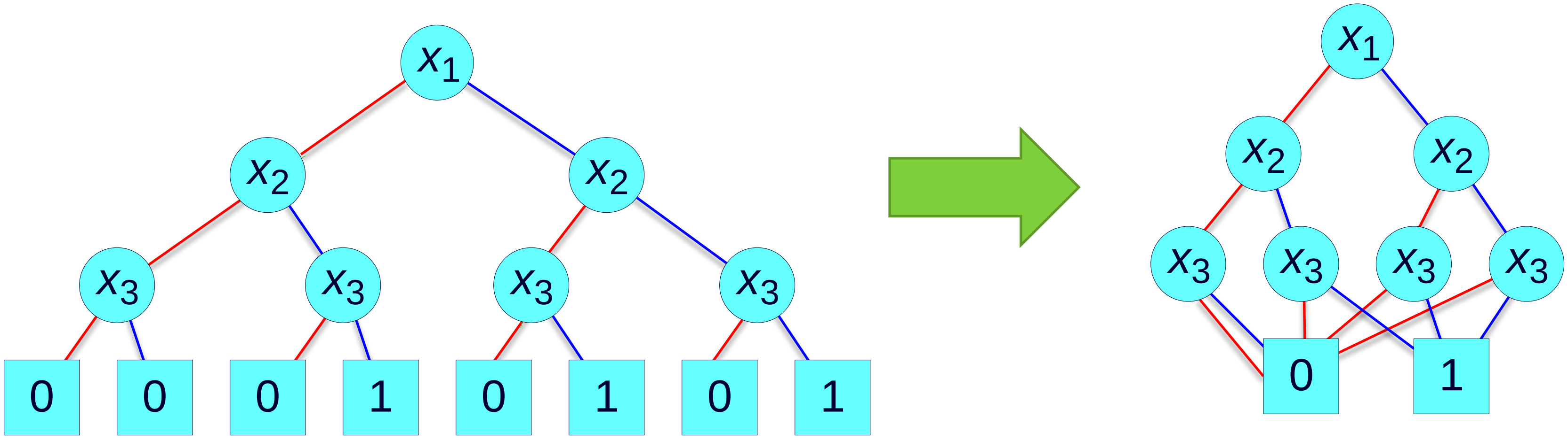
Or representing the table with a (binary decision) tree :



- 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 :

