COMS W4701: Artificial Intelligence

Lecture 7: Optimal Game Playing

Tony Dear, Ph.D.
Department of Computer Science
School of Engineering and Applied Sciences

Today

- Adversarial search problems
- Minimax search
- Alpha-beta pruning
- Move ordering

Multi-Agent Environments

- Consider environment with multiple autonomous agents
- Each agent maintains own state about itself and other agents
- Each agent tries to maximize its own utility, dependent on env outcomes

- Fully **cooperative** if all agents share the same utility function
- Fully **competitive** if an agent effectively seeks to *minimize* others' utilities
  - **Zero-sum** if sum of all utilities in any outcome is always zero

- Most multi-agent environments (*games*) lie somewhere on this spectrum

Competitive Environments

- Several different approaches for competitive, multi-agent environments
- *Aggregate*, e.g., as in an economy; study overall effects and properties without worrying about individual agents

- Consider other agents to be part of a *nondeterministic* environment
- Our agent may perform better if we consider other agents' strategies

- Come up with a *strategy* using **adversarial search** and assuming that other agents are also rational actors

Computer Checkers, Chess, and Go

- 1994: *Chinook* declared computer champion in checkers against Tinsley
- 2007: Checkers is **solved** (predictable from any position assuming perfect play)

- 1997: *Deep Blue* defeats chess world champion Kasparov
- 2016: *AlphaGo* defeats go world champion Lee Sedol
- 2017: *AlphaZero* defeats *Stockfish*, AI chess champion

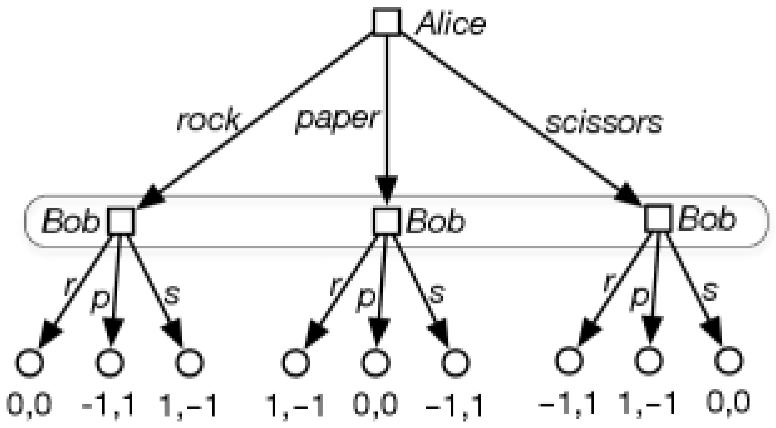
- 2018-today: *Stockfish* continues to improve using alpha-beta search methods
- *Leela Chess Zero* demonstrates comparable performance using methods based on *AlphaZero* (reinforcement learning, self-play)

Adversarial Search

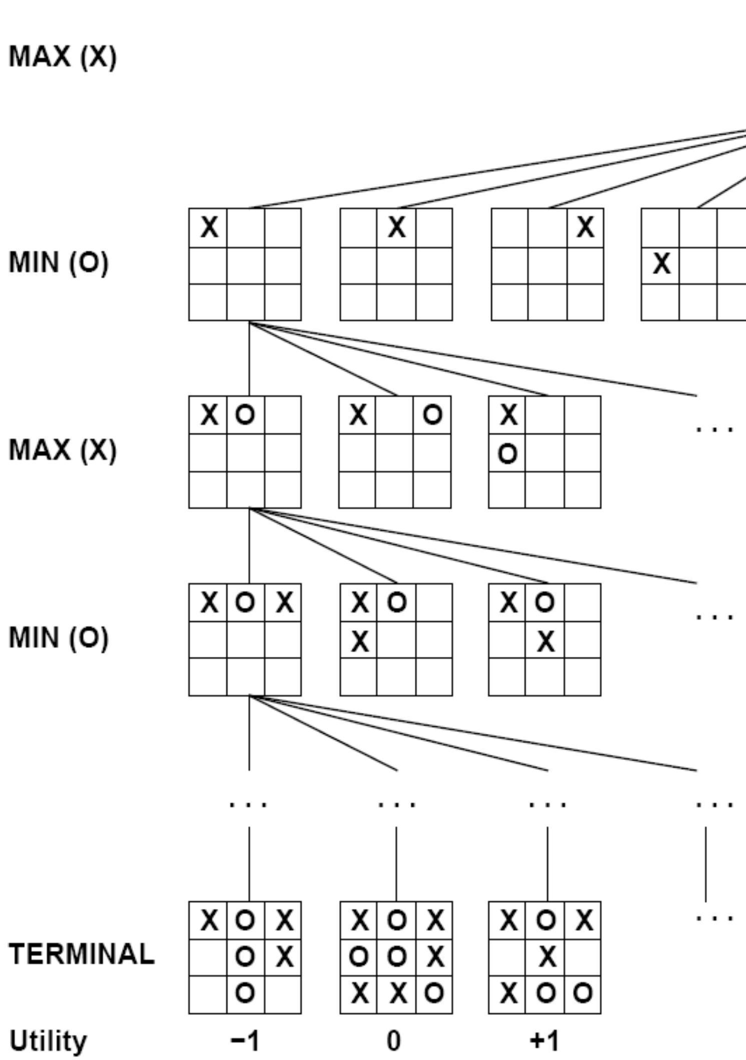
- Assumption: *Perfect information* (fully observable) and turn-taking game
- Represent game progression (and later search) in a **game tree**

