CSC384: Lecture 6

- Last time
  - Costs, Heuristics, LCFS, BeFS, A*, IDS
- Today
  - wrap up last class slides (Misc Search topics)
  - game tree search
- Readings:
  - Today: none (you should use lecture slides)
  - Next week: Ch.8.1, 8.2 (STRIPS, skim Situation Calculus, **skip** Event Calculus), 8.3 (up to STRIPS planning)

Generalizing Search Problems

- So far: our search problems have assumed agent has complete control of environment
  - agent (courier, robot) has the *only* effect on state
  - straight path to goal state is reasonable
- Assumption not always reasonable
  - stochastic environment
  - other agents whose interests conflict with yours
- In these cases, we need to generalize our view of search to handle “uncontrollable” state change

Two-person Zero-Sum Games

- **Two-person, zero-sum games** an extreme case
  - chess, checkers, tic-tac-toe, backgammon, go, Command&Conquer:Renegade, New Adventures with Pooh, travel agents, “find the last parking space”
  - you want to get somewhere (winning position) and opponent wants you to end up somewhere else (losing position)
- Key insight:
  - how you act depends on how other agent acts (or how you think they will act)
  - and vice versa

More General Games

- What makes something a game:
  - there are two (or more) agents influencing state change
  - each agent has their own interests
    - e.g., goal states are different; or we assign different values to different paths/states
- What makes games hard?
  - how you should play depends on how you think the other person will play; but how they play depends on how they think you will play; so how you should play depends on how you think they think you will play; but how they play should depend on how they think you think they think you will play; …

More General Games

- Zero-sum games are “fully competitive”
  - if one player wins, the other player loses
  - e.g., the amt of money I win (lose) at poker is the amount of money you lose (win)
- More general games are “cooperative”
  - some outcomes are preferred by both of us, or at least our values aren’t diametrically opposed
- We’ll look in detail at zero-sum games
  - but first, a couple simple zero-sum and cooperative games for fun

Game 1: Rock, Paper Scissors

- Scissors cut paper, paper covers rock, rock smashes scissors
- Represented as a matrix: Player I chooses a row, Player II chooses a column
- Payoff to each player in each cell   (Pl.I / Pl.II)
- 1: win, 0: tie, -1: loss
  - so it's zero-sum

		Player II		
		R	P	S
Player I	R	0/0	-1/1	1/-1
	P	1/-1	0/0	-1/1
	S	-1/1	1/-1	0/0

Game 2: Prisoner’s Dilemma

- Two prisoner’s in separate cells, DA doesn’t have enough evidence to convict them
- If one confesses, other doesn’t:
  - confessor goes free
  - other sentenced to 4 years
- If both confess (both defect)
  - both sentenced to 3 years
- Neither confess (both cooperate)
  - sentenced to 1 year on minor charge
- Payoff: 4 minus sentence

	C	D
C	3/3	0/4
D	4/0	1/1

Game 3: Battlebots

- Two robots: Blue (Craig's), Red (Fahiem's)
  - one cup of coffee, one tea left
  - both C, F prefer coffee (value 10)
  - tea acceptable (value 8)
- Both robot's go for Cof
  - collide and get no payoff
- Both go for tea: same
- One goes for coffee, other for tea:
  - coffee robot gets 10
  - tea robot gets 8

	C	T
C	0/0	10/8
T	8/10	0/0

Two Player Zero Sum Games

- Key point of previous games: what you should do depends on what other guy does
- Previous games are simple “one shot” games
  - single move each
  - in game theory: *strategic or normal form games*
- Many games extend over multiple moves
  - e.g., chess, checkers, etc.
  - in game theory: *extensive form games*
- We’ll focus on the extensive form
  - that’s where the computational questions emerge