$$\boxed{f(x,y,z) \quad = \quad x.y \ \backslash/ \ z}$$

- In Bryant's paper this is called a : <u>boolean function.</u>

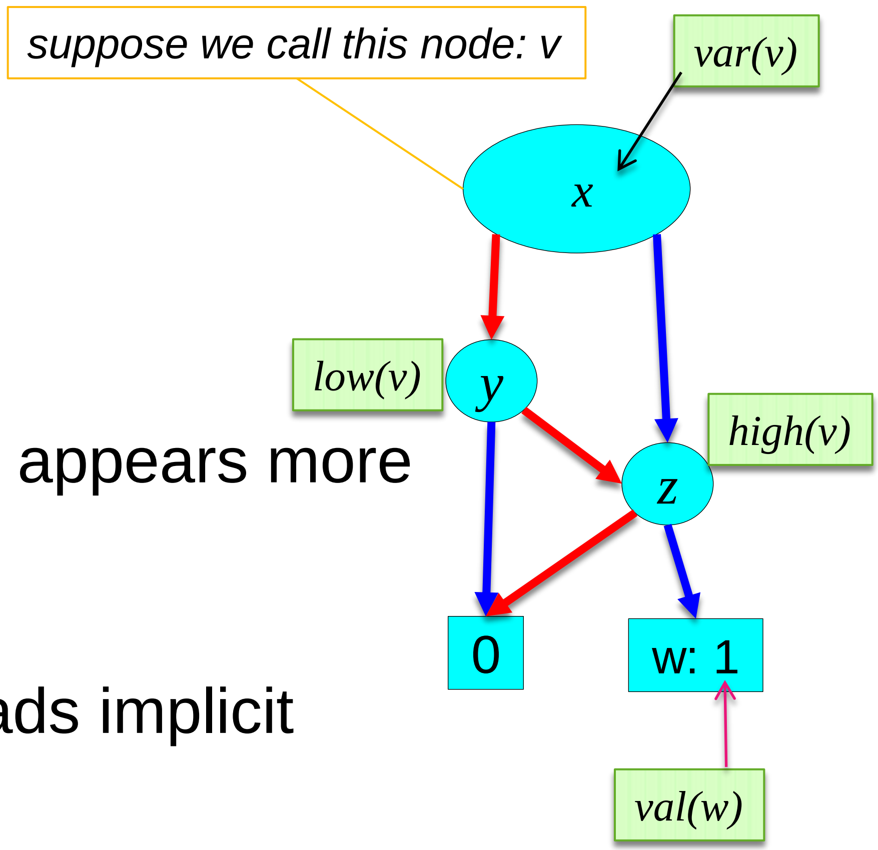
- E.g. 'composing' functions as in

  "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

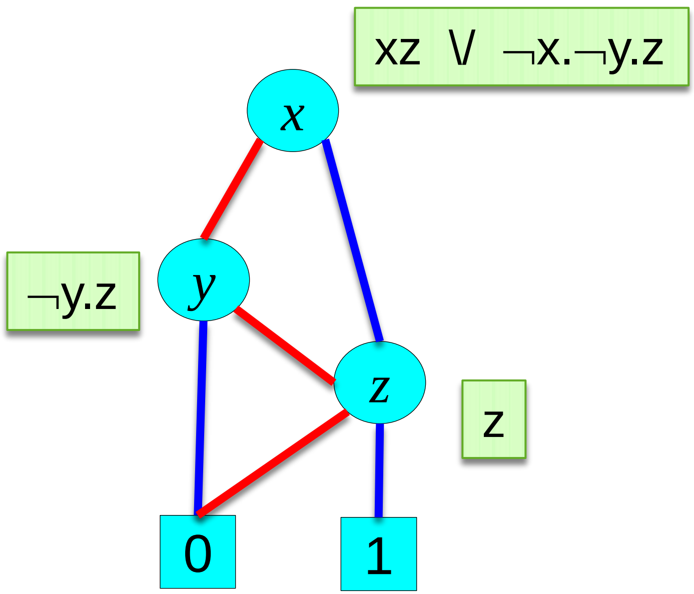


func(G)

$\boxed{x = \mathrm{val}(v)}$

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

$\boxed{\mathrm{func}(G) = \mathrm{func}(\mathrm{root})}$



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

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 var(v) nor val(v) !

  then: func(F) = func(G)

- A BDD G is *reduced* if:

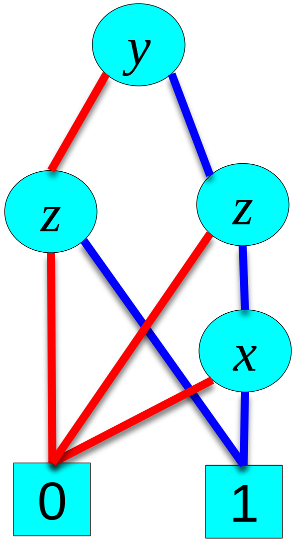
  - for any non-leaf node v, low(v) $\neq$ high(v).

  - for any distinct nodes u and v, the sub-BDDs rooted at them are not isomorphic.

otherwise G can be reduced!

Ordered BDD

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



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

Reduced OBDD

- Reduced OBDD is canonical:

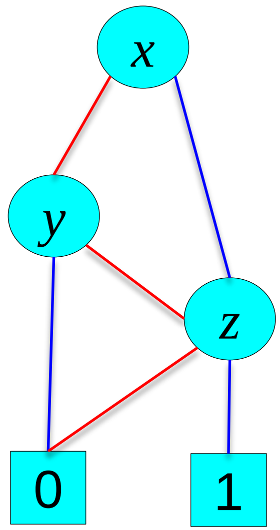
If we fix the variable ordering, every boolean function is uniquely represented by a reduced OBDD (up to isomorphism).