- Each layer (*ply*) of internal nodes and edges is controlled by one agent
- From a given state, the controlling agent may take one action (edge)

- Leaf nodes represent *terminal states*
- **Utility function** specifies utility values for each player at each terminal state



Game Trees



- Game trees can be *huge*, even for games like tic-tac-toe ($O(10^5)$ leaves)
- Chess has $O(10^{123})$, go has $O(10^{360})$!

- If a player wants to take optimal action, they can *search* the tree down to leaf nodes to see all possible utility outcomes

Minimax Values

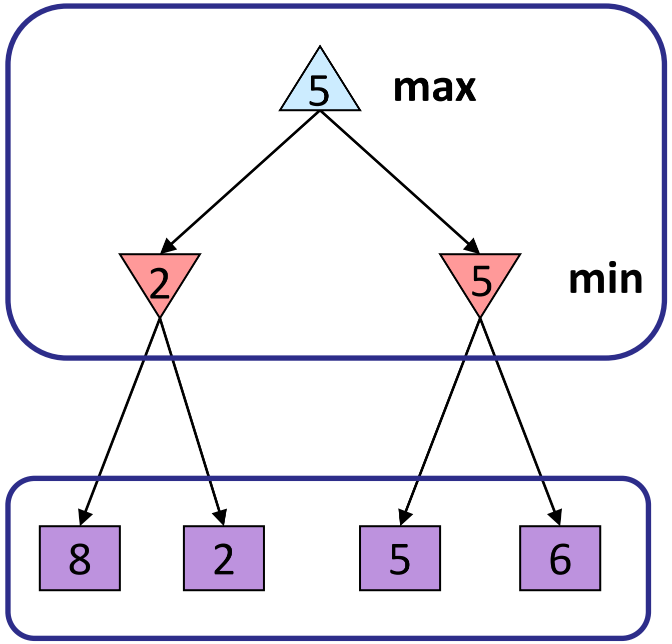
- Suppose we have a 2-player, zero-sum game
- Utility function only assigns one value to a terminal state
- One player wants to maximize; other wants to minimize

- **Minimax value**: Value of a state assuming each player always plays *optimally* at every game state

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

- We can compute all values by *searching* through the tree!

Minimax values: computed *upward* from terminal states



Terminal values: part of the game

Minimax Search Algorithm

function MINIMAX-SEARCH(*game*, *state*) **returns** *an action*
  player $\leftarrow$ *game*.TO-MOVE(*state*)
  value, *move* $\leftarrow$ MAX-VALUE(*game*, *state*)
  return *move*

Assuming root is MAX

function MAX-VALUE(*game*, *state*) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow -\infty$
  for each a **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* $\leftarrow$ MIN-VALUE(*game*, *game*.RESULT(*state*, *a*))
    if $v2 > v$ **then**
      v, *move* $\leftarrow$ $v2$, a
  return v, *move*

