CS357: CTL Model Checking (two lectures worth)

David Dill

CTL

CTL = Computation Tree Logic

It is a “propositional temporal logic” – temporal logic extended to properties of events over time.

CTL is a “branching time logic” – there can be multiple possible next states.

Temporal logic is most useful for reasoning about protocols, circuits, and other concurrent systems, which can be very tricky to get right.

There are many such logics. CTL was invented by Emerson and Clarke in the early 1980’s.

Syntax of CTL

Propositional variables: p, q, etc.

Propositional connectives: $\neg$, $\wedge$, $\vee$, $\rightarrow$

Modal connectives: (f, g are any CTL formulas):

EX f – f is true in some next state

AX f – f is true in all next states

E[f U g] – along some path, f is true until g is true.

A[f U g] -- along all paths, f is true until g is true.

Semantics of CTL

CTL formulas are interpreted on a “Kripke structure” (a state graph).

<AP, S, R, L>

AP – a finite set of atomic propositions (p, q, etc.)

S – a set of states.

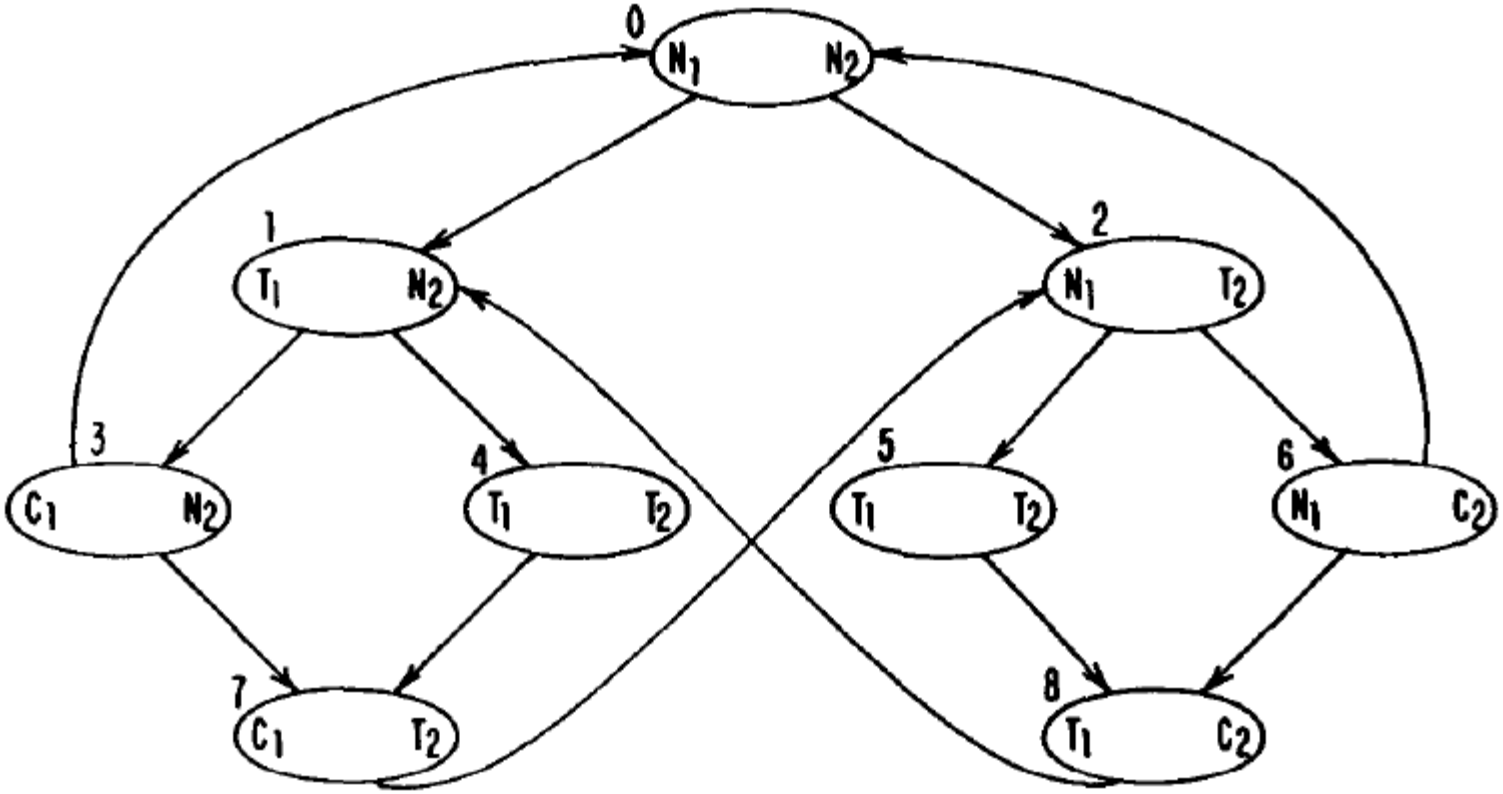
R – A relation from (current) states to (next) states. R must have a next state for every state.

