CSC384: Intro to Artificial Intelligence Game Tree Search II

- Midterm: Everything covered so far plus this lecture slides.

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

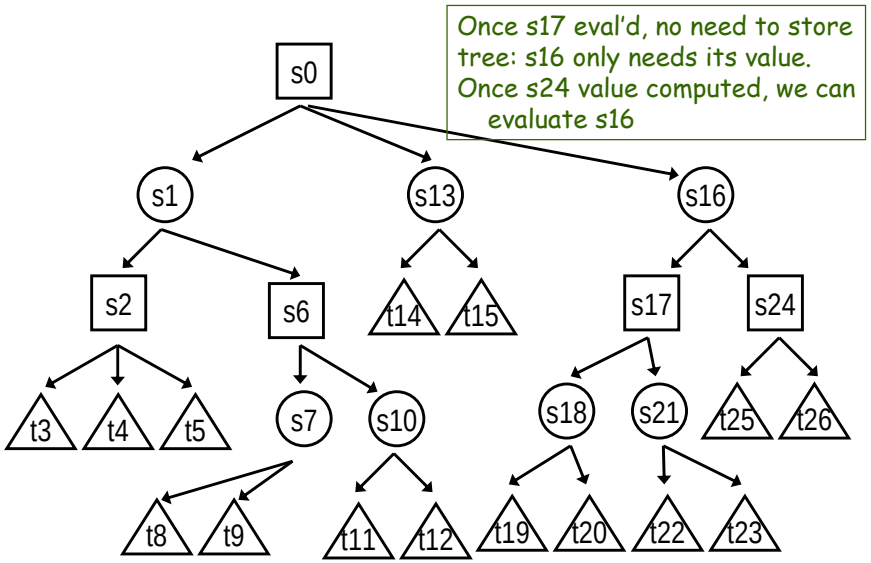
```
utilityList([],[]).
utilityList([N|R],[U|UList])
            :- utility(N,U), utilityList(R,UList).
```

- 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

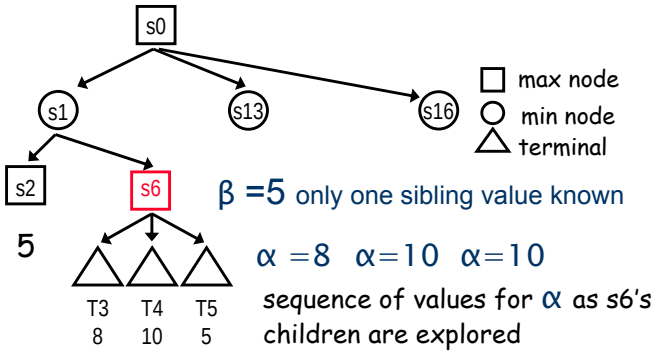


Pruning

- It is 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)

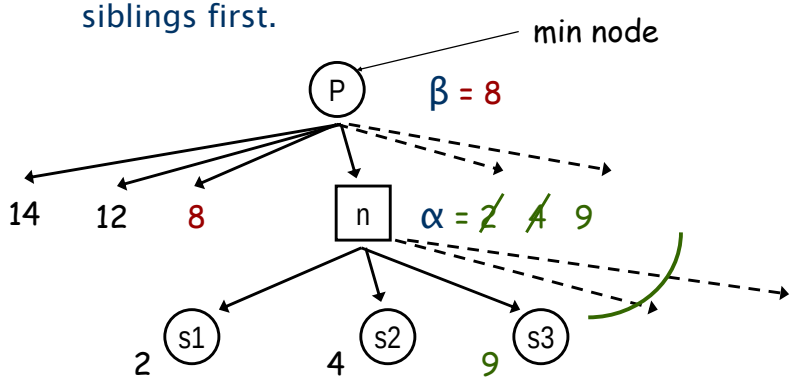
Cutting Max Nodes (Alpha Cuts)

- 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)



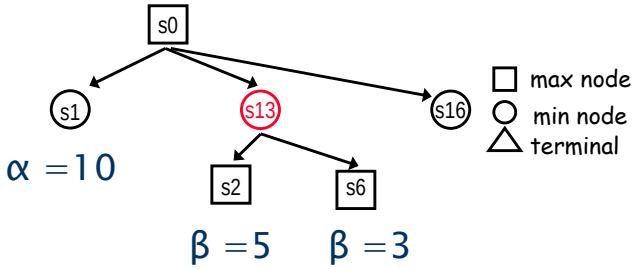
Cutting Max Nodes (Alpha Cuts)

- 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.



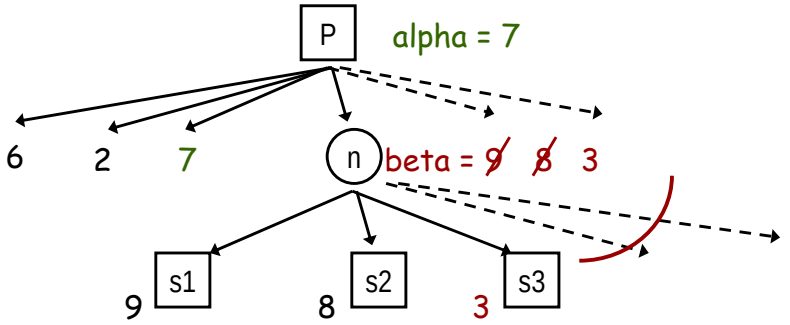
Cutting Min Nodes (Beta Cuts)

- 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"?

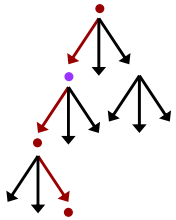
Some Interesting Games

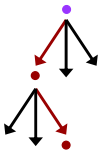
- Tesauro's TD-Gammon
  - champion backgammon player which learned evaluation function; stochastic component (dice)
- Checker's (Samuel, 1950s; Chinook 1990s Schaeffer)
- Chess (which you all know about)
- Bridge, Poker, etc.
- Check out Jonathan Schaeffer's Web page:
  - www.cs.ualberta.ca/~games
  - 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 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*).