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

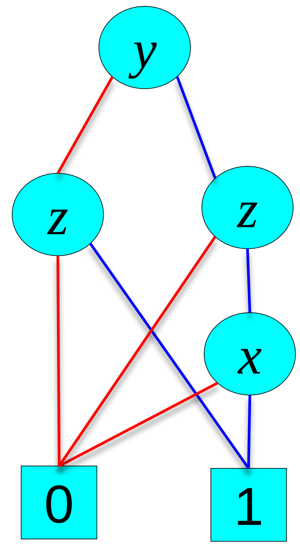
- However, the chosen ordering may influence the size of the OBDD.

Effect of ordering

Consider: $xyz \lor \neg yz$



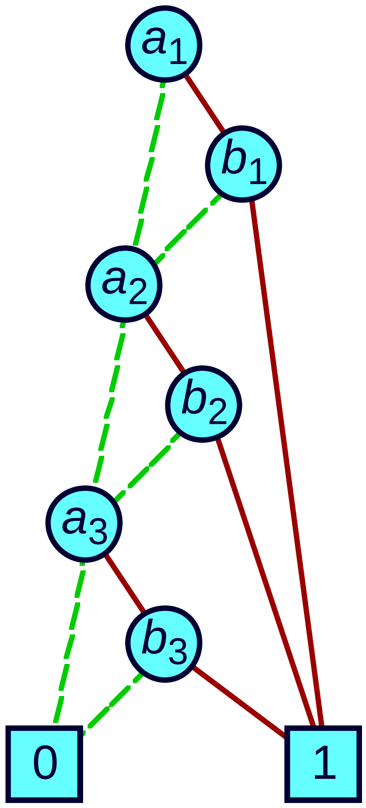
Order: x,y,z