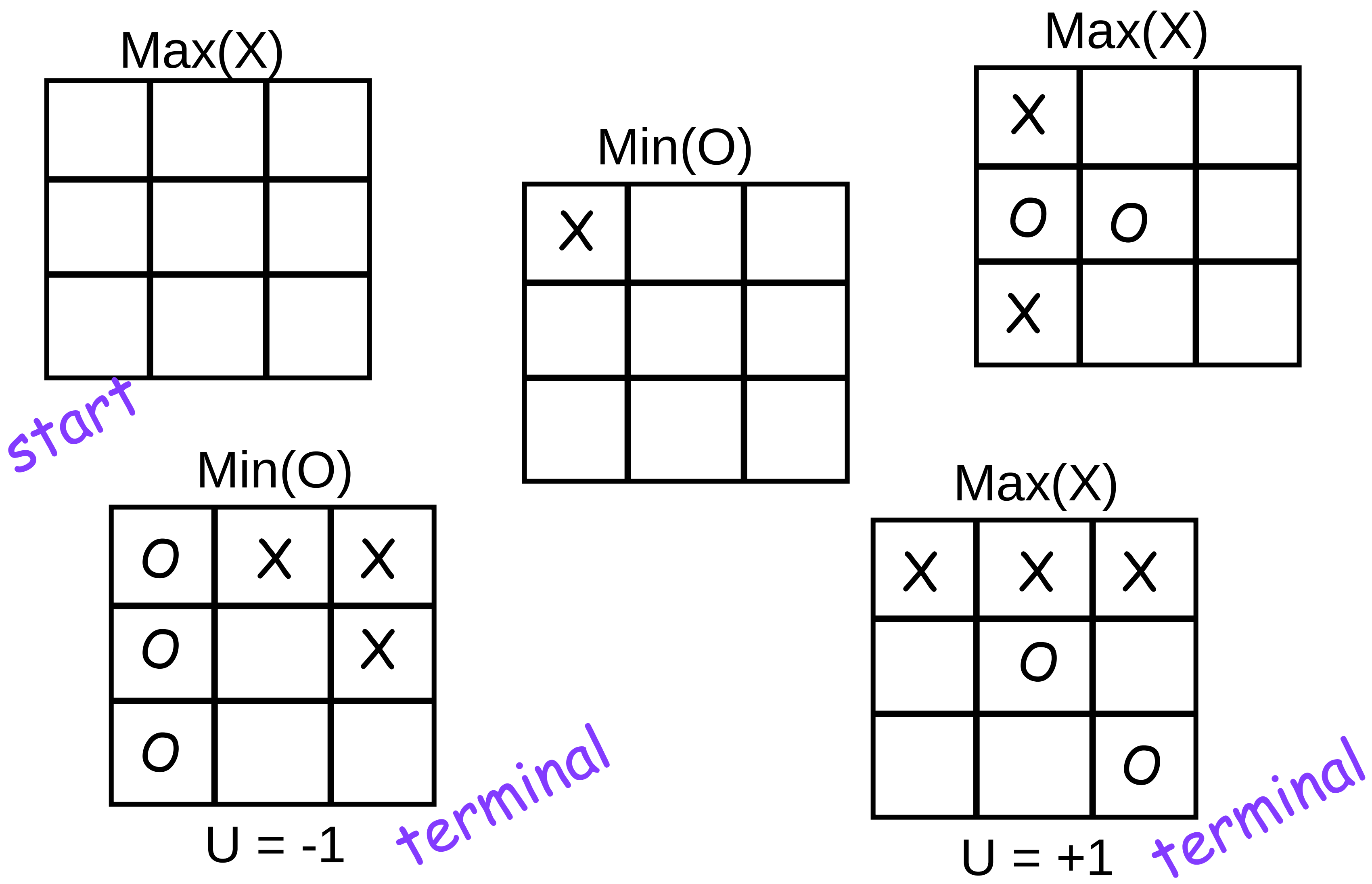
Two-Player, Zero-Sum Game: Defn

- Two *players* A (Max) and B (Min)
- set of *positions* P (states of game)
- a *starting position* $s \in P$ (where game begins)
- *terminal positions* $T \subseteq P$ (where game can end)
- set of directed edges $E_A \subseteq P \times P$ (A's *moves*)
- set of directed edges $E_B \subseteq P \times P$ (B's *moves*)
- *utility* or *payoff function* $U : T \rightarrow \mathbb{R}$