L – a labeling function that maps states to sets of literals (members of AP or their negations).

Model checking algorithms work when S and R are *finite.*

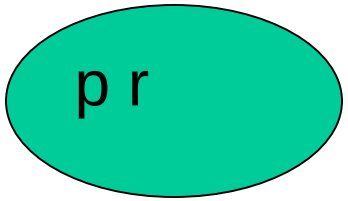
Semantics of CTL, cont.

Example of a Kripke structure
(from Clarke, Emerson, and Sistla, 1986)



Semantics of CTL, cont.

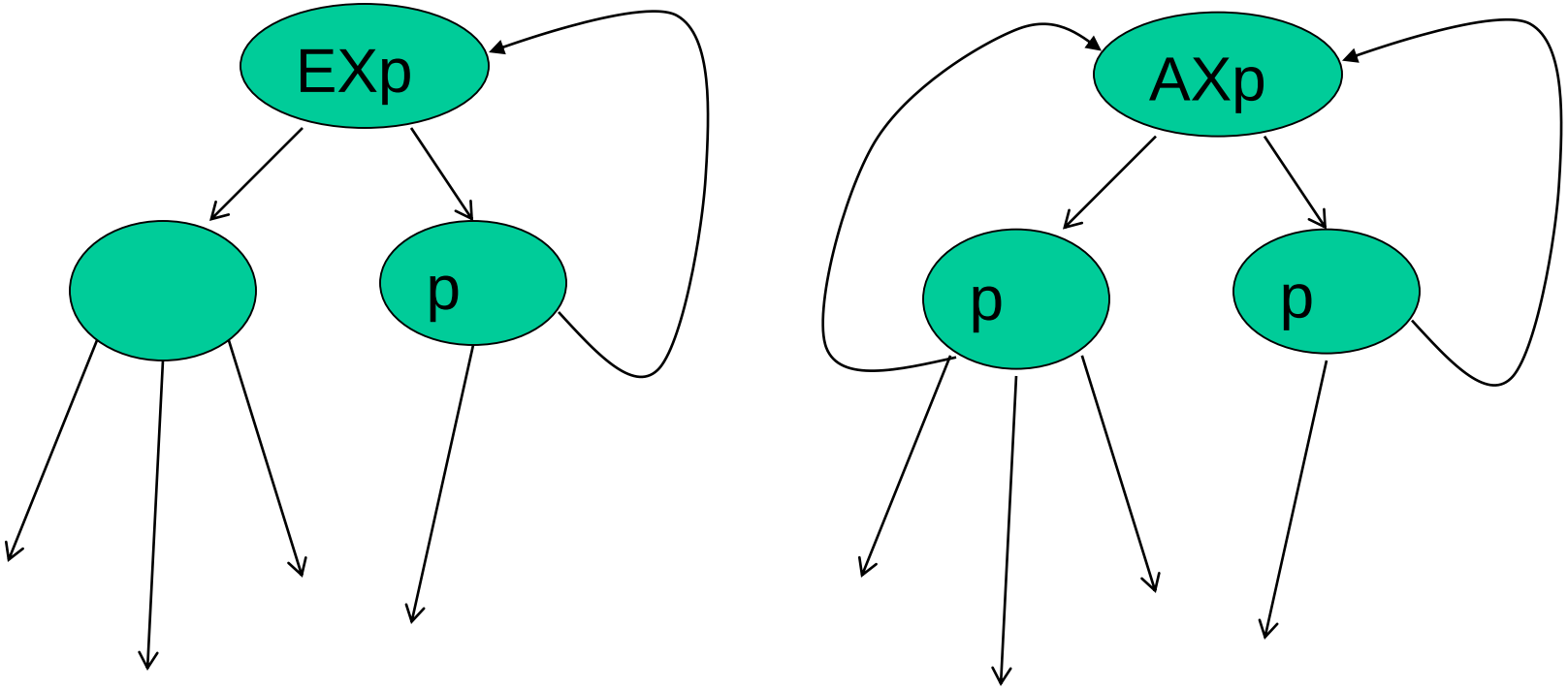
- In CTL, a state of the Kripke structure satisfies a formula: s |= f. This is define recursively:
- s |= p if p $\in$ L(s)
- s |= f $\wedge$ g if s |= f and s |= g (other prop connectives are similar).