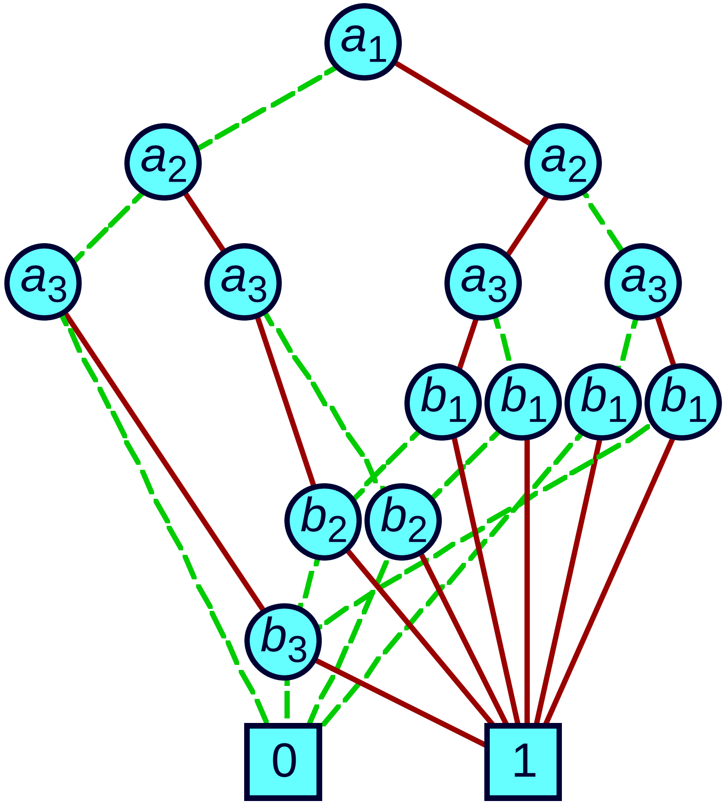
Order: y,z,x

The difference can be huge…

consider: $a_1b_1 \lor a_2b_2 \lor a_3b_3$



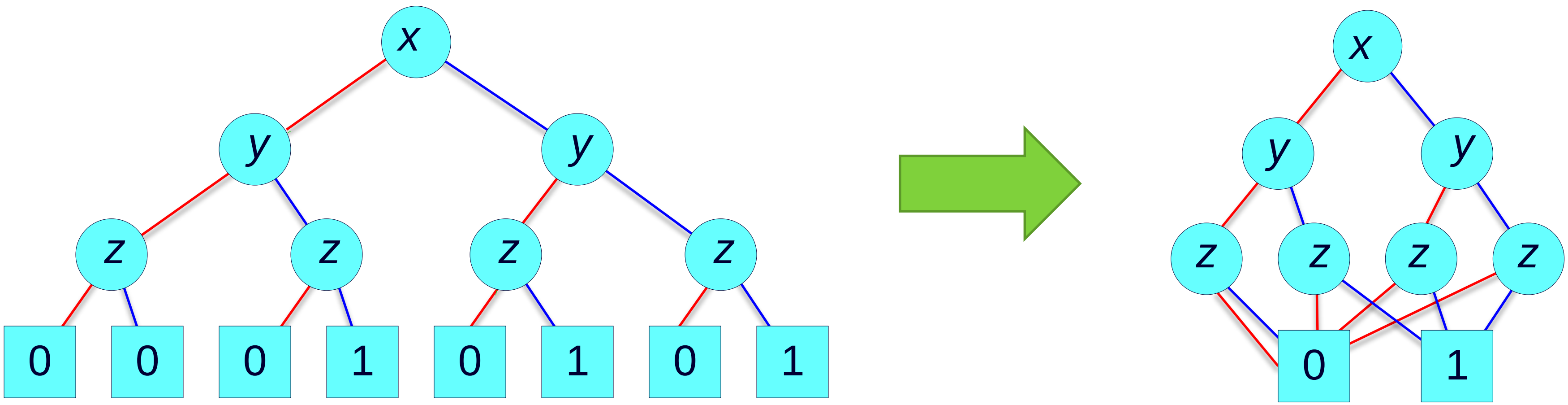
Linear Growth



Exponential Growth

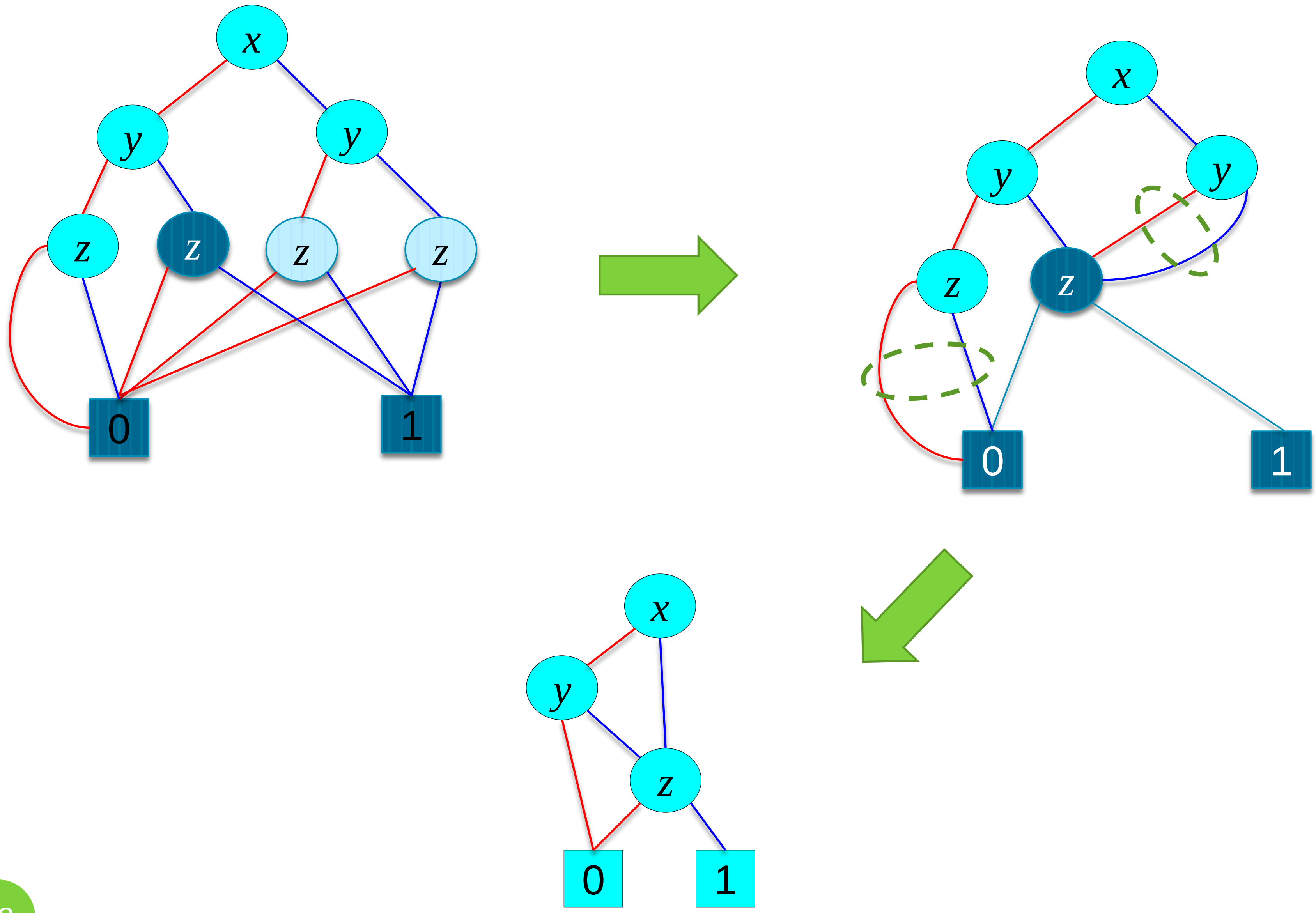
Here: "red" for value 1, "green" for 0.