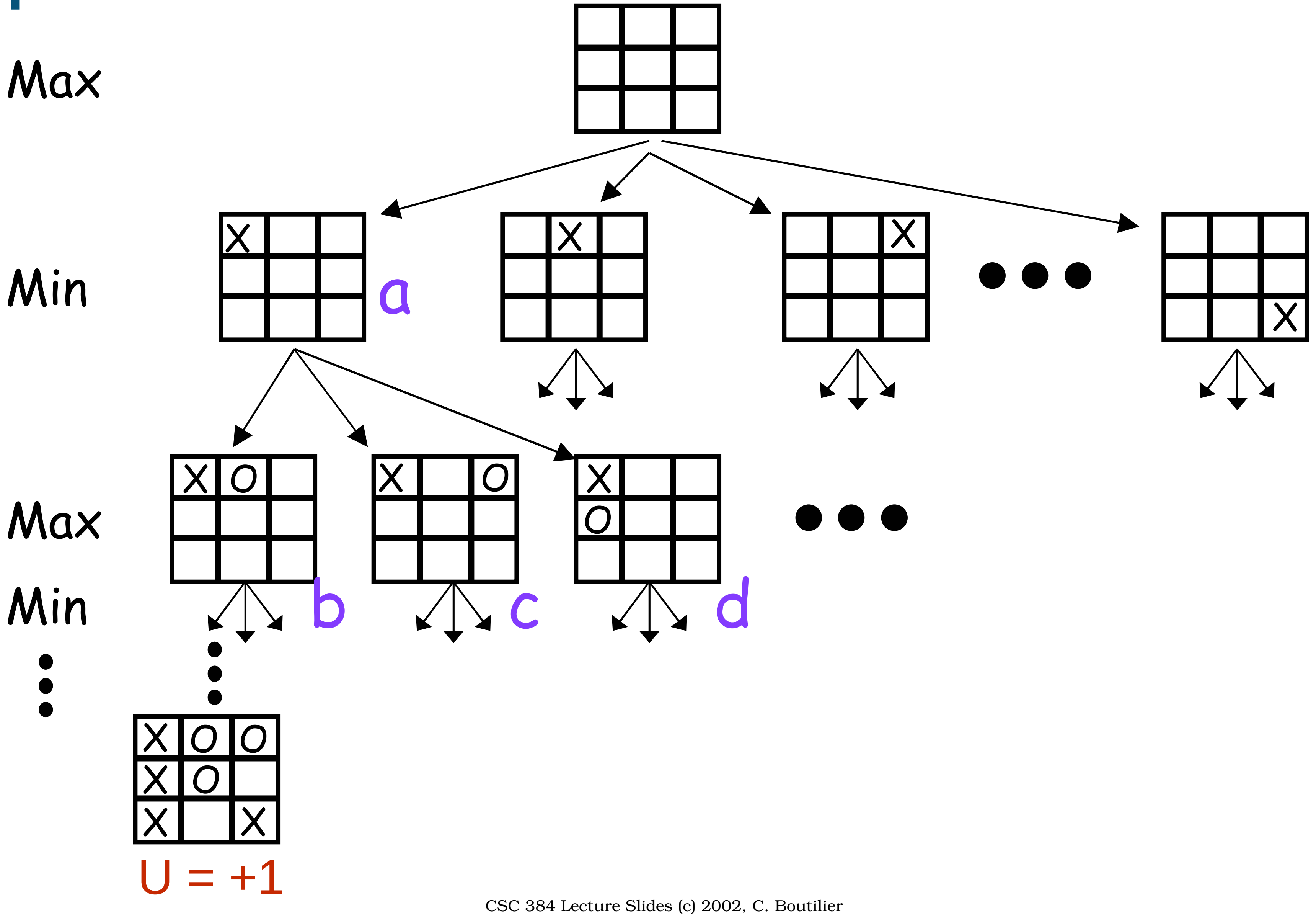
Intuitions

- Players alternate moves (starting with Max)
  - Game ends when some terminal $p \in T$ is reached
- A game **state**: a position-player pair
  - tells us what position we're in, whose move it is
- Utility function and terminals replace goals
  - Max wants to maximize the terminal payoff
  - Min wants to minimize the terminal payoff
- Think of it as:
  - Max gets U(t), Min gets –U(t) for terminal node t
  - This is why it's called zero (or constant) sum