Compute and return max over successors, all of which are MIN

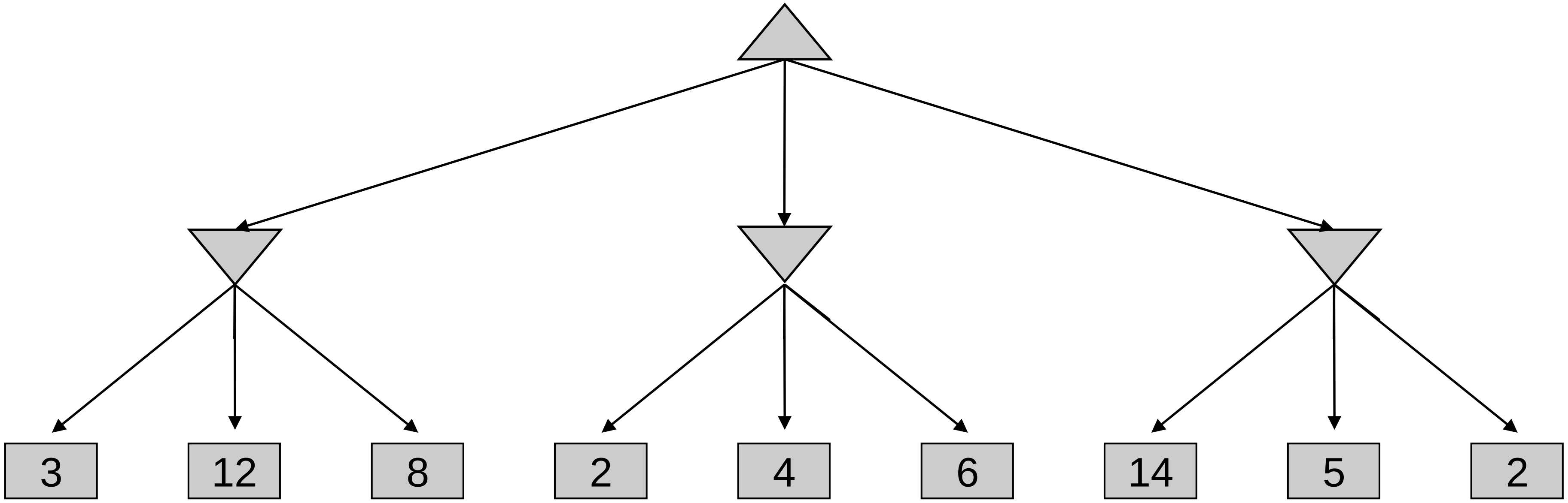
function MIN-VALUE(*game*, *state*) **returns** a (*utility*, *move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state*, *player*), *null*
  $v \leftarrow +\infty$
  for each a **in** *game*.ACTIONS(*state*) **do**
    v2, *a2* $\leftarrow$ MAX-VALUE(*game*, *game*.RESULT(*state*, *a*))
    if $v2 < v$ **then**
      v, *move* $\leftarrow$ $v2$, a
  return v, *move*