satisfies p, p $\wedge$ r, p $\vee$ q
does not satisfy $\neg$p, q,
p $\wedge$ q

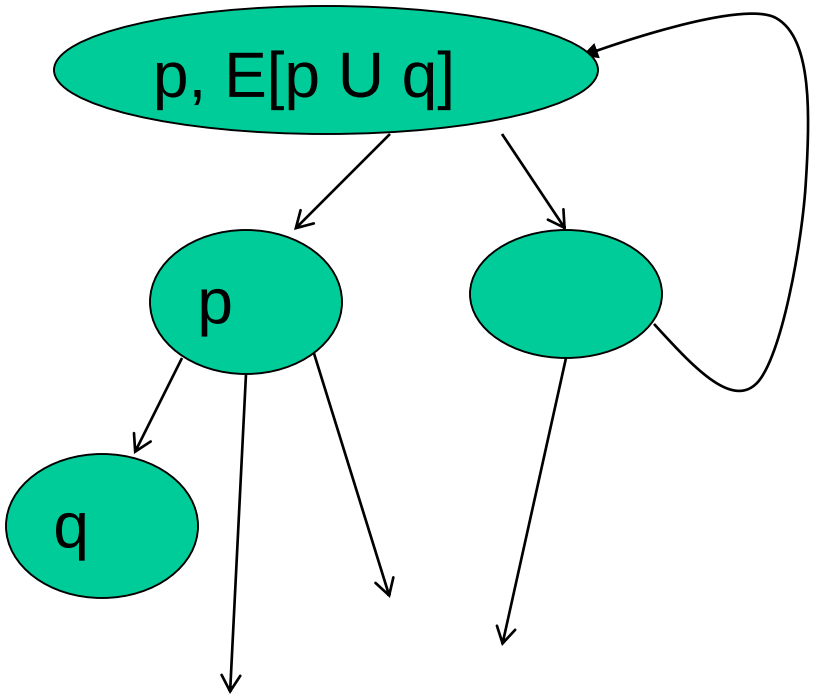
Semantics of CTL, cont.

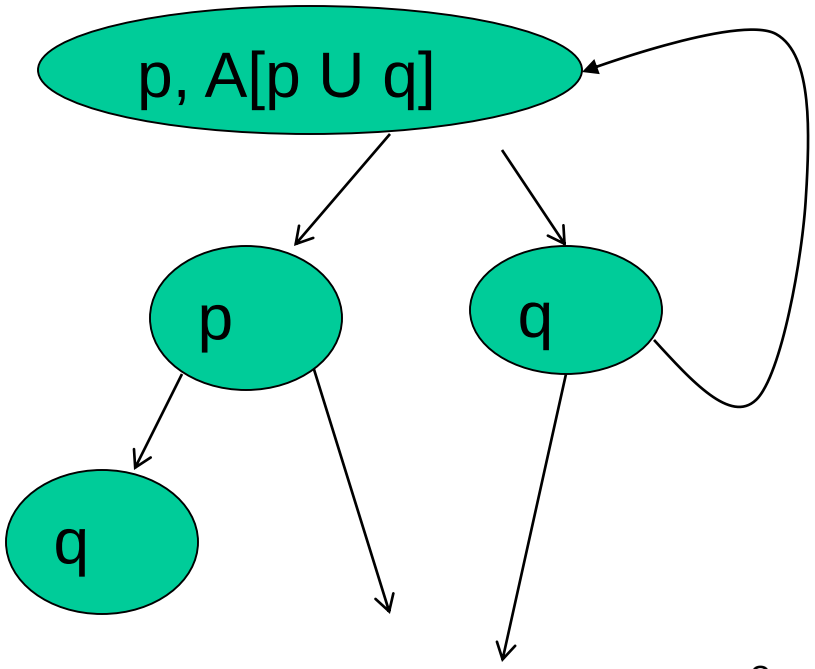
- s |= EX f if ∃t R(s, t) and t |= f
- s |= AX f if ∀t R(s, t) and t |= f



Semantics of CTL, cont.