Reducing BDD



By sharing leaves…

Reducing BDD



The reduction algorithm

- Introduce **id**, function Node $\rightarrow$ Node

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

  Iterate/recurse and maintain this invariant:

  $$\boxed{func(u) = func(id(u))}$$

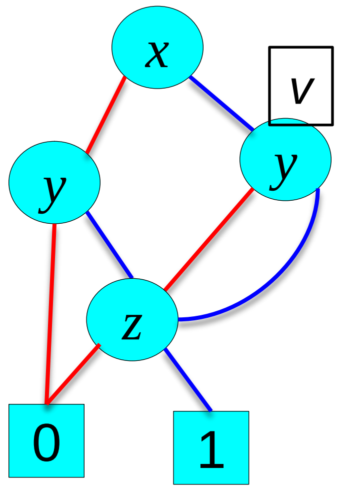
- So, we can remove u from the graph, and re-route arrows to it, to go to id(u) instead.
- Work bottom up, and such that a node decorated with x is processed after all nodes whose decorations come later in the var-ordering are processed first.

The reduction algorithm

- We'll do the relabeling recursively, bottom-up.

  Now suppose we have done the id re-labeling for all non-leaves w with index(w)>i. Suppose index(v)=i

  - **Case-1**, id(low(v)) = id(high(v)) ; suppose var(v) = "x"



func(v) = $\neg$y . $\boxed{\text{func(low(v))}}$ $\vee$ y . $\boxed{\text{func(high(v))}}$

= $\neg$y . func(id(low(v))) $\vee$ y . func(id(high(v)))

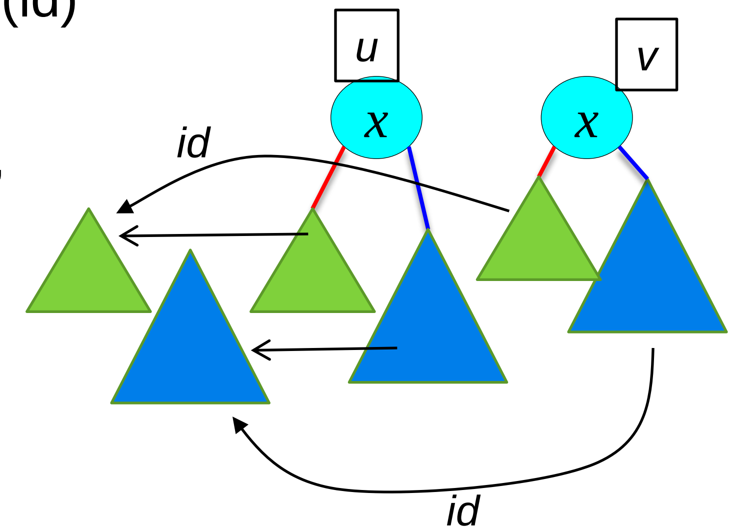
= func(id(low(v))