Tic-tac-toe: States



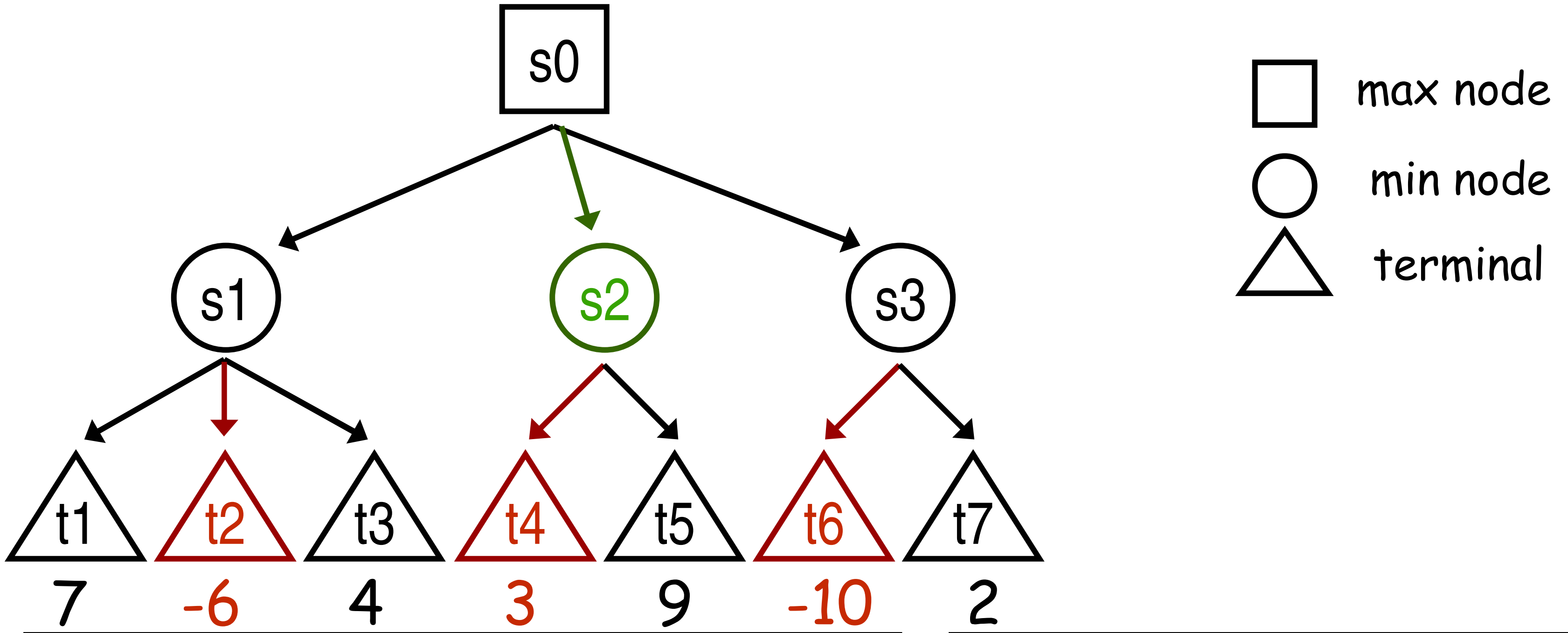
Tic-tac-toe: Game Tree



Game Tree

- Game tree looks like a search tree
  - Layers reflect the alternating moves
- But Max doesn't decide where to go alone
  - after Max moves to state a, Mins decides whether to move to state b, c, or d
- Thus Max must have a *strategy*
  - must know what to do next no matter what move Min makes (b, c, or d)
  - a sequence of moves will not suffice: Max may want to do something different in response to b, c, or d
- What is a *reasonable* strategy?

Minimax Strategy: Intuitions



If Max goes to s1, Min goes to t2
- * U(s1) = min{U(t1), U(t2), U(t3)} = -6

If Max goes to s2, Min goes to t4
- * U(s2) = min{U(t4), U(t5)} = 3

If Max goes to s3, Min goes to t6
- * U(s3) = min{U(t6), U(t7)} = -10

So Max goes to s1: so U(s0)
= max{U(s1), U(s2), U(s3)}
= 3

Minimax Strategy

- Build full game tree (all leaves are terminals)
  - root is start state, edges are possible moves, etc.
  - label terminal nodes with utilities
- Back values *up* the tree
  - *U(t)* is defined for all terminals (part of input)
  - *U(n)* = min {*U(c) : c* a child of *n*} if *n* is a min node
  - *U(n)* = max {*U(c) : c* a child of *n*} if *n* is a max node
- Max chooses, at any (max) state, the action that leads to the highest utility child