- $s_0 \models A[f\ U\ g]$ if there is some path $s_0, s_1, \ldots s_n$ s.t $s_i \models f$ for all $0 \leq i < n$ and $s_n \models g$
- $s_0 \models A[f\ U\ g]$ if for all paths $s_0, s_1, \ldots s_n$ s.t $s_i \models f$ for all $0 \leq i < n$ and $s_n \models g$





Abbreviations

- EF p = E[True U p] -- p is eventually true on some path.
- AF p = A[True U p] -- p is eventually true on every path (“inevitably”)
- EG p = $\neg$AF $\neg$p – p is always true along some path.
- AG p = $\neg$EF $\neg$p – p is true in every reachable state.

Fairness constraints

- Often, we need to assume that certain properties hold "infinitely often". E.g., every process gets scheduled infinitely often.
- We want to rule out unfair paths – e.g, if we try to prove that transaction completes, we don't want the model checker to say: "Nope – here is an example where it didn't complete because a one of the processes never got scheduled."
- CTL allows a finite set of CTL formulas to be specified.
- Definitions of A/E[f U g] formulas modified to quantify over all *fair* paths.

BDD Kripke structure

AP = Boolean state variables.

Each state must be *uniquely labeled*

S (set of states) = all bit vectors.

N(*x*,*x*') = Next state relation

Model checking

Result of checking any formula is BDD of states that satisfy it.

EX $f(x) = \exists y.\ N(x, y) \wedge f(y)$

This is Preds(N,P_i)(y)

AX $f(x) = \forall y.\ N(x, y) \rightarrow f(y)$

E[f **U** g](x) = $\mu Q.\ g(x) \vee (f(x) \wedge$ **EX** $Q)$

A[f **U** g](x) = $\mu Q.\ g(x) \vee (f(x) \wedge$ **AX** $Q)$

AF f = **A**[true **U** f] = $\mu Q.\ f(x) \vee$ **AX** Q

EF f = **E**[true **U** f] = $\mu Q.\ f(x) \vee$ **EX** Q

Model Checking

AG $f = \neg$ **EF** $\neg f = \neg \mu Q.\ \neg f(x) \vee$ **EX** Q

$= \nu Q.\ f(x) \wedge$ **AX** Q

EG $f = \neg$ **AF** $\neg f = \nu Q.\ f(x) \wedge$ **EX** Q

- This gives an excuse to use ν
- CTL can be expanded into big mu-calculus formulas
- But, these formulas don't really have "nested fixed points" (outer fixed point does not bind variables inside inner ones).
- Best method for solving is still recursive bottom-up.

Fair CTL

Fairness constraints { c_k } are BDDs on states

Important formula: $E_C G\ f$ "there is a *fair path* where all states satisfy *f.*"

Largest set of states *Z* satisfying

- All states in *Z* satisfy *f*
- For all c_k and all *s* in *Z*, there is a path of at least length 1 from *s* to a state satisfying c_k *s.t.* all states on the path satisfy *f*.

$E_C G\ f = \nu Z\ .\ (f \wedge \bigwedge_k \mathbf{EX}\ \mathbf{E}[f\ \mathbf{U}(Z \wedge c_k)])$