So, update: ***id(v)*** *:= id(low(v))*

The reduction algorithm

- **Case-2:** there is another non-leaf u$\in$dom(id) (u has been processed) such that:

  1. var(u) = var(v) ; suppose this is "x"
  2. id(low(u)) = id(low(v))
  3. id(high(u)) = id(high(v))



$$
\begin{aligned}
\text{func}(v) &= \neg x\ \text{func}(\text{low}(v)) \lor x\ \text{func}(\text{high}(v)) \\
&= \neg x\ \text{func}(\text{low}(u)) \lor x\ \text{func}(\text{high}(u)) \quad \text{// by inv} \\
&= \text{func}(u) \\
&= \text{func}(\text{id}(u))
\end{aligned}
$$

So, update: ***id(v)*** *:= id(u)*

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:

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

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 :

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

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

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

Restriction

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

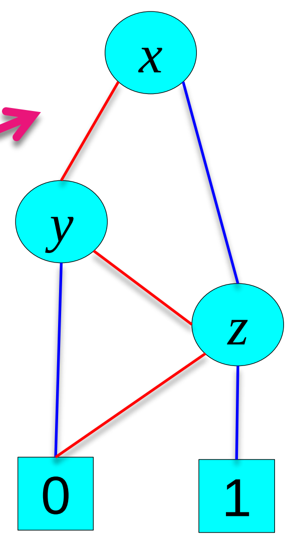
$f(x,y,z) \mid_{y=0}$ → replace all y nodes by low-sub-tree

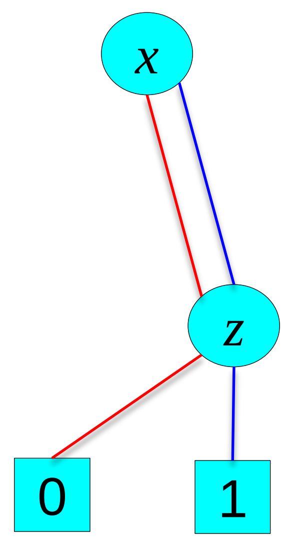
$f(x,y,z) \mid_{y=1}$ → replace all y nodes by high-sub-tree

Example:

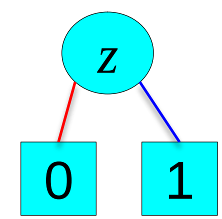
$f(x,y.z) = xz \lor \neg x \neg yz$

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





After replacing "y"



Reduced version.

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*’ :

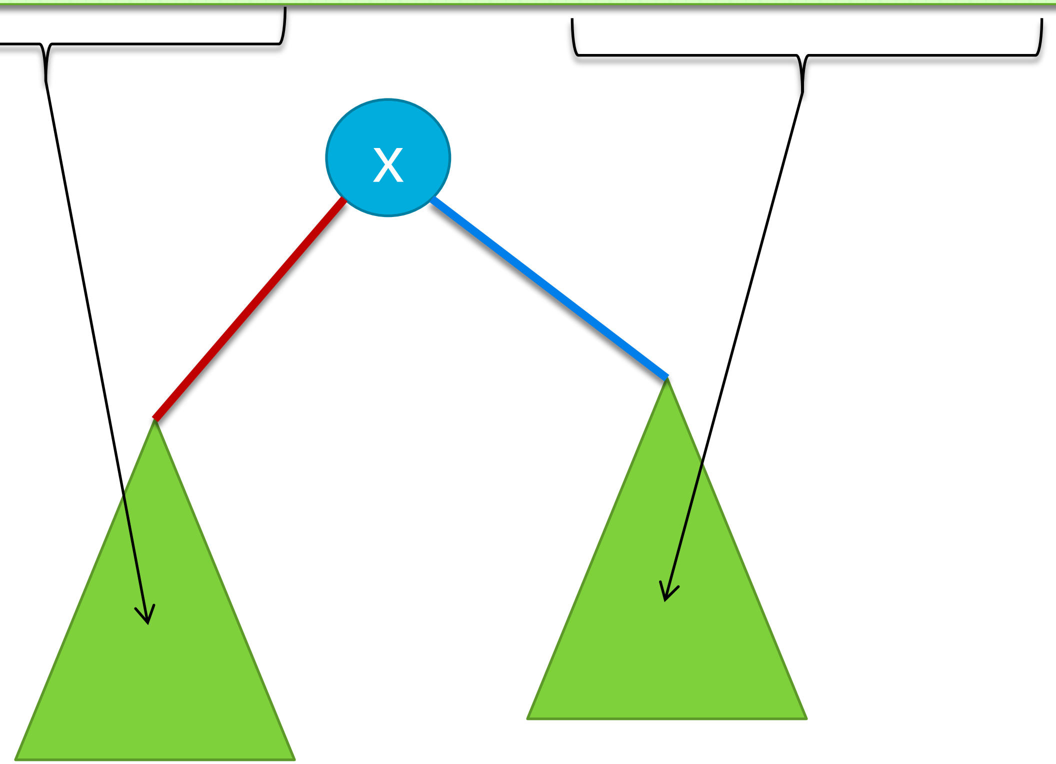
$$f \langle op \rangle g$$
$$=$$
$$\neg x \,.\, (f\,|_{x=0} \langle op \rangle g\,|_{x=0}) \quad \vee \quad x \,.\, (f\,|_{x=1} \langle op \rangle g\,|_{x=1})$$