Depth-first Implementation on MMX

```
utility(N,L,U) :- terminal(L), utility(N,U).
utility(N,L,U) :- maxLevel(L), children(N,CList),
                  maxValue(CList, U).
utility(N,L,U) :- minLevel(L), children(N,CList),
                  minValue(CList, U).
```

- Depth-first evaluation of game tree
  - maxLevel(L) holds if level L is Max’s move; similarly for minLevel(L) and terminal(L).
  - utility of terminals must be specified as input
  - if game isn’t strictly alternating, then move might be associated with the node (or you might need to record whose move it is explicitly)

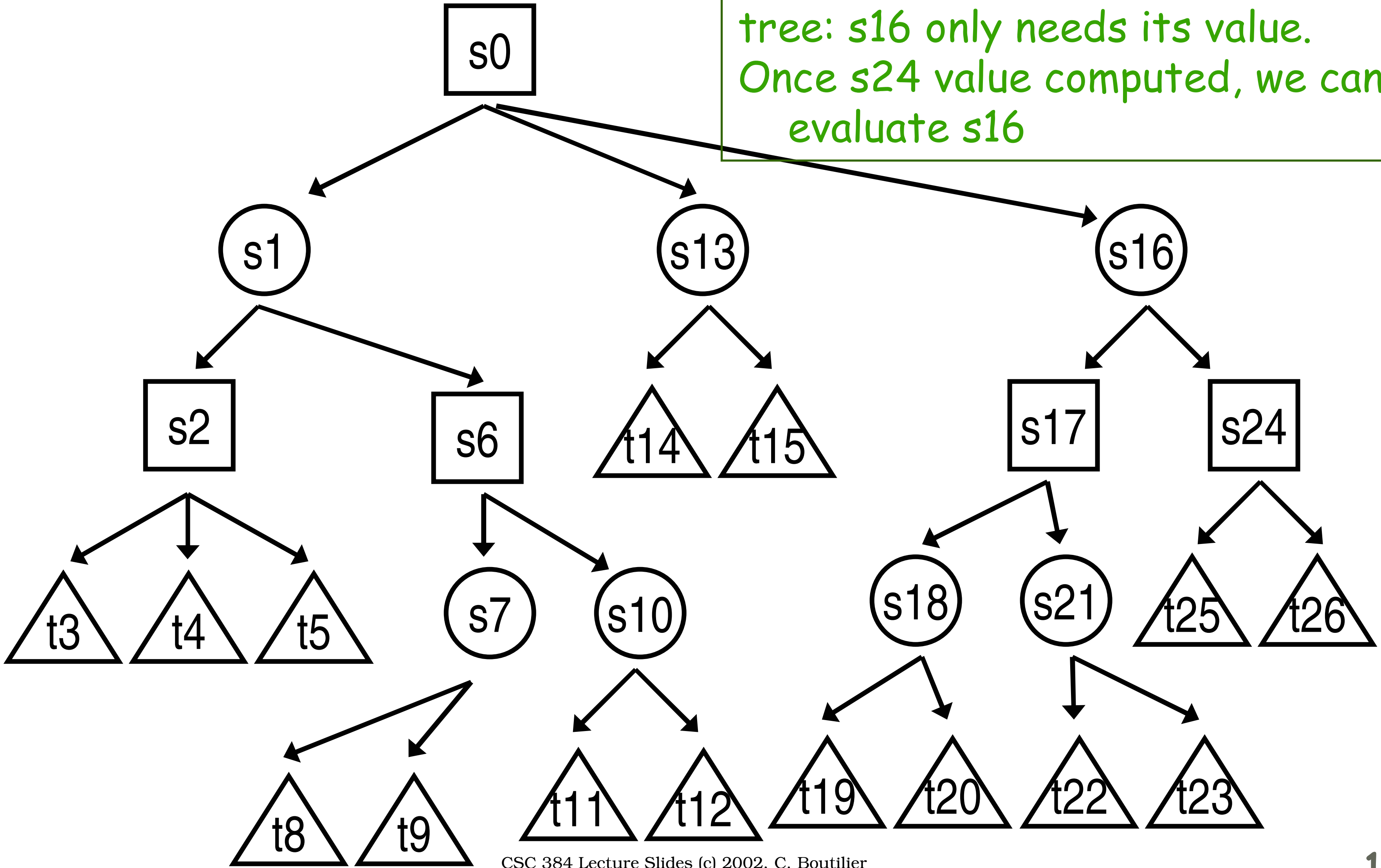
Depth-first Implementation of MMX

```
utility(N,L,U) :- maxLevel(L), children(N,CList),
                  maxValue(CList, U).
utility(N,L,U) :- minLevel(L), children(N,CList),
                  minValue(CList, U).
```