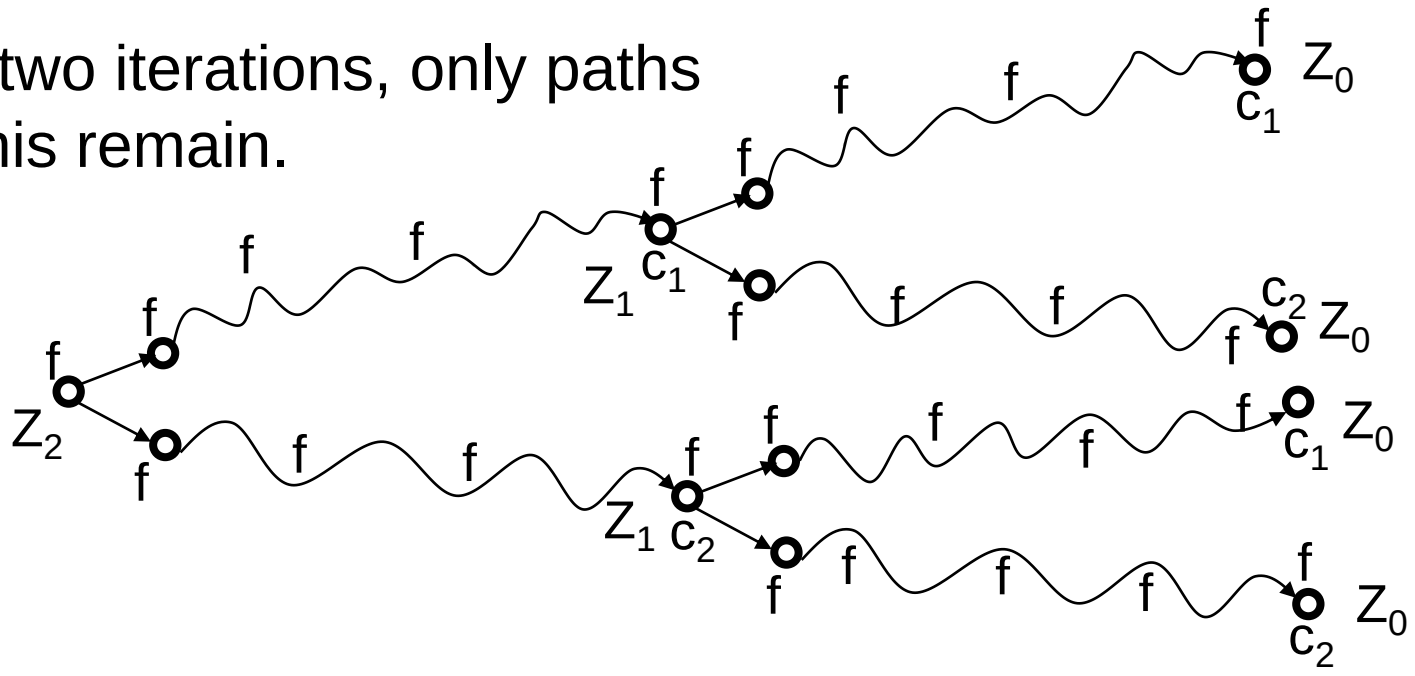
This is *really* nested. Z is in scope of least fixed point for **EU** operator.

Fair CTL

The i^{th} iteration of the GFP eliminates paths that visit any c_k state fewer than i times.

$$E_C G\, f = \nu Z\, .\, (f \wedge \bigwedge_k \mathbf{EX}\, \mathbf{E}[f\, \mathbf{U} (Z \wedge c_k)])$$

After two iterations, only paths like this remain.



Fair CTL

$h = \mathbf{E_C G}$ true ("there exists a fair path from here")

$\mathbf{E_C X}\,(f) = \mathbf{EX}(f \wedge h)$

$\mathbf{E_C}[f\ \mathbf{U}\ g] = \mathbf{E}[f\ \mathbf{U}\ (g \wedge h)]$

This gives a model checking algorithm for Fair CTL (just do all the above using BDD operations)

Symbolic model checking with fairness constraints

We've talked about symbolic reachability with BDDs, but not specifically about CTL model checking, and especially not with fairness constraints.

It helps to understand "fixed points" a little better.

Fixed points are fundamental to program semantics, logic, and also automata and algorithms

Fixed Points

Many problems boil down to solving an equation like $X = F(X)$.

Any solution to the equation is a “fixed point” (input = output).

But there may be many solutions or no solution.

Frequently, there is a least or greatest solution that is particularly interesting.

Lattices and monotonic functions

Suppose we have a finite set S

$\subseteq$ is a partial order on the subsets of S.