- This tells us how to implement “apply” recursively !

Detail, see LN.

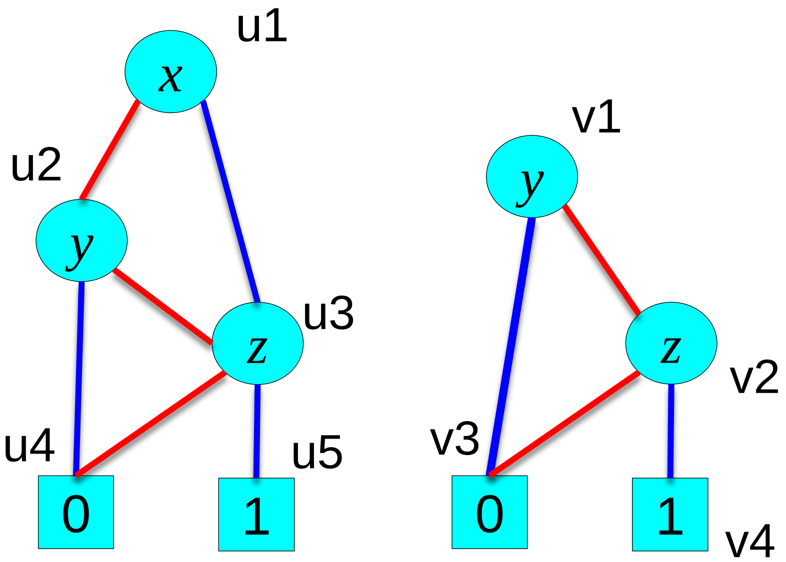
Apply

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



But this can be exponential. Solution: keep track of those sub-expressions you have combined.