- maxValue recursively calls utility on each child and sets U to max of all values over these children (ditto for minValue)
  - note: can’t return until all children evaluated
- Notice that the game tree has to have finite depth for this to work
- Advantage of DF implementation: space efficient

Visualization of DF-MMX

Once s17 eval'd, no need to store tree: s16 only needs its value.
Once s24 value computed, we can evaluate s16



Pruning

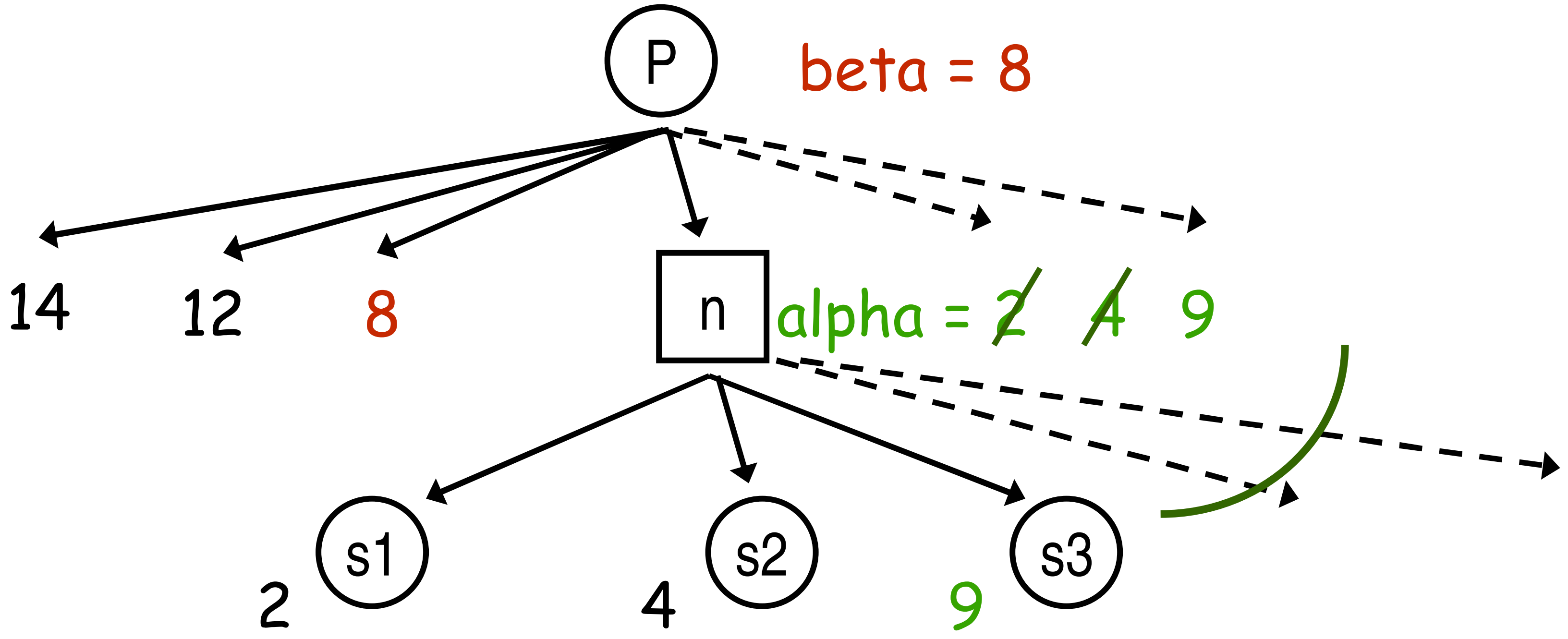
- It is not necessary to examine entire tree to make correct minimax decision
- Assume depth-first generation of tree
  - After generating value for only *some* of node *n*’s children we can prove that we’ll never reach *n*
  - So we needn’t generate or evaluate any further children of *n* !
- Two types of cuts:
  - α-cuts: pruning of max nodes
  - β-cuts: pruning of min nodes