Suppose F is a function that maps a subset of S to another subset of S ($F : 2^S \rightarrow 2^S$)

F is *monotonic* if, for all subsets X and Y of S,

$X \subseteq Y$ implies $F(X) \subseteq F(Y)$.

Lattices and monotonic functions

If these conditions are met, *F* always has *unique least fixed point* and *unique greatest fixed point*.

i.e., there are unique smallest and largest sets satisfying $X = F(X)$.

Properties of fixed points

Fixed points closed under intersection and union:

Least fixed point (*LFP*) is $\cap_i \{ X_i \subseteq S \mid X_i = F(X_i) \}$

Greatest fixed point (*GFP*) is $\cup_i \{ X_i \subseteq S \mid X_i = F(X_i) \}$

Iterative computation

Theorem: If S is a finite set and $F : 2^S \rightarrow 2^S$ is monotonic w.r.t $\subseteq$, there exists n such that $F^n(\emptyset)$ is least fixed point.

Proof:

Claim: For all i, $F^i(\emptyset) \subseteq F^{i+1}(\emptyset)$

Base: $F^0(\emptyset) = \emptyset \subseteq F(\emptyset)$

Induction: If $F^i(\emptyset) \subseteq F^{i+1}(\emptyset)$, then

$F(F^i(\emptyset)) \subseteq F(F^{i+1}(\emptyset))$ [monotonicity of F]

Iterative computation

Since there are only a finite number of sets, there must be some $n < m$ such that $F^n(\varnothing) = F^m(\varnothing)$

But then, for all i between n and m, $F^n(\varnothing) = F^i(\varnothing) = F^m(\varnothing)$.

So $F^n(\varnothing)$ is the desired fixed point.

Similarly, there exists some n where $F^n(S)$ is GFP of F.

Iterative computation

- For least fixed point, start with smallest set and work your way up as necessary.
- For greatest fixed point, start with largest set and work your way down.
- This is a simple instance of a beautiful general theory.
- Same general results hold for lattices, not just 2^S.
- Things change a bit for infinite lattices.
- In general, finite iteration only possible when S is finite.

μ-calculus

"Logic of fixed points"

μ R. $F(R)$ is least fixed point of F (least solution of $R = F(R)$)

νR. $F(R)$ is greatest fixed point of F

$\nu R.F(R) = \neg\mu R.\neg F(R)$

Example: reachable states

$I(x)$ -- x is initial state.

$N(x, y)$ -- y is successor of x.

reachable$(y) = \mu R.[I(y) \vee \exists x.R(x) \wedge N(x, y)]$

μ-calculus

- μ-calculus originally from programming language semantics (Dana Scott & Jaco de Bakker)
- We use it for BDDs, but original model checking was for explicit state graphs.
- μ-calculus model checking on explicit state graphs is not known to be in P or to be NP-complete
  - In the middle ground with graph isomorphism and a few other problems, though it is not known to be equivalent to graph isomorphism.
- As we formalize the problem, everything is PSPACE-complete (I think…)

Reachable states

(Abbreviate $\exists x.R(x) \wedge N(x, y)$] as $N(R)(y)$.)

$R_0(y) = false$ [initially, nothing is reachable]

$R_1(y) = I(y)$ [start states are reachable]

$R_2(y) = I(y) \vee N(R_1)(y) = I(y) \vee N(I)(y)$

$R_3(y) = I(y) \vee N[I(y) \vee N(I)(y)](y)$

$I(y) \vee N(I)(y) \vee N[N(I)](y)$

[start state and their successors are reachable]

...

Efficiency

- As described, what I've described will almost never work.
  - Transition BDD $N(x, y)$ is huge.
  - Image computation is hard.
  - It can be broken into parts.
- Lots of other optimizations
  - Dynamic reordering
  - Exploit "don't cares"
  - Approximate methods
- With all of these … it's sometimes practical.
- Fairness constraints are harder, though.

Relational image - more optimizations

- Combined ANDEXISTS has smaller intermediate BDDs than AND followed by EXISTS.
  - Make one recursive pass over BDDs

Early quantification

- Reduce maximum size of intermediate BDDs by eliminating variables as soon as possible.
- Uses distributive laws for quantifiers:
  - $\exists x\ (P(x) \vee Q(x)) = [\exists x\ P(x)] \vee [\exists x\ Q(x)]$
  - $\exists x\ (P(x,y) \wedge Q(y)) = [\exists x\ P(x,y)] \wedge Q(y)$
- Quantification doesn't always reduce BDD size, but this "usually" works better than not doing it.