Example

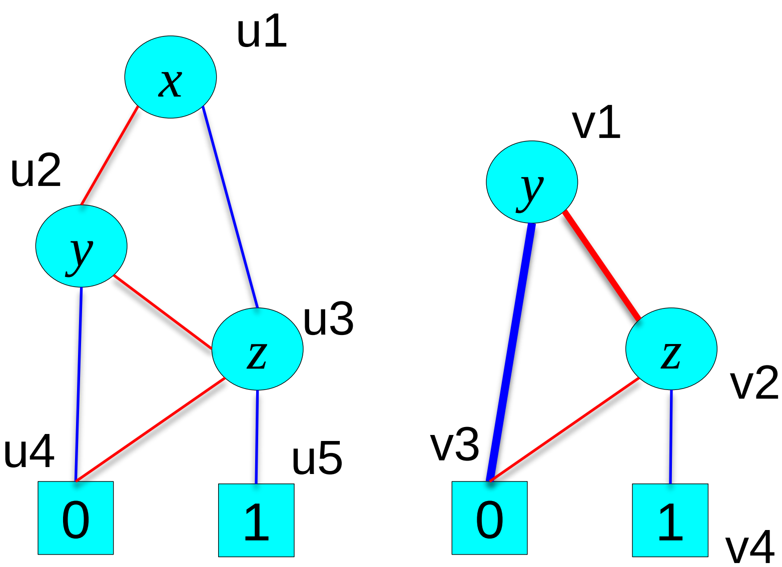


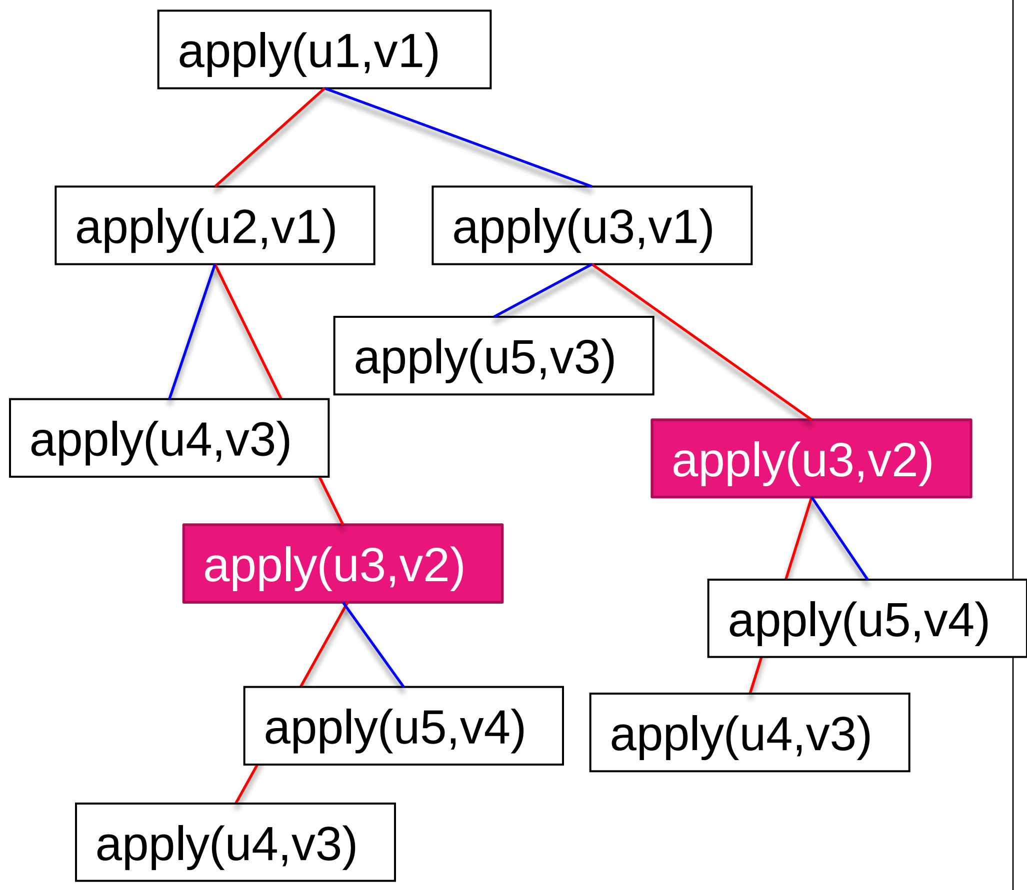
We name the nodes, just so that we can refer to them.

We'll do this by hand.

$$f \text{ <and> } g$$
$$=$$
$$\neg x \;.\; (f\,|_{x=0} \text{ <and> } g\,|_{x=0}) \quad \vee \quad x \;.\; (f\,|_{x=1} \text{ <and> } g\,|_{x=1})$$

Example





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

Satisfy and Compose

- Compose, constructed through :

  $f1|_{x=f2} \;=\; f_2 \,.\, f_1|_{x=1} \;\vee\; \neg f_2 \,.\, f_1|_{x=0}$

- 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.

And substitution…

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

  $$\boxed{\exists x', y' :: \; R \; \wedge \; W_p \, [x', y'/x, y]}$$

  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* $O(n)$

  n is the number of parameters in the target boolean function.