Compute and return min over successors, all of which are MAX

Minimax Example

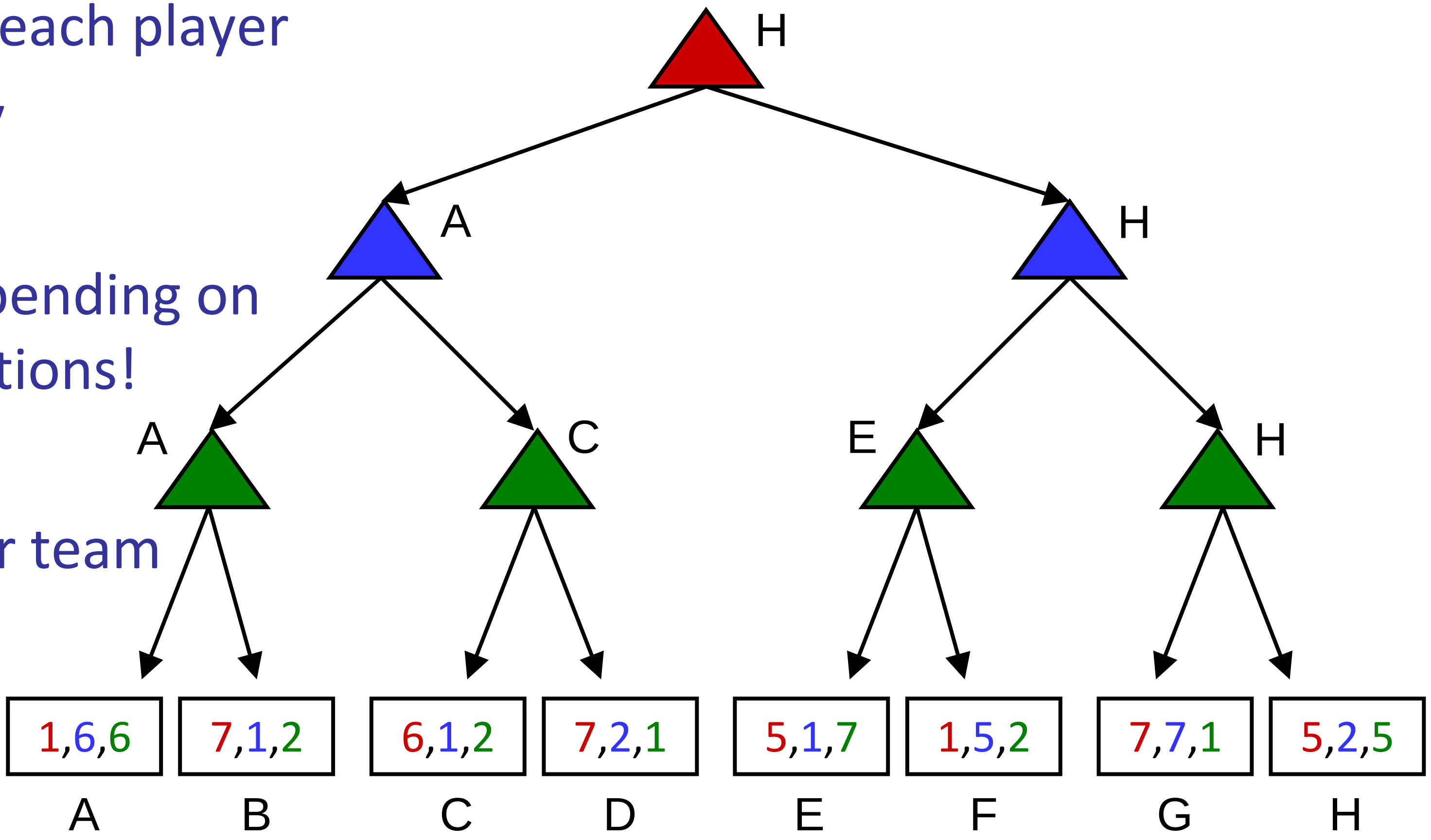
function MAX-VALUE($game$, $state$) **returns** a ($utility$, $move$) pair
  if $game$.IS-TERMINAL($state$) **then return** $game$.UTILITY($state$, $player$), $null$
  $v \leftarrow -\infty$
  for each a **in** $game$.ACTIONS($state$) **do**
    $v2$, $a2 \leftarrow$ MIN-VALUE($game$, $game$.RESULT($state$, a))
    if $v2 > v$ **then**
      v, $move \leftarrow v2$, a
  return v, $move$

function MIN-VALUE($game$, $state$) **returns** a ($utility$, $move$) pair
  if $game$.IS-TERMINAL($state$) **then return** $game$.UTILITY($state$, $player$), $null$
  $v \leftarrow +\infty$
  for each a **in** $game$.ACTIONS($state$) **do**
    $v2$, $a2 \leftarrow$ MAX-VALUE($game$, $game$.RESULT($state$, a))
    if $v2 < v$ **then**
      v, $move \leftarrow v2$, a
  return v, $move$



Non-Zero-Sum Games

- In non-zero-sum games, terminal states must record separate utilities for each player
- Each player maximizes own utility
- Different strategies may arise depending on the players' individual utility functions!
- May see cooperation, alliances, or team beahvior if utilities are aligned



Minimax Efficiency

- Minimax is a form of DFS: Time complexity $O(b^m)$, space $O(bm)$
- Optimal if both players play perfectly, complete if game eventually ends

- Only games with sufficiently small game trees can be solved!
- Example: Chess has $b \approx 35, m \approx 80$

- We will need optimizations or approximations to speed up search
- Pruning the game tree, move ordering, depth limits, heuristic approaches