“Disjunctive partitioning”

- Represent $N(x,y) = N_1(x,y) \vee N_2(x,y) \vee \ldots \vee N_n(x,y)$
- This makes since for *asynchronous* models
  - E.g., Proc 1 goes or Proc 2 goes or Proc 3 goes
- Image: $\exists x.\ R(x) \wedge N(x, y)$

  $= \exists x.\ R(x) \wedge (N_1(x,y) \vee N_2(x,y) \vee \ldots \vee N_n(x,y))$

  $= \exists x.\ [R(x) \wedge N_1(x,y)] \vee [R(x) \wedge N_2(x,y)] \vee \ldots$

  $\vee\ [R(x) \wedge N_n(x,y)]$ (distributive law)

  $= \exists x[R(x) \wedge N_1(x,y)] \vee \exists x\ [R(x) \wedge N_2(x,y)] \vee \ldots$

  $\vee\ \exists x[R(x) \wedge N_n(x,y)]$ ($\exists$ distributes over $\vee$**)**

“Conjunctive partitioning”

- Represent $N(x,y) = N_1(x,y) \wedge N_2(x,y) \wedge \ldots \wedge N_n(x,y)$
- This makes since for *synchronous* models
  - E.g., Proc 1 goes and Proc 2 goes and Proc 3 goes
- Limit variables x,y in $N_i(x,y)$ to the ones that it really depends on.
- This works well when R is deterministic: $N_i(x,y)$ is $y_i = F_i(x)$, where y_i is a single bit, and F_i is some function of a subset of the x bits.

“Conjunctive partitioning”

- To do early quantification of *x*’s
  - Use $\exists x\ (P(x,y) \wedge Q(y)) = [\exists x\ P(x,y)] \wedge Q(y)$
  - Order components so to maximize the number of x bits that appear in early formulas but not later formulas.
  - Loop:
    - AND in next relation in sequence
    - Quantify any variables that do not appear again.
    - Repeat.

Ordering for early quantification

- Define “lifetime” of variable to be the interval [*i*,*j*]
  - *i* least index relation where it appears
  - *j* is greatest index of relation where it appears.
- Goal is to sort relations so that maximum number of live variables at any point is minimized.
- This is a rough heuristic. There are many variations on this idea (see imgIwls95.c in Berkeley VIS verifier).

