CSE 3402: Intro to Artificial Intelligence Game Tree Search

- Required readings: Chapter 5, sections 5.1, 5.2, 5.3, 5.7.

Generalizing Search Problems

- So far: our search problems have assumed agent has complete control of environment
  - state does not change unless the agent (robot) changes it.
    - makes a straight path to goal state feasible.
- Assumption not always reasonable
  - stochastic environment (e.g., the weather, traffic accidents).
  - other agents whose interests conflict with yours

Generalizing Search Problems

- In these cases, we need to generalize our view of search to handle state changes that are not in the control of the agent.
- One generalization yields game tree search
  - agent and some other agents.
  - The other agents are acting to maximize their profits
    - this might not have a positive effect on your profits.

Two-person Zero-Sum Games

- **Two-person, zero-sum games**
  - chess, checkers, tic-tac-toe, backgammon, go, "find the last parking space"
  - Your winning means that your opponent looses, and vice-versa.
  - Zero-sum means the sum of your and your opponent's payoff is zero---any thing you gain come at your opponent's cost (and vice-versa). Key insight:
  - how you act depends on how the other agent acts (or how you think they will act)
    - and vice versa (if your opponent is a rational player)

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
  - Each agent tries to alter the state so as to best benefit itself.

More General Games

- 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 amount of money I win (lose) at poker is the amount of money you lose (win)
- More general games can be "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, some examples of simple zero-sum and cooperative games

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 I \ Player II	R	P	S
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

	Coop	Def
Coop	3/3	0/4
Def	4/0	1/1

Game 3: Battlebots

- Two robots: Blue & Red
  - one cup of coffee, one tea left
  - both robots prefer coffee (value 10)
  - tea acceptable (value 8)
- Both robot’s go for Coffee
  - 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

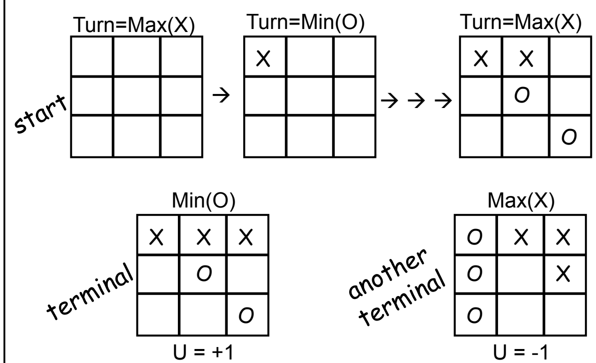
Two-Player, Zero-Sum Game: Defn

- Two *players* A (Max) and B (Min)
- set of *positions* P (states of the 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 between states (A’s *moves*)
- set of directed edges E_B between states (B’s *moves*)
- *utility* or *payoff function* $U : T \rightarrow \mathbb{R}$ (how good is each terminal state for player A)
  - why don’t we need a utility function for B?

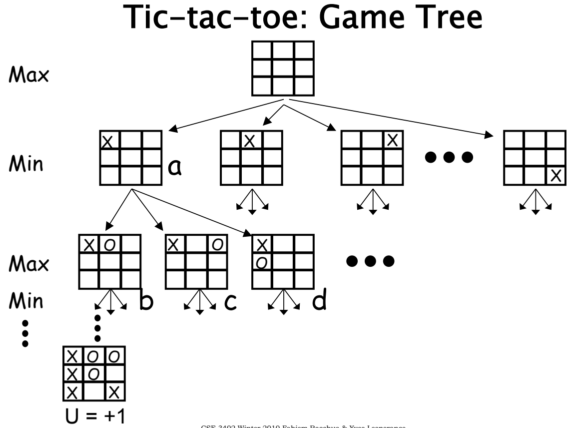
Intuitions

- Players alternate moves (starting with Max)
  - Game ends when some terminal p ∈ 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



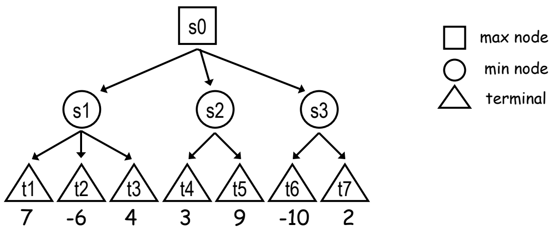
Tic-tac-toe: Game Tree



Game Tree

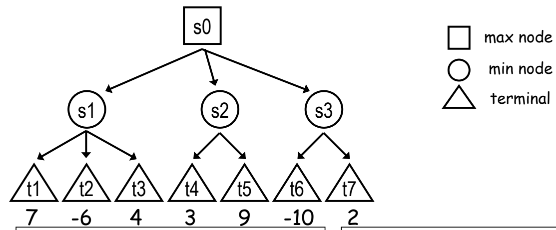
- Game tree looks like a search tree
  - Layers reflect the alternating moves
- But Max doesn't decide where to go alone
  - after Max moves to state a, Min decides whether to move to state b, c, or d
- Thus Max must have a *strategy*
  - must know what to do next no matter what move Min makes (b, c, 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



The terminal nodes have utilities.
But we can compute a "utility" for the non-terminal states, by assuming both players always play their best move.

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 s2: 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

Minimax Strategy

- The values labeling each state are the values that Max will achieve in that state if both he and Min play their best moves.
  - Max plays a move to change the state to the highest valued min child.
  - Min plays a move to change the state to the lowest valued max child.
- If Min plays poorly, Max could do better, but never worse.
  - If Max, however know that Min will play poorly, there might be a better strategy of play for Max than minimax!