Defining Alpha/Beta Values

- At a Max node *n*:
    - α is best value of *n*’s children examined so far (dynamic: changes as children examined)
    - β is worst value of *n*’s parent’s children examined so far (fixed when evaluating *n*)
- At a Min node *n*:
    - β is worst value of *n*’s children examined so far (dynamic: changes as children examined)
    - α is best value of *n*’s parent’s children examined so far (fixed when evaluating *n*)

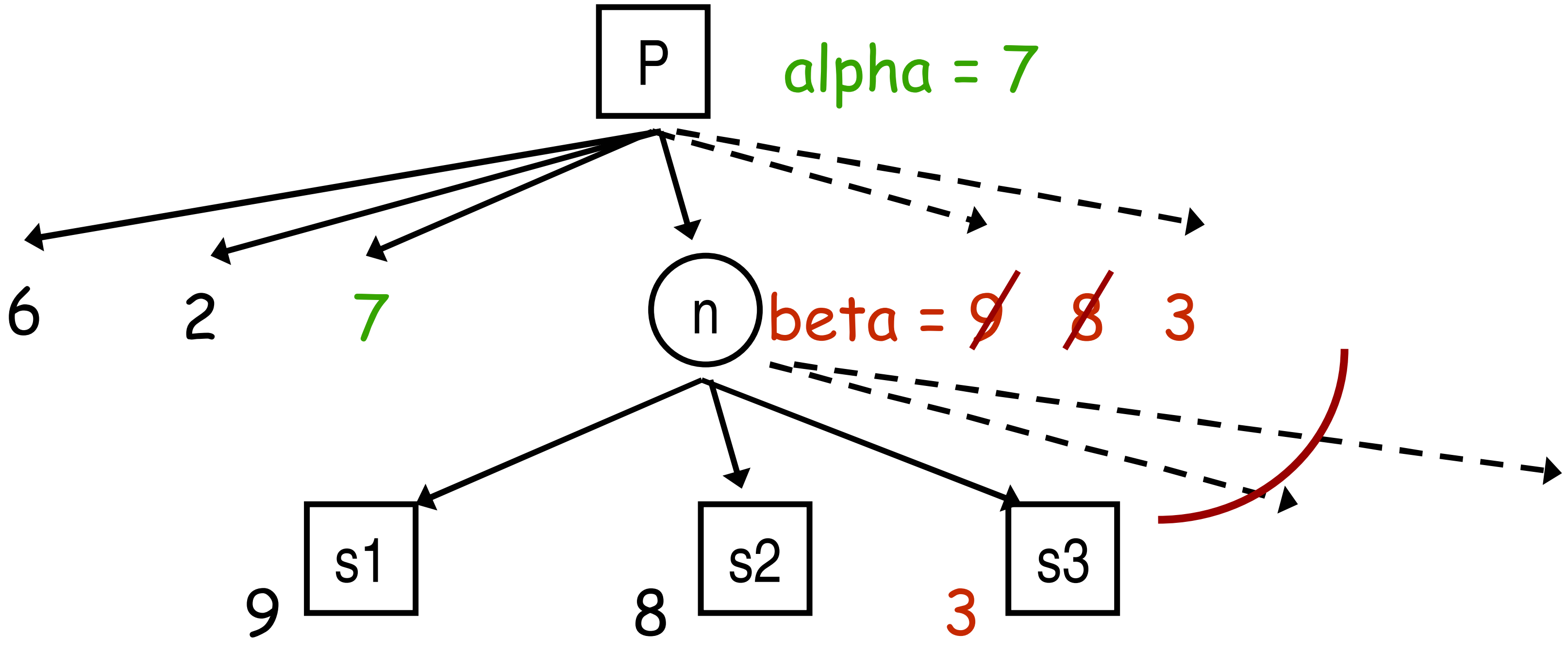
An Alpha Cut

- Once alpha-value at max node *n* reaches its beta-value (i.e., it's parent *P*'s beta-value), we can stop expansion of *n*
  - Min will never choose to move to *n* from *P* since it can guarantee the lower beta-value already