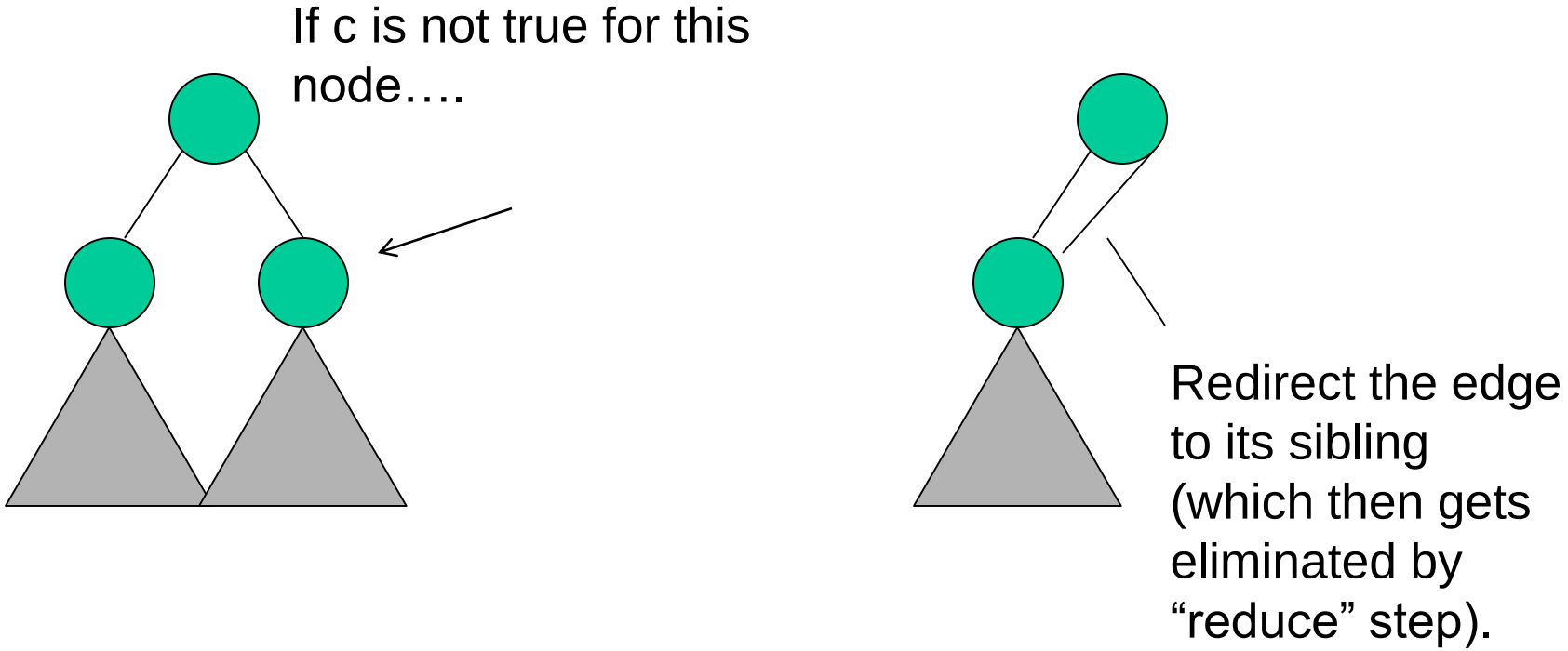
Using “don’t cares”

- Idea: don’t waste BDD nodes representing parts of functions you don’t care about.
- Suppose you are the middle of reachability computation
  - If $R(x)$ is current reached set, $N(x,y)$ is next state relation
  - We don’t care about $\neg R(x)$ part of $N(x,y)$ [because they are eliminated by $R(x) \wedge N(x,y)$]
  - Don’t care about R(y) part of N(x,y) [don’t care about states we have already reached, just the new ones].

Constrain

- Uses don't cares to reduce the size of a BDD
  - It's a heuristic, not guaranteed to reduce size.
  - Looks like a hack, but it has many beautiful and magical properties.
- *Constrain(f, c)* optimizes the BDD *f* while keeping the same values for inputs satisfying *c* (the *care set*).
- *Constrain* is also known as *"generalized cofactor"*
  - Allows cofactoring on a *function*, not just a variable.
  - *Constrain(f, x)* is just $f \downarrow x$
  - We write *constrain(f, g)* as $f \downarrow g$

“Constrain” idea



This would always reduce tree size, but may reduce sharing in a DAG.

If constrain did not sometimes increase BDD size, we would have P=NP.

Constrain algorithm

```
Constrain(f,c):
  if (c==1 || isconst(f)) return f;
  else if (f.var == c.var) {
    if (c.t==0) return Constrain(f.e, c.e);
    else if (c.e==0)
      return Constrain(f.t, c.t);
    else return make_bdd(f.var,
                  Constrain(f.t,c.t)
                  Constrain(f.e,c.e),);
  }
  etc.
```

Geometric interpretation

Let $x = [x_1, x_2, \ldots, x_n]$

If *f' = constrain(f, c),*

for every input *x*, let *y* be the "closest"

point to *x* s.t. *c(y) = 1.*

Then *f'(x) = f(y).*

Closest = minimum numerical value of

x XOR *y.*

[Bits are ordered as in BDD, top var is MSB]

e.g. d(111, 101) = 010 = 2.

Use of constrain in reachability

- Use Constrain(Constrain($N_i(x,y)$, $R(x)$), $\neg$R(y)) instead of $N_i(x,y)$

Find reachable states first

- CTL model checking fixed points work backwards in state graph **EX** $f(x) = \exists y.\ N(x, y) \wedge f(y)$
- Usually, only forward reachable states matter for property of interest.
- Backwards reachable state space might have a big BDD.
- A preliminary forward traversal to limit to reachable states is often helpful.
- BDD for these states can be used as “don’t cares” for the real model-checking.