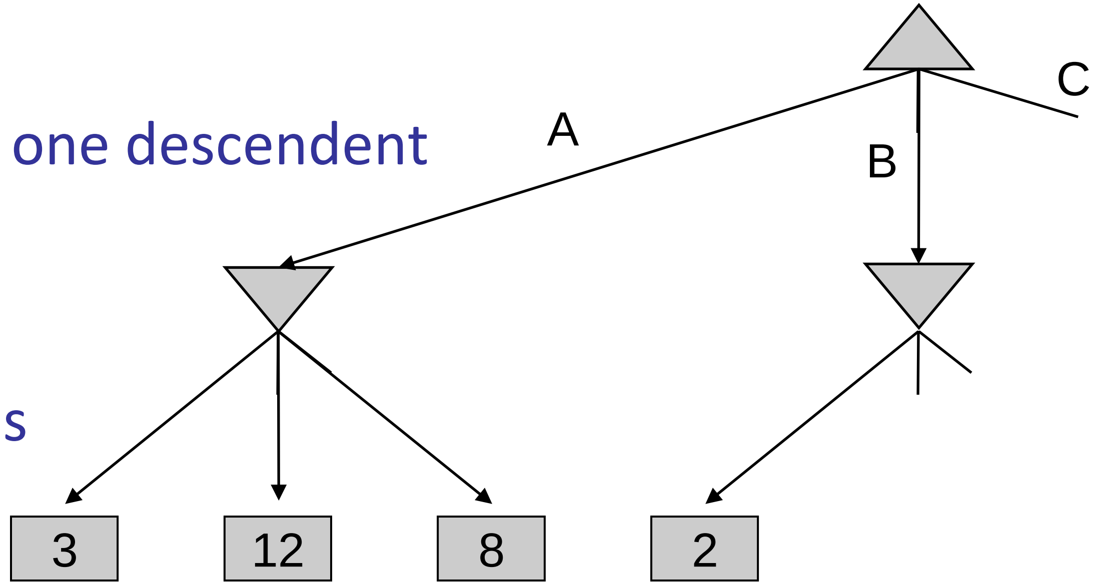
Pruning

- Suppose minimax is searching left to right
- We have finished searching branch A and computed its value (3)

- We start searching branch B and see value of one descendent
- Suppose its value is *smaller* than that of A (2)

- We know successors of root are all MIN nodes
- Regardless of other children, $v(\mathrm{B}) \leq 2$

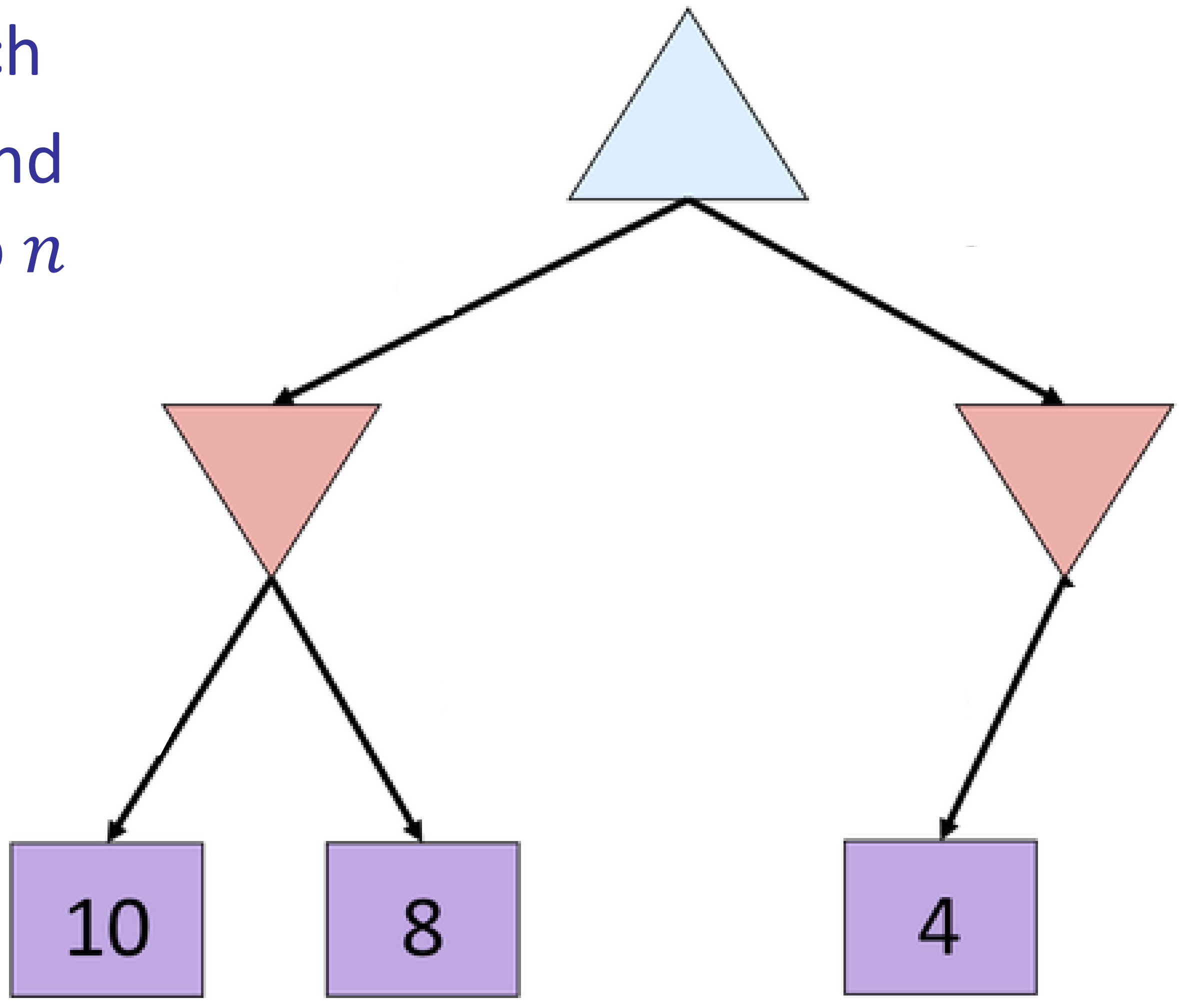
- Root MAX node will never consider B; skip (prune) and move on to C