A Beta Cut

- Once beta-value at min node *n* reaches its alpha-value (i.e., it's parent *P*'s alpha-value), we can stop expansion of *n*
    - Max will never choose to move to *n* from *P* since it can guarantee the greater alpha-value already



Alpha-Beta Algorithm

Pseudo-code algorithm that associates a value with each node. Strategy extracted by moving to Max node (if you are player Max) at each step.

```
Evaluate(startNode):
  /* assume Max moves first */
  MavEval(start, -infnty, +infnty)
```

```
MaxEval(node, alpha, beta):
  If terminal(node), return  U(n)
  For each c in childlist(n)
      val ← MinEval(c, alpha, beta)
      alpha ← max(alpha, val)
      If alpha ≥ beta, return beta
  Return alpha

MinEval(node, alpha, beta):
  If terminal(node), return  U(n)
  For each c in childlist(n)
      val ← MaxEval(c, alpha, beta)
      beta ← min(beta, val)
      If alpha ≥ beta, return alpha
  Return beta
```

Rational Opponents

- This all assumes that your opponent is rational
  - e.g., will choose moves that minimize your max score
- Storing your strategy is a potential issue:
  - you must store “decisions” for each node you can reach by playing optimally
  - if your opponent has unique rational choices, this is a single branch through game tree
  - if there are “ties”, opponent could choose any one of the “tied” moves: must store strategy for each subtree
- What if your opponent doesn’t play rationally?
  - will it affect quality of outcome?
  - what to do if you haven’t stored a full strategy?

Practical Matters

- All “real” games are too large to enumerate tree
    - e.g., chess branching factor is roughly 35
    - Depth 10 tree: 2,700,000,000,000,000 nodes
    - Yikes! Even alpha-beta pruning won’t help here!
- We must limit depth of search tree
    - can’t expand all the way to terminal nodes
    - we must make *heuristic estimates* about the values of the (nonterminal) states at the leaves of the tree
    - *evaluation function* is an often used term
    - evaluation functions are often learned
- Depth-first expansion almost always used for game trees because of sheer size of trees

Heuristics

- Think of a few games and suggest some heuristics for estimating the “goodness” of a position
  - chess?
  - checkers?
  - your favorite video game?
  - “find the last parking spot”?

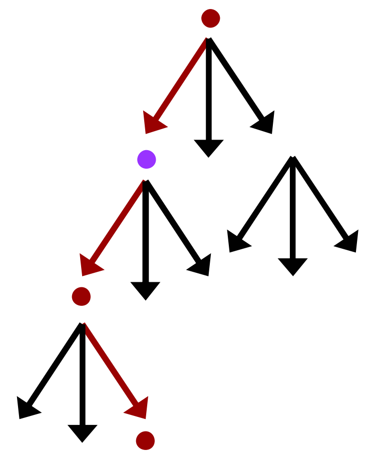
Some Interesting Games

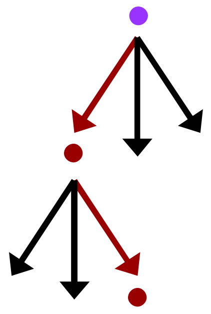
- Tesauro’s TD-Gammon
  - champion backgammon player which learned evaluation function; stochastic component (dice)
- Checker’s (Samuel, 1950s; Schaeffer)
- Chess (which you all know about)
- Bridge, Poker, etc.
- Check out Jonathan Schaeffer’s Web page:
  - www.cs.ualberta.ca/~games
  - they’ve studied lots of games (you can play too)

An Aside on Large Search Problems

- Issue: inability to expand tree to terminal nodes is relevant even in standard search
  - often we can't expect A* to reach a goal by expanding full frontier
  - so we often limit our lookahead, and make moves before we actually know the true path to the goal
  - sometimes called *online* or *realtime* search
- In this case, we use the heuristic function not just to guide our search, but also commits us to moves we actually make
  - in general, guarantees of optimality are lost, but we reduce computational/memory expense dramatically

Realtime Search Graphically





1. We run *A** (or our favorite search algorithm until we are forced to make a move or run out of memory. Note: no leaves are goals yet.

2. We use evaluation function f(n) to decide which path *looks* best (let's say it is the red one).

3. We take the first step along the best path (red), by actually *making that move*.

4. We restart search at the node we reach by making that move. (We may actually cache the results of the relevant part of first search tree if it's hanging around, as it would with *A**).