Depth-first Implementation of MinMax

```
utility(N,U) :- terminal(N), utility(N,U).
utility(N,U) :- maxMove(N), children(N,CList),
                utilityList(CList,UList),
                max(UList,U).
utility(N,U) :- minMove(N), children(N,CList),
                utilityList(CList,UList),
                min(UList,U).
```

- Depth-first evaluation of game tree
  - terminal(N) holds if the state (node) is a terminal node. Similarly for maxMove(N) (Max player's move) and minMove(N) (Min player's move).
  - utility of terminals is specified as part of the input

Depth-first Implementation of MinMax

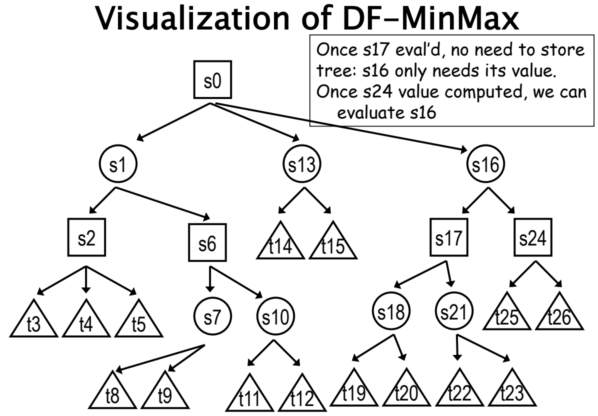
```
utilityList([],[]).
utilityList([N|R],[U|UList])
        :- utility(N,U),utilityList(R,UList).
```

- utilityList simply computes a list of utilities, one for each node on the list.
- The way Prolog executes implies that this will compute utilities using a depth-first post-order traversal of the game tree.
  - post-order (visit children before visiting parents).

Depth-first Implementation of MinMax

- Notice that the game tree has to have finite depth for this to work
- Advantage of DF implementation: space efficient

Visualization of DF-MinMax

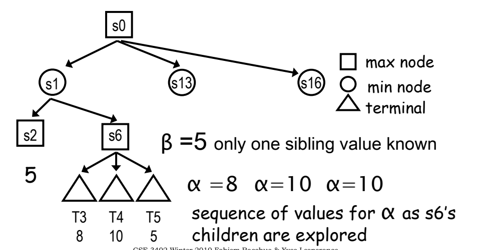


Pruning

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

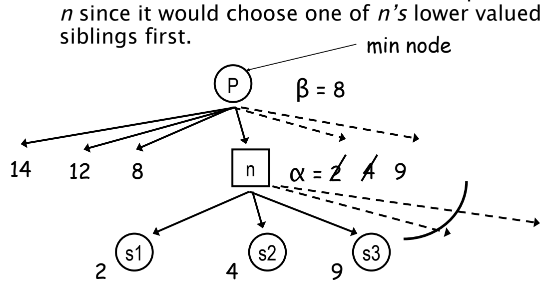
Cutting Max Nodes (Alpha Cuts)

- At a Max node *n*:
  - Let β be the lowest value of *n*'s siblings examined so far (siblings to the left of *n* that have already been searched)
  - Let α be the highest value of *n*'s children examined so far (changes as children examined)



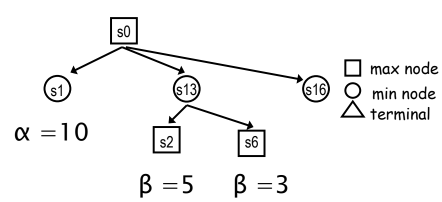
Cutting Max Nodes (Alpha Cuts)

- If α becomes ≥ β we can stop expanding the children of *n*
  - Min will never choose to move from *n*'s parent to *n* since it would choose one of *n's* lower valued siblings first.



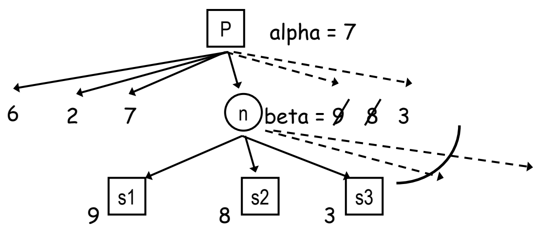
Cutting Min Nodes (Beta Cuts)

- At a Min node *n*:
  - Let β be the lowest value of *n*'s children examined so far (changes as children examined)
  - Let α be the highest value of *n*'s sibling's examined so far (fixed when evaluating *n*)



Cutting Min Nodes (Beta Cuts)

- If β becomes ≤ α we can stop expanding the children of *n.*
  - Max will never choose to move from *n*'s parent to *n* since it would choose one of *n's* higher value siblings first.



Alpha-Beta Algorithm

Pseudo-code 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 */
  MaxEval(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 alpha
  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 beta
  Return beta
```


Rational Opponents

- This all assumes that your opponent is rational
  - e.g., will choose moves that minimize your score
- What if your opponent doesn't play rationally?
  - will it affect quality of outcome?

Rational Opponents

- 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 your stored strategy still work?

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
  - Even alpha-beta pruning won't help here!

Practical Matters

- 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"?

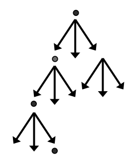
Some Interesting Games

- Tesauro's TD-Gammon
  - champion backgammon player which learned evaluation function; stochastic component (dice)
- Checkers: Chinook 1990s by Schaeffer; solved game in 2005-07
- 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)
- General Game Playing Competition

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 to commits to moves we actually make
  - in general, guarantees of optimality are lost, but we reduce computational/memory expense dramatically

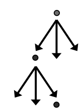
Realtime Search Graphically



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*).