Alpha-Beta Pruning

- Update two new values during search
- A node n is passed the highest (α) and lowest (β) values seen *along path* to n

- Skip remaining actions (prune) if:
- MAX sees new value higher than β, since MIN parent will prefer β
- MIN sees new value lower than α, since MAX parent will prefer α



Alpha-Beta Search

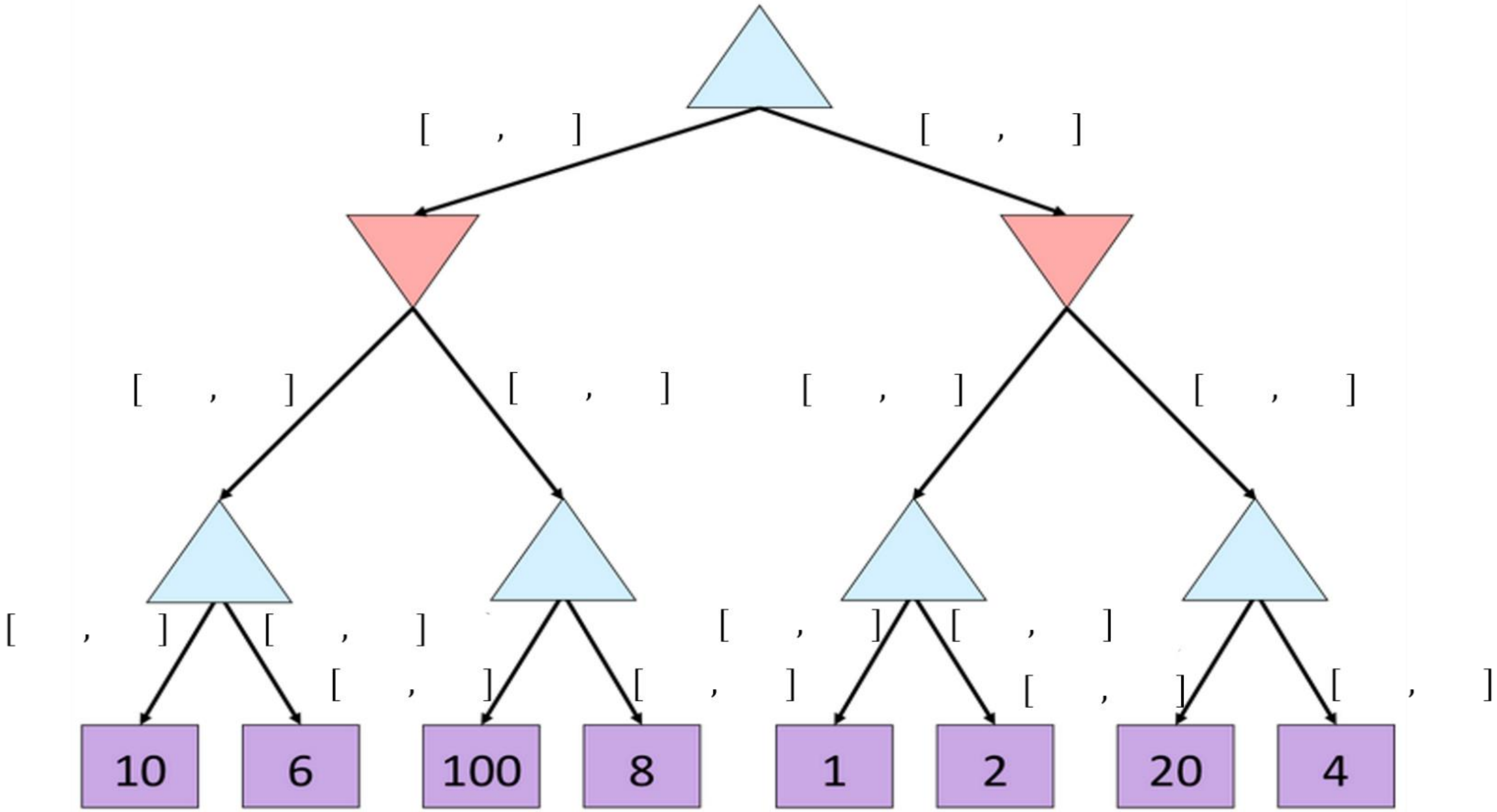
```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Assuming root is MAX

MAX updates α; if node value is $\geq \beta$, prune remaining successors

MIN updates β; if node value is $\leq \alpha$, prune remaining successors

Alpha-Beta Example

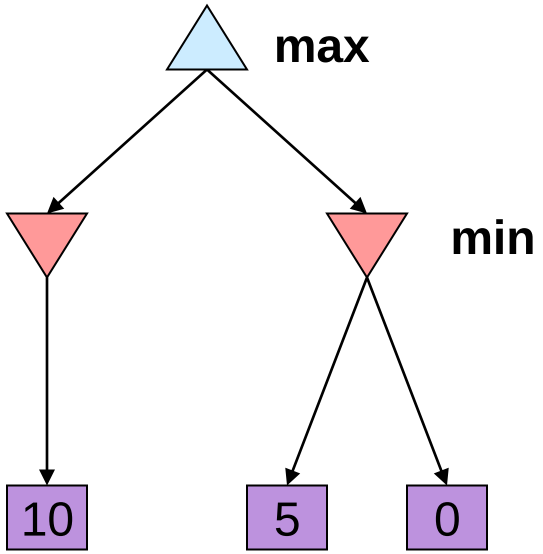


Alpha-Beta Properties

- Pruning still results in correct minimax value at root

- Intermediate (children) node values might be wrong!!

- If pruning, game tree values cannot be reused
- Will have to rerun minimax after each move



- In practice, we can store just the states/values that we know to be correct in a *transposition table* in case they come up again
  - Especially effective when there are multiple paths to a state

Move Ordering

- Good *move ordering* improves effectiveness of pruning
- Alpha-beta with random ordering is roughly ~$O(b^{0.75m})$

- "Perfect ordering" gets us to $O(b^{0.5m})$, doubling solvable depth
- Usually based on domain or expert knowledge
  - Simple chess ordering function: captures, threats, forward moves, backward moves

- Iterative deepening: Since upper game tree layers will be expanded multiple times, can also keep track of their values in order to optimally order them in future iterations

- Search of repeated states can be avoided by caching them in a transposition table

Summary

- Adversarial search can be used to solve multi-agent problems

- Minimax is well suited for two-player, zero-sum, and turn-taking games
- Utility values computed at leaf nodes, backed up to non-end states

- Alpha-beta pruning can make more search more efficient by avoiding useless portions of the game tree
- Effective move ordering and caching can further improve search efficiency